Lancelot Pecquet

Programmation Fonctionnelle

Introduction illustrée en *Objective Caml*

$$\omega = \Big(\lambda x \underbrace{(x\ x)}_{M}\ \underbrace{\lambda x\,(x\ x)}_{A}\Big)$$

Université Paris XII, Licence d'Informatique

Version du 7 mai 2002

Table des matières

Chapitre 1

Introduction

1.1 Qu'attendre de ce document?

1.1.1 Objectifs

Ce document a pour objectif principal de faire découvrir, à un public débutant en la matière, les fondements de la programmation fonctionnelle. L'excellent langage *Objective Caml* est utilisé pour illustrer les concepts théoriques. Ces quelques pages ne constituent aucunement un manuel de référence du langage *Objectif Caml* (Ocaml) et il conviendra de se reporter pour cela au manuel en ligne du langage Ocaml, disponible sur `http://caml.inria.fr`. Des lectures complémentaires pour approfondir les sujets abordés (voire effleurés...) dans cette introduction sont suggérées dans la conclusion, p. 39.

1.1.2 Qu'est-ce que la programmation fonctionnelle?

On classe souvent les langages de programmation en trois familles:

1. impératifs (comme les assembleurs, C, Perl, ...);
2. objets (comme C++, Java,...);
3. fonctionnels (comme lisp, scheme,...).

En fait, cette classification doit plus s'entendre comme des philosophies différentes dans l'approche de la programmation et les langages implémentent même souvent plusieurs de ces aspects.

Un programme est la description des étapes qu'un ordinateur doit effectuer pour réaliser l'objectif du programmeur (dans le meilleur des cas...). La brique élémentaire constituant ces étapes s'appelle une *instruction*.

En caricaturant un peu, on peut dire que les *fonctions* (ou les procédures), vues du point de vue impératif ou objet, ne sont qu'un gadget pratique pour regrouper et réutiliser des instructions qu'il serait fatigant ou inefficace de réécrire.

La programmation fonctionnelle se propose de donner aux fonctions un statut central. Cela se justifie en constatant qu'en un certain sens, « programme » et « fonction » sont deux notions équivalentes. Par exemple, un programme qui convertit les francs en euros correspondra à la fonction $f : x \longmapsto x/6.55957$.

Ces trois philosophies sont complémentaires en fonction des situations. Le cœur du langage *Objective Caml* (Ocaml, développé par le projet Cristal de l'INRIA, est disponible gratuitement sur `http://caml.inria.fr`) est fonctionnel mais il permet également de faire de la programmation impérative ou orientée objet lorsque cela est nécessaire.

1.1.3 Intérêt de la programmation fonctionnelle

La programmation fonctionnelle est fondée sur un ensemble de théories mathématiques puissantes qui garantissent un contrôle sur les programmes. Grâce à cette intéraction entre les mathématiques et l'informatique, il devient alors possible de *prouver* qu'un programme répond à un cahier des charges spécifié.

Si cette possibilité peut être considérée comme une garantie incontestablement supérieure à une serie de tests (qui restent nécessaires, toutefois, ne serait-ce que parce que la spécification a pu être mal faite!) et est souhaitable dans tout programme, elle peut s'avérer critique dans certaines circonstances comme lors du pilotage de centrales nucléaires, de dispositifs médicaux, *etc.*

Du fait de leur grande structure mathématique, les langages fonctionnels permettent également d'être interfacés avec des systèmes de démonstration automatique de théorèmes, comme le système *Coq*, développé par le Projet Logical de l'INRIA (interfacé avec Ocaml `http://logical.inria.fr/`).

Ocaml certainement un des meilleurs langages de programmation à l'heure actuelle. Il bénéficie à la fois de bases théoriques solides, d'une syntaxe très expressive (quoiqu'un peu déroutante, au début, pour les programmeurs n'ayant pas l'habitude des langages fonctionnels) et d'une implémentation redoutablement efficace. Il arrive en deuxième position (après `gcc`) du *Shootout Scorecard* de Doug Bagley disponible à `http://www.bagley.org/~doug/shootout/craps.shtml`.

1.2 Qu'est-ce qu'une fonction?

1.2.1 Définition mathématique

La première nécessité, lorsqu'on adopte la démarche « fonctionnelle », est de définir précisément ce qu'on entend par « fonction ». Comme nous souhaitons rester proche des mathématiques, on peut, dans un premier temps, explorer la définition mathématique de « fonction »:

Définition 1 *Une **relation** sur $E \times F$ est un sous-ensemble $R \subseteq E \times F$. Elle est **fonctionnelle** ssi:*

$$\forall x \in E \ \exists! y \in F \mid (x, y) \in R \ .$$

*On note $f : E \longrightarrow F$, $x \longmapsto f(x)$ où $R = \{(x, f(x)) \ : \ x \in E\}$. Une relation fonctionnelle s'appelle une **fonction**. L'ensemble E s'appelle le **domaine** de f et F son **codomaine**.*

1.2.2 Spécification par une propriété

Lorsqu'on définit un objet en mathématique, on procède souvent ainsi:

1. on donne une propriété P (qui peut être une conjonction de propriétés) que l'objet doit vérifier, par exemple, pour une fonction f:

$$f : \mathbb{R} \longrightarrow \mathbb{R} \quad \text{et} \quad \forall x,\ f(x) \geq 0 \quad \text{et} \quad \forall x,\ f(x) \times f(x) \geq x \ .$$

2. on montre qu'il existe un tel objet vérifiant P;

3. on s'assure éventuellement qu'il est unique;

4. on le nomme; ici on appelera f *racine carrée.*

et on peut l'exprimer avec l'*opérateur de description*: $[f \mid P(f)]$ qui se lit « la fonction f telle que $P(f)$ ». La fonction précédente se réécrit:

$$\text{sqrt} = [f \mid \mathbb{R} \longrightarrow \mathbb{R} \text{ et } \forall x,\ f(x) \geq 0 \text{ et } \forall x,\ f(x) \times f(x) \geq x]\ .$$

Lorsqu'on a pas unicité, l'opérateur de description sert à désigner un représentant de la classe des objets vérifiant ces propriétés; on l'appelle alors *opérateur de choix* [1] Par exemple, si l'on parle d'un objet $[x \mid x \times x = 9]$, on ne peut pas démontrer $x = -3$ ou $x = 3$ mais on peut montrer $x \in \{-3, 3\}$. Remarquons enfin que, si cette définition est parfaitement recevable, elle ne donne aucune information sur la manière dont on peut calculer $\sqrt{x} = \text{sqrt}(x)$, étant donné un réel x.

1.2.3 Notion de fonction « calculable »

La fonction *carré* $f : \mathbb{N} \longrightarrow \mathbb{N}$ est définie par un nombre fini de symboles représentant une expression « calculable » $x \longmapsto x^2 = x \times x$. Il se trouve que l'immense majorité des fonctions dont on a besoin en pratique ne peut pas s'écrire de cette manière. Nous introduirons les trois définitions principales de la notion de fonction « calculables » qui ont été proposées; il s'agit des fonctions:

- λ-définissables;
- récursives;
- calculables par machine de TURING.

Ces définitions de fonction calculable seront données pour des fonctions dont les valeurs et les arguments appartiennent à des ensembles dénombrables (du fait de la nature discrète d'un calcul, qui se fait étape par étape). Il s'agira la plupart du temps de chaînes de caractère ou d'entiers.

1. On trouve également la terminologie *opérateur ϵ de* HILBERT ou *opérateur τ de* BOURBAKI.

Chapitre 2

Théorie de la programmation fonctionnelle

2.1 λ-calcul

2.1.1 Introduction

Le λ-calcul a été inventé par Alonzo CHURCH dans les années 1930. Il permet de donner une définition calculatoire de la notion de fonction et est le fondement des langages fonctionnels. L'une des motivations qui a conduit à sa création était le désir de donner un sens à la notion de substitution des variables liées dans les définitions de fonctions. On voudrait en effet être en mesure d'exprimer le fait que les deux expressions: $f(x) = x + 1$ et $f(y) = y + 1$ définissent toutes les deux la fonction successeur. Les variables x et y dans ces définitions sont *liée*, dans le sens où si l'on remplaçait par exemple toutes les occurences de x où de y par la variable z, on aurait deux expressions égales à $f(z) = z + 1$.

Le λ-calcul est constitué d'expressions, appelées λ-*termes* comprenant des variables comme $x, y, z, \ldots$, des parenthèses et la lettre grecque λ (lambda), et de règles de transformation de ces expressions: c'est la partie « calculatoire » du λ-calcul.

2.1.2 λ-termes purs

Définition 2 *Soit V un alphabet de variables, et $\Sigma = V \sqcup \{\lambda, (,)\}$, le langage des **$\lambda$-termes** sur V est le plus petit sous-ensemble Λ de Σ^* tel que:*

- *si $x \in V$ alors $x \in \Lambda$;*
- *si $x \in V$ et $M \in \Lambda$ alors $\lambda x\, M$ appartient à Λ;*
- *si $M_1 \in \Lambda$ et $M_2 \in \Lambda$ alors $(M_1\ M_2)$ appartient à Λ.*

*Une occurence de la variable x dans le λ-terme A est **liée** dans A ssi elle apparaît comme sous-expression de la forme $\lambda x\, M$ dans A; autrement, elle est dite **libre**. On emploiera la notation abrégée $(M_1\ M_2\ \cdots M_n)$ pour le λ-terme $\big((M_1\ M_2) \cdots M_n\big)$. Un λ-terme de la forme $(\lambda x\, M\ A)$ s'appelle un λ-terme **opérateur-opérande**.*

Exemple 1 *Soit $V = \{x, y\}$, alors x, $\lambda x\, x$, $(x\ y)$, $\lambda x \lambda y\, y$ sont des λ-termes.*

Intuitivement, le terme $f = \lambda x\, M$ correspond à la fonction $f : x \longmapsto M$ et le terme $(f\ A)$ correspond à la valeur $f(A)$, ce qui justifie la terminologie « opérateur-opérande ».

2.1.3 λ-termes typés

Dans la Définition 1, p. 6, on explique ce qu'est une fonction $f : E \longrightarrow F$. Or le langage des λ-termes purs ne permet pas d'exprimer ce que sont le domaine E et le codomaine F de f et si l'on veut appliquer l'interprétation intuitive donnée dans la section précédente au λ-terme $\lambda x\ (x\ x)$, on se rend compte que ce dernier ne correspond pas à une fonction au sens ensembliste.

En 1940, CHURCH a introduit la *hiérarchie des types simples*, issue de celle de WHITEHEAD et RUSSEL, permettant d'introduire d'une part ces restrictions ensemblistes, d'autre part d'éviter des sitations du type « paradoxe de RUSSEL [1] ». L'ordre d'un type mesure le « degré de fonctionnalité » de la fonction qu'il représente.

Définition 3 *Soient $\mathcal{E}$ un ensemble dénombrable de symboles dits de **types élémentaires** et $\Sigma = \mathcal{E} \sqcup \{\rightarrow\}$, le langage des **types** induits par $\mathcal{E}$ est le plus petit sous-ensemble $\mathcal{T}$ de Σ^* tel que:*

- *si $\mathsf{A} \in \mathcal{E}$ alors $\mathsf{A} \in \mathcal{T}$;*
- *si $\mathsf{A}, \mathsf{B} \in \mathcal{T}$ alors $(\mathsf{A} \longrightarrow \mathsf{B}) \in \mathcal{T}$.*

*On désignera par $\mathsf{A}_1 \longrightarrow \cdots \longrightarrow \mathsf{A}_n \longrightarrow B$ le type $\mathsf{A}_1 \longrightarrow \big(\cdots \longrightarrow (\mathsf{A}_n \longrightarrow \mathsf{B}) \cdots\big)$. L'**ordre** d'un type A est défini par $\mathrm{ord}(\mathsf{A}) = 1$ pour tout $\mathsf{A} \in \mathcal{E}$ et $\mathrm{ord}(\mathsf{A} \rightarrow \mathsf{B}) = \max\big(\mathrm{ord}(\mathsf{B}) + 1, \mathrm{ord}(\mathsf{B}')\big)$.*

*Un **type polymorphe** est une chaîne formée selon les règles précédentes mais dont certaines variables peuvent appartenir à un alphabet d'*indéterminées *de telle manière qu'en substituant toutes ces indéterminées par des variables, on obtient un type.*

Nous noterons les variables de types par des lettres majuscules $\mathsf{A}, \mathsf{B}, \ldots$ et les indéterminées par des lettres minuscules $\mathsf{a}, \mathsf{b}, \ldots$

Exemple 2 *Supposons que $\mathcal{E} = \{\mathsf{A}\}$, alors:*

- $\mathrm{ord}(\mathsf{A} \longrightarrow \mathsf{A}) = \max\big(\mathrm{ord}(\mathsf{A}) + 1, \mathrm{ord}(\mathsf{A})\big) = \max(2, 1) = 2$
- $\mathrm{ord}\big(\mathsf{A} \longrightarrow (\mathsf{A} \longrightarrow \mathsf{A})\big) = \max\big(\mathrm{ord}(\mathsf{A}) + 1, \mathrm{ord}(\mathsf{A} \longrightarrow \mathsf{A})\big) = \max(2, 2) = 2$
- $\mathrm{ord}\big((\mathsf{A} \longrightarrow \mathsf{A}) \longrightarrow \mathsf{A}\big) = \max\big(\mathrm{ord}(\mathsf{A} \longrightarrow \mathsf{A}) + 1, \mathrm{ord}(\mathsf{A})\big) = \max(2 + 1, 2) = 3$

Définition 4 *Soit Λ un langage de λ-termes sur un alphabet de variables V et $\mathcal{T}$ un système de types induit par un système de types élémentaires $\mathcal{E}$. Un **typage** est une fonction $\tau : V \longrightarrow \mathcal{E}$ étendue à une fonction de $\Lambda \longrightarrow \mathcal{T}$ par les règles suivantes:*

- *pour tout $x \in V$, et tout $M \in \Lambda$, si $\tau(x) = \mathsf{A}$ et $\tau(M) = \mathsf{B}$, alors $\tau(\lambda x\ M) = (\mathsf{A} \longrightarrow \mathsf{B})$.*
- *pour tout $M_1, M_2 \in \Lambda$, si $\tau(M_1) = (\mathsf{A} \longrightarrow \mathsf{B})$, et $\tau(M_2) = \mathsf{A}$ alors $\tau\big((M_1\ M_2)\big) = \mathsf{B}$.*

*Un λ-terme est **typable** ssi il peut être décomposé de la manière précédente. Les termes typables forment un sous-ensemble strict de Λ. Lorsqu'il n'y a pas d'équivoque sur τ, on note $M : \mathsf{A}$ au lieu de $\tau(M) = \mathsf{A}$.*

1. C'est le paradoxe apparaissant lorsqu'on envisage qu'il existe un ensemble $\mathcal{E}$ contenant tous les ensembles. En particulier, il contient le sous-ensemble $F = \{E \in \mathcal{E} \mid E \notin E\}$. Soit P la proposition « $F \in F$ ». Si P est vraie, alors $F \notin F$ puisque c'est la propriété caractéristique de F dont P est fausse. Si P est fausse, alors $F \notin F$, ce qui fait de F un élément de lui-même et P est vraie. La proposition P est vraie ssi elle est fausse ce qui est paradoxal. Il n'existe donc pas d'ensemble de tous les ensembles.

2.1.4 Inférence de type

Compte-tenu des règles données dans la Définition 4, on peut déduire une méthode pour déduire le type d'un λ-terme donné. Il suffit d'appliquer récursivement les règles de cette définition à des indéterminées de type et résoudre les équations de type pour trouver un typage satisfaisant. Le type pourra être complètement spécifié ou bien polymorphe selon les cas.

Ocaml dispose d'un mécanisme automatique faisant cette inférence de type à partir de la syntaxe des expressions.

On admettra temporairement qu'on peut exprimer les nombres usuels par des λ-termes et les opérations arithmétiques par des transformations de ces λ-termes et qu'on peut donc donner un sens à des termes du genre $\lambda x\ (x+1)$. Cela sera justifié dans la Section 2.1.13, p. 19.

Ocaml dispose d'une *boucle d'intéraction* permettant de dialoguer avec le système, ce qui est très utile pour se familiariser avec ce logiciel. L'invite est représentée par un dièse (`#`). Les commandes se terminent par un double point-virgule (`;;`). Les lignes ne débutant pas par `#` constituent la réponse d'Ocaml. Les commentaires, ignorés par Ocaml, sont compris entre `(*` et `*)`.

La fonction successeur $x \longmapsto x+1$ correspond au λ-terme $\lambda x\ (x+1)$ ce qui correspond au code Ocaml `function x -> x+1`.

```
# 1 ;;                    (* la constante entiere 1 *)
- : int = 1
# function x -> x+1 ;;    (* la fonction successeur *)
- : int -> int = <fun>
```

Analysons ce qui s'est passé: on apprend que le type de la constante entière est `int`. Lorsqu'il cherche le type de `function x -> x+1`, Ocaml fait les déductions suivantes: soit `a` le type de `x`, alors du fait de la présence de l'expression `x+1`, `a` ne peut être que le type `int`, par conséquent l'expression `function x -> x+1` est de type `int`→`int` (la présence du `=<fun>` est là pour insister sur le fait qu'il s'agit d'une fonction).

Le nommage d'une valeur peut être fait avec le mot-clé `let` (attention tous les identificateurs, sauf mention contraire, doivent être en minuscules):

```
# let x = 1 ;;
val x : int = 1
```

La réponse `val x : int = 1` d'Ocaml indique qu'on a déclaré une *valeur* `x`, de type `int` égale à `1`. Au cours de ses déductions Ocaml peut conclure à une incohérence de typage de l'expression entrée. On peut comparer:

```
# 1 + 1 ;;
- : int = 2
# 1.5 +. 1. ;;
- : float = 2.5
# 1.5 + 1. ;;
Toplevel input:
# 1.5 + 1. ;;
  ^^^
This expression has type float but is here used with type int
# 1.5 + 1 ;;
Toplevel input:
# 1.5 + 1 ;;
  ^^^
This expression has type float but is here used with type int
```

D'une manière générale, le λ-terme $\lambda x\ M$ se code en l'expression Ocaml `function` x `->` M. Voyons encore deux exemples:

```
# function x -> 1 ;;      (* la fonction polymorphe x -> 1 *)
- : 'a -> int = <fun>
# function x -> x ;;      (* la fonction polymorphe identite *)
- : 'a -> 'a = <fun>
```

Aucune contrainte sur `x` similaire à celle apparaissant dans la définition de la fonction successeur n'est présente, en revanche, dans la définition de `function x -> 1` et Ocaml déduit que `x` est de type polymorphe `a` (il le note `'a` (lire *apostrophe a* ou *quote a*), c'est-à-dire qu'on pourra appliquer cette fonction à un argument de n'importe quel type. Enfin, Ocaml déduit de l'expression `function x -> x` que `x` peut être de n'importe quel type `a` et que ce terme est de type polymorphe $\mathsf{a} \to \mathsf{a}$.

Constatons maintenant que le terme $\lambda x\ (x\ x)$ ne peut être typé. En effet, ce dernier est de la forme $\lambda x\ M$ avec $M = (M_1\ M_2)$ et $M_1 = M_2 = x$. Le type de M_1 doit être de la forme $\mathsf{a} \to \mathsf{b}$ et le type de M_2 doit être de type a. On en déduit que $\mathsf{a} \to \mathsf{b} = \mathsf{a}$. Cette équation n'a pas de solution (la chaîne de gauche est de longueur 3, celle de droite est de longueur 1). Ocaml s'en rend compte et déclenche une erreur:

```
# function x -> (x x);;
This expression has type 'a -> 'b but is here used with type 'a
```

2.1.5 Types Ocaml définis par l'utilisateur

Ocaml permet à l'utilisateur de définir des types et de spécifier explicitement le type d'une valeur:

```
# type complex = float*float ;;
type complex = float * float
# let re (z:complex) = fst z and im (z:complex) = snd z ;;
val re : complex -> float = <fun>
val im : complex -> float = <fun>
# re (1.5,-3.6) ;;
- : float = 1.5
```

On peut également définir des *types somme* (attention ils doivent nécessairement commencer par une majuscule), séparés par une barre verticale `|` (*pipe*), comme ici pour les jours de la semaine ou l'ensemble $\mathbb{R} \sqcup \{\infty\}$.

```
#  type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche ;;
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche
# type float_bar = Flottant of float | Infinity ;;
type float_bar = Flottant of float | Infinity
```

On les traite habituellement par un *filtrage de motif* (l'étude de cas, aussi appelée *pattern matching*, — correspondance de motif — en anglais). Le filtrage se fait séquentiellement et le résultat correspond au premier motif trouvé. Le motif *trait bas* `_` (*underscore*) signifie « n'importe quelle valeur ».

```
#  let grasse_matinee = function
    Dimanche -> true
```

```
  | _        -> false ;;
val grasse_matinee : jour -> bool = <fun>
# grasse_matinee Lundi ;;
- : bool = false
# let invert = function
    Flottant 0. -> Infinity
  | Flottant x  -> Flottant(1. /. x)
  | Infinity    -> Flottant 0. ;;
val invert : float_bar -> float_bar = <fun>
#  invert (Flottant 2.) ;;
- : float_bar = Flottant 0.5
#  invert (Flottant 0.) ;;
- : float_bar = Infinity
#  invert Infinity ;;
- : float_bar = Flottant 0
```

2.1.6 λ-calcul

Jusqu'à maintenant, nous nous sommes contentés d'envisager des λ-termes de façon syntaxique, c'est-à-dire, comme des chaînes de caractères, formées selon des règles de production. Passons maintenant à la partie « calcul » du λ-calcul. Il s'agit de donner un sens (sémantique) aux λ-terme en établissant quels λ-termes sont équivalents et la manière effective d'en décider. On voudrait, par exemple, exprimer le fait que les deux λ-termes $\lambda x\, x$ et $\lambda y\, y$, bien que syntaxiquement différent, représentent tous deux l'identité. On définira une règle α dite « de renommage » et une règle β, dite « de réduction » afin d'être en mesure de faire ce type d'identification.

Le principe est d'aboutir à une forme dite « réduite » (lorsqu'elle existe). La notion de forme réduite correspond à la notion de calcul arithmétique dans lequel on simplifie les expressions arithmétiques pour obtenir une version finale, expurgée de combinaisons opérateurs-opérande, comme par exemple dans:

$$\big(3 + (4 \times 5)\big) \longrightarrow \big(3 + 20\big) \longrightarrow 23\ .$$

Commençons donc par la règle β, dont les limites justifient l'introduction de la règle α.

2.1.7 Règle β: réduction

La règle de réduction est la transformation que l'on applique à un λ-terme de la forme opérateur-opérande pour le transformer un un λ-terme plus simple qui représente le résultat du calcul de l'application de l'opérateur à l'opérande. Par exemple, considérons le λ-terme opérateur-opérande:

$$\big(\lambda x\ \underbrace{(x+1)}_{M}\quad \underbrace{3}_{A}\big)\ . \tag{2.1}$$

L'intuition que nous avons évoquée dans la section 2.1.2, p. 9 consistait à dire que le λ-terme (2.1) est l'application de la fonction successeur à la valeur 3. Le résultat escompté est l'entier 4, et c'est bien ce qu'Ocaml répond:

```
# (function x -> x+1) 3;;
- : int = 4
```

Ocaml a pu donner cette réponse en faisant la transformation consistant à substituer la valeur $A = 3$ à la variable x dans le λ-terme $M = (x + 1)$ pour se retrouver avec le λ-terme $(3 + 1)$, qui s'évalue en 4. Le principe général de β-réduction est donc le suivant:

$$\begin{array}{c} \lambda x \; (\textit{expression } M \textit{ dépendant de } x) \quad (\textit{argument } A) \\ \downarrow \\ (\textit{expression } M' \textit{ obtenue par substitution de } x \textit{ par } A \textit{ dans } M) \end{array} \tag{2.2}$$

L'exemple précédent en Ocaml peut se réécrire:

```
# let x = 3 in x+1 ;;
- : int = 4
```

Ainsi l'expression Ocaml `((function` x `->` M`)` A`)` qui correspond au λ-terme $\lambda x\, M\; A$ équivaut à l'expression `let` x `=` A `in` M qui correspond au λ-terme M' comme défini dans (2.2). On pourrait la traduire en français par « soit $x = A$ dans M ». Autre exemple; dans le λ-terme:

$$\lambda h \; \underbrace{\big((h\ 3) + (h\ 4)\big)}_{M} \quad \underbrace{\big(\lambda x\ (x+1)\big)}_{A}\ , \tag{2.3}$$

on applique le λ-terme $\lambda h\big((h\ 3)+(h\ 4)\big)$ à l'argument constitué du λ-terme $\lambda x\,(x+1)$. En appliquant la réduction (2.2), on obtient le λ-terme équivalent:

$$\big(\lambda x\ (x+1)\big)\ 3 \quad + \quad \big(\lambda x\ (x+1)\big)\ 4\ ,$$

qui se réécrit de même $(3+1)+(4+1)$ qui vaut 9.

```
# (function h -> h(3) + h(4)) (function x -> x+1);;
- : int = 9
```

Une définition employant la terminologie classique pourrait être l'une des deux expressions:

- soit la valeur obtenue en définissant h comme la fonction successeur dans l'expression $h(3) + h(4)$;
- soit la valeur $h(3) + h(4)$ dans laquelle on a défini h comme étant la fonction successeur.

Un formalisme intermédiaire permet de rester proche du λ-calcul tout en conservant une certaine lisibilité:

- soit $h = \lambda x\ (x+1)$ dans $h(3) + h(4)$;
- $h(3) + h(4)$ où $h = \lambda x\ (x+1)$.

Ocaml permet l'emploi d'expressions de la première forme comme on l'a vu plus haut, avec la construction `let` x `=` A `in` M, ce qui donne, dans le cas présent:

```
# let h = function x -> x+1 in h(3) + h(4);;
- : int = 9
```

2.1.8 Conflits empêchant les β-réductions

On vient de le voir, la réduction de $(\lambda x\, M\ A)$ est le λ-terme M' obtenu en toutes les occurences de x par A dans M; par exemple, si $M = \lambda y\, x$ et $A = a$, ce λ-terme est $(\lambda x\ \lambda y\ x\ a)$ et se réduit à $\lambda y\, a$. Mais il faut observer que cela n'a un sens que s'il n'y a pas de conflits de noms de variables, au sens où il est nécessaire que:

- M ne contienne aucune occurence liée de x; on ne peut pas réduire *e.g.* :

$$(\lambda x\ \underbrace{\lambda x\ x}_{M}\ \underbrace{\ a\ }_{A}) \tag{2.4}$$

- M ne contient aucune variable liée qui apparaissent librement dans A; on ne peut pas réduire *e.g.* :

$$(\lambda x\ \underbrace{\lambda y\ x}_{M}\ \underbrace{\ y\ }_{A}) \tag{2.5}$$

La section suivante traite de la façon de résoudre de tels conflits par renommage des variables problématiques.

2.1.9 Règle α: renommage

Cette règle énonce qu'une variable liée x dans un λ-terme M peut être remplacée par n'importe quelle variable y à condition que cette variable n'apparaisse pas dans M. En particulier, on pourra lever tout conflit de noms dans un λ-terme opérateur-opérande de la forme $(\lambda x\, M\ A)$ en le réécrivant sous la forme $(\lambda x\ \widetilde{M}\ A)$ où $\widetilde{M}$ s'obtient à partir de M en remplaçant toutes les occurences de variables provoquant des conflits par d'autres variables. Par exemple,

$$(2.4) \underset{\alpha}{\longrightarrow} (\lambda x\ \underbrace{\lambda y\ y}_{\widetilde{M}}\ \ a) \underset{\beta}{\longrightarrow} \lambda y\ y\ ,$$

tandis que:

$$(2.5) \underset{\alpha}{\longrightarrow} (\lambda x\ \underbrace{\lambda z\ x}_{\widetilde{M}}\ \ y) \underset{\beta}{\longrightarrow} \lambda z\ y\ .$$

Observons la stratégie adoptée par Ocaml pour traiter les situations (2.4) et (2.5):

Définition 5 *Un λ-terme qui n'a pas d'expression de la forme opérateur-opérande $(\lambda x\ M\ A)$ est dit **réduit** et une expression qui n'est pas réductible à une expression réduite par une succession de renommages et de réductions est dite **irréductible**.*

L'exemple typique de λ-terme irréductible est:

$$\omega \overset{\text{def}}{=} \big(\lambda x\ \underbrace{(x\ x)}_{M}\ \underbrace{\lambda x\ (x\ x)}_{A}\big) \tag{2.6}$$

Il n'y a en effet pas de conflits de noms et la seule règle de réduction applicable laisse ce λ-terme invariant. Il faut toutefois noter que ce terme *n'est pas typable* (cela provient du fait que le sous-terme $\lambda x\ (x\ x)$ ne l'est pas, comme on l'a vu dans la Section 2.1.4, p. 11. On voit immédiatement un autre avantage du typage pour éviter certaines situations pathologiques.

Observons à cette occasion le type d'un λ-termes opérateur-opérande $(\lambda x\ M\ A)$. Il est de la forme $(M_1\ M_2)$ avec $M_1 = \lambda x\ M$ et $M_2 = A$ donc $M_1 : (\mathsf{a} \rightarrow \mathsf{b})$, $M_2 : \mathsf{a}$ et $(M_1\ M_2) : \mathsf{b}$, où a et b sont indéterminés. Comme M_1 est de la forme $\lambda x\ M$ et de type $(\mathsf{a} \rightarrow \mathsf{b})$, on en déduit que $x : \mathsf{a}$ et $M : \mathsf{b}$.

La question de savoir si un λ-terme admet une expression réduite est *indécidable*: il n'existe pas d'algorithme qui, étant donnée un λ-terme E peut toujours déterminer en un nombre fini d'étapes si E a, ou pas, une forme réduite.

2.1.10 Théorème de Church-Rosser

Il se trouve que lorsqu'un λ-terme possède plusieurs sous-expressions de la forme opérateurs-opérande, plusieurs réécritures sont possibles. Dans la Section 2.1.6, nous avons fait le parallèle entre le calcul arithmétique et le λ-calcul. Nous allons maintenant évoquer certaines de leurs différences.

Étant donnée une expression arithmétique, on a une certaine liberté dans la progression du calcul. Par exemple, on peut calculer selon les deux branches du graphe suivant:

$$\begin{tikzcd}
 & (2+3)\times(4+5) \arrow[dl] \arrow[dr] & \\
5\times(4+5) \arrow[dr] & & (2+3)\times 9 \arrow[dl] \\
 & 5\times 9 \arrow[d] & \\
 & 45 &
\end{tikzcd}$$

Le calcul arithmétique possède deux propriétés fondamentales:

- *finitude:* tous les calculs se terminent en un nombre fini d'étapes;
- *cohérence:* tous les calculs aboutissent au même résultat.

Cette dernière propriété résulte d'une propriété plus générale, dite *de confluence.*

Définition 6 *Un ensemble de règles de réécriture est dit* ***confluent*** *ssi pour toute expression E, si E_1 et E_2 sont des réécritures de E alors il existe une réécriture E' de E telle que E_1 et E_2 peuvent se réécrire en E', selon le* ***schéma du diamant:***

$$\begin{tikzcd}
 & E \arrow[dl] \arrow[dr] & \\
E_1 \arrow[dr] & & E_2 \arrow[dl] \\
 & E' &
\end{tikzcd}$$

La puissance d'expressivité du λ-calcul empêche une telle régularité et on doit se contenter d'une propriété plus modeste, dite *cohérence faible*: tous les calculs qui terminent donnent le même résultat.

Théorème 1 (Théorème de Church-Rosser) *Si deux suites de réécritures d'un λ-terme aboutissent à une forme réduite, alors celles-ci sont égales à un renommage des variables liées près.*

Attention: il existe des λ-termes possédant simultanément des réécritures réduites et d'autres irréductibles. Par exemple le λ-terme opérateur-opérande:

$$\Big(\lambda x\ \underbrace{\lambda y\, y}_{M}\ \underbrace{\big(\lambda x\ (x\ x)\ \ \lambda x\ (x\ x)\big)}_{A}\Big)$$

peut se réduire, d'une part, en $\lambda y\, y$ en substituant par A les occurences de x dans M (il n'y en a pas), mais aussi en lui-même en applicant une réduction sur $A = \omega$ (défini en (2.6), p. 15) qui, comme on l'a vu, est irréductible, ce qui conduit à une suite infinie de réécritures.

2.1.11 Appel par nom, appel par valeur

Le choix de l'ordre des réécritures dans le λ-calcul a une implication dans l'évaluation des fonctions dans les langages de programmation qui sont fondés sur ce formalisme. En effet, l'évaluation de $f\big(g(x)\big)$ peut se faire soit en évaluant $g(x)$, puis en passant la valeur évaluée à f (appel intérieur-extérieur, dit « par valeur »), soit en substituant l'expression $g(x)$ dans la définition de f (appel extérieur-intérieur, dit « par nom »). Par exemple, soit $g : x \longmapsto x + 1$ et $f : y \longmapsto y^2 + y + 1$, on a:

$$\begin{tikzcd}
 & f(g(3)) \arrow[dl, "\text{appel par valeur}"'] \arrow[dr, "\text{appel par nom}"] & \\
f(4) \arrow[d] & & (3+1)^2 + (3+1) + 1 \arrow[d] \\
4^2 + 4 + 1 \arrow[dr] & & (9+6+1) + (3+1) + 1 \arrow[dl] \\
 & 21 &
\end{tikzcd}$$

L'appel par valeur est plus efficace que l'appel par nom lorsque la valeur de $g(x)$ est réutilisée plus d'une fois mais est moins efficace si $g(x)$ n'est, de fait, jamais utilisée. En particulier, si l'évaluation de $g(x)$ requiert beaucoup de calcul voire, ne termine pas alors que f n'utilise pas sa valeur, l'appel par valeur est catastrophique tandis que l'appel par nom est toujours valable. Le λ-terme:

$$\Big(\lambda x\ \underbrace{\lambda y\, y}_{M}\ \underbrace{\big(\lambda x\ (x\ x)\ \ \lambda x\ (x\ x)\big)}_{A}\Big)$$

est de la forme $f\big(g(x)\big)$ où $f = \lambda x\, M$ et $g(x) = A$ et l'évaluation de $g(x)$ ne termine pas. L'appel par nom correspond, en λ-calcul, à la réduction dite « par la gauche » des λ-termes opération-operande $(\lambda x\ M\ A)$ dont les composants λx sont les plus à gauche. Le théorème suivant donne le cadre d'application de l'appel par nom:

Théorème 2 *Si un λ-terme admet une suite de réductions qui aboutissent à une forme réduite, alors la suite de réductions par la gauche conduisent à une forme réduite qui lui est équivalente, à renommage des variables liées près.*

L'appel par valeur, plus efficace en pratique est le choix fait par Ocaml. On pourra se reporter à [WL99, pp. 325–326] pour une discussion sur ce sujet.

2.1.12 Curryfication des fonctions multivariables

En mathématique usuelle, on définit les fonctions « à plusieurs variables » comme des fonctions dont le domaine est un produit cartésien. Par exemple: $f : \mathbb{N}\times\mathbb{N} \longrightarrow \mathbb{N}$ définie par $(x,y) \longmapsto x+y$, ce qui s'écrit en Ocaml:

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

Le λ-calcul ne traite que de fonctions à une variable mais on a une correspondance bijective, popularisée par Haskell CURRY et naturellement baptisée *curryfication*, entre celles-là et les fonctions multivariées. En effet, on peut voir la fonction f de l'exemple précédent comme une fonction: $g : \mathbb{N} \longrightarrow (\mathbb{N} \longrightarrow \mathbb{N})$ définie par $x \longmapsto (y \longmapsto x+y)$.

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

On dit que g est la *version curryfiée* de f. En λ-calcul, cette dernière fonction s'écrirait $\lambda x \lambda y\ (x+y)$. Afin d'alléger l'écriture, on peut aussi écrire en Ocaml:

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

Ainsi on peut définir la correspondance de CURRY:

```
# let curry = fun f x y -> f(x,y) ;;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry = fun f (x,y) -> f x y ;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
# let g = curry f ;;
val g : int -> int -> int = <fun>
# let f = uncurry g ;;
val f : int * int -> int = <fun>
```

La fonction `g` est du même type que l'opérateur `+`. En effet, la version préfixe de celui-ci s'obtient en l'entourant de parenthèses:

```
# (+) ;;
- : int -> int -> int = <fun>
# (+) 2 3 ;;
- : int = 5
```

L'utilisation de fonctions curryfiées permet la manipulation de fonctions « partielles »:

```
# let succ = (+) 1;;
val succ : int -> int = <fun>
# succ 1;;
- : int = 2
```

Signalons enfin qu'on peut parfois se passer des mots-clé `function` et `fun` lorsqu'on emploie les variantes syntaxique:

```
# let succ x = x+1 ;;
val succ : int -> int = <fun>
# let sum x y = x+y ;;
val sum : int -> int -> int = <fun>
```

Un résumé de la correspondance entre notation classique, λ-calcul et syntaxe Ocaml est donnée dans la Tab. 2.1.

notation classique	λ-terme	syntaxe Ocaml
soit $x = 1$	soit $x = 1$	`let` x `=` 1
soit $f : x \longmapsto M$	soit $f = \lambda x\ M$	`let` f `= function` x `->` M
$f(A)$	$(f\ A)$	`(`$f\ A$`)`
$x \longmapsto M$	$\lambda x\ M$	`function` x `->` M
$x_1 \longmapsto \big(\cdots(x_n \longmapsto M)\cdots\big)$	$\lambda x_1 \cdots \lambda x_n\ M$	`fun` x_1 `...` x_n `->` M
$(x \longmapsto M)(A)$	$(\lambda x\ M\ A)$	`((function` x `->` M`)` A`)` ou `let` x `=` A `in` M

TAB. 2.1: *Correspondance entre notation classique, λ-calcul et syntaxe Ocaml*

2.1.13 Fonction λ-définissable

On a supposé dans la Section 2.1.4, p. 11 que les entiers pouvaient être représentés par des λ-termes. Montrons maintenant comment on s'y prend.

Définition 7 *Soit $k \in \mathbb{N}$, on appelle **k-ième entier de** **CHURCH** le λ-terme $\underline{k} = \lambda f \lambda x\ (f^k\ x)$, où on définit $(f^k\ x) \stackrel{\text{def}}{=} \big(f^{k-1}\ (f\ x)\big)$ pour $k \geq 1$ et $(f^0\ x) \stackrel{\text{def}}{=} x$.*

Ocaml n'utilise pas explicitement cette définition des entiers car ce serait épouvantablement inefficace. Par ailleurs, comme on l'a vu dans les exemples précédents, il représente les entiers par les symboles `0`, `1`, `2`,... Ce qu'il est important de retenir est qu'il existe une correspondance théorique entre le λ-calcul et l'arithmétique usuelle, ce qui permet d'avoir un contrôle mathématique sur les programmes qu'on fabrique.

Définition 8 *Le λ-terme M **représente** une fonction partielle $f : \mathbb{N}^p \longrightarrow \mathbb{N}$ ssi $\forall (k_1, \ldots, k_p) \in \mathbb{N}^p$:*

- *si $f(k_1, \ldots, k_n) = k$, alors $(M\ \underline{k_1} \cdots \underline{k_p})$ est β-réductible à $\underline{k}$;*
- *si $f(k_1, \ldots, k_n)$ n'est pas définie, alors $(M\ \underline{k_1} \cdots \underline{k_p})$ est irréductible.*

*Une fonction partielle telle qu'il existe un λ-terme qui la représente est dite **λ-définissable** ou **λ-représentable**.*

Exemple 3 *La fonction successeur* succ : $x \longmapsto x + 1$ *peut être représentée par le λ-terme* $\underline{\text{succ}} = \lambda n \lambda f \lambda x\ \big((n\ f)\ (f\ x)\big)$ *puisqu'on a, pour tout entier k:*

$$
\begin{aligned}
(\underline{\text{succ}}\ \underline{k}) = \Big(\lambda n\, \overbrace{\lambda f \lambda x\ \big((n\ f)\ (f\ x)\big)}^{M}\ \ \overbrace{\lambda f \lambda x\ (f^k\ x)}^{A}\Big) \xrightarrow[\beta]{} \lambda f \lambda x\ \big((\lambda f\ \overbrace{\lambda x\ (f^k\ x)}^{M'}\ \overbrace{f}^{A'}\ )\ (f\ x)\big) \\
\xrightarrow[\beta]{} \lambda f \lambda x\ \big(\lambda x\ \underbrace{(f^k\ x)}_{M''}\ \underbrace{(f\ x)}_{A''}\big) \xrightarrow[\beta]{} \lambda f \lambda x\ \big(f^k\ (f\ x)\big) = \lambda f \lambda x\ (f^{k+1}\ x) = \underline{k+1}\ .
\end{aligned}
\tag{2.7}
$$

2.1.14 Conclusion

Nous avons introduit la notion de λ-calcul, typé ou non, et donné la correspondance entre celui-ci et le langage Objective Caml. Nous avons évoqué le fait que le λ-calcul permettait de représenter les entiers et opérations usuelles de l'arithmétique.

2.2 Fonctions récursives

2.2.1 Introduction

Une autre approche pour définir ce que peut être une fonction « calculable », consiste à définir ce qu'est une fonction *récursive*. Prenons un exemple: la fonction carré est explicitable par une formule mais on peut s'interroger la formule définissant la fonction puissance en général:

$$\mathrm{pow} : (x, n) \longmapsto x^n \stackrel{\mathrm{def}}{=} \underbrace{x \times \cdots \times x}_{n \text{ fois}} .$$

En effet, comment interpréter les points de suspension? Une autre définition met en évidence un procédé de calcul *récursif* (c'est-à-dire, que la formule définissant la fonction fait intervenir la fonction elle-même):

$$\mathrm{pow} : (x, n) \longmapsto \begin{cases} 1 & \text{si } n = 0 \\ x \times \mathrm{pow}(x, n-1) & \text{sinon} \end{cases}$$

Cela s'exprime en Ocaml avec le mot clé `let rec` qui autorise le nom qui le suit à apparaître à droite du signe égal:

```
# let rec pow = fun x n -> match n with
  0 -> 1
| _ -> x * pow x (n-1) ;;
val pow : int -> int -> int = <fun>
```

Il s'agit d'une fonction curryfiée de type `int` $\to$ `int` $\to$ `int`. La construction `match n with` relève du filtrage de motifs. Donnons le détail du calcul:

```
# pow 3 2 ;;
- : int = 9
```

1. $r =$ `pow x n` avec `x` $= 3$ et `n` $= 2$:
   (a) est-ce que `n` est égal à `0`? non;
   (b) est-ce que `n` est égal à « n'importe quoi »? oui, donc $r =$ `x *` r' avec $r' =$ `pow 3 1`;
2. $r' =$ `pow x n` avec `x` $= 3$ et `n` $= 1$:
   (a) est-ce que `n` est égal à `0`? non;
   (b) est-ce que `n` est égal à « n'importe quoi »? oui, donc $r' =$ `x *` r'' avec $r'' =$ `pow 3 0`;
3. $r'' =$ `pow x n` avec `x` $= 3$ et `n` $= 0$:
   (a) est-ce que `n` est égal à `0`? oui, donc le résultat r'' de `pow x n`$''$ est 1.

Le dépilement se fait en déduisant de (2.b) que $r' =$ `x *` $1 = 3$, puis de (1.b), que $r =$ `x *` $3 = 9$.

2.2.2 Récursivité terminale

En fait, dans l'exemple précédent, on peut remarquer qu'on a pas besoin de stocker dans la pile $r' =$ `pow 3 1` pour en déduire $r =$ `x *` r' à partir du moment où, au lieu de se contenter de la définition $r' =$ `x *` r'', on effectue le calcul pour déduire que $r =$ `x * (x *` r''`)`. Une définition de fonction récursive f est *terminale* lorsque le résultat du calcul de f ne nécessite pas le stockage dans la pile de résultats intermédiaires. Évidemment, les fonctions récursives terminales sont beaucoup plus efficaces. Comparons deux définitions récursives de la fonction successeur.

```
# let rec succ n = if (n=0) then 1 else 1 + succ (n-1) ;;
val succ : int -> int = <fun>
# succ 100000 ;;
Stack overflow during evaluation (looping recursion?).
```

Ocaml annonce un débordement de pile (*stack overflow*), et une explication probable (boucle infinie de récursion? (*looping recursion?*) ce n'est pas le cas ici). On peut construire une version recursive terminale en utilisant une fonction auxilliaire qui stocke dans un accumulateur le résultat des calculs intermédiaires:

```
# let succ n =
    let rec succ_aux i accu = (if i=0 then accu else succ_aux (i-1) (accu+1)) in
      succ_aux n 1 ;;
val succ : int -> int = <fun>
# succ 100000 ;;
-: int = 100001
```

De même on démontre par récurrence que les expressions:

$$f(n) = \begin{cases} 1 & \text{si } n \leq 0\ ; \\ n \cdot f(n-1) & \text{sinon}\ ; \end{cases} \quad \text{et} \quad g(n) = \gamma(n,1) \text{ avec } \gamma(i,a) = \begin{cases} a & \text{si } i \leq 0\ ; \\ \gamma(i-1, i \cdot a) & \text{sinon}\ ; \end{cases}$$

sont toutes deux égales à $n!$ pour tout $n \in \mathbb{N}$; la fonction $f : n \longmapsto f(n)$ est récursive non terminale tandis que $g : n \longmapsto g(n)$ est récursive terminale.

2.2.3 Nécessité de préciser la définition de fonction récursive

On a vu que la définition d'une fonction f en termes d'opérateur de description $[f \mid P(f)]$ n'était possible qu'à partir du moment où: il existe une unique fonction f telle que $P(f)$. Nous pouvons redéfinir la fonction puissance:

$$\text{pow} = \left[f \;\middle|\; f = \Big((x,n) \longmapsto \big(\text{si } n = 0 \text{ alors } 1 \text{ sinon } x \times f(x, n-1)\big) \Big) \right].$$

et constater que, dans un tel cas, la propriété $P(f)$ est de la forme « $f = G(f)$ », c'est-à-dire que f est le point fixe de la fonction G. Si, d'un point de vue formel, une telle propriété ne pose pas de problème, du point de vue effectif, le calcul – s'il est défini – de $f(x)$ pour un certain x semble requérir l'évaluation de $G\big(f(x)\big)$ qui, lui-même, requiert l'évaluation de $(G \circ G)(f(x))$, *etc.* Ce processus ne définit une valeur effective qu'à certaines conditions « de décroissance » sur G.

La construction `let rec` de Ocaml permet à une fonction de faire référence à son propre nom dans sa définition. Cependant, on ne peut pas détecter ce genre de définition pathologique:

```
# let rec f = function x -> f x ;;
val f : 'a -> 'b = <fun>
```

```
# f(0);;
Interrupted.
# let rec f = function x -> 1 + (f x);;
val f : 'a -> int = <fun>
# f(0);;
Stack overflow during evaluation (looping recursion?).
```

2.2.4 Fonctions récursives primitives

On va partir d'un ensemble de fonctions de base, puis enrichir l'ensemble des fonctions que l'on va considérer grâce à un mécanisme appelé récurrence primitive:

Définition 9 *Une **fonctions récursive élémentaire** est l'une des fonctions suivantes:*

- *la fonction constante nulle* $0 : \mathbb{N} \longrightarrow \mathbb{N},\ x \longmapsto 0$
- *la fonction **successeur*** $\mathrm{succ} : \mathbb{N} \longrightarrow \mathbb{N},\ x \longmapsto x+1$
- *les fonctions **projections*** $\mathbb{N}^p \longrightarrow \mathbb{N},\ \mathrm{pr}_p^i : (x_1, \ldots, x_n) \longmapsto x_i$.

Soient $g : \mathbb{N}^p \longrightarrow \mathbb{N}$ *et* $h : \mathbb{N}^{p+2} \longrightarrow \mathbb{N}$ *la **fonction définie par récurrence primitive** par* g *et* h *est la fonction* $f : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}$*:*

- $f(x_1, \ldots, x_p, 0) = g(x_1, \ldots, x_p)$ *(condition initiale)*
- $f(x_1, \ldots, x_p, y+1) = h\big(x_1, \ldots, x_p, y, f(x_1, \ldots, x_p, y)\big)$ *(étape de récurrence primitive)*

*Le plus petit ensemble de fonctions contenant les fonctions de base et clos par composition et récurrence primitive s'appelle l'ensemble des **fonctions récursives primitives**. Un sous-ensemble* A *de* $\mathbb{N}^p$ *est **récursif primitif** ssi sa fonction caractéristique est récursive primitive.*

Proposition 1 *Les fonctions récursives de base sont* λ*-représentables:*

- *la fonction constante nulle* $0 : \mathbb{N} \longrightarrow \mathbb{N},\ x \longmapsto 0$ *est représentable par* $\lambda x\, \underline{0}$ *où* $\underline{0}$ *est le* 0*-ième entier de* CHURCH.
- *la fonction **successeur*** $\mathrm{succ} : \mathbb{N} \longrightarrow \mathbb{N},\ x \longmapsto x+1$ *est représentable par le* λ*-terme* $\underline{\mathrm{succ}} = \lambda n \lambda f \lambda x\ \big((n\ f)\ (f\ x)\big)$.
- *les fonctions **projections*** $\mathbb{N}^p \longrightarrow \mathbb{N},\ \mathrm{pr}_p^i : (x_1, \ldots, x_n) \longmapsto x_i$ *sont* λ*-représentables par* $\lambda x_1 \cdots \lambda x_n\, x_i$.

Exemple 4 *La fonction d'addition* $f : \mathbb{N}^2 \longrightarrow \mathbb{N},\ (x_1, x_2) \longmapsto x_1 + x_2$ *est récursive primitive car elle peut se définir par récurrence primitive par* $g : \mathbb{N} \longrightarrow \mathbb{N},\ x_1 \longmapsto \mathrm{pr}_1^1(x_1)$ *et* $h : \mathbb{N}^3 \longrightarrow \mathbb{N}$, $(x_1, y, z) \longmapsto \mathrm{succ}\Big(\mathrm{pr}_3^3(x_1, y, z)\Big)$ *de telle sorte qu'on a bien:*

- $f(x_1, 0) = g(x_1) = \mathrm{pr}_1^1(x_1)$
- $f(x_1, y+1) = h\big(x_1, y, f(x_1, y)\big) = \mathrm{succ}\big(\mathrm{pr}_3^3(x_1, y, x+y)\big)$

```
# let g = function x1 -> x1 ;;
val g : 'a -> 'a = <fun>
# let pi_3_3 = function (x1,x2,x3) -> x3 ;;
val pi_3_3 : 'a * 'b * 'c -> 'c = <fun>
# let h = function (x1,y,z) -> succ (pi_3_3 (x1,y,z)) ;;
val h : 'a * 'b * int -> int = <fun>
# let rec plus = function
    (x1,0)  -> g x1
  | (x1,y) -> h (x1,y, plus(x1,y-1)) ;;
val plus : int * int -> int = <fun>
# plus 3 5 ;;
- : int = 8
```

Il en est de même de la fonction multiplication, de la définition par cas, des sommes et produits bornés, de la quantification bornée.

2.2.5 Traduction récursive primitive des boucles `for`

Dans un langage impératif, une boucle `for` est une suite d'instructions du genre:

pour i de 0 à n faire
  corps de la boucle
fin pour

Nous allons montrer qu'on peut traduire cette notion en terme de fonctions récursives primitives. Tout d'abord, il est nécessaire d'expliciter ce qu'est un corps de boucle. Le principe est que chaque passage dans la boucle transforme l'*état* de la machine. Dans un ordinateur, cet état est constitué des suites binaires qui occupent le processeur, la RAM, les disques durs, *etc.* Nous le représenterons cet état par une variable `etat`, de type polymorphe `a`. Nous interdisons que la valeur i ou la valeur n soient modifiables par le corps de la boucle (pas d'effet de bord). Le i-ième passage dans la boucle fera donc passer la machine d'un $\mathtt{etat}_{i-1}$ à un état $\mathtt{etat}_i$ comme voici:

$$\mathtt{etat}_0 \xrightarrow{f_1} \mathtt{etat}_1 \xrightarrow{f_2} \cdots\cdots \xrightarrow{f_n} \mathtt{etat}_n\ , \quad \text{où} \quad f_i : \mathtt{a} \to \mathtt{a}\ .$$

On veut disposer d'une fonction `pour` n qui prend une fonction f_i et un état $\mathtt{etat}_{i-1}$ et renvoie $f_i(\mathtt{etat}_i)$, ce qui s'écrira `etat' = pour n f i etat`. On définit donc:

```
# let rec pour n f i etat = if (i<=n) then (pour n f (succ i) (f i etat)) else etat ;;
val pour : int -> (int -> 'a -> 'a) -> int -> 'a -> 'a = <fun>
```

Prenons un exemple où l'état est une chaîne de caractères et où la fonction f_i concatène à l'état courant la chaîne représentant le nombre i:

```
# let concat i etat = etat^string_of_int(i) ;;
val concat : int -> string -> string = <fun>
# pour 9 concat 0 "" ;;
- : string = "0123456789"
```

2.2.6 Fonctions calculables non récursives primitives

Théorème 3 *Il existe des fonctions « calculables » qui ne sont pas récursives primitives.*

Démonstration: Par construction, les fonctions récursives primitives sont dénombrables. Soit $\varphi_0, \varphi_1, \ldots$ une énumération de ces fonctions. Pour tout i, si φ_i est une fonction à n_i arguments, on notera:

$$f_i : j \longmapsto \varphi_i(\underbrace{j, \ldots, j}_{n_i \,\text{termes}}) .$$

Considérons alors la fonction $g : n \longmapsto f_n(n) + 1$. Si g était primitive récursive, alors il existerait $k \in \mathbb{N}$ tel que $g = \varphi_k = f_k$ et on aurait $g(k) = g(k) + 1$ *i.e.* $0 = 1$, ce qui est absurde. En revanche, g est calculable. En effet pour calculer $g(n)$, on procède comme suit:

1. on énumère toutes les fonctions primitives récursives jusqu'à la fonction f_n (il suffit d'énumérer toutes les chaînes de caractères, de longueur croissante, de programmes dont la syntaxe est celle d'une fonction primitive récursive);
2. une fois cette définition obtenue, on évalue $f_n(n)$ (f_n est primitive récursive, donc calculable);
3. on évalue $f_n(n) + 1$.

2.2.7 Fonctions récursives

Définition 10 *Une* ***fonction partielle*** *de $\mathbb{N}^p \longrightarrow \mathbb{N}$ est un couple (D, f) où $D \subseteq \mathbb{N}^p$ et $f : D \longrightarrow \mathbb{N}$. On dit que f est définie* ***par minimisation*** *à partir d'une fonction partielle $g : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}$ ssi:*

- *dès qu'il existe $z \in \mathbb{N}$ tel que simultanément $g(x_1, \ldots, x_p, z) = 0$ et $g(x_1, \ldots, x_p, z')$ pour tout $z' < z$ alors on a $f(x_1, \ldots, x_p) = z$;*
- *si un tel z n'existe pas, alors $f(x_1, \ldots, x_p)$ n'est pas défini.*

*On définit l'****ensemble des fonctions récursives partielles*** *comme le plus petit ensemble de fonctions contenant les fonctions de base et clos par composition, récurrence primitive et minimisation. Un sous-ensemble de $\mathbb{N}^p$ est dit* ***récursif*** *ssi sa fonction caractéristique est récursive.*

Exemple 5 *Soit la fonction $A : \mathbb{N}^2 \longrightarrow \mathbb{N}$, dite* fonction d'Ackermann, *définie par:*

- *$A(0, x) = x + 2$ pour tout entier x;*
- *$A(1, 0) = 0$ et $A(y, 0) = 1$ pour tout $y \geq 2$;*
- *$A(y + 1, x + 1) = A\big(y, A(y + 1, x)\big)$;*

est récursive mais pas récursive primitive (ce n'est pas évident!).

```
# let rec ack = function
    (0,n) -> n+1
  | (m,0) -> ack(m-1,1)
  | (m,n) -> ack(m-1,ack(m,n-1));;
val ack : int * int -> int = <fun>
```

2.2.8 Traduction récursive des boucles `while`

On dans la Section 2.2.5, p. 23 qu'on pouvait donner une définition récursive primitive des boucles `for`. Nous allons montrer qu'on peut donner une version récursive des boucles `while` qui sont de la forme suivante:

tant que b faire
corps de la boucle
fin tant que

Étant donnée une condition de test `test` : a $\longrightarrow$ bool, la condition booléenne $b =$ `test(etat)` dépend de l'état de la machine qui est modifiable dans le corps de la boucle: c'est là la différence fondamentale avec les boucles `for`. Ainsi, il y a donc des boucles qui ne terminent pas; celles-ci représentent des fonctions récursives partielles qui ne sont pas définies. Un passage dans la boucle sera modélisé par une fonction f : a $\to$ a qui prend un état `etat` : a et renvoie un état. On va donc créer une fonction `tant_que` qui prend une fonction de test `test` : a $\longrightarrow$ bool, une fonction f : a $\to$ a et un état `etat` : a et renvoie l'état etat$'$ obtenu par itération de la fonction f à `etat` jusqu'à ce que `test(etat)` soit faux.

```
# let rec tant_que test f etat = if (test etat) then tant_que test f (f etat) else etat ;;
val tant_que : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

```
# let test = (>) 10 and etat = 0 and f = succ in tant_que test f etat ;;
- : int = 10
```

On peut, en particulier, simuler la boucle `for` qu'on avait vue dans la Section 2.2.5:

```
# let test_pour ((i:int),(n:int),e) = (i<=n) ;;
# let pour = fun n f i etat ->
    let r = tant_que test (function (i,n,e) -> ((succ i),n,(f i e))) (i,n,etat) in
      match r with (u,v,e) -> e ;;
# let concat i etat = etat^string_of_int(i) ;;
val concat : int -> string -> string = <fun>
# pour 9 concat 0 "" ;;
- : string = "0123456789"
```

2.3 Calculabilité par machine de Turing

2.3.1 Introduction

Nous faisons maintenant une troisième tentative pour définir ce qu'est une fonction « calculable ». Cette fois-ci, notre objectif sera de rester très proche d'un modèle physique de machine. Chacun des modèles de machine qu'on peut définir est associé au langage que cette machine reconnaît. Dans le cas des automates, il s'agit des langages rationnels, dans le cas de automates à pile, des langages hors-contexte. La définition de ces langages fait apparaître des structures qui suggèrent que leur reconnaissance peut être automatisée; c'est en particulier le cas du langage $L = \{a^n b^n c^n \ : \ n \in \mathbb{N}\} \subset \{a, b, c\}^*$ qui a tout l'air d'être reconnaissable par une procédure effective mais on peut montrer que les automates à pile, et *a fortiori* les automates finis, ne sont pas assez puissants pour reconnaître les mots de L.

Cette classe de machines, plus puissantes que les automates à pile, s'appelle les *machines de* TURING, et ont été introduites par Alan TURING en 1936. Leur différence principale avec les automates à pile est que l'accès à la mémoire ne se fait pas selon la logique LIFO (*Last In First Out*).

2.3.2 Définition d'une machine de TURING

Définition 11 *Une **machine de TURING (déterministe, à un ruban)** peut s'imaginer comme un dispositif M caractérisé par:*

- *un ruban unidimensionnel contenant une infinité dénombrable (le ruban peut être considéré comme infini « à droite » ou alors comme infini dans les deux sens, compte-tenu de la façon dont elle est accédée, cela ne change rien) de cases sur lequel est inscrit une chaîne x de caractères d'un **alphabet** Σ, toutes les autres cases étant remplies avec le **symbole blanc**;*
- *un ensemble fini $Q = \{q_0, \ldots, q_n\}$ d'**états** internes de la machine possibles. La machine part de l'**état initial** (que l'on note traditionnellement q_0) pour se retrouver, à un instant donné, dans un état q et une tête de lecture/écriture est placée sur le ruban en une certaine case x_i.*
- *une partie non-vide $F \subseteq Q$ d'états dits **acceptants** correspondant à la fin d'un calcul.*
- *une fonction δ partielle dite **de transition** décrivant la marche à suivre lorsque la machine est dans l'état q, sa tête de lecture placée sur x_i, c'est-à-dire:*
  - *le nouvel état q' dans lequel la machine se trouve;*
  - *le symbole x'_i que la tête de lecture/écriture va mettre à la place de x_i;*
  - *le déplacement éventuel de la tête de lecture d'une case vers la gauche (sur x_{i-1}), vers la droite (sur x_{i+1}) ou son maintien dans la position actuelle (sur x'_i).*
- *on a donc $\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{\mathsf{g}, \mathsf{d}, \mathsf{s}\}$; la machine s'**arrête** si $\delta(q, x_i)$ n'est pas défini ou si q est un état acceptant.*

Pour tout mot $x \in \Sigma^*$, on a trois cas de figure:

- *il existe une succession de transitions de M partant de (q_0, x_1) et conduisant à un état (q, y_i) où $q \in F$ est un état acceptant et on dit que M **accepte** x;*
- *il existe une succession de transitions de M partant de (q_0, x_1) et conduisant à un état (q, y_i) duquel aucune transition n'est définie et on dit que M **rejette** x;*
- *l'exécution de M sur x à partir de (q_0, x_1) se poursuit indéfiniment et on dit que M **ne termine pas** sur x.*

*Le **langage accepté** par M est l'ensemble L des mots acceptés. Le **langage rejeté** par M est l'ensemble $\bar{L}$ des mots rejeté. Si M termine toujours, alors $\Sigma^* = L \sqcup \bar{L}$ et on dit que L est **décidé** par M.*

*Un langage $L \subseteq \Sigma^*$ est dit **récursivement énumérable** ssi il existe une machine de* TURING *qui l'accepte. Si en outre elle le décide, il est dit **récursif**.*

Si M termine toujours, la ***fonction calculée*** *par M est la fonction $f : \Sigma^* \longrightarrow \Sigma^*$ qui associe à x le mot y qui est écrit sur le ruban lorsque la machine s'arrête (soit dans un état acceptant si x est dans le langage décidé par L, soit dans un état rejetant sinon). Une fonction $f : \Sigma^* \longrightarrow \Sigma^*$ est* ***MT-calculable*** *ssi il existe une machine de* TURING *qui calcule f.*

Le nombre de transitions de M lors du calcul de $f(x)$ s'appelle la ***complexité en temps*** *que met M pour calculer $f(x)$. Le nombre maximal de cases du ruban de M utilisées lors du calcul s'appelle la* ***complexité en espace*** *qu'utilise M pour calculer $f(x)$.*

On peut remarquer une différence intéressante entre les points de vue du λ-calcul et des machines de Turing. En effet, le ruban de ces dernières est une structure qui est altérée au cours du *temps*. C'est cette vision qui est le point de départ de la programmation impérative. Lorsqu'on écrit, lors d'une *affectation*, $x \leftarrow 1$ dans un langage impératif, cela se traduit en termes de machines de Turing comme l'écriture du symbole 1 dans la case d'adresse x et on écrira $x = 1$. Cependant, une « fonction » f, au sens impératif, appliquée à x peut tout-à-fait, à l'étape suivante, écrire 2 dans la case d'adresse x et rendre caduque cette équation $x = 1$. Dans la vision impérative, on peut, en quelque sorte, dire que les équations dépendent du temps. Voilà qui éclaire sur la difficulté, voire l'impossibilité, de faire des preuves sur des programmes écrits dans les langages impératifs car une égalité à l'instant t n'en est plus une à l'instant $t + 1$...

Voici qui donne en ce domaine un avantage incontestable aux langages fonctionnels comme Ocaml. En effet une déclaration telle que `let x = 1` fait de `x` le *synonyme* de `1`, et aucune fonction appliquée à x ne pourra remettre en cause l'égalité `x` = `1`.

2.3.3 Théorème fondamental de la calculabilité

Le théorème fondamental qui unifie les trois définitions de fonctions « calculables » que nous avons abordées est le suivant:

Théorème 4 *Soit $f : \mathbb{N}^p \longrightarrow \mathbb{N}$, alors les propositions suivantes sont équivalentes:*

- *f est λ-définissable;*
- *f est récursive;*
- *f est MT-calculable.*

Nous ne démontrerons pas ce théorème mais on a pu constater à travers les exemples de programme que nous avons donnés que le λ-calcul, dont Ocaml représente une implémentation, nous a permis de définir les fonctions récursives primitives et même, plus généralement, les fonctions récursives (comme la fonction d'Ackermann ou les fonctions définissables avec une boucle `while`). En ce qui concerne les fonctions MT-calculables, nous nous contenterons d'un exemple en Ocaml pour nous convaincre que le langage $L = \{a^n b^n c^n \ : \ n \in \mathbb{N}\} \subset \{a, b, c\}^*$ est bien décidable:

```
# let test = function
    "" -> true
  | s  -> let l = String.length s in
           let n = l/3 in
             if (l = 3*n) then
               let rec test_aux = function index -> match index with
                 i when (i<n)   -> if s.[i]='a' then test_aux (index + 1) else false
               | i when (i<2*n) -> if s.[i]='b' then test_aux (index + 1) else false
               | i when i=(l-1) -> s.[i]='c'
```

```
                | _ as i         -> if s.[i]='c' then test_aux (index + 1) else false
                in test_aux 0
              else false ;;
val test : string -> bool = <fun>
# test "" ;;
- : bool = true
# test "aabcc" ;;
- : bool = false
# test "abc" ;;
- : bool = true
# test "aaabbbccc" ;;
- : bool = true
```

Avant de refermer cette section, il faut remarquer que les fonctions λ-définissables *bien typées* représentent un sous-ensemble strict des fonctions λ-définissables et que le λ-calcul typé est donc théoriquement moins puissant que le λ-calcul pur. Par conséquent, du fait du Théorème 4, il existe des fonctions récursives ou MT-calculables, qu'on ne peut pas représenter par un λ-terme typé. Ces cas pathologiques n'apparaissent que dans les laboratoires d'informaticiens théoriciens amateurs de curiosités et ne se produisent jamais en pratique.

2.3.4 Thèse de Church-Turing

Existe-t-il des fonctions qui sont « calculables par une procédure effective » et qui ne soient pas MT-calculables? Comme on l'a vu, toutes les tentatives pour donner une définition de ce que notre intuition appellerait une « procédure effective » se sont soldées par des définitions équivalentes. Church s'est exprimé pour donner son opinion, selon laquelle notre interprétation intuitive de ce signifie l'expression « procédure effective » conduit immanquablement à l'une des définitions équivalentes que l'on a vues. Cet « acte de foi » est connu sous le nom de *Thèse de* Church ou encore Thèse de Church-Turing et que l'on peut énoncer:

> « Les fonctions que l'on peut réussir à calculer "par un procédé effectif" sont les fonctions calculables par une machine de Turing ».

2.4 Fonctions non-calculables et indécidabilité

2.4.1 Problèmes de décision

Parmi les fonctions que l'on peut considérer, se trouvent une classe de fonctions dont la sortie ne prend que deux valeurs (*e.g.* vrai ou faux); il s'agit des problèmes de décision.

Définition 12 *Soit Σ un alphabet, un* ***problème de décision*** *est une fonction $f : \Sigma^* \longrightarrow \Sigma^*$ telle que $|f(\Sigma^*)| = 2$. Si f n'est pas MT-calculable, on dit aussi que f est* ***indécidable****.*

2.4.2 Exemples

Un problème de décision célèbre est le *problème de l'arrêt.* Il consiste à décider si l'excécution d'une machine de Turing M sur le ruban de laquelle on a écrit le mot x, va terminer ou pas.

Par construction, l'ensemble $\mathcal{M}$ des machines de Turing est dénombrable, et donc en bijection avec les entiers, ou encore avec l'ensemble $\{0,1\}^*$ des chaînes binaires. Soit $\varphi : \mathcal{M} \longrightarrow \{0,1\}^*$ cette bijection. Nous supposerons, sans perte de généralité, que toute machine $M \in \mathcal{M}$ fonctionne sur l'alphabet binaire.

Considérons l'alphabet $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{(}, \texttt{)}, \texttt{,}\}$ et le langage $L \subseteq \Sigma^*$ défini par

$$L = \{\, \texttt{(}\varphi(M)\texttt{,}x\texttt{)} \;:\; M \in \mathcal{M}\ ,\ x \in \{\texttt{0},\texttt{1}\}^* \;\big|\; M \text{ accepte } x \,\}\ .$$

Théorème 5 *Le langage L est indécidable.*

Démonstration: Sinon, soit $H \in \mathcal{M}$ décidant L, alors pour tout $M \in \mathcal{M}$ et tout $x \in \{\texttt{0},\texttt{1}\}^*$:

$$H\big(\texttt{(}\varphi(M)\texttt{,}x\texttt{)}\big) = \begin{cases} \textsf{accepte} & \text{si } M \text{ accepte } x \\ \textsf{rejette} & \text{si } M \text{ rejette } x \text{ ou ne termine pas sur } x\ . \end{cases}$$

Soit maintenant $D \in \mathcal{M}$ la machine telle que:

$$D\big(\texttt{(}\varphi(M)\texttt{)}\big) = \begin{cases} \textsf{accepte} & \text{si } H\big(\texttt{(}\varphi(M)\texttt{)}\texttt{,}\texttt{(}\varphi(M)\texttt{)}\big) = \textsf{rejette}\ ; \\ \textsf{rejette} & \text{si } H\big(\texttt{(}\varphi(M)\texttt{)}\texttt{,}\texttt{(}\varphi(M)\texttt{)}\big) = \textsf{accepte}\ , \end{cases}$$

alors on a:

$$D\big(\texttt{(}\varphi(D)\texttt{)}\big) = \begin{cases} \textsf{accepte} & \text{si } D\big(\texttt{(}\varphi(D)\texttt{)}\big) = \textsf{rejette}\ ; \\ \textsf{rejette} & \text{si } D\big(\texttt{(}\varphi(D)\texttt{)}\big) = \textsf{accepte}\ , \end{cases}$$

ce qui est une contradiction, donc une telle machine H n'existe pas et L est donc indécidable.

Ce théorème démontre en particulier qu'il est vain de chercher à écrire un *debugger* qui détecterait toutes les boucles infinies des programmes C (c'est aussi vrai pour n'importe quel langage de programmation).

On démontre également que l'égalité fonctionnelle est indécidable, c'est pourquoi, malgré le type polymorphe du signe `=`, Ocaml lève une exception lors de la comparaison de deux fonctions:

```
# (=) ;;
- : 'a -> 'a -> bool = <fun>
# (function x -> (x+1)) = (function y -> (y+1)) ;;
Exception: Invalid_argument "equal: functional value".
```

2.5 Logique et programmation fonctionnelle

2.5.1 Introduction

Nous avons motivé une partie de notre discours sur la programmation fonctionnelle par le fait que les langages fonctionnels pouvaient intéragir beaucoup mieux avec les mathématiques que les langages impératifs et permettaient, en particulier, la preuve de programmes.

L'isomorphisme de Curry-Howard [Ld93, p. 212] relie λ-calcul typé et les preuves de programme, mais nous n'aborderons pas le sujet ici. Mentionnons juste qu'y est associé un mécanisme appelé *extraction de programme* qui permet de fabriquer automatiquement, à partir de la preuve qu'un algorithme est correct, un programme implémentant celui-ci! Par exemple, en prouvant que l'algorithme d'Euclide calcule le pgcd de deux entiers avec un système permettant l'extraction (comme *Coq*), ce système génère un programme (Ocaml pour *Coq*) qui implémente cet algorithme.

Nous illustrerons ici l'autre point de l'intéraction entre mathématiques et programmation fonctionnelle, en donnant un exemple de la manière dont on peut déterminer qu'une formule mathématique (en calcul des propositions) est vraie ou non.

2.5.2 Formules propositionnelles

Définition 13 *Un* ***alphabet propositionnel*** *est un ensemble de la forme* $\mathcal{A} = \mathcal{P} \sqcup \mathcal{C} \sqcup \{(,)\}$ *où: $\mathcal{P}$ de symboles appelés* ***variables propositionnelles*** *et* $\mathcal{C} = \{\neg, \wedge\}$ *est un ensemble de symboles appelés* ***connecteurs propositionnels****. Les* ***formules du calcul propositionnel*** *sur $\mathcal{P}$ est le plus petit sous-ensemble $\mathcal{F}$ de $\mathcal{A}^*$ tel que:*

- *toute variable $P \in \mathcal{P}$ est un élément de $\mathcal{F}$;*
- *si $F \in \mathcal{F}$ alors la* ***négation*** *de F, notée $\neg F$, est dans $\mathcal{F}$;a*
- *si $F, F' \in \mathcal{F}$ alors la* ***conjonction*** *de F et F', notée $(F \wedge F')$ est dans $\mathcal{F}$.*

On définit la ***profondeur*** *π d'une formule:*

- $\pi(P) = 1$ *pour tout* $P \in \mathcal{P}$;
- $\pi(\neg P) = 1 + \pi(P)$
- $\pi(F \wedge F') = 1 + \pi(F) + \pi(F')$.

On définit également la ***disjonction*** $(F \vee F') \stackrel{\text{def}}{=} \neg(\neg F \wedge \neg F')$, *l'****implication*** $(F \rightarrow F') \stackrel{\text{def}}{=} (\neg F \vee F')$, *et l'****équivalence*** $(F \leftrightarrow F') \stackrel{\text{def}}{=} (F \rightarrow F') \wedge (F' \rightarrow F)$ *de F et F'.*

On représente naturellement les formules par un type somme:

```
# type formula =
    Var of string
  | Neg of formula
  | Con of formula * formula ;;
type formula = Var of string | Neg of formula | Con of formula * formula
```

et on définit des fonctions de construction:

```
# let var = function f -> Var(f);;
val var : string -> formula = <fun>
# let neg = function f -> Neg(f);;
val neg : formula -> formula = <fun>
# let con = fun f f' -> Con(f,f');;
val con : formula -> formula -> formula = <fun>
# let dis = fun f f' -> neg (con (neg f) (neg f'));;
val dis : formula -> formula -> formula = <fun>
# let imp = fun f f' -> dis (neg f) f';;
val imp : formula -> formula -> formula = <fun>
# let equ = fun f f' -> con (imp f f') (imp f' f);;
val equ : formula -> formula -> formula = <fun>
```

Une formule pourra être affichée avec la fonction:

```
# let print_formula f =
  let rec print_aux = function
      Var a -> a
    |  Neg a -> "not "^(print_aux a)
    |  Con (a,b) -> "("^(print_aux a)^" and "^(print_aux b)^")" in
    print_string (print_aux f);;
val print_formula : formula -> unit = <fun>
```

2.5.3 Sémantique

Définition 14 *Une fonction $v : \mathcal{P} \longrightarrow \{\mathsf{faux}, \mathsf{vrai}\}$ s'appelle une* ***valuation****. On étend v aux formules propositionnelles $\mathcal{F}$ en notant:*

- $v(F) = \mathsf{vrai}$ *ssi* $v(\neg F) = \mathsf{faux}$*;*
- $v(F \wedge F') = \mathsf{vrai}$ *ssi* $v(F) = v(F') = \mathsf{vrai}$*;*

on peut vérifier qu'on a:

- $v(F \vee F') = \mathsf{faux}$ *ssi* $v(F) = v(F') = \mathsf{faux}$*;*
- $v(F \rightarrow F') = \mathsf{faux}$ *ssi* $v(F) = \mathsf{vrai}$ *et* $v(F') = \mathsf{faux}$.
- $v(F \leftrightarrow F') = \mathsf{faux}$ *ssi* $v(F) = v(F')$.

*Soient $P_1, \ldots, P_n$ les variables apparaissant dans une formule F, l'****interprétation*** *de F associée à v est le n-uplet de booléens $\big(v(P_1), \ldots, v(P_n)\big)$; elle est dite* ***satisfaisante*** *ssi $v(F) = \mathsf{vrai}$; elle est dite* ***falsifiante*** *ssi $v(F) = \mathsf{faux}$. Une formule F est une* ***tautologie*** *ssi toute interprétation la satisfait; c'est une* ***antilogie*** *ssi toute interprétation la falsifie; elle est* ***contingente*** *si certaines interprétations la satisfont et d'autres la falsifient. Deux formules F, F' sont* ***équivalentes****, ce que l'on note $F \equiv F'$ ssi, pour toute valuation, $v(F) = v(F')$. Soit $\Sigma \subseteq \mathcal{F}$, une formule F est* ***déductible*** *(on dit aussi* ***est une conséquence logique****) de Σ ssi, pour toute valuation v, le fait que v satisfasse toute formule de Σ implique que v satisfait F. L'ensemble Σ est* ***satisfaisable*** *ssi il existe une valuation qui satisfasse toutes les formules de Σ.*

Le type `interpretation` est une liste de couples qui associent une chaîne représentant une variable à un booléen:

```
# type interpretation = (string * bool) list ;;
type interpretation = (string * bool) list
```

et on définit une fonction subsituant une interprétation dans une formule:

```
# let rec subs i = function
    Var a -> (List.assoc a i)
  | Neg f -> not (subs i f)
  | Con(f,f') -> (subs i f) && (subs i f');;
val subs : (string * bool) list -> formula -> bool = <fun>
```

puis, une fonction d'affichage:

```
# let rec print_interpretation (i:interpretation) = match i with
    [] -> print_string ""
  | (c,valeur)::i' -> begin print_string("  "^c^" = "^string_of_bool(valeur)^"\n");
                            print_interpretation i' end ;;
val print_interpretation : interpretation -> unit = <fun>
```

2.5.4 Problème SAT et arbres de Beth

Le problème de décision consistant, étant donnée une formule F, à répondre à la question: « existe-t-il une valuation qui satisfasse F » s'appelle le problème SAT. Soient $p_1, \ldots, p_n$ les variables apparaissant dans F, on peut répondre à la question précédente en examinant les 2^n valuations correspondant aux substitution de p_i par faux ou par vrai, dans F, pour $1 \leq i \leq n$. COOK et LEVIN ont démontré [Deh00, Proposition 3.3, p. 237] qu'il était difficile de trouver une méthode rapide pour décider de la satisfaisabilité d'une formule en général. Nous utiliserons cependant une méthode non-triviale, recourant aux *arbres de* BETH.

On va se ramener récursivement à des sous-formules de plus en plus petites jusqu'à aboutir à des variables. En effet, supposons qu'il existe une interprétation I satisfaisant une formule F,

- si F est de la forme $\neg F'$, alors I falsifie F';
- si F est de la forme $(F' \wedge F'')$, alors I satisfait F' et F''.

De même, supposons qu'il existe une interprétation I falsifiant une formule G,

- si G est de la forme $\neg G'$, alors I satisfait G';
- si G est de la forme $(G' \wedge G'')$, alors I falsifie au moins l'une des deux sous-formule G' ou G'';

En poursuivant récursivement ce raisonnement, on construit, étape par étape, un ensemble de sous-formules Φ à satisfaire et un ensemble Ψ de sous-formules à falsifier pour chaque cas envisagé. À l'issue d'un nombre fini d'étapes, on aboutit à une situation où on n'a que des formules atomiques (*i.e.* des variables).

Définition 15 *Un* ***arbre de*** **BETH** *est arbre dont les nœuds sont étiquetés par un couple* (Φ, Ψ) *d'ensemble de formules telles que, pour tout nœud* N *de l'arbre, étiqueté par* (Φ, Ψ)*, avec* $\Phi = \{F_1, \ldots, F_n\}$ *et* $\Psi = \{G_1, \ldots, G_m\}$*, on ait:*

- *pour tout* $i \in \{1, \ldots, n\}$*:*
  - *si* $F_i = \neg F'$*, alors* N *a un fils, étiqueté par* $\big(\Phi \setminus \{F_i\}, \Psi \cup \{F'\}\big)$*;*
  - *si* $F_i = (F' \wedge F'')$*, alors* N *a un fils, étiqueté par* $\Big(\big(\Phi \setminus \{F_i\}\big) \cup \{F', F''\}, \Psi\Big)$*;*
- *pour tout* $j \in \{1, \ldots, m\}$*:*
  - *si* $G_i = \neg G'$*, alors* N *a un fils, étiqueté par* $\big(\Phi \cup \{G'\}, \Psi \setminus \{G'\}\big)$*;*
  - *si* $G_i = (G' \wedge G'')$*, alors* N *a deux fils, l'un étiqueté par* $\Big(\Phi, \big(\Phi \setminus \{G_i\}\big) \cup \{G'\}\Big)$*, l'autre, par* $\Big(\Phi, \big(\Phi \setminus \{G_i\}\big) \cup \{G''\}\Big)$*.*

Un tel arbre dont la racine la racine est étiquetée par $\big(\{F\}, \varnothing\big)$ *(resp.* $\big(\varnothing, \{F\}\big)$*) s'appelle* ***l'arbre de satisfaction (resp. de falsification) de*** **BETH** *de* F*.*

Un nœud d'un arbre de BETH est donc représenté par un couple de liste de formules:

```
# type node = (formula list * formula list) ;;
type node = formula list * formula list
```

La proposition suivante est immédiate:

Proposition 2 *Soit une feuille de l'arbre de satisfaction (resp. falsification) de* Beth *de F, étiquetée par (Φ, Ψ). Si Φ et Ψ sont disjoints, alors la substitution par* vrai *des variables de F apparaissant dans Φ et la substitution par* faux *de celles apparaissant dans Ψ correspond à une interprétation satisfaisant (resp. falsifiant) F. En outre:*

- *si aucune feuille de l'arbre de satisfaction de* Beth *de F n'est de la forme précédente, F est une antilogie;*
- *si aucune feuille de l'arbre de falsification de* Beth *de F n'est de la forme précédente, F est une tautologie.*

D'un point de vue algorithmique, la non-disjonction des deux sous-ensemble de formules étiquetant un nœud de l'arbre de satisfaction (resp. de falsification) constitue une incohérence qui suffit à conclure que la branche en cours d'examen ne peut pas conduire à une interprétation satisfaisante (resp. falsifiante). Lors de la détection d'une telle incohérence, on lèvera une exception:

```
# exception Incoherence ;;
exception Incoherence
```

En particulier, si l'on souhaite ajouter un couple `(p,valeur)` où `p` est une variable propositionnelle et `valeur` un booléen, à une interprétation `i` en cours d'élaboration, on peut utiliser la fonction `List.assoc p i` qui renvoie une certaine valeur `v` si la liste `i` contient déjà un couple de la forme `(p,v)` (ce qui constitue une incohérence si `v` $\neq$ `valeur`) ou lève l'exception `Not_found` si aucun couple dont le premier élément est `p` n'est trouvé dans `i`:

```
# let append = fun (i:interpretation) (p:string) (valeur:bool) ->
  try if (List.assoc p i)=valeur then i else raise Incoherence
  with Not_found -> (p,valeur)::i ;;
val append : interpretation -> string -> bool -> interpretation = <fun>
```

La fonction `beth`, appliquée à une interprétation `i` et à un nœud `n`, va renvoyer une interpretation compatible avec `i` et `n` si elle existe, et propager le signal `Incoherence`, sinon. La methode consiste à parcourir recursivement un arbre de Beth. Une fois atteinte une feuille, on renvoie l'interpretation construite au cours de cette construction. La decouverte d'une incoherence fait cesser les investigations dans la branche en cours. Cette decouverte est faite lors de l'appel a append dans le cas ou on a des variables. Dans le cas ou on a la conjonction de `f1` et `f2` a falsifier, si la branche de Beth sur `f1` conduit conduit a une incoherence, on tente avec la branche sur `f2`. Si aucune incoherence n'est detectee, on aboutit a une feuille et on renvoie l'interpretation correpondante:

```
# let rec beth = fun (i:interpretation) (n:node) -> match n with
    ([],[])                -> i (* C'est une feuille *)
  | ([],(Var p)::l)        -> beth (append i p false) ([],l)
  | ([],(Neg f)::l)        -> beth i ([f],l)
  | ([],(Con(f1,f2))::l)   -> begin try beth i ([],f1::l) with Incoherence -> beth i ([],f2::l) end
  | ((Var p)::l,l2)        -> beth (append i p true) (l,l2)
  | ((Neg f)::l,l2)        -> beth i (l,f::l2)
  | ((Con(f1,f2))::l,l2)   -> beth i (f1::f2::l,l2) ;;
val beth : interpretation -> node -> interpretation = <fun>
```

Enfin, on écrit une fonction `test` qui va tenter dans un premier temps de calculer une interprétation satisfaisant une formule `f`. Si le signal `Incoherence` est percu, cela signifie qu'aucune interprétation satisfaisante n'a ete trouvée: `f` est une antilogie. On tente ensuite de calculer une interprétation falsifiant `f`. Si le signal `Incoherence` est perçu, cela signifie qu'aucune interprétation falsifiante n'a ete trouvee: `f` est une tautologie. Si aucune de ces situations n'a eu lieu, on a trouve une interprétation satisfaisante et une interpretation falsifiante: `f` est contingente:

```
# let test = function f ->
  print_string "la formule";
  try  let i_satisfy = beth [] ([f],[]) in
    try  let i_falsify = beth [] ([],[f]) in
      assert (subs i_satisfy f) ;
      assert (not (subs i_falsify f)) ;
      print_string (" est contingente:\nelle est satisfaite pour:\n");
      print_interpretation i_satisfy;
      print_string ("et falsifiee pour\n");
      print_interpretation i_falsify;
    with Incoherence -> print_string (" est une tautologie.\n")
  with Incoherence -> print_string (" est une antilogie.\n") ;;
val test : formula -> unit = <fun>
```

2.5.5 Application aux clubs écossais

Le maire libéral McArthur d'une petite bourgade d'Écosse, souhaite fusionner en un « club McArthur » les membres du club McPherson et ceux du club McKinnon. Or les membres de ces clubs tiennent énormément à respecter scrupuleusement leurs traditions séculaires: le club McPherson obéit, depuis 1304, aux règles suivantes:

- tout membre non-écossais porte des chaussettes rouges;
- tout membre porte un kilt ou ne porte pas de chaussettes rouges;
- les membres mariés ne sortent pas le dimanche;

et le club McKinnon respecte la charte de 1431:

- un membre sort le dimanche si et seulement s'il est écossais;
- tout membre qui porte un kilt est écossais et marié;
- tout membre écossais porte un kilt.

Définissons tout d'abord les différentes variables:

```
# let e = var "est ecossais" and
  c = var "a des chaussettes rouges" and
  k = var "porte un kilt" and
  m = var "est marie" and
  d = var "sort le dimanche";;
val e : formula = Var "est ecossais"
val c : formula = Var "a des chaussettes rouges"
val k : formula = Var "porte un kilt"
val m : formula = Var "est marie"
val d : formula = Var "sort le dimanche"
```

puis, les règles des deux clubs:

```
# let r1 = imp (neg e) c and
  r2 = dis k (neg c) and
  r3 = imp m (neg d) and
  r4 = con (imp d e) (imp e d) and
  r5 = imp k (con e m) and
  r6 = imp e k;;
val r1 : formula =
  Neg (Con (Neg (Neg (Neg (Var "est ecossais"))), Neg (Var "a des chaussettes rouges")))
val r2 : formula =
  Neg (Con (Neg (Var "porte un kilt"), Neg (Neg (Var "a des chaussettes rouges"))))
val r3 : formula =
  Neg (Con (Neg (Neg (Var "est marie")), Neg (Neg (Var "sort le dimanche"))))
val r4 : formula =
  Con (Neg (Con (Neg (Neg (Var "sort le dimanche")), Neg (Var "est ecossais"))),
   Neg (Con (Neg (Neg (Var "est ecossais")), Neg (Var "sort le dimanche"))))
val r5 : formula =
  Neg (Con (Neg (Neg (Var "porte un kilt")), Neg (Con (Var "est ecossais", Var "est marie"))))
val r6 : formula =
  Neg (Con (Neg (Neg (Var "est ecossais")), Neg (Var "porte un kilt")))
```

Définissons une fonction qui construit la formule qui est la conjonction d'une liste de formules:

```
# let rec (con_formulas:(formula list) -> formula) = function
    [] -> failwith "con_formulas: list shouldn't be empty"
  | [f] -> f
  | h::t -> con h (con_formulas t);;
val con_formulas : formula list -> formula = <fun>
```

et appliquons-la pour constituer la charte `rules1` du club McPherson:

```
# let rules1 = con_formulas [r1;r2;r3];;
val rules1 : formula =
  Con (Neg (Con (Neg (Neg (Neg (Var "est ecossais"))), Neg (Var "a des chaussettes rouges"))),
  Con (Neg (Con (Neg (Var "porte un kilt"), Neg (Neg (Var "a des chaussettes rouges")))),
  Neg (Con (Neg (Neg (Var "est marie")), Neg (Neg (Var "sort le dimanche"))))))
```

puis testons que ce club peut contenir au moins un membre et est un minimum sélectif:

```
# test rules1;;
la formule est contingente:
elle est satisfaite pour:
  est ecossais = true
  porte un kilt = true
  est marie = false
et falsifiee pour
  est ecossais = false
  a des chaussettes rouges = false
- : unit = ()
```

Toute personne écossaise, pourtant un kilt et mariée a ses chances d'être dans ce club, alors que toute personne n'étant ni écossaise, ni pourvue de chaussettes rouges, ne peut se voir accueillie dans cette grande institution. Faisons maintenant de même pour le club McKinnon, dont voici la charte `rules2`:

```
# let rules2 = con_formulas [r4;r5;r6];;
val rules2 : formula =
  Con (Con (Neg (Con (Neg (Neg (Var "sort le dimanche")), Neg (Var "est ecossais"))),
  Neg (Con (Neg (Neg (Var "est ecossais")), Neg (Var "sort le dimanche")))),
  Con (Neg (Con (Neg (Neg (Var "porte un kilt")),
  Neg (Con (Var "est ecossais", Var "est marie")))),
  Neg (Con (Neg (Neg (Var "est ecossais")), Neg (Var "porte un kilt")))))
```

Ce club-là peut envisager l'adhésion d'une personne ni écossaise, ni même vêtue d'un kilt, ne sortant pas le dimanche, mais n'accepterait pas un non-écossais sortant le dimanche:

```
# test rules2;;
la formule est contingente:
elle est satisfaite pour:
  sort le dimanche = false
  porte un kilt = false
  est ecossais = false
et falsifiee pour
  sort le dimanche = true
  est ecossais = false
- : unit = ()
```

Montrons maintenant que les plans de M. McArthur sont compromis, en envisageant le règlement extrêmement contraignant qu'il souhaite imposer:

```
# let rules = Con(rules1,rules2);;
val rules : formula =
  Con (Con (Neg (Con (Neg (Neg (Neg (Var "est ecossais"))),
  Neg (Var "a des chaussettes rouges"))),
  Con (Neg (Con (Neg (Var "porte un kilt"), Neg (Neg (Var "a des chaussettes rouges")))),
  Neg (Con (Neg (Neg (Var "est marie")), Neg (Neg (Var "sort le dimanche")))))),
  Con (Con (Neg (Con (Neg (Neg (Var "sort le dimanche")), Neg (Var "est ecossais"))),
  Neg (Con (Neg (Neg (Var "est ecossais")), Neg (Var "sort le dimanche")))),
  Con (Neg (Con (Neg (Neg (Var "porte un kilt")),
  Neg (Con (Var "est ecossais", Var "est marie")))), Neg (Con (Neg (Neg (Var "est ecossais")),
  Neg (Var "porte un kilt"))))))
# test rules;;
la formule est une antilogie.
- : unit = ()
```

Le club McArthur est si strict, qu'il ne peut contenir aucun membre...

Chapitre 3

Objective Caml

3.1 Historique

Le *Lisp*, créé au *Massachusetts Institute of Technology (MIT)* dans les années 1960 fut le patriarche des langages fonctionnels. Beaucoup de dialectes de ce langage en ont été dérivés, comme *elisp* qui sert à programmer le surpuissant éditeur de texte *emacs* et *scheme* qui est utilisé dans le standard de la retouche d'image dans le monde du logiciel libre: *Gimp*.

Les langages ML (*Meta Language*) furent développés au début des années 1980. Les trois dialectes ML les plus connus sont *SML-NJ* (USA), *Moscou-ML* (Russie) et *Objective Caml* (France).

« CAML » est un acronyme en langue anglaise: « Categorical Abstract Machine Language » (langage de la machine abstraite catégorique) qui souligne que ce système est fondé sur une intéraction entre la théorie des catégories et le λ-calcul. La CAM est une machine abstraite (on dit aussi *machine virtuelle*) capable de définir et d'exécuter les fonctions. Le premier compilateur générant du code pour cette machine abstraite fut écrit en 1984.

La deuxième filiation de ce nom est ML (acronyme pour *Meta Language*): Caml est aussi issu de ce langage de programmation crée par Robin MILNER en 1978. La première version de Caml fut développée entre 1985 et 1990. Une implantation à machine virtuelle baptisée *Caml-Light* lui a succédé au début des années 1990. *Objective Caml* lui succède. Ce dernier est largement utilisé dans le monde pour l'éducation, la recherche (INRIA,CEA,...) et également par des industriels (France Télécom, Dassault...).

3.2 Objective Caml

3.2.1 Héritage ML

Dans *Objective Caml* comme dans tous les dialectes ML, on a les caractéristiques suivantes.

- **Les fonctions sont des valeurs de première classe**: elles peuvent être arguments d'autres fonctions, ou résultat d'un calcul.
- **Le langage est fortement typé**: la vérification de la compatibilité entre les types des paramètres formels et des paramètres d'appel permet d'éliminer la plupart des erreurs introduites par maladresse ou étourderie et contribue à la sûreté de l'exécution.
- **Le typage est inféré automatiquement**: l'utilisateur n'a pas à faire de déclarations de type, tous les types sont déduits automatiquement de la syntaxe des expressions.

- **Le typage est statique**, c'est-à-dire décidé à la compilation. Dès lors, il n'est pas nécessaire de faire ces vérifications durant l'exécution du programme ce qui accroît son efficacité.
- **Le typage est polymorphe paramétrique**: l'algorithme de typage reconnaît le type générique d'une fonction dont les paramètres — dits polymorphes — ne sont pas totalement spécifiés. Cela permet de développer un code générique réutilisable dans tous les contextes compatibles avec ce type polymorphe.
- **Un mécanisme d'exceptions** permet de rompre l'exécution normale d'un programme à un endroit et de la reprendre à un autre endroit du programme prévu à cet effet.
- **Le filtrage de motifs** donne une très grande expressivité au langage lors de définitions par cas;
- **La mémoire est gérée automatiquement** par un *ramasse-miette* (*garbage collector (GC)*).

3.2.2 Traits impératifs et orientation objet

Ocaml possède des traits impératifs qu'on peut utiliser lorsque la philosophie fonctionnelle s'avère inadaptée. En particulier, pour la gestion des entrées-sorties, les modifications physiques de valeurs. Les structures de contrôle itératives (`for` et `while`) sont également disponibles. Le mélange des deux styles offre une grande souplesse de développement. Pour ce qui est de l'orientation objet, Ocaml dispose d'un système de modules permettant la dissociation de l'interface avec l'implantation et d'un système de classes incluant l'héritage multiple.

3.2.3 Bibliothèques

Ocaml est activement développé à travers le monde et bénéficie de nombreuses bibiliothèques: entiers en précision arbitraire, analyses lexicale et syntaxique, processus légers (*multithreading*), interface graphique, réseau...

Ocaml peut également intéragir avec le langage C *via* l'appel de fonctions C à partir d'un programme Ocaml et *vice versa*. On peut alors exploiter, depuis Ocaml, les bibliothèques C.

3.2.4 Environnement de travail

Ocaml dispose d'un environnement de programmation complet incluant:

- **une machine virtuelle Ocaml** (`ocamlrun`);
- **un compilateur vers du code portable** (`ocamlc`), exécutable par la machine virtuelle Ocaml;
- **un compilateur vers du code natif** (`ocamlopt`) optimisé pour l'architecture spécifique choisie;
- **une boucle d'intéraction** (`ocaml`) permettant l'utilisation interactive de la machine virtuelle d'Ocaml; ces boucles peuvent être personnalisées avec *ocamlmktop*.
- **un *debugger*** (`ocamldebug`) permettant la pose de points d'arrêt, l'exploration de variables, pour trouver et réparer facilement les erreur*etc.*
- **un *profiler*** (`ocamlprof`) permettant d'optimiser les programmes écrits en Ocaml

– **un mode emacs** (`tuareg`) permettant de développer plus rapidement (colorisation automatique de la syntaxe, y compris lors de l'utilisation de la boucle d'intéraction dans emacs, indentation automatique, raccourcis-clavier, *etc*.

3.3 Conclusion et lectures suggérées

Pour la partie théorique, je n'ai malheureusement pas beaucoup de références à vous suggérer car assez peu d'écrits vraiment élémentaires sont disponibles à ma connaissance. Le livre *Approche fonctionnelle de la programmation* [CM95] de Guy Cousineau et Michel Mauny constitue cependant un très bon ouvrage sur le sujet. Pour tout ce qui a trait aux automates, machines de Turing, langages, *etc*, je vous invite à lire *Introduction à la calculabilité* [Wol91] de Pierre Wolper qui est très clair et pédagogique, ou encore *Mathématiques de l'Informatique*, de Patrick Dehornoy, qui est plus formel et plus complet. Quant à l'intéraction du λ-calcul avec la logique, elle est bien mise en évidence dans *Logique et fondements de l'informatique* de Richard Lassaigne et Michel de Rougemont. Je vous signale également la présence de plusieurs documents remarquables, en particulier sur les types et les preuves automatiques, sur la page web de Gilles Dowek (`http://pauillac.inria.fr/~dowek/`).

Pour la partie pratique, outre la documentation en ligne d'Ocaml (`http://caml.inria.fr`), *Le langage Caml* [WL99] de Pierre Weis and Xavier Leroy constitue une excellent premier contact avec les systèmes Caml. Il conviendra de faire attention à la syntaxe qui diffère légèrement de celle d'Ocaml. Pour Ocaml proprement dit, *Développement d'applications avec Objective Caml* [CMP00] d'Emmanuel Chailloux, Pascal Manoury et Bruno Pagano est la certainement la référence actuelle en la matière (bien que la syntaxe diffère également déjà de celle de la version courante d'Ocaml, qui se développe toujours très activement). Une version électronique intégrale est disponible sur `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html`.

Index

Bibliographie

[CM95] Guy COUSINEAU et Michel MAUNY. *Approche fonctionnelle de la programmation*. Ediscience international, 1995.

[CMP00] Emmanuel CHAILLOUX, Pascal MANOURY, et Bruno PAGANO. *Développement d'applications avec Objective Caml*. O'Reilly, 2000. disponible en ligne sur `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html`.

[Deh00] Patrick DEHORNOY. *Mathématiques de l'Informatique*. Dunod, 2000.

[Ld93] Richard LASSAIGNE et Michel DE ROUGEMONT. *Logique et fondements de l'informatique*. Hermes, 1993.

[WL99] Pierre WEIS et Xavier LEROY. *Le langage Caml*. Dunod, 1999. 2$^{\text{ème}}$ édition.

[Wol91] Pierre WOLPER. *Introduction à la calculabilité*. InterEditions, 1991.