Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec)

Titre	**État des lieux des langages fonctionnels**
Identifiant	**Livrable L2.1.2**
Version	**6.0**
Date	**2011-06-16**
Pages	**59**
Approbation	**Christèle Faure, SafeRiver**
	Date: Signature:

Table des révisions

Version	Date	Description et changements	Parties modifiées
1.0	2011-01-14	Version initiale	Tout le document
2.0	2011-01-02	Version suivant la revue de la version 1.0 du 2 février 2011	Tout le document
3.0	2011-05-11	Consolidation et réécriture pour livrable L2.1.2	Introduction, premier chapitre, fiches langages
4.0	2011-05-31	Version intégrant les commentaires de l'ANSSI	Tout le document
5.0	2011-06-06	Prise en compte de certains commentaires, version pour la réunion d'avancement du 7 juin 2011	Tout le document
6.0	2011-06-16	Prise en compte des remarques après réunion, livraison finale	Tout le document

Résumé

Ce document réalisé dans le cadre du projet LaFoSec propose un état de l'art des langages fonctionnels. Il présente d'abord les caractéristiques principales des langages fonctionnels. Suit une description succincte de quelques langages fonctionnels généralistes, afin de faciliter le choix de l'un de ces langages pour le développement d'applications.

Table des matières

Introduction

Objet du document

Ce document a été produit dans le cadre de l'étude LaFoSec, relative au marché n° 2010027960021207501 notifié le 8 novembre 2010 par le Secrétariat Général de la Défense et de la Sécurité Nationale (SGDSN). Il présente un état de l'art des langages fonctionnels, sans ambition d'en étudier la sécurité.

Présentation du projet LaFoSec

Les langages de programmation dits *fonctionnels* sont réputés offrir de nombreuses garanties facilitant le développement de logiciels soumis à des exigences de sûreté ou de sécurité. Par exemple, la société *Ericsson* a développé le langage fonctionnel Erlang, dédié à la concurrence, pour ses applications de communication. La société *Esterel Technologies* a développé le langage fonctionnel SCADE, dédié au traitement synchrone de flots de données, pour le traitement de logiciel critique.

Dans le cadre de ses activités d'expertise, l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) souhaite bénéficier d'une assistance scientifique et technique sur l'adéquation de ces langages au développement d'applications de sécurité et disposer d'une étude permettant d'améliorer la confiance vis-à-vis de la sécurité de ces applications. Ce travail est l'objet du projet LaFoSec.

Le projet LaFoSec consiste en une étude prospective des langages fonctionnels, visant à déterminer les caractéristiques propres à ces langages susceptibles de répondre aux exigences de sécurité.

Tout d'abord, les caractéristiques des langages fonctionnels sont décrites et plusieurs langages généralistes sont décrits selon ces caractéristiques. Ensuite, cette étude est approfondie en se restreignant à trois langages, OCaml, F$^{\#}$ et Scala, pour recenser leurs avantages et inconvénients du point de vue de la sécurité.

Présentation de l'objet du document

Ce document présente l'approche fonctionnelle de la programmation et plusieurs langages fonctionnels généralistes, afin de faciliter le choix du langage de développement d'un système logiciel industriel répondant à des exigences de sûreté et de sécurité, les deux problématiques étant distinctes mais non disjointes.

La construction d'un système logiciel dans n'importe quel langage, quelles que soient les fonctionnalités requises de ce système, comporte plusieurs grandes étapes :

- la spécification des besoins et des exigences auxquels ce système doit répondre ;
- la conception pendant laquelle la décomposition en sous-systèmes et la détermination de leurs interactions est définie ;
- le développement du code ;
- la démonstration de la conformité du système obtenu aux besoins et exigences à l'aide d'analyses, de preuves de propriétés, de vérification et de tests, de techniques de traçabilité ;
- en parallèle de ces phases, la production de la documentation.

Le choix d'un langage de programmation pour une application donnée se doit de prendre en compte son adaptation aux pratiques et connaissances des développeurs, sa popularité, son efficacité, etc. Ce choix devrait aussi être déterminé par des objectifs sur le programme produit : correction (absence de bogues), conformité vis à vis de sa spécification, traçabilité. Le projet LaFoSec doit aider à étayer ou infirmer l'affirmation que certains langages fonctionnels répondent mieux à ces objectifs que des langages fondés sur d'autres paradigmes.

Le chapitre 1 décrit les grands traits des langages fonctionnels afin de fournir une base à la présentation succincte des langages choisis dans le chapitre 2. Le chapitre 3 fournit une synthèse permettant une rapide comparaison de ces langages.

À la fin de ce document, se trouve un index des termes principaux donnant pour chacun la page à laquelle il est défini.

Chapitre 1

Traits des langages fonctionnels

Les langages fonctionnels ne sont pas d'invention récente puisque le premier langage fonctionnel, ISWIM, date des années 50 et est donc contemporain du premier langage impératif Fortran. Pendant que les langages impératifs se multipliaient avec ALGOL, Pascal, C, APL, ADA, etc. la liste des langages fonctionnels s'étoffait avec les langages de la famille LISP ou dérivés comme Scheme, les langages de la famille ML (conçu par Robin MILNER au tout début des années 80 à Édimbourg) avec CAML, SML, puis OCaml, Haskell, F#, etc. et des langages dédiés comme Erlang. Parallèlement, la plupart de ces langages, fonctionnels ou impératifs, ont été étendus par des traits modulaires et des traits objet, conduisant à des langages comme Java, C++, Scala. Ainsi, la majorité des langages actuels proposent plusieurs paradigmes de programmation parmi ceux qui viennent d'être évoqués.

Il est important de comprendre la différence entre les paradigmes impératif et fonctionnel (quelles que soient leurs extensions) car elle oriente fortement la manière de concevoir les programmes. C'est pourquoi ces deux paradigmes sont rapidement présentés ici.

Les **langages impératifs** sont fondés sur la notion d'*état d'une machine* défini par les **variables du programme**, qui sont des noms associés à des zones mémoire. L'instruction fondamentale de ces langages est l'**affectation**, qui permet de changer le contenu de la zone mémoire. Les autres instructions ne servent *in fine* qu'à combiner des affectations. Un programme impératif décrit, par une succession d'affectations, les changements d'état nécessaires à l'obtention d'un état final depuis un état initial. La valeur contenue dans une variable évolue donc au cours de l'exécution, ce sont donc ces évolutions qu'il faut suivre pour comprendre ce que fait un programme. Cela n'est pas toujours simple : tout

lecteur de code impératif produit par un tiers en a fait l'expérience.

L'exemple suivant, écrit en syntaxe C, permet de montrer l'influence de l'ordre d'évaluation des opérandes d'un opérateur binaire (en général non spécifié donc laissé au choix du compilateur) sur les valeurs calculées.

```
int a = 25;
int P1 (int b) {
  a = a + 10;
  return (a + b);
}
```

L'expression `(P1(2) - P1(2))` ne vaut pas `0` comme on pourrait le supposer, mais `10` si on évalue d'abord l'opérande de droite et `-10` si on choisit l'ordre inverse. Cet exemple illustre bien les ambiguïtés dues à l'affectation : une même expression comme `P1(2)` peut désigner des valeurs différentes selon sa position dans le programme. Ce programme `P1` n'a pas d'équivalent dans un langage fonctionnel pur. Plus généralement, le style impératif ne facilite ni la relecture du code, ni sa maintenance. Ainsi, les normes de développement des systèmes critiques telles que EN 61508 déconseillent l'utilisation de certaines constructions des langages impératifs comme les instructions de branchement inconditionnel (`goto`), les pointeurs de fonction, etc.

Les **langages fonctionnels** sont fondés sur les notions mathématiques de fonction et d'application de fonction. Les langages fonctionnels dits **purs** ne proposent pas d'affectation ni d'autre structure de contrôle que la composition de fonctions (pas de `for`, de `while` ni de `goto`). Ils offrent en revanche des expressions conditionnelles et la récursivité. Soit le programme `p2` [1] écrit dans un langage fonctionnel :

```
let a = 25
let p2 (b) = a + b
```

L'expression `(p2 (2) - p2 (2))` vaut `0` quel que soit l'ordre d'évaluation des opérandes de la soustraction.

En lieu et place de l'affectation et des manipulations de pointeur, les langages fonctionnels fournissent souvent des outils de construction et de traitement de données structurées (listes, arbres, etc.) permettant d'éviter totalement la manipulation de pointeurs. Cela conduit à des implémentations beaucoup plus proches de la spécification qu'elles ne le sont dans un langage impératif. De plus,

1. Parmi les syntaxes possibles, une syntaxe à la ML a été choisie pour tous les exemples de ce chapitre.

les compilateurs des langages fonctionnels offrent souvent des analyses statiques puissantes, comme le typage. Tous ces points contribuent à l'amélioration de la traçabilité de la spécification dans l'implémentation et à la qualité du code produit. C'est pourquoi ces langages peuvent être préférés lorsque l'on considère la facilité de développement et l'analyse de conformité de l'implémentation à la spécification.

Tous les langages fonctionnels disposent d'un noyau fonctionnel pur, suffisant pour programmer n'importe quelle application. Cependant, la plupart d'entre eux ont été étendus avec des traits impératif et objet, qui sont succinctement décrits dans la suite de ce chapitre.

Les langages fonctionnels ont une efficacité d'exécution comparable aux langages impératifs compilés, voire même un peu supérieure pour des applications manipulant de grosses structures de données [Harrop, 2007, Marceau, 2009]. Cela est attesté par les utilisations industrielles de ces langages dans les domaines classiques de l'informatique (manipulation de bases de données, gestion de contrats financiers, services Web, etc.) mais également dans le domaine du développement d'outils d'analyse de langages, de compilation, de systèmes d'aide à la preuve, etc.

1.1 Noyau fonctionnel

Cette partie décrit les caractéristiques essentielles des langages fonctionnels généralistes modernes : définition et application de fonctions, typage et filtrage.

1.1.1 Programmes fonctionnels

Un **programme fonctionnel** est une suite de déclarations et d'évaluations d'**expressions**, construites à partir d'applications de fonctions à des données. Une **déclaration** est une construction syntaxique qui permet de nommer une valeur, cette valeur étant définie par une expression du langage : le nom et la valeur sont dits **liés**. La **portée** d'une déclaration est la portion du texte source où l'identificateur est lié par cette déclaration.

Déclarations

Une déclaration est dite **globale** au programme si le nom peut être utilisé dans toute la partie du programme suivant son introduction. Par exemple, les

déclarations globales sont introduites par la construction `let ...` dans les langages à la ML et par la construction `(define ...)` en Scheme. Une déclaration est dite **locale** si le nom ne peut être utilisé que dans une portion délimitée du programme, en général une expression. Par exemple, les déclarations locales sont introduites par la construction `let ... in ...` dans les langages à la ML et par la construction `(let ...)` en Scheme.

```
let perimetre (a, b) = 2 * (a + b)
let main (long, larg) =
   let surface = function (x, y) -> x * y
   in
   (surface (long, larg), perimetre (long, larg))
main(5, 4)
```

Dans l'exemple ci-dessus (où ne manquent que les entrées-sorties), la déclaration globale introduit le nom `perimetre` et le *lie* à la fonction de **paramètre formel** `(a, b)` et de **corps** `2 * (a + b)`. La déclaration locale de `surface` nomme l'**expression fonctionnelle** `function (x, y) -> x * y` définissant la fonction liée à `surface`. Enfin la fonction `main` calcule la surface et le périmètre d'un rectangle à partir des longueur et largeur fournies en paramètre. La portée de `surface` est restreinte au corps de `main`.

Evaluer une **application de fonction**[2] à un **paramètre effectif** consiste à remplacer, dans le corps de la fonction, le paramètre formel par la valeur du paramètre effectif puis à évaluer le corps ainsi modifié. Ce mode de calcul très simple et intuitif est celui de tout programme purement fonctionnel. Ainsi le résultat du programme ci-dessus est défini par l'application de `main` à `(5, 4)` notée `main (5, 4)` et a pour valeur le couple `(20, 18)`.

Stratégies d'évaluation

Il reste à préciser à quel moment l'expression passée en paramètre est évaluée. Cette évaluation est réalisée soit dès le début de l'évaluation — le langage est alors dit **à évaluation stricte** — soit uniquement si cette valeur est utilisée — le langage est alors dit **à évaluation paresseuse**. Ce dernier mode a l'avantage de ne pas faire de calculs inutiles (par exemple, si `f (x, y) = x`, alors le calcul de `f (25, e)` ne requiert pas l'évaluation de `e`). Mais le premier mode permet de déterminer sans ambiguïté à quel moment le paramètre est évalué, un point qui

2. L'application d'une fonction correspond à l'"appel d'une fonction" dans les langages impératifs et à l'"envoi de message" dans certains langages orientés objet.

est fondamental dès que le langage comporte des traits non-fonctionnels. Quelle que soit la stratégie d'évaluation choisie par le langage, la valeur d'une expression est la même.

Le mécanisme d'exécution des langages fonctionnels purs est donc celui du calcul mathématique, de sorte qu'un code source fonctionnel pur peut se lire en faisant totalement abstraction de la gestion des données en mémoire. Appliquer une fonction à une valeur donnée retourne toujours le même résultat, quelle que soit la localisation de cette application dans le programme. Cette propriété, appelée **transparence référentielle**, est importante car elle simplifie grandement la compréhension des programmes.

1.1.2 Types et construction de données

La construction de données complexes a été considérée très tôt dans le développement des langages fonctionnels puisque c'est LISP, créé en 1958 par John McCarthy, qui a proposé le premier une construction du langage dédiée à la manipulation de listes (LISP signifie *LISt Processing*), d'arbres, etc. Il s'agissait de faciliter le développement d'applications en intelligence artificielle et en calcul symbolique et LISP a proposé en même temps la gestion automatique de la mémoire. La construction de données se fait donc, depuis LISP, sans affectation ni manipulation de pointeurs dans tous les langages fonctionnels.

Notion de type

La notion de **type** n'est pas explicitement requise pour la construction de données complexes (à titre d'exemple LISP n'était pas typé), mais elle facilite grandement leur manipulation. En effet, un type détermine une catégorie de données. Dans les langages de la famille ML, la théorie assigne un type et un seul à une entité donnée. Cette définition de type est contraignante mais permet beaucoup de vérifications de cohérence détaillées ci-après. Un algorithme de **typage** sert à vérifier que les données sont manipulées conformément à leur type et que les fonctions sont appliquées à des arguments ayant le type requis, ce qui élimine des erreurs de codage ou d'exécution. La qualité de l'algorithme de typage peut être garantie par une preuve de correction (reposant sur la logique employée), démontrant que si le programme est bien typé, alors toute valeur retournée est du type attendu. Une telle preuve nécessite une définition formelle de la sémantique du langage. En général, de tels fondements théoriques sont mieux étudiés pour

les langages fonctionnels, ce qui donne plus d'assurance quant à la correction des programmes développés.

Propriétés du typage

Un langage attribuant un et un seul type à toute expression bien formée du langage est dit **fortement typé**. Par contraste, un langage est dit **faiblement typé** si une même expression peut avoir plusieurs types (par exemple, un entier considéré comme un booléen). La vérification du typage y est donc moins précise et le niveau de détection d'erreurs plus faible. Le typage peut être fait à la compilation et donc avant toute exécution du programme, auquel cas le langage est dit **statiquement typé**. Lorsque des vérifications de typage sont faites au chargement du code ou en cours d'exécution, le langage est dit **dynamiquement typé**. Les erreurs sont dans ce cas détectées beaucoup plus tard, ce qui diminue la robustesse du langage. Un langage statiquement et fortement typé apporte donc plus de garanties en terme de sécurité ou de sûreté de fonctionnement.

Inférence de type

Les langages de la famille ML sont statiquement et fortement typés et proposent de plus l'**inférence de types**. Cela signifie que l'algorithme de typage détermine le type des expressions sans requérir d'annotations de type par le programmeur. De plus, les types déduits par inférence peuvent être automatiquement ajoutés dans le source par le compilateur, à des fins de documentation par exemple. A titre d'exemple l'inférence du type de la fonction `let f (x, y) = x + y + 1` se déroule comme suit : à partir des types connus de la fonction `+` noté `+ : int * int -> int` et de la constante `1 : int`, l'inférence de type déduit automatiquement les types de `f : int * int -> int`, de `x : int` et `y : int`. La definition en C de la même fonction contient trois déclarations de type `int f (int x, int y) { return x+y+1; }`. Dans l'exemple précédent, le type est simple et facilement calculé mais ce mécanisme permet la déduction de types arbitrairement complexes.

Types et structures de données

Tous les langages fonctionnels typés proposent des **types atomiques** (entiers, booléens, flottants, etc.) et des **types fonctionnels** (`t1 -> t2` est le type

des fonctions dont le paramètre est de type `t1` et le résultat de type `t2`). Les langages fonctionnels offrent souvent des **types produit** correspondant aux couples et n-uplets, et des **types enregistrement**. Le programmeur dispose également de **types somme** (encore appelés **types union** ou **types variants**) permettant de différencier les formes possibles d'une valeur par l'emploi d'un constructeur adapté à chaque forme. Par exemple, le type `figure` est défini par :

```
type figure =
 | Carre of float
 | Rectangle of float * float
```

`Carre`, `Rectangle` sont appelés **constructeurs des valeurs** du type `figure`. Le seul moyen de créer une valeur de type `figure` est d'appliquer l'un de ces deux constructeurs à une expression du type indiqué dans la définition. Par exemple, `Carre (5.0)`, `Rectangle (3.0, 4.0)` sont des valeurs de type `figure`, alors que `Carre ("n")`, `25` et `Rectangle (2)` n'en sont pas.

Comme l'exemple ci-dessus l'indique, les types permettent de créer des structures de données complexes, sans aucune manipulation de pointeurs. De tels types peuvent être **récursifs**, c'est-à-dire que leur définition fait référence à elle-même, comme le type des expressions arithmétiques `arith` très simples (somme d'entiers et d'occurrences de la variable `X`).

```
type arith =
  | X
  | Cst of int
  | Add of arith * arith
```

Polymorphisme paramétrique

Certains langages fonctionnels, tels ceux de la famille ML, proposent un mécanisme de paramétrisation des types, appelé **polymorphisme paramétrique**. L'exemple suivant illustre cette notion. Une suite de valeurs de type `int` est soit vide et représentée par le constructeur `Vide`, soit construite grâce au constructeur `Node` à partir d'un entier et d'une suite.

```
type suite =
  | Vide
  | Node of int * suite
```

Le type `int` n'est pas déterminant pour cette définition. Il pourrait être remplacé par `bool`, `arith` ou tout autre type, la structure de liste serait identique.

Cette généricité ou polymorphisme peut être indiquée en paramétrant le type par une **variable de type** nommée ici `'element`, comme présenté dans l'extrait suivant :

```
type 'element suite =
  | Vide
  | Node of 'element * ('element suite)
```

ce qui signifie que, quel que soit le type effectif `t` remplaçant `'element`, une suite est soit vide (représentée par `Vide`) soit obtenue en appliquant le constructeur `Node` à un couple formé d'une valeur de type `t` et d'une suite de valeurs de type `t`. Ainsi, le type des suites d'entiers est `int suite`, et celui des suites d'expressions arithmétiques est `arith suite`.

Les types polymorphes favorisent la généricité des développements, les possibilités de réutilisation tout en permettant l'inférence de types. L'inférence non seulement simplifie la tâche du développeur, mais permet aussi de mesurer le degré de généricité du code produit.

Surcharge

Le **surcharge** est un mécanisme permettant de désigner par le même nom des éléments — en général des fonctions — dont les spécifications sont au moins partiellement similaires. Par exemple, l'identificateur + peut désigner, dans certains langages, à la fois l'addition des entiers et l'addition des flottants, la concaténation de deux chaînes de caractères ou la fusion de deux listes ... mais aussi n'importe quelle fonction pourvu qu'elle ait deux paramètres (et encore !). Même si cela peut être commode, ce mécanisme est difficilement compatible avec une notion de typage fort et statique et il est soit rejeté soit restreint par les langages fortement et statiquement typés.

Pour clore cette présentation des types, mentionnons qu'il existe des extensions de la notion de type qui regroupent la description d'une structure de données constituant le type lui-même, et les fonctions manipulant les valeurs de ce type. Ces extensions sont proposées sous la forme de types algébriques ou abstraits (décrits section 1.1.4), de modules (section 1.1.5) et de classes (section 1.2.3).

1.1.3 Construction de fonctions

Une fonction est définie par une expression fonctionnelle comportant un paramètre formel et un corps. Si elle n'est utilisée qu'une seule fois dans un pro-

gramme, cette expression peut ne pas être liée à un identificateur ; la fonction est alors dite **anonyme**.

Récursivité

Le corps d'une fonction est une expression qui peut utiliser des fonctions connues, ce qui correspond à la **composition** de fonctions. Les fonctions dites **récursives** contiennent dans leur corps un appel à elles-mêmes, soit directement (`fact (n) = ... n * fact (n - 1) ...`) soit indirectement (`pair (n) = ... impair (n - 1) ... and impair(n) = ... pair (n - 1) ...`). Les fonctions récursives ne peuvent pas être anonymes et selon les langages, leur caractère récursif doit être explicitement mentionné ou non. Par exemple, en OCaml, les fonctions récursives sont introduites par `let rec`.

Filtrage

La définition d'une fonction sur une structure de donnée est grandement facilitée par le mécanisme de **filtrage** (*pattern matching* en anglais), proposé par de nombreux langages fonctionnels typés. Le filtrage permet de décomposer une structure de données en sous-structures, à une profondeur choisie par le développeur. Un **filtre** se compose du préfixe d'une valeur structurée, les sous-structures absentes étant représentées soit par des paramètres, soit par `_` appelé "attrape-tout".

```
let perimetre figure = match figure with
  | Carre c -> 4.0 * c                    (* Filtre 1 *)
  | Rectangle (l, h) -> 2.0 * (l + h)     (* Filtre 2 *)
```

Dans la fonction `Perimetre` ci-dessus, le premier filtre `Carre (c)` est un préfixe de toutes les valeurs construites avec le constructeur `Carre`.

La superposition d'une valeur structurée et d'un filtre — ou **filtrage** — ne réussit que si le filtre est bien un préfixe de cette valeur : les paramètres du filtre sont alors associés aux sous-structures correspondantes. L'expression `Rectangle (4.0, 3.0)` est filtrée par le second filtre qui associe respectivement `4.0` et `3.0` aux paramètres `l` et `h`.

Une fonction peut donc être définie par cas, en utilisant des filtres sur ces arguments. Les tentatives de filtrage sont faites en suivant l'ordre des filtres dans le texte de la fonction, le premier filtrage réussi étant retenu pour calculer le résultat correspondant. Ce choix élimine toute possibilité d'ambiguïté.

Le filtrage est dit **exhaustif** si toute valeur du type de l'argument est filtrée par au moins un filtre figurant dans la définition. Les compilateurs détectent en général les filtrages non exhaustifs (donc incomplets) et émettent alors un avertissement. Ils mentionnent également les filtres qui ne seront jamais utilisés, car cela révèle souvent une erreur de codage.

La fonction `bad_ev` ci-dessous calcule récursivement la valeur d'une expression arithmétique `arith`. Son filtrage est incomplet puisque le cas où la valeur est de la forme `X` a été oublié. La détection de cet oubli par le compilateur évite une erreur lors de l'évaluation de `bad_ev (X)`.

```
let rec bad_ev (exp) = match exp with
  | Cst (n) -> n                              (* Filtre 1 *)
  | Add (Cst(0), b) -> bad_ev (b)             (* Filtre 2 *)
  | Add (a, b) -> bad_ev (a) + bad_ev (b)     (* Filtre 3 *)
  | Add (a, Cst(0)) -> bad_ev (a)             (* Filtre 4 *)
```

La définition de `bad_ev` contient une deuxième erreur puisque le quatrième filtre `Add (a, Cst (0))` est un cas particulier du troisième `Add (a, b)` et ne peut donc jamais être utilisé. Cette erreur est en général aussi signalée par les compilateurs.

La définition de la fonction `eval` suivante est complète et attribue la valeur `vx` passée en paramètre à la variable `X`. Le compilateur calcule son type `eval : arith * int -> int` par inférence.

```
let rec eval (exp, vx) = match exp with
  | X -> vx                                   (* Filtre 1 *)
  | Cst (n) -> n                              (* Filtre 2 *)
  | Add (Cst (0), b) -> eval (b, vx)          (* Filtre 3 *)
  | Add (a, Cst (0)) -> eval (a, vx)          (* Filtre 4 *)
  | Add (a, b) ->                             (* Filtre 5 *)
    eval (a, vx) + eval (b, vx)
```

Le calcul de `eval (Add (Cst (25), Cst (0)), 10)` provoque le filtrage de son argument `Add (Cst (25), Cst (0))`. Les trois premiers filtres sont essayés sans succès. Le filtrage par le quatrième filtre réussit et lie le paramètre `a` à `Cst (25)`. Le calcul se poursuit par l'appel récursif `eval (Cst (25))` qui retourne 25 après filtrage par le deuxième filtre.

Fonctions d'ordre supérieur

Une fonction est dite d'**ordre supérieur** si elle prend en paramètre une autre fonction ou si son résultat est une fonction. Par exemple, la fonction `map` (voir extrait de code ci-dessous) applique une fonction `f` à chaque élément d'une suite `seq` et reconstruit une suite avec les résultats de ces applications. Son type `map : ('t1 -> 't2) * ('t1 suite) -> ('t2 suite)` peut être inféré. Le type fonctionnel `('t1 -> 't2)` de son premier argument contient deux variables de type `'t1` et `'t2`. Ce code est générique : le paramètre `f` peut être remplacé a priori par n'importe quelle fonction. Mais la cohérence est garantie : si `f` a pour type `a -> b`, alors le paramètre effectif remplaçant `seq` doit être de type `a suite`, et le résultat est de type `b suite`.

```
let rec map (f,seq) = match seq with
 | Vide -> Vide
 | Node (elt, reste) -> Node (f (elt), map (f, reste))
```

Soit la fonction (anonyme) `function x -> (x < 6)` qui teste si son argument est plus petit que 6, l'évaluation de `map ((function x -> x < 6), Node (5, Node (7, Vide)))` retourne la suite `Node (False, Node (True, Vide))` [3].

Les fonctions d'ordre supérieur fournissent un puissant outil d'abstraction. Couplées à l'inférence de types, elles permettent d'exploiter la généricité du développement tout en maintenant la simplicité de l'écriture.

1.1.4 Types abstraits

Le choix de la forme des données repose souvent sur un invariant. Par exemple, on peut représenter un intervalle d'entiers par un couple `(a, b)` qui respecte l'invariant `a < b`. Un simple algorithme de typage assure seulement la cohérence de la manipulation des données conformément à leur type et ne permet pas toujours de vérifier le respect d'un invariant.

Pour aider au maintien des invariants, certains langages proposent une extension de la notion de type appelée **type abstrait**. Il s'agit d'un type associé à des fonctions de manipulation, le tout **encapsulé** dans une construction syntaxique dédiée. L'encapsulation peut être également réalisée via les notions de module ou de classe, présentées plus loin.

3. Les langages fonctionnels fournissent souvent de bons outils pour améliorer la lisibilité de valeurs structurées.

L'utilisateur d'un type abstrait ne connaît pas l'implantation des données du type et ne peut donc les créer ou les manipuler que par l'intermédiaire des fonctions encapsulées, ce qui peut être vérifié par un algorithme de typage **étendu** pour contrôler l'utilisation de ces types abstraits. Ce mécanisme permet donc un contrôle du respect des invariants de représentation des données.

Pour l'exemple des intervalles, le type abstrait pourrait contenire les déclarations suivantes :

```
type t = int * int
let intervalle (a, b) = if a < b then (a, b) else (b, a)
let borne_inf (a, b) = a
let borne_sup (a, b) = b
```

et exporter (permettre l'utilisation) des noms `intervalle`, `borne_inf` et `borne_sup` tout en cachant (encapsulant) la représentation par un couple d'entiers.

1.1.5 Modules

Un **module** est une structure syntaxique regroupant des déclarations et des définitions, en vue de leur exportation. Un élément d'un module peut être importé par tout autre programme, en suivant une convention de nommage fournie par le langage.

Les langages typés attribuent non seulement un type à chaque élément d'un module mais ils attribuent également un **type de module** au module lui-même qui regroupe les noms et types des éléments du module. Ils proposent de plus la notion d'**interface de module**, qui permet d'interdire l'exportation de certains éléments d'un module. Une interface de module est construite à partir d'un type de module en enlevant les noms et types des éléments que l'on ne souhaite pas exporter en ne fournissant que les noms que l'on veut abstraire. Définir une interface pour un module donné permet donc une forme d'encapsulation, facilement ajustable aux besoins.

La plupart des langages fonctionnels proposent des modules compilables séparément, ce qui permet de construire des bibliothèques logicielles. Dans ce cas, les exportations sont gérées uniquement par les interfaces, ce qui permet de remplacer une implantation de fonction par une autre de même type, sans que le programme importateur doive être recompilé, seule l'édition de liens devant être refaite.

1.2 Intégration de traits non fonctionnels

Les langages fonctionnels ont dès leur début intégré des traits impératifs, le plus souvent pour répondre à des problèmes d'efficacité. Ces problèmes ont disparu mais les traits impératifs sont restés. De même, la plupart des langages fonctionnels ont été étendus avec des traits objet, réputés faciliter les développements. Ces extensions sont décrites ci-après.

1.2.1 Langages fonctionnels et effets de bord

Sans aller jusqu'à des traits véritablement impératifs, les langages fonctionnels possèdent tous des primitives effectuant des **effets de bord**, ne serait-ce que pour recevoir et transmettre des données.

En particulier ils peuvent proposer des structures de données appelées **flots de données** pour représenter des données qui évoluent au cours de l'exécution. Ce sont des suites de données potentiellement infinies, qui se consomment ou se construisent au fur et à mesure de l'exécution. Un flot de données peut par exemple servir à représenter un flot d'informations provenant d'un canal d'entrée, l'état courant d'une machine, ou une suite de caractères qui peut être soumise à une analyse syntaxique. La lecture de la tête d'un flot de données efface cette tête et donne accès à la valeur suivante. L'écriture d'un élément dans un flot de données provoque l'émission de cet élément et la mise en attente d'un nouvel élément. La gestion de ces flots de données requiert donc un schéma spécifique de compilation dans les langages à évaluation stricte. Les fonctions sur les flots de données peuvent être définies par filtrage, ce qui facilite la manipulation de ces flots.

L'utilisation d'un flot de données, pour représenter les différentes valeurs prises par la donnée au cours de l'exécution, est une alternative à l'affectation. L'aspect purement fonctionnel du programme est conservé, tout en bénéficiant des indications fournies par les constructions syntaxiques des flots pour suivre l'évolution temporelle des données au cours de la relecture.

1.2.2 Traits impératifs

Les langages fonctionnels intégrant des traits impératifs sont parfois dits **impurs**.

Mutabilité

Les langages impurs proposent une notion de **variable mutable**, encore appelée **référence**. Il s'agit d'un identificateur lié à une adresse-mémoire dont le contenu peut être changé par une **affectation**. Ces langages offrent également des structures de donnée mutables plus complexes comme des tableaux dont les éléments peuvent être modifiés par une affectation. Les chaînes de caractères sont souvent des valeurs mutables, même si certains langages récents proposent des chaînes de caractères non-mutables dans leur noyau fonctionnel pur. De plus, il est souvent possible de contrôler finement la mutabilité de structures de donnée complexes, en limitant la mutabilité aux sous-structures qui la nécessitent. On peut définir par exemple des enregistrements dont seuls certains champs sont mutables. Cela permet de contrôler la manière dont sont utilisés ces traits impératifs.

Avec la notion de zone mémoire mutable, les langages fonctionnels proposent généralement les structures de contrôle classiques des langages impératifs telles que les boucles `for` et `while`.

Typage

Les systèmes de type sont étendus afin de typer aussi les éléments mutables et les structures de contrôle. L'algorithme de typage contrôle que l'affectation respecte le type de la structure mutable et sa manipulation. Par exemple, les opérations arithmétiques sur les adresses-mémoire sont en général interdites par le typage. L'inférence de types peut être étendue aux traits impératifs, avec quelques restrictions sur le polymorphisme paramétrique.

Exceptions

Les langages fonctionnels proposent souvent un mécanisme d'**exceptions**, un trait considéré comme impératif dans la mesure où la levée d'une exception et son rattrapage modifient le déroulement de l'exécution. Les systèmes de typage peuvent être étendus pour intégrer la notion d'exception.

L'usage de traits impératifs peut se justifier dans certains cas, par exemple lorsque le programme doit conserver une donnée qui évolue au cours de l'exécution. Cela évite de la passer en paramètre aux fonctions qui la manipulent et permet d'alléger le code. La mise à jour se fait alors par effet de bord réalisé par une affectation. Comme cela a déjà été dit dans l'introduction, l'affectation

peut conduire à des difficultés importantes de mise au point et de correction d'anomalies du code source, dues en partie au fait que la valeur associée à un identificateur évolue au cours de l'exécution. La relecture du code doit donc prendre en compte cette évolution, ce qui n'est pas toujours simple.

Il est recommandé aux utilisateurs des langages fonctionnels de limiter et de justifier l'usage de traits impératifs, en ajustant la portée des identificateurs mutables aux réels besoins, en ne rendant mutables que les sous-structures de données dont la mutabilité est jugée nécessaire, en contrôlant l'utilisation de la mutabilité par le typage et par des mécanismes d'abstraction tels que l'encapsulation dans un module ou dans un type abstrait.

1.2.3 Traits objet

Certains langages fonctionnels sont étendus avec une notion d'objet dont les détails varient d'un langage à un autre. Le mécanisme objet est le plus souvent donné sous la forme de **classes**. Tout comme un type abstrait ou un module, une classe regroupe un certain nombre d'éléments : des constantes nommées, des variables de classe, des variables mutables appelées parfois **variables d'instance** ou **attributs**, des fonctions appelées **méthodes**.

Une classe n'est pas en général directement exécutable mais sert à créer des **objets** par un mécanisme dit d'**instanciation**, qui construit la structure représentant l'objet par duplication des variables d'instance, et en général partage des constantes et méthodes entre tous les objets d'une classe. Toutes les méthodes possèdent un paramètre implicite, qui est l'objet sur lequel la méthode est **invoquée**, c'est-à-dire appliquée. Un programme objet procède donc par création et instanciation de classes puis invocations de méthodes sur les objets créés.

Les mécanismes de construction de classes proposent tous une forme d'**héritage**, qui permet d'étendre la définition d'une classe connue (dite super-classe) pour construire une nouvelle classe (dite sous-classe). L'héritage est dit **multiple** s'il est possible d'hériter de plusieurs classes en même temps et il est dit **simple** s'il n'est possible que d'hériter d'une seule classe. Dans ce dernier cas, les langages proposent généralement un mécanisme d'**interface de classe** qui permet de simuler l'héritage multiple. Les méthodes héritées peuvent être **redéfinies**, ce qui permet de spécialiser une méthode à la sous-classe héritière. Il est possible de restreindre la portée et la visibilité de certaines méthodes à l'aide de labels comme `private` et ainsi de les encapsuler.

Le typage peut être étendu aux traits objet, avec une perte partielle de l'inférence de types. Le premier rôle du typage est toujours de garantir que l'invocation

d'une méthode sur un objet donné est licite. Les différentes restrictions de visibilité sont en général prises en compte par tous les algorithmes de typage, cependant les systèmes de type proposés pour les traits objet peuvent différer profondément d'un langage à un autre.

Les traits objet facilitent l'utilisation des langages, au prix d'une complexification certaine de la syntaxe et surtout de la sémantique.

1.3 Compilation et exécution

L'interprétation et la compilation sont les deux techniques classiques permettant d'exécuter un programme source. Les premiers langages fonctionnels étaient interprétés et certains langages récents ont conservé ce choix, au moins optionnellement.

Les premières versions de LISP ne proposaient qu'un interpréteur du programme source comme mode d'exécution. Cette notion classique d'interprétation ne sera pas présentée ici car elle est surtout utilisée actuellement par des langages de script.

La majorité des langages fonctionnels sont maintenant compilés. Les compilateurs effectuent plusieurs passes de traduction, brièvement décrites ci-après.

1.3.1 Analyses syntaxique et sémantique d'un source

La première passe de compilation effectue la traduction de la syntaxe concrète en un arbre de syntaxe abstraite, qui est ensuite typé, si le langage est statiquement typé. L'arbre de syntaxe abstraite est alors traduit vers un code intermédiaire. Les informations de typage peuvent ou non y être conservées. Les compilateurs effectuent ensuite différentes analyses statiques telles que la vérification de l'exhaustivité du filtrage. Par ailleurs, ils peuvent **instrumenter le code** pour effectuer différentes vérifications à l'exécution, comme le non-débordement des tableaux.

1.3.2 Production de code

La seconde phase de compilation traduit le code intermédiaire vers du code cible, qui peut être du code source pour C, Java, du bytecode ou encore du code natif.

Le **code natif** (parfois appelé code assembleur ou code machine) est dédié à une architecture donnée et est directement exécutable sur cette architecture, mais seulement sur cette architecture. Le compilateur peut donc fortement optimiser le code en fonction des caractéristiques de l'architecture cible. Par ailleurs, certains compilateurs produisent un programme en C, qui est ensuite compilé en code natif. Cette option est parfois utilisée pour couvrir les architectures non ciblées par la production directe de code natif.

Les compilateurs peuvent également produire un **bytecode** à partir du code source. Le bytecode est un langage intermédiaire, proche de la machine mais non exécutable. Ce bytecode doit donc être exécuté par une **machine virtuelle** (ou interpréteur de bytecode).

Certains langages fonctionnels proposent de plus une compilation par une boucle interactive (Read-Eval-Print Loop (REPL)), qui lit une phrase du programme, la compile et l'exécute en imprimant les résultats. Cela permet l'écriture pas-à-pas d'un programme et la validation immédiate du code produit. Chaque nouvelle phrase du programme peut ainsi être testée au plus vite. Les langages fonctionnels garantissent une (quasi)-absence de différence entre la compilation pas-à-pas et la compilation en bloc.

1.3.3 Gestion de la mémoire

Depuis la première version de LISP, les langages fonctionnels proposent la gestion automatique de la mémoire décrite ci-après.

- L'allocation est organisée et optimisée par le compilateur qui déduit du typage l'espace à allouer pour chaque valeur et le partage possible des données en mémoire.
- La désallocation est prise en charge par un programme spécial appelé **ramasse-miettes** ou *garbage collector* en anglais (GC). Programme utilisateur et GC sont exécutés concurremment, le GC désallouant dynamiquement les données qui ne sont plus accessibles.
- Aucune primitive d'allocation et de désallocation mémoire n'est en général fournie (pas de `malloc`/`free` à la C ni `delete` à la C++). En revanche, le GC peut être déclenché manuellement si besoin.

L'exemple suivant permet de comparer la programmation de la concaténation d'un entier en tête d'une suite d'entiers en C et dans un langage fonctionnel typé avec filtrage et GC.

```
type suite =
```

```
  | Vide
  | Node of int * suite
let concatenation (l, v) = Node(v, l)
```

Aucune indication explicite d'allocation ou de désallocation n'est nécessaire, le compilateur déduit de la définition du type `suite` la méthode d'allocation pour une valeur de ce type. On obtient alors un code plus bref et lisible que dans les langages impératifs classiques.

La manière traditionnelle de coder ce même exemple en C est de manipuler une structure chaînée. En C, le programmeur doit explicitement allouer l'espace mémoire pour chaque nouvel élément de la suite (`malloc`) et gère lui-même la désallocation (`free`).

```
struct suite {
  struct suite *node ;
  int valeur ;
};

struct suite *concatenation(struct suite *l, int v){
  struct suite *res =
    (struct suite *) malloc (sizeof (struct suite)) ;
  res->valeur = v ;
  res->node = l ;
  return res ;
}

void desalloue_suite (struct suite *l) {
  while (l) {
    struct suite *node = l->node;
    free (l);
    l = node;
  }
}
```

1.3.4 Parallélisme et concurrence

Certains langages fonctionnels permettent la gestion de **processus légers** (threads) et de processus classiques (i.e. lourds). Les threads peuvent être directement ceux du système d'exploitation sous-jacent ou disposer de constructions

spécifiques, qui renforcent leur intégration au langage (typage par exemple). Le traitement des processus par les langages fonctionnels ne diffère pas fondamentalement de celui offert par les systèmes d'exploitation. Cependant, l'approche fonctionnelle apporte des garanties particulières. En effet, l'absence d'effets de bord permet d'éviter des problèmes de cohérence de la mémoire globale (en anglais, *race conditions*).

1.4 Conclusion

Les langages fonctionnels apportent un grand nombre d'automatisations qui tout à la fois facilitent la programmation et vérifient automatiquement le comportement du programme lors du développement bien avant la phase de test. Le typage et l'analyse statique du filtrage permettent de détecter des erreurs et d'identifier des cas oubliés. La gestion automatique de la mémoire libère le programmeur de cette tâche difficile sujette à de nombreuses erreurs. Toutes ces automatisations ne dépossèdent toutefois pas le programmeur du contrôle sur la manière dont les calculs sont effectués. Les programmes écrits dans un langage fonctionnel sont souvent plus lisibles et restent plus proches de leur spécification.

Les programmeurs habitués à développer dans un langage impératif comme C ou orienté objet comme Java peuvent cependant rencontrer quelque difficulté à la prise en main d'un langage fonctionnel. Par exemple, manipuler des arbres binaires sans aucune manipulation de pointeurs peut demander une adaptation des techniques de codage.

Les langages fonctionnels sont souvent étendus par des traits impératifs et objet, ce qui peut apporter une facilité pour l'écriture de certains programmes. Cependant la manipulation des traits impératifs augmente fortement la difficulté de compréhension d'un programme et est souvent source d'erreurs. Les traits objet quant à eux facilitent le développement de logiciels et la réutilisation de code, mais la relecture et l'analyse de code source utilisant de tels traits peut devenir beaucoup plus complexe. Les utiliser nécessite de les maîtriser finement. Les systèmes de modules sont souvent beaucoup plus simples à maîtriser et offrent une solution efficace pour l'encapsulation.

Bibliographie

[ANA-SÉCU, 2011] Analyse des langages OCaml, Scala et F#. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L2.2.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[Cousineau et Mauny, 1995] Cousineau, G. et Mauny, M. (1995). *Approche fonctionnelle de la programmation*. Dunod.

[Felleisen et Friedman, 1997] Felleisen, M. et Friedman, D. P. (1997). *The Little MLer*. The MIT Press.

[Harrop, 2007] Harrop, J. (2007). Ray tracer language comparison. Site Web de la Flying Frog Consultancy Ltd. `http://www.ffconsultancy.com/languages/ray_tracer/` (dernière visite le 15 décembre 2010).

[Marceau, 2009] Marceau, G. (2009). The speed, size and dependability of programming languages. Blog de Guillaume Marceau. `http://blog.gmarceau.qc.ca/2009/05/speed-size-and-dependability-of.html` (dernière visite le 15 décembre 2010).

[Paulson, 1996] Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge University Press, 2e édition.

[Pierce, 2002] Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Chapitre 2

Description de quelques langages fonctionnels

L'objectif de ce chapitre est de donner un aperçu de sept langages fonctionnels actuellement utilisés : chaque langage est décrit succinctement par une fiche.

Les langages choisis sont les six langages généralistes OCaml, F$^{\#}$, Scala, Haskell, Scheme, CLISP ainsi que le langage Erlang utilisé dans les télécommunications. Ce choix n'implique aucun jugement de valeur sur ces langages ni sur ceux qui ne sont pas présentés. Il aurait été possible de décrire d'autres langages fonctionnels généralistes comme SML, Clean Clojure, Bigloo, des langages fonctionnels synchrones "flot de données" comme Lustre/SCADE, Lucid Synchone, des langages fonctionnels pour la concurrence asynchrone comme JoCaml ou pour la programmation réactive comme ReactiveML.

2.1 OCaml

Le langage OCaml[1] est la version courante de Caml, langage de la famille ML distribué dès 1987 par l'Institut National de Recherche en Informatique et en Automatique (INRIA) en *open source*. L'équipe *Gallium* de l'*INRIA* est en charge de l'évolution et de la distribution de OCaml, qui contient un compilateur (vers du code natif et du bytecode), les outils associés et la bibliothèque standard.

La fiche qui suit fait référence à la version 3.12.0 de OCaml et à sa distribution standard, datant d'août 2010.

2.1.1 Principaux traits du langage

OCaml est un langage fonctionnel à évaluation stricte, possédant tous les outils avancés de la fonctionnalité (fonctions anonymes, ordre supérieur, filtrage). OCaml est fortement et statiquement typé, avec inférence des types et polymorphisme paramétrique. Le langage de types est très riche : produits, unions, types récursifs, flots de données avec évaluation paresseuse, types définis par l'utilisateur. La surcharge est interdite. Les fonctions peuvent être définies par filtrage de profondeur quelconque. Le compilateur vérifie l'exhaustivité du filtrage et la présence de filtres non utilisés.

OCaml possède aussi des traits impératifs : références et affectation, structures de données partiellement mutables, tableaux et chaînes de caractères mutables, structures de contrôle `for` et `while`. L'inférence de types prend en compte les traits impératifs, les références ne pouvant pas être polymorphes. OCaml offre un mécanisme de traitement des exceptions, permettant à l'utilisateur de définir ses propres exceptions, paramétrées ou non, et de les rattraper.

OCaml offre un système de modules définis par un type de module, une implémentation et un mécanisme d'interface. Un module est paramétrable par d'autres modules (foncteurs et foncteurs d'ordre supérieur). Le système de modules permet la compilation séparée.

OCaml propose un langage de classes et d'objets, les classes pouvant être paramétrées. Le type d'un objet est défini à partir des types des méthodes qui lui sont applicables et non par héritage. Le système de types intègre les traits fonctionnels, les classes, les modules, la mutabilité en conservant une forme un peu restreinte d'inférence de types.

1. `http://caml.inria.fr/`

2.1.2 Mécanismes d'exécution

Il existe une seule distribution du compilateur, faite par l'*INRIA*, sous licence libre [2].

OCaml peut être compilé vers du code natif, vers du bytecode ou exécuté en mode REPL. Le compilateur natif (`ocamlopt`) produit du code assembleur pour les architectures AMD64 (Opteron) sous Linux, MacOS X et MS Windows, IA32 (Pentium) sous Linux, FreeBSD, MacOS X et MS Windows, et PowerPC sous MacOS X. Le compilateur (`ocamlc`) produit du bytecode pour la machine virtuelle (`ocamlrun`) appelée système *run-time* de OCaml, exécutable sur toute architecture disposant d'un compilateur C ANSI sous un système d'exploitation POSIX ou Windows. Une version pas-à-pas (`ocaml`) du compilateur permet une exécution en mode REPL sur la machine virtuelle OCaml.

Les trois modes de compilation ne diffèrent que par l'étape de production de code. La phase commune de compilation gère le typage, la vérification de couverture du filtrage et met en place des vérifications à l'exécution (respect des bornes des tableaux ou des chaînes de caractères, détection de la division par 0). Le mécanisme de gestion de la mémoire est toujours intégré ; il est reconnu particulièrement performant et est repris par plusieurs langages fonctionnels récents.

Les trois modèles d'exécution sont également fortement compatibles : les exécutions d'un même code source ne diffèrent que par la vitesse d'exécution. Le modèle d'exécution natif est économe en mémoire et offre des performances du même ordre de grandeur que celles du langage C [Harrop, 2007, Marceau, 2009]. Mais OCaml offre beaucoup plus de garanties grâce au typage et à la gestion automatisée de la mémoire.

2.1.3 Utilisation du langage

Pérennité et portabilité

Un consortium regroupant des industriels et des centres de recherche, créé en 2002, fédère les efforts de conception et de développement du langage et de son environnement, en assurant une forte compatibilité ascendante des versions du langage.

2. Détails disponibles à la page `http://caml.inria.fr/ocaml/license.fr.html`.

Outillage, support

La distribution OCaml contient un mode pour l'éditeur Emacs, offrant indentation et coloration du code, interaction avec le mode REPL, etc. L'outil **`Tuareg`** est une version très évoluée de ce mode. Il existe des modules d'extension (*plugins*) pour Eclipse et Netbeans. L'outil **`ocamldoc`**, fourni avec la distribution, est un générateur automatique de documentation de sources OCaml. La distribution contient également un lexeur **`ocamllex`**, un générateur de parseurs **`ocamlyacc`** et un outil de pré-processing très puissant **`camlp4`**.

La distribution inclut de plus un débogueur puissant (**`ocamldebug`**) qui s'interface avec le mode Emacs, un gestionnaire de bibliothèques **`Findlib`** et des outils d'observation du code.

Le document [OUTILS-OCAML, 2011] contient une description détaillée des principaux outils développés pour OCaml.

Écosystème

La distribution OCaml inclut une bibliothèque standard fournissant des types et des fonctions pour la manipulation de structures de données, pour les interfaces système, pour la programmation concurrente par *Threads*, pour l'évaluation paresseuse et pour le chargement dynamique.

Il existe de nombreuses bibliothèques et outils hors distribution qui sont généralement référencés sur le site officiel via la Caml Hump. Parmi ceux-ci *GODI* permet l'installation automatique de bibliothèques, et *ExtLib* et *Batteries Included* fournissent des distributions de bibliothèques.

Un mécanisme d'interfaçage avec C permet l'appel à du code C depuis un programme OCaml.

La communauté des utilisateurs de OCaml est internationale, active et variée (chercheurs, enseignants, ingénieurs). Elle fournit un support très réactif aux développeurs et elle propose de nombreux outils et langages dérivés.

OCaml est utilisé dans de nombreux domaines, à la fois pour développer des outils et environnements dédiés à une famille d'applications et pour développer les applications elles-mêmes. Sont écrits en OCaml les langages Jocaml (calcul distribué), LucidSynchrone (synchronisme), OCaml, Focalize, les prouveurs Coq, Zenon, les analyseurs ASTRÉE, Frama-C, SLAM, le logiciel pair-à-pair MIdonkey. On peut citer également les outils pour le Web 2.0 développés par Ocsigen, le langage de modélisation de contrats financiers MLFi développé par la société LexiFi, les outils pour l'arbitrage boursier de la société Jane Street Capital, le

compilateur Scade développé par Esterel Technologies (certifié SIL4), le système ControlBuild de TNI-Software (certifié SIL2), etc.

Complexité

La programmation en OCaml demande simplement une bonne formation à la programmation. Notons que l'utilisation de la boucle REPL fournit une aide importante au développeur. La connaissance du noyau du langage (fonctionnalité, modularité et éventuellement traits objet simples et valeurs mutables) est très souvent largement suffisante au développement d'applications conséquentes.

OCaml et sa version allégée Caml-Light étant beaucoup utilisés dans l'enseignement, de nombreux ouvrages et notes de cours leur sont dédiés. Il n'existe en revanche que peu de livres sur OCaml spécifiquement dédiés au développement d'applications, mais [Weis et Leroy, 1999, Chailloux *et al.*, 2000] sont disponibles en ligne.

Bibliographie

[Accart Hardin et Donzeau-Gouge Viguié, 1997] Accart Hardin, T. et Donzeau-Gouge Viguié, V. (1997). *Concepts et outils de programmation*. Dunod.

[Chailloux *et al.*, 2000] Chailloux, E., Manoury, P. et Pagano, B. (2000). *Développement d'applications avec Objective Caml*. O'Reilly. Version originale française (`http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/`) et traduction anglaise (`http://caml.inria.fr/pub/docs/oreilly-book/`) disponibles en ligne et dans certaines distributions Linux.

[Dubois et Ménissier Morain, 2004] Dubois, C. et Ménissier Morain, V. (2004). *Apprentissage de la programmation avec OCaml*. Hermes Science Publications.

[Harrop, 2007] Harrop, J. (2007). Ray tracer language comparison. Site Web de la Flying Frog Consultancy Ltd. `http://www.ffconsultancy.com/languages/ray_tracer/` (dernière visite le 15 décembre 2010).

[Harrop, 2005] Harrop, J. D. (2005). *OCaml for Scientists*. Flying Frog Consultancy Ltd.

[Leroy *et al.*, 2010] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. et Vouillon, J. (2010). The Objective Caml system release

3.12 – documentation and user's manual. INRIA. Disponible en ligne `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[Marceau, 2009] MARCEAU, G. (2009). The speed, size and dependability of programming languages. Blog de Guillaume MARCEAU. `http://blog.gmarceau.qc.ca/2009/05/speed-size-and-dependability-of.html` (dernière visite le 15 décembre 2010).

[Narbel, 2005] NARBEL, P. (2005). *Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml*. Vuibert.

[OUTILS-OCAML, 2011] Outils associés au langage OCaml. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L3.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[Weis et Leroy, 1999] WEIS, P. et LEROY, X. (1999). *Le langage Caml*. Dunod. Version française disponible en ligne `http://caml.inria.fr/pub/distrib/books/manuel-cl.pdf`.

2.2 F#

Le langage F# a été créé en 2002 par Don Syme de *Microsoft Research* afin d'offrir un langage fonctionnel à la OCaml pour la plateforme de développement d'applications .NET. Il est maintenant développé par la *Microsoft Developer Division* et distribué avec la plateforme .NET. Depuis novembre 2010 les codes sources du compilateur et de la bibliothèque standard sont *open source*. La fiche fait référence à la version 2.0 d'avril 2010 du langage et de son compilateur.

2.2.1 Principaux traits du langage

F# est grandement inspiré de OCaml dont il reprend une partie de la syntaxe : F# peut être considéré comme un « OCaml pour .NET ». F# est d'abord un langage fonctionnel à évaluation stricte, possédant tous les outils avancés de la fonctionnalité (fonctions anonymes, ordre supérieur) et une notion étendue de filtrage par des filtres dits actifs, permettant de cataloguer des données selon certaines propriétés. Son noyau fontionnel est identique à celui de OCaml, à quelques différences minimes près.

F# est fortement et statiquement typé. Son langage de types est riche et permet de définir de nouveaux types : produits, unions, types récursifs, flots de données, chaînes de caractères non mutables. Il permet la gestion d'unités de mesure et propose le polymorphisme paramétrique. Les types sont inférés.

F# est étendu par des traits impératifs (variables mutables, chaînes de caractères et tableaux mutables, structures partiellement mutables) qui sont un peu plus contrôlés que ceux de OCaml. Il possède des exceptions analogues à celles de OCaml.

F# offre un système de modules analogue à celui de OCaml mais ne propose pas de foncteurs de module.

En revanche, F# offre des traits objets différents de ceux de OCaml. Ceux-ci sont directement issus du langage de spécification de .NET appelé Common Language Infrastructure (.NET CLI). Les principaux traits objet sont l'héritage simple d'implémentations, la simulation de l'héritage multiple à l'aide d'interfaces de classe et la réflexion (ou introspection) qui permet d'obtenir des informations sur les valeurs d'un type au cours de l'exécution.

2.2.2 Mécanismes d'exécution

Le compilateur F# produit du bytecode Common Intermediate Language (.NET CIL), qui est éxécutable par la machine virtuelle, Common Language Runtime (.NET CLR), de la plateforme .NET. Le langage .NET CIL est typé statiquement par un système de types *génériques*, le Common Type System (.NET CTS), qui garantit une exécution sans dépassement de tampon (*buffer overflow*). La gestion de la mémoire et des *Threads* est à la charge de la machine virtuelle. Le code F# peut faire appel à du bytecode .NET CIL et vice-versa.

F# peut également être utilisé en mode REPL.

2.2.3 Utilisation du langage

Pérennité et portabilité

F# fait partie de la plateforme .NET. F# est donc portable via les implémentations .NET (celles de *Microsoft* sous Windows et Mono de *Novell* sous Linux). Son développement par *Microsoft* garantit sa pérennité.

Outillage, support

L'outil principal de développement pour F# est Microsoft Visual Studio. Certaines des fonctionnalités de cet IDE sont disponibles pour F# : gestion de projets, aide contextuelle et débogueur.

Écosystème

La bibliothèque standard de F# (F# PowerPack) inclut des fonctions d'entrées/sorties, des types de données et les fonctions associées pour la réflexion, la gestion d'évènements et la programmation parallèle. Outre sa bibliothèque standard, le langage F# permet d'accéder aux fonctionnalités de la bibliothèque standard de la plateforme .NET (Base Class Library) et s'interface aisément aux bibliothèques développées pour .NET. Cela permet d'intégrer dans un programme F# des développements réalisés dans les autres langages de la plateforme et vice versa. On peut par exemple développer en F# les composantes algorithmiques complexes d'une application tout en développant un interface utilisateur en C#.

La proximité de F# avec OCaml et l'ouverture du code du compilateur et de la bibliothèque standard devraient permettre le développement de nombreux

échanges entre les communautés d'utilisateurs d'OCaml et de F# et le partage de développements.

Le langage est déjà utilisé pour la paramétrisation de jeux sur console (XBox Live TrueSkill) et par plusieurs sociétés comme Credit Suisse, Trafigura, E.ON Energy Trading pour des applications financières.

Complexité

F# est moins expressif que OCaml mais plus abordable en raison du nombre plus restreint de constructions du langage et de la présence de certaines extensions syntaxiques rendant son code plus lisible. C'est donc un langage relativement facile à maîtriser, mais qui requiert une connaissance réelle de .NET pour en exploiter tout le potentiel.

Le nombre d'ouvrages consacrés à la présentation théorique du langage et à sa mise en œuvre est déjà important. Ce langage est cependant encore peu enseigné.

Bibliographie

[Harrop, 2009] Harrop, J. (2009). *F# for Numerics*. Flying Frog Consultancy Ltd.

[Harrop, 2010] Harrop, J. (2010). *Visual F# 2010 for Technical Computing*. Flying Frog Consultancy Ltd.

[Neward *et al.*, 2010] Neward, T., Erickson, A., Crowell, T. et Minerich, R. (2010). *Professional F# 2.0*. Wrox.

[Petricek et Skeet, 2009] Petricek, T. et Skeet, J. (2009). *Functional Programming for the Real World : With Examples in F# and C#*. Manning Publications.

[Pickering, 2007] Pickering, R. (2007). *Foundations of F#*. Apress.

[Smith, 2009] Smith, C. (2009). *Programming F#*. O'Reilly Media.

[Syme, 2010] Syme, D. (2010). The F# 2.0 language specification. Microsoft Research and the Microsoft Developer Division. Disponible en ligne `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf`.

[Syme *et al.*, 2007] Syme, D., Granicz, A. et Cisternino, A. (2007). *Expert F#*. Apress.

2.3 Scala

Le langage Scala a été créé par Martin ODERSKY en 2003, pour étendre le langage Java en offrant à la fois le paradigme objet pour architecturer les programmes et le paradigme fonctionnel pour l'implémentation, tout en restant parfaitement interopérable avec Java, la Java Virtual Machine (JVM) et l'ensemble de son écosystème.

La distribution du langage est assurée par le groupe *LAMP* à l'École Polytechnique Fédérale de Lausanne (EPFL). La fiche qui suit fait référence à la version 2.8.1 de Scala [3] de novembre 2010.

2.3.1 Principaux traits du langage

Scala est fondé sur le paradigme objet intégral : pas de types primitifs, pas de champs de classe à portée globale. Les traits fonctionnels (fonctions récursives, fonctions d'ordre supérieur) sont tous implantés via les traits objet. Les fonctions sont en fait des instances de classes possédant un champ par paramètre formel de la fonction et un champ `apply` implicitement appelé à chaque application de la fonction. L'évaluation est stricte par défaut mais peut être rendue paresseuse.

Les classes Scala peuvent posséder des champs mutables (`var`) et des champs non mutables évalués soit à la création de l'objet (`val`), soit au premier appel d'une méthode de l'objet (`lazy val`), soit à chaque appel d'une méthode (`def`). La visibilité des champs peut être spécifiée par des labels (`public`, `private` et `protected`).

Les classes Scala peuvent être paramétrées. Scala offre un mécanisme d'héritage simple compatible avec celui de Java et propose également une extension de la notion d'interface de classe de Java, appelée *abstract trait* et souvent seulement *trait*. Ces traits Scala peuvent contenir des implantations, ils permettent de simuler l'héritage multiple. Ils fournissent un mécanisme d'abstraction en permettant la spécialisation de classes et d'objets.

Scala est un langage statiquement et fortement typé, les types étant construits à partir des noms et contenus des classes. En raison de la surcharge inhérente à l'utilisation de méthodes, l'inférence de types de Scala est locale à une définition de classe. Les types des fonctions récursives et des arguments des méthodes et des constructeurs de classes doivent être explicités. L'inférence locale fournit une bonne localisation des erreurs de sous-typage.

3. `http://www.scala-lang.org/`

2.3.2 Mécanismes d'exécution

Il existe un seul compilateur pour Scala (`scalac`) qui produit du *bytecode* Java, totalement compatible avec Java 1.5 et 1.6, exécutable sur toute implémentation de la machine virtuelle Java (JVM). Le code Scala peut faire appel à du code Java et vice versa. Du point de vue de la JVM, Scala n'ajoute qu'une Java ARchive (JAR) dédiée à Scala et sa bibliothèque standard.

Scala peut également être utilisé en mode REPL.

2.3.3 Utilisation du langage

Pérennité et portabilité

Martin Odersky a créé, en octobre 2010, la société *Scala-Solutions*[4] pour le développement et le support de Scala. La compatibilité binaire devrait être assurée entre les versions mineures depuis la version courante (version 2.8).

Outillage et support

Le langage Scala dispose de tous les outils associés à la JVM : débogueur, vérificateur de bytecode, serveurs d'applications. Il existe des *plugins* pour les Integrated Development Environment (IDE) généralistes et pour ceux spécifiques à Java. Netbeans, Eclipse, IntelliJ IDEA et Emacs (projet Ensime) offrent des outils de complétion contextuelle, de refactorisation de code, d'interface avec le débogueur et d'intégration des données de typage.

Écosystème

Outre son interfaçage transparent avec les bibliothèques Java, certains gros projets comme Scalaz, Lift Web Framework et AKKA Actor Kernel fournissent des bibliothèques dédiées pour Scala. La communauté Scala est très réactive (IRC, listes de diffusions). Elle est formée de chercheurs et d'ingénieurs développant principalement des applications Web et bases de données. Elle contribue activement au développement de ces bibliothèques, en particulier en *open source*.

Plusieurs sociétés utilisent Scala, par exemple FourSquare, Twitter, LinkedIn, Novell, EDF trading pour des applications client-serveur ou d'arbitrage financier.

4. `http://www.scalasolutions.com/`

Complexité

Développer en Scala demande de connaître la programmation objet à la Java, l'utilisation de traits fonctionnels se faisant dans ce cadre. La possibilité de choix du style de développement (fonctionnel ou objet) facilite l'accès au langage et l'écriture du programme. Mais ce choix peut rendre la programmation collaborative plus complexe. La syntaxe du langage, relativement flexible, permet un bon niveau d'expressivité et de lisibilité. Le système d'inférence de types peut décontenancer un programmeur Java habitué au typage explicite. Il existe une documentation abondante et un certain nombre d'ouvrages dédiés à la programmation en Scala dont [Wampler et Payne, 2009] et l'ouvrage de référence [Odersky *et al.*, 2008] disponibles en ligne.

Le langage Scala n'est enseigné que dans un nombre restreint d'universités [5] et il n'existe que peu de sociétés capables de fournir du service sur ce langage.

Bibliographie

[Odersky, 2010] ODERSKY, M. (2010). The Scala language specification version 2.8. Programming Methods Laboratory, EPFL. Disponible en ligne `http://www.scala-lang.org/node/198`.

[Odersky *et al.*, 2008] ODERSKY, M., SPOON, L. et VENNERS, B. (2008). *Programming in Scala*. Artima Inc.

[Wampler et Payne, 2009] WAMPLER, D. et PAYNE, A. (2009). *Programming Scala*. O'Reilly Media. Version anglaise disponible en ligne `http://programming-scala.labs.oreilly.com/`.

5. `http://www.scala-lang.org/node/203`

2.4 Haskell

Le langage Haskell a été élaboré par des chercheurs souhaitant disposer d'un langage commun, purement fonctionnel et à évaluation paresseuse. Il en existe plusieurs implémentations, le premier rapport standardisant le langage datant d'avril 1990. La plus complète et la plus utilisée actuellement est celle fournie, en *open source*, par l'Université de Glasgow, avec son compilateur le Glasgow Haskell Compiler (GHC).

La fiche fait référence à la version 7.0.1 du 16 novembre 2010 de GHC qui implémente le standard Haskell 2010.

Parmi les autres systèmes implémentant Haskell, on peut citer l'interpréteur `Hugs`, porté sur de nombreuses architectures, et le compilateur `nhc98` cherchant à optimiser l'espace mémoire des programmes.

2.4.1 Principaux traits du langage

Haskell est un langage fonctionnel pur, à évaluation paresseuse, possédant tous les outils avancés de la fonctionnalité (fonctions anonymes, ordre supérieur, filtrage). Il ne possède pas de traits impératifs, objet, etc. La transparence référentielle est donc garantie. Il reste possible de simuler des aspects impératifs ou objet en Haskell, les effets de bord nécessaires (entrées-sorties par exemple) devant être encapsulés dans une structure spéciale appelée monade.

Haskell est fortement et statiquement typé. Son langage de types permet de définir de nouveaux types (produits, unions, types récursifs, flots de données) et propose le polymorphisme paramétrique. Les types sont inférés. Les fonctions peuvent être définies par filtrage de profondeur quelconque. Le compilateur vérifie l'exhaustivité du filtrage et la présence de filtres non utilisés.

Haskell propose des *type classes* (à ne pas confondre avec les classes d'un langage objet) qui sont des sortes d'interfaces formées d'un paramètre de type et de déclarations de fonctions sur ce type. Ainsi le *type class* `Num` contient le paramètre de type `t` et la fonction `(+)` de type `t -> t -> t`. Tout type déclaré comme une instance d'un *type class* doit fournir une implantation des opérations de ce *type class*. La fonction `(+)` peut alors être utilisée pour n'importe quelle instance de `Num`. Les *type classes* proposent donc une forme de surcharge maîtrisée.

Haskell dispose de modules et d'interfaces de modules pour gérer l'encapsulation. Les modules peuvent être imbriqués pour constituer des hiérarchies de

modules (comme `Data.Array.MArray`). Haskell ne propose pas de foncteurs de modules. Le système de modules permet la compilation séparée.

La sémantique de Haskell est donnée en langue naturelle (anglais) dans les rapports Haskell 98 puis Haskell 2010 [Peyton Jones, 2003, Marlow, 2010], de manière claire, précise et détaillée.

La stratégie d'évaluation de GHC est paresseuse, mais une analyse statique (*strictness analysis*) recherche si certaines fonctions peuvent être considérées comme strictes, ce qui permet d'optimiser leur compilation.

2.4.2 Mécanismes d'exécution

Le compilateur GHC produit du code natif pour les systèmes d'exploitation MS Windows, Linux, Mac OS X, FreeBSD pour l'architecture x86 et Linux et FreeBSD pour x86-64. Il peut produire également du code C compilé par `gcc`. Enfin, il peut également produire du code intermédiaire pour la plate-forme LLVM [6] qui peut être ensuite compilé en code natif pour de nombreuses architectures (x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, etc). GHC peut également être utilisé en mode REPL (outil `GHCi`). L'outil `runghc` permet d'interpréter un programme en ligne de commande.

Il est possible d'interfacer du code écrit dans un autre langage via la *Foreign Function Interface* intégrée à Haskell 2010.

2.4.3 Utilisation du langage

Pérennité et portabilité

La pérennité de Haskell est assurée par la communauté Haskell, qui gère son évolution, via un processus appelé Haskell', dont l'objectif est de mettre à jour régulièrement le standard du langage.

Outillage et support

GHC est fourni avec un débogueur intégré au mode REPL `GHCi`, ainsi qu'un profileur de temps et d'espace mémoire. Ce dernier outil est essentiel, car l'évaluation paresseuse de Haskell fait qu'il est difficile pour un développeur d'évaluer la complexité d'une fonction en temps mais surtout en espace. Il permet aussi

6. `http://llvm.org/`

de repérer certains problèmes d'utilisation de la mémoire appelés abusivement [7] *memory leaks* et de les résoudre en forçant l'exécution de certains arguments.

Écosystème

Le rapport Haskell 2010 définit une bibliothèque standard et GHC fournit une bibliothèque bien plus conséquente [8]. Il existe un outil appelé `Cabal` qui permet de créer, d'empaqueter et d'installer des réalisations Haskell en gérant leurs dépendances. Un grand nombre de paquets créés par `Cabal` sont disponibles sur Hackage [9].

Haskell est utilisé pour construire des langages dédiés à un domaine (DSL) : des langages de spécification de matériel (comme ceux de la société Bluespec) ou des langages avec des primitives pour le parallélisme (par exemple Eden ou Parallel Haskell), etc. Haskell est utilisé pour développer des prouveurs comme Agda, des applications client-serveur comme le gestionnaire de version Darcs ou le serveur d'applications Happstack, des compilateurs comme Pugs pour Perl. Plusieurs sociétés, par exemple Standard Chartered, Starling Software, Deutsche Bank, etc. l'utilisent pour développer leurs propres applications dans différents domaines. Un groupe d'industriels (the Industrial Haskell Group) a été créé en 2009 pour assurer la prise en compte des besoins des utilisateurs commerciaux de Haskell.

Complexité

L'approche purement fonctionnelle de Haskell et son évaluation paresseuse en font un langage difficile à aborder pour quelqu'un habitué à d'autres paradigmes de programmation. Néanmoins, la plupart des utilisateurs ayant maîtrisé le langage reconnaissent son élégance. La présence d'un mode REPL peut aider son apprentissage.

Il existe de nombreux livres sur Haskell, et une documentation abondante est disponible sur le Web. Il existe un site consacré au langage [10] qui comporte notamment un Wiki assez complet. La communauté Haskell est active, interna-

7. Les fuites mémoire décrites par ce terme désignent une consommation importante de mémoire à un moment donné, et non la perte définitive d'une partie de l'espace mémoire d'un programme.

8. `http://www.haskell.org/ghc/docs/7.0-latest/html/libraries/index.html`

9. `http://hackage.haskell.org/packages/hackage.html`

10. `http://www.haskell.org/`

tionale et variée. Ce langage est enseigné dans de nombreuses universités, aussi bien comme premier que comme second langage.

Bibliographie

[Hudak *et al.*, 2007] HUDAK, P., HUGHES, J., PEYTON JONES, S. et WADLER, P. (2007). A history of Haskell : being lazy with class. *In Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, NY, USA. ACM.

[Hutton, 2007] HUTTON, G. (2007). *Programming in Haskell*. Cambridge University Press.

[Marlow, 2010] MARLOW, S., éditeur (2010). *Haskell 2010 Language Report*. `http://www.haskell.org/onlinereport/haskell2010`.

[O'Sullivan *et al.*, 2008] O'SULLIVAN, B., STEWART, D. et GOERZEN, J. (2008). *Real World Haskell*. O'Reilly Media. Version anglaise disponible en ligne `http://book.realworldhaskell.org/`.

[Peyton Jones, 2003] PEYTON JONES, S., éditeur (2003). *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England.

2.5 LISP

Le langage LISP, créé au début des années 50 par John McCarthy, est directement inspiré du λ-calcul de Alonzo Church. Il a depuis été implanté à de nombreuses reprises et a fait l'objet de quelques évolutions. Il est décrit ici à travers le dialecte CLISP (Common LISP), langage normalisé par l'ANSI [Pitman et Chapman, 1994] et distribué en *open source*. La fiche fait référence à la version 2.6.7 du compilateur GNU Common LISP (GCL) de CLISP, datant de août 2005 [11].

Il existe de nombreuses autres distributions de LISP, dont Emacs LISP (dans lequel est écrit l'éditeur GNU Emacs) et Allegro CLISP, qui est la version commerciale la plus connue [12]. Scheme présenté dans la section 2.6 de ce document peut également être vu comme un dialecte de LISP.

2.5.1 Principaux traits du langage

LISP est un langage fonctionnel, à l'origine non typé et interprété. La notion de base en LISP est la S-expression (S pour Symbolique), qui sert à représenter données, fonctions, applications de fonction de manière uniforme. Une S-expression peut être évaluée ou non à l'exécution, selon la sémantique que le programmeur lui attribue. Ainsi une S-expression représentant une liste n'a pas à être évaluée et cela sera indiqué en la faisant précéder par une apostrophe. Une S-expression représentant une application de fonction doit être évaluée et pourra l'être soit en évaluation stricte soit en évaluation paresseuse, au choix du programmeur. De plus, LISP offre un mécanisme de macros, permettant de créer de nouvelles S-expressions et donc de modifier, pendant la compilation, le programme à exécuter, permettant ainsi la réflexivité et l'introspection (en mode interprété).

CLISP permet de regrouper les données manipulées en sous-ensembles appelés *types*, ordonnés par inclusion : une entité peut appartenir à plusieurs types. Les opérations usuelles sur les ensembles sont proposées sur les types. Ainsi, CLISP propose des types atomiques (entiers, flottants, nombres complexes, etc. et atomes), des types structurés (chaînes de caractères, listes, tableaux, etc.). Le programmeur peut définir ses propres types. La vérification du typage est dynamique et repose sur l'inclusion (non obligatoire) d'annotations de typage dans les

11. `http://savannah.gnu.org/projects/gcl`
12. `http://www.franz.com/`

S-expressions par le programmeur. Ainsi, un programme peut contenir des parties typées et des parties non-typées, le compilateur n'effectuant que les vérifications de type déductibles de la syntaxe utilisée et des annotations.

CLISP propose également des traits impératifs : variables et affectation, boucles et exceptions, appelées conditions. CLISP possède une extension objet, appelée CLOS, à base de classes et d'héritage simple. Le système de types de CLISP est étendu pour prendre en compte les classes en tant que types.

2.5.2 Mécanismes d'exécution

CLISP est un langage qui peut être interprété (boucle REPL) ou compilé, sans qu'il y ait de différence sémantique entre les deux choix. En revanche, le mode interprété est beaucoup plus lent.

La gestion mémoire est automatisée avec un GC. L'interfaçage avec du code C est possible. Le compilateur GCL produit du code C qui, une fois compilé, offre des performances comparables à des programmes écrits en C ou en Fortran.

2.5.3 Utilisation du langage

Pérennité et portabilité

Une partie de CLISP est normé par l'ANSI [Pitman et Chapman, 1994], le reste est spécifié dans le livre de Guy Steele [Steele, 1990]. Le compilateur GCL est maintenu dans le cadre du projet GNU. Tout code source peut être compilé sur toute architecture disposant d'un compilateur C.

CLISP, malgré l'apparition de nouveaux venus dans le monde LISP, reste très utilisé. Certains dialectes de LISP apparus récemment comme Clojure permettent une exécution sur la JVM et sur .NET.

Outillage et support

Il existe un grand nombre d'IDE pour LISP comme l'éditeur Emacs. Il existe aussi un plugin pour Eclipse.

GCL fournit des outils pour le débogage comme requis par la norme CLISP, par exemple `trace` qui trace les appels d'une fonction, `step` qui évalue pas-à-pas une expression ou `inspect` qui permet interactivement d'inspecter n'importe quel objet CLISP.

Écosystème

Il existe énormément de codes et de librairies disponibles pour CLISP vu l'ancienneté du langage. L'utilisation de CLISP restant encore très importante, de nombreuses bibliothèques sont constamment proposées et maintenues. Les groupes d'utilisateurs de LISP sont toujours actifs.

LISP est très largement utilisé pour les applications de l'intelligence artificielle et du calcul non numérique. Sont écrits en CLISP le démonstrateur automatique ACL2, les prouveurs PVS, Nuprl, certains langages de la famille Prolog et plusieurs outils de calcul symbolique, tels Axiom, Maxima et Kenzo. CLISP est utilisé également dans de nombreuses applications client-serveur et pour la gestion de bases de données.

Complexité

LISP est enseigné dans beaucoup d'universités françaises et étrangères comme premier langage de programmation fonctionnelle car sa syntaxe est très simple. Il existe un grand nombre d'ouvrages dédiés à l'enseignement de la programmation ou à la programmation en LISP. Il est réputé faciliter le codage d'algorithmes compliqués, ce que démontre sa large utilisation en Intelligence Artificielle. Il faut toutefois noter que la sémantique d'un programme LISP peut être difficile à cerner car la souplesse du typage et les différentes manières d'effectuer un calcul (évaluations aux formes multiples, structures de contrôle impératives, appel de méthodes, etc.) complexifient la lecture du code source.

Bibliographie

[Graham, 1993] Graham, P. (1993). *On LISP : Advanced Techniques for Common LISP*. Prentice Hall. Disponible en ligne `http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf`.

[Pitman et Chapman, 1994] Pitman, K. et Chapman, K. (1994). Information Technology – Programming Language – Common Lisp. Rapport technique 226–1994, ANSI INCITS.

[Queinnec, 1997] Queinnec, C. (1997). *Les Langages LISP*. IIA. Interéditions.

[Seibel, 2005] Seibel, P. (2005). *Practical Common Lisp*. APress.

[Steele, 1990] Steele, G. L. (1990). *Common Lisp : The Language*. HP Technologies. Digital Press.

2.6 Scheme

Le langage Scheme est un langage de la famille LISP, conçu par Guy STEELE en 1975 avec l'objectif de définir un noyau de taille minimale, pouvant être facilement standardisé et permettant de définir des extensions de langage selon les besoins des applications. Le standard R6RS (Revised[6] Report on the Algorithmic Language Scheme) [Sperber *et al.*, 2010] a été ratifié en août 2007. Scheme possède plusieurs implémentations et de nombreux compilateurs (13 compilateurs et 29 interpréteurs recensés).

La présentation fait référence à la dernière version MIT/GNU Scheme 9.0.1, conforme à R6RS.

2.6.1 Principaux traits du langage

Comme pour tout langage de la famille LISP, la notion de base est la S-expression, structure syntaxique permettant de représenter données, fonctions, applications de fonction, et toute extension, de manière uniforme. Scheme est un langage fonctionnel à évaluation stricte. Il offre des fonctions anonymes, l'ordre supérieur.

Le compilateur sait détecter les récursions terminales et les compiler sans utiliser de pile d'exécution ; ce point est requis par le standard. Il travaille principalement en portée statique. Scheme propose un mécanisme de macros, permettant de créer de nouvelles S-expressions à la compilation et gérant les conflits de noms de variables (par exemple, les paramètres de fonction créés par des macros ne peuvent pas masquer un identificateur présent dans l'environnement, ils seront renommés).

Scheme propose un typage dynamique des valeurs mais les identificateurs ne sont pas typés, contrairement aux langages de la famille ML. Les types répartissent les valeurs en plusieurs catégories : nombres, caractères, symboles, vecteurs, enregistrements, tables de hachage, etc. qui peuvent elles-mêmes être partitionnées.

Comme pour tous les langages de la famille LISP, les variables sont mutables. Il existe un mécanisme d'exceptions, appelées `conditions`. Le MIT/GNU Scheme ne possède pas d'extension objet mais de nombreux dialectes Scheme proposent de telles extensions.

2.6.2 Mécanismes d'exécution

Le MIT/GNU Scheme possède un interpréteur (boucle REPL) et un compilateur qui produit du code natif pour les architectures i386 et x86-64 sous les systèmes GNU/Linux, MacOS X et Windows. Il existe aussi une version du compilateur produisant du C.

Le MIT/GNU Scheme fournit une interface avec les principaux systèmes d'exploitation et possède des primitives permettant la programmation concurrente asynchrone.

La gestion mémoire est automatisée avec un GC. L'interfaçage avec du code C est possible.

2.6.3 Utilisation du langage

Pérennité et portabilité

Le langage MIT/GNU Scheme est distribué sous la licence GPL. Il est distribué par la Free Software Foundation. Son standard R*RS est régulièrement mis à jour.

Outillage et support

MIT/GNU Scheme dispose d'un environnement de développement intégré, à la Emacs, appelé EDWIN, écrit dans ce langage. GNU Emacs offre un mode dédié à MIT/GNU Scheme. La boucle REPL est accessible depuis ces deux éditeurs.

MIT/GNU Scheme offre un outil de débogage et une librairie standard très complète.

Écosystème

MIT/GNU Scheme est d'abord un langage fortement utilisé dans les universités européennes et américaines pour l'enseignement de la programmation et l'algorithmique. Il est utilisé pour des applications d'intelligence artificielle, pour définir des langages de script comme celui du logiciel de manipulation d'images GIMP, pour construire des applications graphiques (par exemple, la composante GUILE de Gnome), pour des applications de calcul symbolique comme l'outil JACAL qui effectue des calculs algébriques ou différentiels.

Complexité

Suivant la volonté de ses créateurs de construire un langage de syntaxe simple, MIT/GNU Scheme est un langage qui s'apprend facilement. C'est pour cela qu'il est présent dans de nombreux établissements d'enseignement. Il existe donc beaucoup d'ouvrages le présentant, certains étant des cours complets de programmation comme [Abelson *et al.*, 1996]. Cela étant dit, la programmation d'applications complexes en Scheme est peu outillée mais le programmeur peut assez facilement définir ses propres outils. Il lui faut pour cela maîtriser finement les différentes possibilités d'évaluation offertes par le langage.

Bibliographie

[Abelson *et al.*, 1996] Abelson, H., Sussman, G. J. et Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press.

[Brygoo *et al.*, 2004] Brygoo, A., Durand, T., Pelletier, M., Queinnec, C. et Soria, M. (2004). *Initiation à l'informatique : Principes fondateurs de la programmation avec Scheme*. Sciences Sup. Dunod.

[Dybvig, 2009] Dybvig, R. K. (2009). *The Scheme Programming Language*. MIT Press.

[Queinnec, 2007] Queinnec, C. (2007). *Principes d'implantation de Scheme et Lisp*. Paracamplus.

[Sperber *et al.*, 2010] Sperber, M., Dybvig, R. K., Flatt, M., van Straaten, A., Findler, R. et Matthews, J., éditeurs (2010). *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press. Version disponible en ligne `http://www.r6rs.org/`.

2.7 Erlang

Erlang est un langage développé pour la programmation des équipements de télécommunications, par la société *Ericsson* depuis 1987. Sa première implémentation efficace a été diffusée en 1991, le langage a été porté dès 1992 sur de nombreuses plate-formes (Windows, MacOS, etc) et utilisé par *Ericsson* dans ses produits industriels. *Ericsson* le commercialise depuis 1993 et il est diffusé en *open source* depuis 1998.

Cette fiche fait référence à la version R14B02, publiée en mars 2011.

2.7.1 Principaux traits du langage

Le langage Erlang est dédié à la construction d'applications constituées de processus distribués (nommés également acteurs) communiquant par passage de messages. Cette forme de parallélisme est à la base de la structuration des programmes Erlang.

Les processus Erlang sont décrits à l'aide d'un langage fonctionnel à évaluation stricte, proposant fonctions anonymes et ordre supérieur. Ce langage est presque pur : une variable ne peut être affectée qu'une fois, au moment de sa définition. Les données mutables sont limitées à des fonctions de manipulation d'un *dictionnaire* local à chaque processus. Les seuls effets de bord couramment utilisés sont les envois de messages entre processus.

Erlang est faiblement dynamiquement typé. Son langage de types contient les types de base (nombres, atomes et identificateurs de processus) et les types structurés (tuples et listes).

Erlang est muni d'un système de modules simple. Il ne fournit aucun trait objet.

Le langage permet de gérer facilement la résistance aux pannes et la reprise sur erreur avec un mécanisme de moniteurs intégrés au langage. Un moniteur est un processus qui surveille le fonctionnement d'un autre processus et peut le relancer en cas de besoin.

Erlang propose un mécanisme de mise à jour du code à chaud : le programmeur peut modifier le code du programme sans interrompre son exécution. Cette possibilité est essentielle pour les réseaux de télécommunication et toutes les applications nécessitant une disponibilité ininterrompue. L'absence de traits impératifs dans le langage rend ce mécanisme plus contrôlable.

2.7.2 Mécanismes d'exécution

Erlang peut être compilé soit vers du code natif avec le compilateur (HiPE) soit vers du bytecode pour la machine virtuelle (BEAM), propre à Erlang. Le compilateur natif gère les architectures x86 en 32 et 64 bits, PowerPC, SPARC, et ARM. La machine virtuelle BEAM est disponible sur toute architecture disposant d'un compilateur gcc.

2.7.3 Utilisation du langage

Pérennité et portabilité

Erlang est produit par la société *Ericsson* qui l'utilise dans les systèmes de télécommunications qu'elle développe (notamment pour la téléphonie). Le compilateur HiPE a été développé à l'université d'Uppsala, et il est maintenant inclus dans la distribution diffusée par *Ericsson*. Le développement et l'utilisation par *Ericsson*, ainsi que la diffusion en *open source* donnent de bonnes garanties de pérennité au langage Erlang.

Outillage et support

La distribution Erlang inclut le compilateur et des outils de débogage et de gestion de code. Il existe aussi de nombreux outils externes : générateurs d'analyseurs lexicaux et syntaxiques, outil de couverture de tests, intégration dans de nombreux environnements de développement (Emacs, VIM, Eclipse, etc), outils d'analyse statique de code (`XREF` et `Dialyzer`).

Écosystème

La distribution Erlang inclut une bibliothèque conséquente, nommée Open Telecom Platform (OTP). Elle inclut notamment une base de données distribuée, des supports pour SNMP, CORBA et ASN.1, des bibliothèques d'interfaces graphiques (GUI) et d'outils pour les applications client-serveur et pour la résistance aux pannes. La distribution Erlang propose aussi un mécanisme d'interfaçage avec d'autres langages.

Il existe aussi de nombreuses bibliothèques développées par la communauté d'utilisateurs. Jungerl et Trapexit sont des projets cataloguant ce type de développements.

Erlang est utilisé par Ericsson mais aussi par d'autres sociétés comme Facebook pour le Facebook Chat System, Amazon pour la gestion de bases de données avec SimpleDB, Goldman Sachs pour de l'arbitrage financier. Il sert aussi à développer des applications pour le Web comme Twitterfall et des protocoles de communication comme RabbitMQ.

Complexité

Erlang est un langage sans concepts compliqués et qui inclut peu de constructions, ce qui le rend simple et facile à aborder. Il existe de nombreux tutoriels bien conçus.

Les traits inhabituels du langage sont la communication par passage de messages, la gestion des pannes et la mise à jour des programmes en cours de fonctionnement.

Bibliographie

[Armstrong, 2007] Armstrong, J. (2007). *Programming Erlang : Software for a Concurrent World*. The Pragmatic Programmers.

[Armstrong *et al.*, 1996] Armstrong, J., Virding, R., Wikström, C. et Williams, M. (1996). *Concurrent Programming in Erlang*. Prentice Hall.

[Cesarini et Thompson, 2009] Cesarini, F. et Thompson, S. (2009). *Erlang Programming – A Concurrent Approach to Software Development*. O'Reilly Media.

[Logan *et al.*, 2010] Logan, M., Merritt, E. et Carlsson, R. (2010). *Erlang and OTP in Action*. Manning Publications.

[Rémond, 2003] Rémond, M. (2003). *Erlang programmation*. Coming Next. Eyrolles.

Chapitre 3

Récapitulatif de l'étude

Ce chapitre présente une synthèse des langages étudiés dans ce document sous la forme d'un tableau (table 3.1). Les indications fournies dans ce tableau ne donnent qu'un aperçu synthétique du langage, voir la section dédiée dans ce document pour plus de détails. La sécurité n'est pas abordée dans ce document mais dans un autre document de l'étude LaFoSec [ANA-SÉCU, 2011] : pour les langages OCaml, F# et Scala le tableau référence ce document, pour les autres langages il signale que cet aspect est hors du périmètre de l'étude par la marque n. t..

Le tableau présente, pour chaque langage considéré, les six qualités recherchées pour un développement industriel. Le tableau est divisé en six sous-tableaux. Chaque sous-tableau associe au langage son niveau d'adéquation, relativement aux autres langages, représenté par les symboles ✚, ✚✚ ou ✚✚✚. Chacun des sous-tableaux recense les caractéristiques qui ont conduit à la détermination du niveau d'adéquation. Ces caractéristiques sont classées à l'aide de trois marques, listées par degré d'appropriation croissant : ○, ◐, ●. La marque n. a. signifie que le critère n'est pas applicable.

Les caractéristiques prises en compte pour chaque qualité traitée sont détaillées ci-après.

Facilité de prise en main. Les caractéristiques tiennent compte de l'existence de matériel d'enseignement dédié à ce langage (ouvrages pédagogiques, supports de cours, tutoriels, etc.), de l'intégration de ces langages dans des IDEs dédiés ou généralistes. Sont également considérées d'une part la simplicité de la syntaxe facilitant les débuts d'utilisation du langage, et d'autre part la simplicité de l'expression adressant plus les besoins d'écriture de code de taille importante.

Qualité de développement. Ce sous-tableau recense quelques points permettant de faciliter et d'améliorer la qualité des développements : lisibilité, modularité, maintenabilité et réutilisabilité du code source. L'inférence de types et la facilité de structuration de données complexes aide à la fois l'écriture du code et sa relecture. Traits modulaires et classes aident à architecturer le code. Conjointement avec le polymorphisme, ils facilitent également la réutilisabilité.

Les outils d'aide au développement associés ou intégrés au compilateur (débogueur, profileur, boucle d'interaction, etc.) ne figurent pas dans ce sous-tableau car tous les langages sont munis de tels outils.

Robustesse. Figurent dans ce sous-tableau quelques caractéristiques garantissant plus de robustesse vis-à-vis des erreurs d'exécution : typage statique fort, couverture du filtrage, analyses statiques formellement décrites et réalisées systématiquement au cours de la compilation. La richesse du noyau purement fonctionnel a également été considérée, ainsi que la possibilité d'identifier facilement l'utilisation de traits non fonctionnels dans un code source.

Efficacité. Ce sous-tableau porte sur les performances des applications développées dans le langage. Cette efficacité est estimée vis-à-vis d'applications semblables écrites en langage C sur la base de *benchmarks* présentés dans la bibliographie [Harrop, 2007, Marceau, 2009].

Interopérabilité. Ce sous-tableau couvre la problématique de l'interfaçage avec d'autres langages et leurs bibliothèques ainsi que la portabilité mesurée par le nombre d'architectures ou de machines virtuelles supportées par le compilateur.

Sécurité. Ce sous-tableau référence le document [ANA-SÉCU, 2011] qui traite de la sécurité des langages OCaml, F$^{\#}$ et Scala. Pour les langages non traités, la marque n. t. qui signifie "non traitée dans l'étude LaFoSec" apparaît.

Langage	OCaml	F#	Scala	Haskell	Scheme	CLISP	Erlang
Version	3.12.0	2.0	2.8.1	GHC 7.0.1	GNU 9.0.1	GCL 2.6.7	R14B02
Licence principale	QPL	Apache	à la BSD	BSD	GPL	GPL	MPL
Facilité de prise en main	+++	++	++	+	++	++	+++
Matériel d'enseignement	●	○	○	◐	●	●	◐
Intégration dans des IDEs	◐	●	●	◐	◐	◐	◐
Simplicité de la syntaxe	◐	◐	○	○	●	●	●
Simplicité de l'expression	●	●	◐	◐	○	○	◐
Qualité de développement	+++	+++	+++	+++	+	+	++
Inférence de type	●	●	◐	●	n. a.	n. a.	n. a.
Structuration de données complexes	●	●	●	●	○	○	◐
Modularité	●	●	●	●	◐	◐	◐
Polymorphisme	●	●	●	●	n. a.	n. a.	n. a.
Robustesse	+++	+++	+++	+++	+	+	++
Typage statique fort	●	●	●	●	○	○	○
Couverture du filtrage	●	●	●	●	○	○	○
Richesse et identification du noyau fonctionnel pur	●	●	◐	●	○	○	●
Efficacité	+++	++	++	+++	+	++	++
Production de code natif	●	JIT	JIT	●	●	via C	●
Interopérabilité	++	+++	+++	++	+	+	++
Interopérabilité avec C	●	●	●	●	◐	◐	●
Machines virtuelles	ZAM	.NET	JVM	LLVM	n. a.	n. a.	BEAM
Sécurité	[ANA-SÉCU, 2011]			n. t.	n. t.	n. t.	n. t.

TABLE 3.1 – Caractéristiques principales des langages

Acronymes

ANSI	American National Standards Institute
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information
BEAM	Bogdan/Björn's Erlang Abstract Machine
CLISP	Common LISP
DSL	Domain Specific Language
ECMA	European Computer Manufacturers Association
EPFL	École Polytechnique Fédérale de Lausanne
FSF	Free Software Foundation
GCL	GNU Common LISP
GHC	Glasgow Haskell Compiler
GPL	GNU General Public License
GUI	Graphical User Interface
HiPE	High Performance Erlang
IDE	Integrated Development Environment
INRIA	Institut National de Recherche en Informatique et en Automatique
JAR	Java ARchive
JIT	Compilation Just-In-Time
JVM	Java Virtual Machine
LISP	LISt Processing
LLVM	Low Level Virtual Machine
MPL	Mozilla Public License
.NET BCL	Base Class Library de .NET

.NET CIL	Common Intermediate Language de .NET
.NET CLI	Common Language Infrastructure de .NET (standard ECMA)
.NET CLR	Common Language Runtime de .NET
.NET CTS	Common Type System de .NET
OTP	Open Telecom Platform de Erlang
POSIX	Portable Operating System Interface pour UniX
QPL	Q Public License
REPL	Read-Eval-Print Loop
SCADE	Safety Critical Application Development Environment
SGDSN	Secrétariat Général de la Défense et de la Sécurité Nationale
SML	Standard ML
ZAM	ZINC Abstract Machine (machine virtuelle de OCaml)

Index