Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec)

Titre	**Analyse des langages OCaml, F# et Scala**
Identifiant	**Livrable L2.2.2**
Version	**5.0**
Date	**2011-09-20**
Pages	**281**
Approbation	**Christèle Faure, SafeRiver**
	Date: Signature:

Table des révisions

Version	Date	Description et changements	Parties modifiées
1.0	2011-02-16	Version initiale	Tout le document
2.0	2011-04-06	Version incorporant les commentaires de l'ANSSI	Tout le document
3.0	2011-07-16	Version du document correspondant au livrable L2.2.2	Tout le document
4.0	2011-08-22	Version du document incorporant les remarques de l'ANSSI des 16 et 19 août 2011	Tout le document
5.0	2011-09-20	Version du document répondant aux remarques de l'ANSSI	Tout le document

Résumé

Ce document, réalisé dans le cadre du projet LaFoSec propose une étude et une analyse des langages OCaml, F$^{\#}$ et Scala. Ces langages appartiennent à la famille des langages fonctionnels mais offrent également des traits impératifs, des traits objet et des traits modulaires. Leurs caractéristiques principales sont décrites et analysées du point de vue de la sécurité.

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 une analyse de sécurité des langages OCaml, F# et Scala.

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 pour déterminer 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. C'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 présentés 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

Les systèmes à logiciel prépondérant doivent de plus en plus être conçus de manière à résister à différentes attaques de sécurité. La sécurité d'une application en service dépend non seulement de l'application elle-même, du langage dans lequel elle est développée, mais aussi de son environnement d'exécution : système d'exploitation, architecture matérielle, etc. La sécurité d'un logiciel peut dépendre du (ou des) langage(s) de programmation dans lequel il a été développé. Dans ce document, seule la sécurité liée au langage de programmation est étudiée.

Ce document porte sur trois langages de programmation OCaml, F# et Scala. OCaml est un langage de la famille ML, qui propose à la fois une compilation native et une compilation vers du bytecode exécutable par sa propre machine virtuelle (la ZAM) sur la majorité des architectures. Scala est un langage présenté comme une extension fonctionnelle de Java. De même F# a été créé par *Microsoft* en s'inspirant fortement de OCaml pour doter sa plateforme .NET d'un langage fonctionnel. Ce sont deux langages compilés en bytecode, respectivement pour les plateformes Java Virtual Machine (JVM) et .NET.

Les différentes constructions des trois langages OCaml, F# et Scala ont été étudiées pour savoir si elles concourent (positivement ou non) à la sécurité des applications qui les utilisent. Il s'agit, plus précisément, de déterminer l'impact (positif ou négatif) sur la sécurité d'une application de l'utilisation de certaines constructions du langage, de certaines options ou modes de compilation.

L'amélioration ou la détérioration de ces propriétés est souvent due à l'utilisation conjointe de plusieurs constructions du langage, alliée ou non à celle de bibliothèques et d'options du compilateur.

L'influence de chaque trait fonctionnel, impératif, objet ou modulaire du langage sur la sécurité de l'application a donc été étudiée ainsi que les possibilités de modifier cette influence, par extension de l'application (à l'aide de l'héritage par exemple), par compilation, par modification de l'environnement d'exécution.

Chaque point étudié est décrit en terme d'intérêts, de dangers, et de recommandations permettant de faire apparaître les aspects principaux de son influence sur la sécurité. Les intérêts, dangers, et recommandations sont indexés par type (I, D, R), par langage (O, F, S) et par ordre d'apparition dans le document.

Le chapitre 1 est consacré à l'étude du langage OCaml. Cette étude est com-

plétée par le document [MODE-EX, 2011] pour tout ce qui concerne la compilation et l'exécution. Le chapitre 2 présente l'étude de F$^{\#}$ et le chapitre 3 porte sur Scala. Il faut noter que F$^{\#}$ est construit sur .NET et Scala sur Java. Les aspects de Scala se rapportant au modèle d'exécution sur JVM ne sont pas abordés dans ce document mais l'étude renvoie aux chapitres de [JavaSec exécution, 2009].

Bibliographie

[ETAT-LANG, 2011] État des lieux des langages fonctionnels. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L2.1.2, ANSSI (2011). Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[JavaSec exécution, 2009] Rapport d'étude sur les modèles d'exécution de Java. Rapport d'étude JavaSec JAVASEC_NTE_003, version : 1.2, ANSSI (2009). Étude menée par un consortium composé de SILICOM, AMOSSYS et de l'INRIA, dans le cadre formel d'un marché du SGDSN. Disponible en ligne à l'adresse `http://www.ssi.gouv.fr/IMG/pdf/JavaSec-Execution.pdf`.

[MODE-EX, 2011] Modèles d'exécution du 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.

Chapitre 1

Analyse du langage OCaml

Le langage OCaml [1] est un langage fonctionnel de la famille ML distribué par l'Institut National de Recherche en Informatique et en Automatique (INRIA) en *open source* depuis 1996. Il existe une seule distribution de OCaml, développée par l'INRIA, qui contient un compilateur (vers du code natif et du bytecode), les outils associés et la bibliothèque standard. L'étude porte sur la version 3.12.0 de OCaml et à sa distribution standard, datant d'août 2010. La section 1.1 étudie les traits fonctionnels et le typage. Les traits impératifs sont présentés dans la section 1.2. Le système de modules fait l'objet de la section 1.3. La section 1.4 est dédiée aux objets et classes de OCaml.

1.1 Traits fonctionnels et typage

Un programme purement fonctionnel OCaml est une suite de déclarations et d'expressions. Un **environnement** est un ensemble de liaisons d'identificateurs à leurs valeurs. Une expression est construite à partir d'applications de fonctions et est évaluée dans un environnement précisé par la sémantique.

1.1.1 Gestion des noms

Déclaration, portée, initialisation et allocation

Une **déclaration globale** est introduite par la construction `let`. Sa portée est toute la partie du texte source (donc du module courant cf. 1.3.1) suivant son

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

introduction. Si son introduction est faite par `sig`, alors elle est accessible par son nom qualifié après la fin du module (jusqu'à la fin du texte source contenant le module, à moins d'être masquée).

Une **déclaration locale**, introduite par la construction `... (let idf = exp1 in exp2) ...`, crée la liaison de `idf` à la valeur de `exp1`, le temps de l'évaluation de l'expression `exp2`.

Le compilateur OCaml détecte les variables locales non utilisées et émet un avertissement `Warning 26: unused variable`.

OCaml utilise la **portée statique** : la portion du texte source où un identificateur peut être utilisé est déterminée par la syntaxe du programme et donc aisée à appréhender par le programmeur. Elle est implémentée à l'aide de la notion de fermeture (introduite ci-dessous en 1.1.2).

La portée statique des identificateurs facilite l'analyse et la maintenance du texte source. Le compilateur garantit que les liaisons respectent la sémantique donnée dans son texte source.En particulier il garantit que toutes les lectures d'un identificateur non-mutable sont incluses dans sa portée.

> *Recommandation 1* (R-O-1) : *La portée de tout identificateur doit être choisie en accord avec son utilisation.*
> Afin de bénéficier pleinement des garanties apportées par la portée statique, il est souhaitable de n'utiliser des identificateurs globaux que si cela est pleinement justifié et de déterminer précisément quelle doit être la portée des identificateurs locaux. ■

L'évaluation d'une application (voir section 1.1.2) ne permet l'accès à aucune liaison de l'environnement d'appel de la fonction. Soit `x` un identificateur lié dans l'environnement d'appel d'une fonction `f`. Si `f` a été définie avant la liaison de `x`, si la valeur de `x` n'est pas passée en argument à `f`, alors l'évaluation de l'appel de `f` ne peut pas accéder à `x`. Le compilateur garantit le cloisonnement entre l'environnement d'appel d'une fonction et l'environnement d'évaluation de cet appel.

Dans la portée d'une variable non-mutable `x`, sont incluses toutes les lectures de cette variable (pas d'aliasing possible).

La valeur d'un identificateur `x` ne peut fuir que de l'une des manières suivantes, facilement détectables par lecture du code :

- Appel d'une fonction faisant fuir son paramètre avec `x` en argument ;
- Affichage de la valeur de `x` ;
- Affectation de la valeur de `x` dans un mutable (cf. section 1.2.2) ;
- Retour de `x` à la fin de la portée de `x` ;

- Levée d'une exception avec `x` comme argument ;
- Liaison d'un nouvel identificateur `y` à la valeur de `x` et fuite de `y` par l'une des méthodes de cette liste.

Si la valeur d'un identificateur ne fuit pas comme décrit précédemment, sa confidentialité est assurée au niveau langage par la portée statique.

> *Recommandation 2* (R-O-2) : *Vérifier l'absence de fuite des données confidentielles par relecture du code.*

> *Danger 3* (D-O-3) : *L'utilisation de mutables diminue la lisibilité apportée par la portée statique.*
> voir section 1.2.2 ■

> *Recommandation 4* (R-O-4) : *Limiter l'utilisation des variables mutables.*

La valeur d'initialisation est la valeur permanente d'un identificateur non mutable. Un identificateur mutable est lié à son adresse, qui n'est pas modifiable et c'est seulement la valeur à cette adresse qui peut évoluer. Une liaison n'est donc pas modifiable (mais elle peut être masquée par un autre identificateur qui a le même nom, voir plus loin). Elle est supprimée à la fin de sa portée. Le GC se charge de la **désallocation** des valeurs qui ne sont plus liées (cf. section 1.6).

OCaml ne propose ni `malloc` ni `free` pour manipuler la mémoire car ces manipulations sont réalisées par le GC. A partir d'une déclaration et des informations sur le type calculées par le typage (cf. section 1.1.3), le compilateur détermine la manière de stocker la valeur liée dans la déclaration.

> *Intérêt 5* (I-O-5) : *Le GC réalise automatiquement les manipulations mémoire.*
> Le GC réalise seul l'allocation et la désallocation de la mémoire. Ce mécanisme automatique garantit que l'allocation est correcte vis à vis du type de l'identificateur (en particulier sa taille) et permet un placement des données efficace. Le GC garantit aussi que la désallocation est réalisée une fois et une seule après la dernière utilisation. Il est ainsi impossible de désallouer prématurément une valeur. Ceci évite la manipulation de valeurs non initialisées ainsi que les erreurs et attaques par *double-free*. ■

Intérêt 6 (I-O-6) : *Déclaration, initialisation et allocation mémoire sont indissociables.*
Il n'est pas possible d'introduire une nouvelle déclaration sans donner la valeur de cet identificateur, y compris lorsqu'il est mutable. Cela évite les oublis d'initialisation. Il n'est pas possible de déclarer un identificateur (même mutable) sans que le GC lui associe un espace mémoire adéquat et lui attribue une valeur initiale. ■

Recommandation 7 (R-O-7) : *L'utilisation d'un type option permet de gérer l'absence de valeur plausible.*
Le type `option` (voir 1.1.3) permet de dénoter l'absence de valeur plausible en fournissant la valeur `None`, les valeurs plausibles étant, elles, dénotées par `Some (..)`. Cela permet par exemple d'éviter un choix arbitraire de valeur initiale, si aucune valeur ne convient, tout en conservant les avantages du typage. ■

Transparence référentielle

Le noyau purement fonctionnel de OCaml possède la propriété de **transparence référentielle** : toutes les occurrences dans le texte source d'une même expression ont la même valeur.

L'exemple suivant propose une fonction transparente référentiellement (`add_x` définie à la ligne 3) et une autre non transparente (`add_m` définie à la ligne 7). Des appels successifs retournent systématiquement le même résultat pour la première fonction (lignes 9–12) et des résultats différents pour la seconde (lignes 13–16).

```
1  let x = 1;;
2  val x : int = 1
3  let add_x (y) = x + y;;
4  val add_x : int -> int = <fun>
5  let m = ref 1;;
6  val m : int ref = {contents = 1}
7  let add_m (y) = m:= !m + y; !m;;
8  val add_m : int -> int = <fun>
9  add_x (25);;
10 - : int = 26
11 add_x (25);;
12 - : int = 26
13 add_m (25);;
```

```
14 - : int = 26
15 add_m (25);;
16 - : int = 51
```

Cette propriété facilite la division d'un texte source en unités de taille plus petite et donc la réutilisation de code. Elle procure surtout une aide importante à l'analyse et à la maintenance de texte source.

Intérêt 8 (I-O-8) : *La transparence référentielle facilite l'écriture, la relecture et la maintenance de texte source.*

Danger 9 (D-O-9) : *Les effets de bord cassent la transparence référentielle.*
Les effets de bord par entrées-sorties ou par affectation de valeurs mutables cassent la transparence référentielle. En particulier, l'exécution du programme est plus difficilement prédictible. ■

Recommandation 10 (R-O-10) : *Favoriser la transparence référentielle du texte source.*
Il est conseillé, en particulier pour les textes source sensibles, d'éviter ou au moins d'isoler les effets de bord et l'utilisation de données mutables. ■

Masquage

Une liaison peut être **masquée**, de manière temporaire ou permanente, par une nouvelle déclaration de l'identificateur lié. La liaison masquée n'est plus accessible dans l'environnement courant mais existe toujours en concordance avec sa portée. Les déclarations masquée et masquante sont totalement indépendantes et aucune concordance de type n'est demandée. Le masquage est à distinguer d'une redéfinition de méthode d'une classe : le type de la méthode redéfinie doit être conservé (voir 1.4.2). Le masquage n'a non plus aucun lien avec l'affectation. L'exemple suivant montre que la liaison `(x,1)` de la ligne 1, masquée par la déclaration globale de la liaison `(x,100)` de la ligne 7 reste accessible à travers la définition de `add_x` et que la liaison courante `(x,100)` est masquée par la déclaration locale de la liaison `(x,"y = ")` de la ligne 11 le temps de l'évaluation de `g`. Cette liaison `(x,"y = ")` est conservée dans la fermeture de `g` après sa disparition de l'environnement courant.

```
let x = 1;;
val x : int = 1
let add_x (y) = x + y;;
val add_x : int -> int = <fun>
add_x (25);;
- : int = 26
let x = 100;;
val x : int = 100
add_x (25);;
- : int = 26
let g (y) = let x =  "y = " in  x ^ string_of_int (y);;
val g : int -> string = <fun>
g(35);;
- : string = "y = 35"
x;;
- : int = 100
```

Le masquage peut parfois être utile pour éviter l'introduction d'un nouvel identificateur, mais il risque d'obscurcir le texte source et peut induire en erreur le développeur.

> *Recommandation 11* (R-O-11) : *Éviter tout masquage global et justifier tout masquage local.*
> En particulier, il faut interdire tout masquage (voir 1.1.4) même temporaire, de l'égalité et des opérateurs de comparaison prédéfinis. De manière générale, il est préférable d'éviter le masquage de fonctions importées depuis un module ouvert (cf. 1.3). ■

1.1.2 Étude de la fonctionnalité

OCaml propose la pleine fonctionnalité. Une fonction est une valeur de première classe : elle peut être passée en argument d'une autre fonction, retournée en tant que valeur d'une expression, etc. Une fonction est dite d'**ordre supérieur** si elle prend une fonction en argument ou si son application retourne une valeur fonctionnelle.

Fermetures et évaluation des applications

La valeur d'une expression fonctionnelle e, appelée **fermeture** ou **clôture**, est constituée du corps de la fonction et des liaisons des identificateurs libres dans ce corps présentes au moment de l'évaluation de e. L'application d'une fonction à un paramètre effectif se fait en étendant l'environnement présent dans la fermeture par la liaison du paramètre formel à la valeur du paramètre effectif. Cette méthode d'évaluation des applications est le fondement de la portée statique (voir 1.1.1).

> *Intérêt 12* (I-O-12) : *Le mécanisme de fermeture garantit la portée statique.*

Récursivité

Les **fonctions récursives** (introduites par les mots-clés `let rec`) sont des fonctions dont le corps contient un appel à elles-mêmes. Elle sont indispensables pour décrire les manipulations de données structurées récursives (listes, arbres, etc.) non-mutables. Leur définition, utilisant le filtrage, peut rester très proche de leur spécification.

Il n'y a pas de vérification de la terminaison des fonctions récursives. L'exécution d'une fonction récursive requiert l'utilisation d'une pile d'exécution. Lorsque l'exécution ne termine pas, la pile peut s'agrandir indéfiniment.

OCaml implémente un mécanisme qui permet de limiter la taille de la pile et évite que le programme monopolise la mémoire. Quand la pile atteint sa limite, on dit qu'il y a débordement de pile, la fonction récursive s'arrête par un des mécanismes suivants :

- En mode bytecode, où la limite de la pile est définie par OCaml, l'exception `Stack_overflow` est levée. Elle est rattrapable par le programme.
- En mode natif, où la limite de la pile est définie par l'OS, selon l'architecture, le programme lève l'exception `Stack_overflow` ou engendre un signal système (`Segmentation fault`).

Attention, un programmeur peut lancer manuellement l'exception `Stack_overflow`, ce qui peut induire un diagnostic erroné.

> *Danger 13* (D-O-13) : *Débordement de la pile d'exécution par appels récursifs.*

> *Recommandation 14* (R-O-14) : *S'assurer du non-débordement de la pile d'exécution.*

Il faut identifier les fonctions récursives en cherchant les occurrences du mot-clé `rec`, puis estimer le nombre d'appels récursifs engendrés par chaque application d'une fonction récursive. Et ceci particulièrement lors de la manipulation de grosses structures de données ou de calculs arithmétiques complexes. ■

Si l'évaluation du corps d'une fonction se termine sur un appel de fonction (une autre fonction ou elle-même), alors le compilateur évite d'empiler les instructions d'appel : l'appel est remplacé par un saut inconditionnel. Un tel appel est dit **terminal**. Ce type de mécanisme est particulièrement utile pour un langage fonctionnel qui fait usage de récursion.

Intérêt 15 (I-O-15) : *Les appels récursifs terminaux ne font pas grossir la pile.*

Danger 16 (D-O-16) : *Un appel, placé dans le contexte d'un rattrapage d'exceptions, n'est jamais terminal.*
Si on englobe un appel dans un `try with`, l'appel n'est plus considéré comme terminal car il faut dépiler le bloc de gestion d'exception. ■

Recommandation 17 (R-O-17) : *Vérifier le caractère terminal des appels récursifs.*
L'utilisation de l'option `-annot` permet d'observer les appels terminaux (voir [OUTILS-OCAML, 2011]) . ■

Une **valeur récursive** non-fonctionnelle est une valeur définie en terme d'elle-même. Elle ne peut être définie qu'avec un constructeur (comme `type seq = Vide | C of int * seq` utilisé avec `let rec seq_circ = C(1,seq_circ`), un enregistrement ou un n-uplet.

L'égalité structurelle `=`, qui s'appuie sur la structure des valeurs pour les comparer, peut boucler sur ces valeurs récursives. L'égalité physique `==` qui compare les adresses ne boucle pas.

Recommandation 18 (R-O-18) : *Prendre garde à la comparaison et au parcours de valeurs récursives.*

Fonctions anonymes

Une expression fonctionnelle peut être utilisée sans être nommée, on la désigne alors sous le vocable **fonction anonyme**.

> *Intérêt 19* (I-O-19) : *Laisser anonyme une fonction utilisée une seule fois allège le code.*

> *Intérêt 20* (I-O-20) : *L'utilisation d'une fonction anonyme permet son confinement.*
> Une fonction anonyme n'est utilisable que localement. ■

Malgré ces intérêts, l'utilisation de fonctions anonymes doit rester très limitée car la relecture du code est plus difficile en leur présence.

> *Recommandation 21* (R-O-21) : *Limiter l'usage de fonctions anonymes aux cas où elles sont strictement nécessaires.*

Application partielle

Une fonction OCaml peut retourner une fonction en résultat. Par exemple, la fonction définie par `let f x y = x + y` a le type `int -> int -> int` , ou de manière plus lisible, `int -> (int -> int)`. Cette fonction, dite à deux arguments, peut être **appliquée partiellement** à un argument : par exemple `(f 1)` est une fonction équivalente à `function y -> 1 + y`.

> *Intérêt 22* (I-O-22) : *Fonctions d'ordre supérieur et applications partielles permettent un partage optimal de texte source.*
> Utiliser une fonction d'ordre supérieur permet de bien transcrire la généricité d'une spécification, tout en permettant une spécialisation progressive du texte source par passage successif d'arguments. ■

Dans l'exemple suivant, la fonction `fold_left` possède trois arguments, `op` et `accu` d'une part, qui vont permettre de spécialiser son code et `l` d'autre part, qui est la liste sur laquelle le traitement spécialisé est appliqué. La fonction `encode_ABC` spécialise la fonction `fold_left` en une fonction qui concatène les encodages les entiers de la liste passée en argument et ajoute un préfixe `ABC`.

```
1 let rec fold_left op accu l =
2   match l with
```

```
3     | [] -> accu
4     | a::l -> fold_left op (op accu a) l;;
5   val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> ↵
        'a = <fun>
6   let encode i =
7     Digest.to_hex (Digest.string (string_of_int i));;
8   val encode : int -> string = <fun>
9   let encode_ABC =
10    fold_left (fun x y -> x ^ (encode y)) "ABC";;
11  val encode_ABC : int list -> string = <fun>
12  encode_ABC [1234;9876;5050];;
13  - : string = "ABC81dc9bdb52d04dc20036dbd8313ed055912e ↵
      79cd13c64069d91da65d62fbb78c8977ecbb8cb82d77fb091c7 ↵
      a7f186163"
```

Si une donnée est utilisée plusieurs fois dans l'exécution d'un traitement, il peut être intéressant de définir ce traitement par une fonction à plusieurs arguments et de nommer l'application partielle de ce traitement à cette donnée. La donnée sera alors encapsulée dans la fermeture et n'aura pas à être présente dans l'environnement courant (comme par exemple la fonction de codage `encode` ou le préfixe `"ABC"` qui sont encapsulées dans la fonction `encode_ABC`).

> *Intérêt 23* (I-O-23) : *L'application partielle permet d'encapsuler des données.*

Séparation programme–données

La lecture de programmes fonctionnels peut donner l'impression que des fonctions sont créées dynamiquement selon le chemin d'exécution du programme alors que seules les fermetures sont créées dynamiquement. Le compilateur n'ajoute à la séquence d'instructions codant le corps de la fonction que les instructions d'accès aux valeurs des identificateurs libres dans ce corps. Ces valeurs n'apparaissent donc pas dans le jeu d'instructions produit par la création dynamique d'une fermeture et leurs liaisons aux identificateurs libres sont gérées par le GC.

> *Intérêt 24* (I-O-24) : *Le compilateur garantit la séparation programme–données.*
> Même s'il contient des traits impurs, le code qui s'exécute est produit à la compilation sous la forme de fermetures et ne peut pas

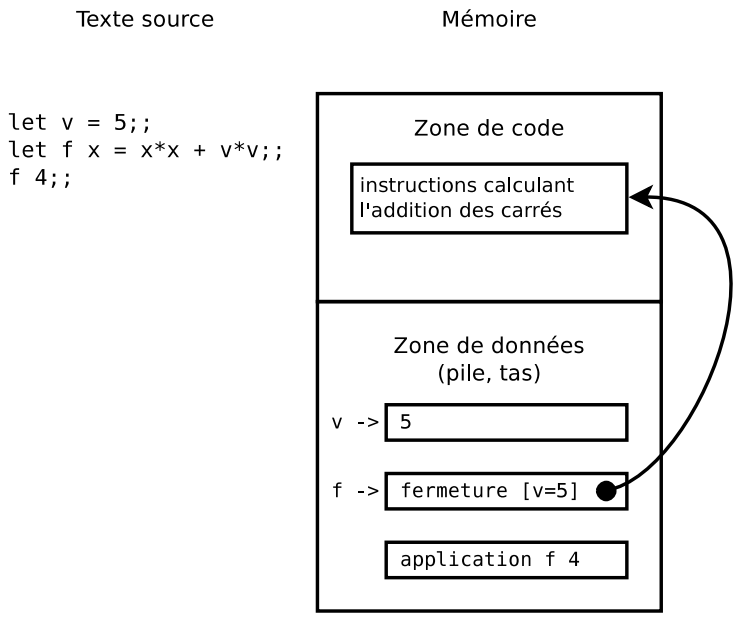

FIGURE 1.1 – Séparation entre bloc d'instructions et données du programme

s'altérer lui-même. Son intégrité est donc garantie. Il n'y a pas de création ni de modification dynamique de code, ce qui permet en mode natif de placer le code dans la zone en lecture seule. ■

Danger 25 (D-O-25) : *En présence de chargement dynamique de code les garanties du compilateur sont affaiblies.*
Le chargement dynamique de code modifie ipso-facto le code exécuté. ■

La figure 1.1 illustre la séparation entre programme et données par un exemple. Le bloc d'instructions correspondant au calcul effectué par la fonction `f` est enregistré statiquement dans la zone de code alors que les données manipulées dynamiquement par le programme sont dans une zone différente composée d'une pile et d'un tas. L'identifiant `f` est ainsi une fermeture capturant la valeur de `v`.

1.1.3 Étude du typage

Cette section présente les types OCaml associés aux paradigmes fonctionnel et impératif. Les types associés aux classes sont présentés avec les traits objet OCaml (cf. section 1.4).

Le mécanisme de typage comprend trois facettes. L'**inférence de types** attribue un type à toute expression. L'algorithme de **typage** contrôle la conformité de l'utilisation des expressions à leur type. Le typage est aussi utilisé pour calculer le placement des données en mémoire (voir intérêt I-O-5). Ces trois facettes sont étudiées dans cette section.

Propriétés du typage

OCaml est un langage **fortement statiquement typé**. Cela signifie que toute expression possède un type et un seul.

> *Intérêt 26* (I-O-26) : *Le typage statique fort de OCaml apporte plusieurs garanties.*
> Le typage apporte plusieurs garanties : accès cohérent aux variables, contrôle strict de la conformité de la manipulation des données à leur type, constance des valeurs non-mutables, protection des données de types abstraits, exhaustivité du filtrage. Ces garanties sont présentées dans les sections décrivant les points en question. ■

> *Intérêt 27* (I-O-27) : *Le compilateur détecte les erreurs de typage.*
> Le compilateur détecte les erreurs de typage du code source. Si il n'y en a pas, il produit un code compilé qui offre les garanties listées ci-dessus. ■

Le typage peut être vu comme un mécanisme d'abstraction des identificateurs et des expressions, qui facilite le prototypage et permet de rendre compte de la spécification du programme. Le langage de types de OCaml est suffisamment expressif pour décrire toutes les structures de données requises par le développement d'algorithmes complexes, et offre ainsi une assistance importante au programmeur.

> *Intérêt 28* (I-O-28) : *Expressivité apportée par la richesse du langage de types.*

Dans la suite du texte, on appellera informellement **type structuré** un type somme ou un type enregistrement ou encore un type produit, sauf dans l'étude de certains points où les types produit seront traités à part.

> *Recommandation 29* (R-O-29) : *Définir la représentation des données par un type.*

> Il est recommandé d'utiliser les types structurés pour refléter au mieux la structure des données manipulées par le programme. Cela facilite la traçabilité de la spécification dans le texte source et permet de bénéficier de vérifications automatiques et systématiques. Cela libère de plus le programmeur de la gestion bas niveau de ces structures. ■

L'utilisation des types structurés va de pair avec le mécanisme de filtrage, qui permet de décrire ces traitements selon les différentes formes possibles des données (cf. section 1.1.5).

Comme la construction ou la déconstruction de données ne passe pas par des manipulations de pointeurs définies par le programmeur, toute une classe d'erreurs disparaît. En effet, le compilateur OCaml se base sur les informations de types pour instrumenter automatiquement le code du programmeur avec des instructions bas niveau de construction et de déconstruction de données.

De plus, comme mentionné ci-dessus, le compilateur effectue une vérification de la cohérence de l'utilisation des données par typage et analyse de filtrage.

Inférence de types

OCaml réalise l'**inférence de types** : les types de toutes les expressions du texte source sont automatiquement calculés par le compilateur. Ceci signifie qu'aucune déclaration du type d'une expression n'est nécessaire. Le type inféré par le compilateur est correct, au sens où il est conforme aux règles de typage.

> *Intérêt 30* (I-O-30) : *L'inférence de types allège l'écriture et la relecture du texte source.*

L'inférence de type est complète dans le sens où tout programme bien typé est accepté par le compilateur. Dans le cas où l'inférence de type échoue, le compilateur signale quelles expressions posent problème.

Par défaut, les types inférés durant la compilation ne sont pas présentés in extenso au développeur car le volume d'informations produit serait trop important.

> *Intérêt 31* (I-O-31) : *Tous les types inférés peuvent être présentés au développeur.*
> L'option `-i` du compilateur produit la liste des identificateurs globaux liés dans le texte source et de leur type (voir section 1.3).

L'option `-annot` du compilateur donne de plus le type de toutes les sous-expressions. Cette option utilisée en association avec les outils d'édition de OCaml, permet d'examiner les types inférés (voir [OUTILS-OCAML, 2011]). ■

Intérêt 32 (I-O-32) : *L'inférence de type révèle des erreurs de programmation.*
Une différence entre le type inféré par le compilateur et celui attendu par le développeur peut révéler une erreur dans le texte source. ■

Recommandation 33 (R-O-33) : *Vérifier que le type inféré correspond au type attendu.*

L'exemple ci-dessous explicite un cas courant d'erreur qui peut être détecté en vérifiant les types inférés. Dans cet exemple, `instruction` fait un effet de bord d'impression (ligne 2). Le type inféré pour `instruction` est `unit` (ligne 3) car l'effet de bord est effectué au moment de la déclaration, donc l'identificateur `instruction` est lié à la valeur `()`. L'évaluation de l'appel `main 4`, évalue deux fois l'expression `instruction` soit `()` ce qui ne fait rien, puis calcule `fact (4)`.

```
1 let instruction = print_endline "Hello world";;
2 Hello world
3 val instruction : unit = ()
4 let main n = instruction; instruction; fact (n);;
5 val main : int -> int = <fun>
6 main 4;;
7 - : int = 24
```

Si l'intention du développeur était d'imprimer deux fois "Hello world" puis d'évaluer la factorielle de `4`, le code de l'exemple est faux et le type inféré pour `instruction` le montre. Pour que le code agisse comme attendu par le développeur, il aurait fallu que le type inféré pour `instruction` soit le type fonctionnel `unit -> unit`.

La version correcte du code est :

```
1 let instruction () = print_endline "Hello world";;
2 val instruction : unit -> unit = <fun>
3 let main n = instruction (); instruction (); fact (n);;
4 val main : int -> int = <fun>
```

```
5 main 4;;
6 Hello world
7 Hello world
8 - : int = 24
```

Danger 34 (D-O-34) : *La confusion entre les types* `unit` *et* `unit -> unit` *conduit à des erreurs de programmation.*

Recommandation 35 (R-O-35) : *La vérification des occurrences du type* `unit` *inférées permet un contrôle des effets de bord.*

Types atomiques

OCaml dispose de **types atomiques** non-mutables : entiers (`int`, `nativeint`, `int32`, `int64`), flottants (`float`), booléens (`bool`), caractères (`char`) et le **type unit** (`unit`). Le **type exception** (`exn`) est celui des exceptions : ses particularités sont étudiées dans la section 1.2.3.

OCaml n'effectue pas de conversion implicite. Il fournit en revanche des conversions explicites, le typage vérifiant leur bonne utilisation. Il ne faut pas confondre conversion implicite et sous-typage. Une conversion transforme la représentation de la donnée pour adapter sa forme à une opération qui ne lui est pas a priori destinée. La relation de sous-typage de OCaml, examinée dans la section 1.4.2 assure qu'une valeur d'un sous-type peut être considérée comme une valeur de son super-type car elle possède toutes les caractéristiques définissant ce super-type.

Intérêt 36 (I-O-36) : *Absence de conversion implicite.*
L'absence de conversion implicite rend le programme plus robuste car elle évite les erreurs dues à des conversions implicites non souhaitées. ■

Types fonctionnels

Les **types fonctionnels**, notés `t1 -> t2`, sont ceux des expressions fonctionnelles prenant en argument une valeur de type `t1` et calculant une valeur de type `t2`.

Intérêt 37 (I-O-37) : *Contrôle de la conformité des applications des fonctions à leur type.*
Le typage fort et l'absence de conversion implicite assurent la cohérence du texte source, grâce à l'inférence de type des fonctions. ■

Intérêt 38 (I-O-38) : *Documentation automatique des fonctions par leur type.*
Le type d'une fonction fournit une première information sur cette fonction. En particulier, les fonctions de type `t -> unit` sont souvent des fonctions effectuant des effets de bord. ■

Types produit et enregistrement

En OCaml, un groupe de valeurs peut être représenté de trois manières différentes. Les **types enregistrement** représentent des valeurs composées d'un groupe de champs étiquetés fixé dans la définition du type. Les **types produit cartésien** représentent des n-uplets, dont les composantes ne sont pas nommées. Les objets de OCaml sont, eux aussi, des groupes de valeurs nommées (voir section 1.4.2) et peuvent a priori, servir également à représenter des n-uplets de valeurs étiquetées. Cependant, pour représenter des structures de données, les enregistrements sont à préférer aux classes et aux types produit cartésien.

Recommandation 39 (R-O-39) : *Représenter un n-uplet de données par un enregistrement plutôt qu'une classe ou un type produit cartésien.*
Cette recommandation complète la recommandation R-O-29. Les types enregistrement ont un certain nombre d'avantages par rapport aux produits cartésiens, qui sont présentés ci-dessous. Par rapport aux classes, les types enregistrement possèdent les avantages suivants :
- Contrairement aux objets, l'égalité et la comparaison d'enregistrements sont définies structurellement.
- Les fonctions sur les enregistrements peuvent être définies par filtrage, ce qui est impossible sur les objets.
- La structure des valeurs des types enregistrement est définie par le type et ne peut évoluer. Les classes peuvent être héritées et leurs objets peuvent en fait appartenir à une sous-classe. ■

Les types enregistrement autorisent l'égalité, la comparaison structurelle et le filtrage. Les champs d'un enregistrement sont par défaut non mutables mais peuvent être individuellement déclarés mutables (cf. section 1.2.2).

Ainsi, si un enregistrement possédant un champ mutable (ligne 2) est partagé par plusieurs identificateurs (`enreg` en ligne 2 et `paire_enreg` en ligne 4), la modification via l'un des identificateurs (ligne 6) aura pour effet de changer la valeur des autres (ligne 10) rendant la compréhension du programme plus difficile.

```
1  type enreg = { mutable m : int };;
2  let enreg = { m = 1 };;
3  val enreg : enreg = {m = 1}
4  let paire_enreg = (enreg, { m = 2 });;
5  val paire_enreg : enreg * enreg = ({m = 1}, {m = 2})
6  enreg.m <- 10;;
7  enreg;;
8  - : enreg = {m = 10}
9  paire_enreg;;
10 - : enreg * enreg = ({m = 10}, {m = 2})
```

> *Recommandation 40* (R-O-40) : *Restreindre l'utilisation de champs mutables dans les enregistrements.*
> Un champ d'un enregistrement ne doit être déclaré mutable que si les besoins de la spécification l'imposent. ■

La construction `{ e with ... }` permet de définir un nouvel enregistrement à partir des champs d'un enregistrement connu `e`. L'enregistrement `e` n'est pas modifié, les champs non mentionnés avec `with` sont dupliqués, ceux qui sont mentionnés sont recréés. Dans l'exemple, le champ `n` est recréé pour `e2`. Le champ mutable `m` est dupliqué pour `e2` et reçoit la même valeur, qui pourra être mutée indépendamment dans les deux enregistrements. Le champ non mutable `r` est, lui aussi, dupliqué et reçoit la valeur de `e1.r`. Celle-ci est une référence, qui est partagée entre `e1` et `e2`. Donc, toute affectation de cette référence modifie les deux enregistrements.

```
1  type t = { n : int; mutable m : int; r : int ref };;
2  let e1 = { n = 1; m = 2; r = ref 3 };;
3  val e1 : t = {n = 1; m = 2; r = {contents = 3}}
4  let e2 = { e1 with n = 10 };;
5  val e2 : t = {n = 10; m = 2; r = {contents = 3}}
6  e1.m <- 20;;
7  - : unit = ()
```

```
8  e1.r := 30;;
9  - : unit = ()
10 e1;;
11 - : t = {n = 1; m = 20; r = {contents = 30}}
12 e2;;
13 - : t = {n = 10; m = 2; r = {contents = 30}}
```

La copie partielle de champs permet d'alléger la manipulation des enregistrements car elle facilite la définition de nouveaux enregistrements. Il est à noter que les champs, déclarés mutables, de copies d'enregistrements sont toujours distincts, ce qui évite le problème de l'*aliasing*. Ce n'est pas le cas des champs non mutables dont la valeur est de type `t ref`.

Il est à noter que les types référence `t ref` (cf. section 1.2.2) sont des enregistrements ayant un champ unique mutable appelé `contents`, comme l'illustre l'exemple ci-dessus.

Bien que représentant tous deux des n-uplets de valeurs, les types enregistrement et produit cartésien sont traités différemment par OCaml car les types enregistrement sont obligatoirement nommés et définis par l'utilisateur. Cela a des conséquences sur l'égalité entre types (voir partie 1.1.3).

Types somme

Les **types somme** (ou **types union**) permettent de définir des valeurs ayant plusieurs formes possibles. Chacune des formes est définie à l'aide d'un **constructeur de valeur** qui lui est propre, ce constructeur de valeur pouvant être constant ou appliqué à une ou plusieurs valeurs de n'importe quel type. Le nom du type peut figurer dans sa définition, permettant de créer ainsi un **type récursif**. Par exemple, la déclaration suivante introduit un nouveau type défini par la donnée d'un constructeur de valeur `Suitevide` définissant une valeur constante et un constructeur de valeur `ConsInt` pouvant être appliqué à deux valeurs de types `int` et `suite_int` pour créer une valeur de type `suite_int`. Ce type permet de représenter des suites finies d'entiers.

```
1 type suite_int =
2   | Suitevide
3   | ConsInt of int * suite_int
```

Intérêt 41 (I-O-41) : *Le mécanisme de filtrage est disponible sur les types somme.*

Les fonctions sur les valeurs de type somme peuvent être définies par filtrage. ■

Intérêt 42 (I-O-42) : *Les types somme permettent de représenter des données complexes sans manipulation de pointeurs.*
Aucune manipulation de pointeur n'est nécessaire pour la création et la manipulation de structures de données, aussi complexes soient-elles. Leur désallocation est assurée par le GC (voir intérêt I-O-5), sans aucune intervention du programmeur. ■

Intérêt 43 (I-O-43) : *Contrôle de la construction et déconstruction de valeurs de type somme.*
Le compilateur vérifie qu'une construction d'une valeur d'un type somme ou sa déconstruction sont effectuées en accord avec la définition du type. Ces vérifications réduisent fortement les erreurs de manipulation de données. ■

Recommandation 44 (R-O-44) : *Utiliser les types somme pour représenter des structures de données linéaires et arborescentes.*
Cette recommandation complète la recommandation R-O-29. Les types somme permettent de représenter directement toutes les structures de données linéaires ou arborescentes, exactement comme elles sont décrites dans les ouvrages d'algorithmique. ■

Types polymorphes

Un **type polymorphe** (ou type paramétré) est un type dont la définition dépend d'autres types, représentés par des paramètres de type (encore appelés variables de type). Ainsi, il est possible de définir des types enregistrement, des types somme, des types associés à des classes, etc. en utilisant des paramètres de type pour désigner le type des sous-structures qui apparaissent génériques dans la spécification de la structure.

Intérêt 45 (I-O-45) : *La généricité des structures de données se traduit dans les types polymorphes.*
Pouvoir exprimer la généricité d'une structure de données évite la duplication de code. Si l'inférence de types attribue un type polymorphe à une expression, cela signifie que cette expression

possède un degré de généricité qui peut être voulu ou témoigner d'une erreur de programmation (voir recommandation R-O-33). ■

Type option

Le **type option** (`'a option`) est un type prédéfini OCaml. C'est un type somme polymorphe paramétré par un type quelconque `'a`. Il possède deux constructeurs : `None` et `Some v` où `v` dénote une valeur de type `'a`.

Certains traitements nécessitent l'utilisation de variables ayant soit une valeur significative, soit une valeur non significative. Cette valeur non significative est utilisée lorsque la valeur n'a pas encore été initialisée par le traitement, lorsque le traitement a échoué, etc. De plus, OCaml oblige le développeur à initialiser toutes les variables à la déclaration (cf intérêt I-O-6) ce qui augmente encore ce besoin car au moment de la déclaration il n'existe pas toujours de valeur significative.

Dans la plupart des langages la valeur non significative et les valeurs significatives sont du même type. Par exemple `-1` est souvent utilisé pour représenter une valeur non significative de type entier positif ou la chaîne vide pour marquer la valeur par défaut d'une chaîne de caractères. Les valeurs non significatives ne peuvent être repérées de manière sûre ni à la compilation, ni à l'exécution ce qui est la source de nombreuses erreurs. Certains langages utilisent la valeur `NULL` comme valeur non significative et cela provoque des erreurs telles que erreur de segmentation en C, exception pointeur null en Java.

Intérêt 46 (I-O-46) : *Le type option permet de dénoter explicitement les valeurs non significatives.*

Recommandation 47 (R-O-47) : *Utiliser le type option pour représenter les valeurs non significatives.*
Ceci permet non seulement d'expliciter dans le texte source les utilisations de valeurs non significatives, mais aussi de vérifier statiquement qu'elles sont traitées systématiquement (par typage et filtrage, voir section 1.1.5). Cette recommandation complète la recommandation R-O-29. ■

Types format

La bibliothèque standard offre des fonctions pour lire et écrire des valeurs selon un format donné : `fprintf/fscanf, printf/scanf, eprintf/escanf,`

`sprintf/scanf`. Alors que dans les langages courants les formats sont des chaînes de caractères, en OCaml les **formats** sont des valeurs du type `Pervasives.format`. Un format contient des spécifications de conversion qui indiquent le type des valeurs attendues : `%c` pour un caractère, `%s` pour une chaîne de caractères, `%i` pour un entier, etc. Un format est constitué d'une séquence de spécifications de conversion et s'applique à une séquence de valeurs.

> *Intérêt 48* (I-O-48) : *Garanties apportées par le typage des formats.*
> Grâce au typage, la bonne formulation des formats d'impression est vérifiée automatiquement. Une erreur est signalée à la compilation si :
> – les nombres de spécifications et de valeurs ne sont pas égaux,
> – le type de la spécification et de la valeur ne correspondent pas.■

L'exemple suivant illustre le typage des formats en OCaml. Le format de la ligne 1 attend un entier. Utiliser ce format avec plus d'un argument (ligne 4) ou à une valeur n'étant pas un entier (ligne 11) aboutit à une erreur de typage. Certains erreurs de typage impliquant des formats peuvent être difficiles à lire, il est souvent utile pour déboguer de rajouter des parenthèses ou des annotations de type. La ligne 9 est équivalente à la ligne 4 mais l'ajout de parenthèses a rendu le message d'erreur de typage plus compréhensible. Formats et chaînes de caractères sont différenciés au niveau du typage mais pas au niveau syntaxique. Il n'est donc pas possible de manipuler les formats avec des fonctions sur les chaînes de caractères, le typage détecte ce genre d'erreurs (lignes 14–18). La fonction `format_of_string` permet de créer un format à partir d'un littéral chaîne de caractères (ligne 19), l'opérateur `^^` permet ensuite de combiner les formats (ligne 21).

```
1  Printf.printf "nombre: %i\n" 4;;
2  nombre: 4
3  - : unit = ()
4  Printf.printf "nombre: %i\n" 4 5;;
5  Error: This expression has type
6           (int -> 'a -> 'b, out_channel, unit, unit, ↵
       unit, 'a -> 'b) format6
7         but an expression was expected of type
8           (int -> 'a -> 'b, out_channel, unit) format = ↵
       (int -> 'a -> 'b, out_channel, unit, unit, unit, ↵
       unit) format6
9  ((Printf.printf "nombre: %i") 4) 5;;
```

```
10 Error: This expression is not a function; it cannot be ↵
     applied
11 Printf.printf "nombre: %i\n" "4";;
12 Error: This expression has type string but an ↵
     expression was expected of type
13          int
14 let fmt_erreur titre = titre ^ "%i\n";;
15 val fmt_erreur : string -> string = <fun>
16 Printf.printf (fmt_erreur "nombre: ") 4;;
17 Error: This expression has type string but an ↵
     expression was expected of type
18          ('a -> 'b, out_channel, unit) format = ('a -> ↵
     'b, out_channel, unit, unit, unit, unit) format6
19 let fmt_titre = (format_of_string "%s: ");;
20 val fmt_titre : (string -> '_a, '_b, '_c, '_d, '_d, ↵
     '_a) format6 = <abstr>
21 let fmt t = t ^^ (format_of_string "%i\n");;
22 val fmt : ('a, 'b, 'c, 'd, 'e, int -> 'f) format6 -> ↵
     ('a, 'b, 'c, 'd, 'e, 'f) format6 = <fun>
```

Type abstraits

Un type est qualifié d'**abstrait** en OCaml si une interface (de module ou de classe) définie par le développeur interdit l'accès à sa représentation et restreint l'accès aux fonctions qui permettent la création et la manipulation des valeurs de ce type. Il n'y a pas de mot-clé marquant la qualité de type abstrait.

L'abstraction de type peut être réalisée en OCaml par l'utilisation de modules (cf. section 1.3). On ne confondra pas avec les classes abstraites (cf. section 1.4.2).

La représentation d'un type abstrait n'est pas visible, aucune fonction de construction ou de déconstruction sur ce type ne peut être ajoutée à l'extérieur. De plus, seules les fonctions exportées par le type abstrait peuvent être utilisées pour manipuler les valeurs de ce type. Un type abstrait apporte donc une garantie de confidentialité sur les valeurs de ce type, garantie contrôlée par le compilateur.

> *Intérêt 49* (I-O-49) : *L'utilisation de types abstraits protège la représentation des données.*

Pour maintenir des propriétés de la représentation d'un ensemble de données d'un type donné, il suffit de fournir des fonctions de construction de ces valeurs qui maintiennent ces propriétés et de cacher la définition du type. C'est précisément le rôle d'un type abstrait.

> *Recommandation 50* (R-O-50) : *Utiliser des types abstraits pour maintenir des invariants de représentation des données.*

> *Recommandation 51* (R-O-51) : *Utiliser des types abstraits pour masquer la représentation de données sensibles.*

Même si la protection offerte par les types abstraits est contournable via la comparaison et une certaine utilisation des exceptions, l'abstraction des types reste un moyen très sûr pour encapsuler des données ou traitements à protéger. Les risques peuvent être réduits en suivant les recommandations R-O-65 et R-O-110.

> *Danger 52* (D-O-52) : *Le mécanisme d'exception peut violer l'encapsulation.*
> Voir 1.2.3. ■

> *Danger 53* (D-O-53) : *Les comparaison prédéfinies peuvent violer l'encapsulation.*
> Voir 1.1.4. ■

OCaml propose une notion intermédiaire entre type concret (dit encore type manifeste) et type abstrait, qui utilise le mot-clé `private` dans sa définition. On appelle ces types des **types abstraits algébriques**

> *Danger 54* (D-O-54) : *Une déclaration de type incluant le mot-clé* `private` *laisse visibles les constructeurs de valeur de ce type.*
> Le danger ne vient que de la confusion possible sur la signification de ce mot-clé `private`, qui diffère dans d'autres langages. ■

Abréviation et définition de type

Les types OCaml sont définis par des expressions de type qui sont évaluées en des valeurs de type. Ces valeurs ne sont pas directement manipulables par l'utilisateur.

La notion de nom utilisée pour les relations entre types est celle de **nom qualifié**, obtenu en concaténant l'identificateur figurant dans la déclaration avec le nom du module où cette déclaration figure (voir 1.3). Dans un module donné, le nom qualifié se réduit au nom. Un nom de type peut être masqué, de la même manière qu'un nom de valeur.

Une **expression de type** est construite à partir de constructeurs de type (par exemple, `->`, `*`), de types connus et de variables de type `'a`. Une expression de type peut être nommée par la déclaration `type nom_type = expression_type`, une telle déclaration est appelée une **abréviation** car elle allège l'écriture mais n'ajoute pas de type.

> *Danger 55* (D-O-55) : *Toutes les abréviations du même type sont égales.*

Dans l'exemple suivant, les abréviations `masse` et `taille` du type `float` ne permettent pas de différencier leurs valeurs. Ainsi une fonction définie comme manipulant explicitement des `masse` (ligne 3) peut recevoir des valeurs de type `taille` en paramètre (ligne 9) car au niveau du typage, les types `masse` et `taille` sont égaux au même type `float`.

```
1  type masse = float;;
2  type taille = float;;
3  let plus_masse (x : masse) (y : masse) : masse =
4    x +. y;;
5  val plus_masse : masse -> masse -> masse = <fun>
6  let t : taille = 3.2 and m : masse = 4.5;;
7  val t : taille = 3.2
8  val m : masse = 4.5
9  plus_masse t m;;
10 - : masse = 7.7
```

Une **définition explicite de type** est une déclaration `type nom_type = definition_type` où `definition_type` est soit une liste de nouveaux champs introduisant un nouveau type enregistrement (i.e. `type enreg = { a : string; b : int}`) soit une liste de nouveaux constructeurs introduisant un nouveau type somme (i.e. `type somme = A of int | B of string | C`). Le type ainsi introduit est différent de tout type préexistant.

L'exemple suivant définit les types `masse` et `taille` comme des types somme n'ayant qu'un constructeur. Une fonction définie comme manipulant des `masse` (ligne 3) n'admet pas des valeurs d'un autre type en paramètre (ligne 9).

```
type masse = Masse of float;;
type taille = Taille of float;;
let plus_masse (Masse x) (Masse y) =
  Masse (x +. y);;
val plus_masse : masse -> masse -> masse = <fun>
let t = Taille 3.2 and m = Masse 4.5;;
val t : taille = Taille 3.2
val m : masse = Masse 4.5
plus_masse t m;;
Error: This expression has type taille but an ↵
    expression was expected of type
           masse
```

Relations entre types

Les relations entre types permettent de contrôler finement la manipulation des expressions.

Il existe trois relations sur les types : l'égalité entre types, la compatibilité liée à l'instanciation et la compatibilité liée au sous-typage.

L'**égalité de deux expressions de type** se fait sur la structure des expressions : deux types `t1 * t2` et `t3 * t4` sont égaux si `t1` et `t3` sont égaux, et `t2` et `t4` le sont également. Autrement dit, deux expressions de type sont égales si et seulement si elles ont la même structure. Deux abréviations de type sont égales si les expressions de type qu'elles dénotent sont égales.

Un type introduit par une définition de type ne peut être égal à un autre type et de même pour un type abstrait.

Les types enregistrement sont obligatoirement introduits par une définition de type. Dans un même module, déclarer un type enregistrement en reprenant le nom et la définition d'un type précédemment introduit masque ce dernier type. Les valeurs du type masqué ne sont pas converties en des valeurs du type masquant.

Un type `t1` est une **instanciation** d'un type polymorphe `'a t` si `t1` est le résultat du remplacement de `'a` par un type quelconque. Toute instanciation d'un type peut remplacer ce type dans le typage d'une expression (voir I-O-45). Par exemple `('b list) list` et `int list` sont deux instances du type `'a list`.

Le sous-typage définit une autre relation de compatibilité entre types définis via des classes et est donc traité en 1.4.2.

Un type `t1` est dit **compatible** avec un type `t2` si `t1` et `t2` sont égaux ou si `t1` est une instance de `t2` ou bien si `t1` est un sous-type de `t2`. Le type `t1` peut dans ce cas remplacer `t2` au cours du typage.

Types variants polymorphes

Les types **variants polymorphes** sont une extension avancée des types somme de OCaml, qui permettent d'ajouter des constructeurs de valeur selon les besoins. Ils n'ont pas besoin d'être introduits par une définition de type, leur égalité est définie structurellement et ils fournissent une notion de sous-typage. Cette extension augmente l'expressivité du langage au prix d'une complexification certaine du texte source et d'une difficulté accrue de relecture.

> *Recommandation 56* (R-O-56) : *Préférer les types somme aux types variants polymorphes.*
> Les types somme sont à préférer car leur définition ne peut pas évoluer et ils permettent de vérifier l'exhaustivité du filtrage. Les types variants polymorphes apparaissent avec le symbole back-quote `` [`Variant] ``. ■

Types objet

Les types associés aux objets sont définis par l'ensemble des types de leurs méthodes : voir section 1.4.2. Leur égalité est également définie structurellement et ils fournissent une notion de sous-typage, expliquée dans la section 1.4.2.

Conversion de types

> *Intérêt 57* (I-O-57) : *Les conversions de types sont explicites.*
> Les **conversions de type** de OCaml sont toujours explicites et qui plus est, restreintes à des types objet ou des types variants polymorphes. ■

Elles seront étudiées pour les objets dans la partie qui leur est dédiée. En revanche, l'utilisation des types variants polymorphes n'étant pas recommandée, les conversions les concernant ne seront pas détaillées.

Typage et fichiers `.mli`

Un texte source OCaml est un fichier d'extension `.ml`. Le compilateur lui associe un fichier de spécification, portant le même nom, d'extension `.mli`, qui contient la liste des noms déclarés dans le fichier et de leurs types. Ce fichier `.mli` peut être vu comme une spécification des éléments du fichier. Il est géré par de nombreux outils d'édition. Il est possible de créer un fichier `.mli` et de le compiler avant de définir le fichier `.ml` de même nom et de compiler celui-ci. Ce faisant, le compilateur vérifie que les éléments du fichier `.ml` ont bien le type annoncé dans le fichier `.mli`. Cette possibilité est intéressante à deux points de vue, d'une part pour faciliter les interactions entre différents développeurs, d'autre part à des fins de vérification.

> *Recommandation 58* (R-O-58) : *Écrire des fichiers de spécification* `.mli`.
> Écrire la spécification d'un développement OCaml permet de contrôler la visibilité et le typage du code. On améliore ainsi la robustesse du code. ■

Contournement du typage

Une construction du langage, une option du compilateur, et certaines fonctions de la bibliothèque standard permettent de contourner les contraintes imposées par le typage. On les appelle **constructions non-sûres**. Leur utilisation invalide toutes les garanties apportées par le typage, que ce soit pour la sûreté de fonctionnement ou pour l'encapsulation et la sécurité.

> *Danger 59* (D-O-59) : *Les constructions non sûres permettent de contourner le typage.*
> Ces constructions non-sûres sont listées ci-dessous :
> – la déclaration de fonction externe avec le mot-clé `external` ;
> – l'option `-unsafe` du compilateur ;
> – les fonctions du module `Obj` ;
> – les fonctions du module `Marshal` ;
> – les fonctions du module `Printexc` ;
> – les fonctions `unsafe_get` et `unsafe_set` des modules `Array`, `Bigarray`, et `String` ;
> – la fonction `create` du module `String`.
> La description détaillée des constructions non-sûres et des re-

commandations associées sont disponibles dans [MODE-EX, 2011] et [RECOM-OCAML, 2011]. ■

Recommandation 60 (R-O-60) : *Éviter l'emploi de constructions non-sûres.*
L'utilisation des constructions non-sûres est à proscrire absolument. Deux exceptions peuvent être justifiées par la spécification :
– le mot-clé `external` s'il est indispensable d'utiliser des fonctions externes (par exemples écrites en C),
– et le module `Marshal` s'il faut transmettre ou recevoir des données d'un autre programme OCaml qui utilise le format de `Marshal`. ■

Recommandation 61 (R-O-61) : *Vérifier les fonctions externes.*
Quand on utilise une bibliothèque ne provenant pas d'une source de confiance, il est important d'examiner son code source pour vérifier que les fonctions externes qu'elle utilise respectent les contraintes du typage. ■

Recommandation 62 (R-O-62) : *Vérifier les utilisations de* `Marshal`.
Quand on utilise une bibliothèque ne provenant pas d'une source de confiance, il est important d'examiner son code source pour vérifier que les appels aux fonctions du module `Marshal` ne sont pas appliquées à des données confidentielles, et qu'elles respectent les contraintes du typage. Ceci ne donne pas une garantie totale, mais limite les risques. ■

1.1.4 Mécanismes de gestion des expressions

Comparaison de valeurs

Le langage OCaml propose un opérateur d'égalité, des opérateurs et fonctions d'ordre polymorphes prédéfinis : le type de `(=)`, `(!=)`, `(<)`, `(>)`, `(<>)`, `(<=)`, `(>=)` est `'a -> 'a -> bool`, le type de `compare` est `'a -> 'a -> comparison` et le type de `min, max` est `'a -> 'a -> 'a`. Cela signifie que ces opérateurs et fonctions peuvent être utilisés avec des arguments de n'importe quel type. La seule propriété énoncée et garantie sur `<=` est qu'il s'agit bien d'un ordre total (sauf sur les nombres flottants). Les autres opérateurs et fonctions prédéfinis sont

basés sur le même ordre. Dans le reste de cette section, nous utiliserons le terme générique de comparateur pour représenter l'un quelconque de ces opérateurs ou fonction car tous les points soulevés s'appliquent à tous.

Un comparateur prédéfini compare les valeurs pour les types atomiques et la structure (récursivement) pour les types structurés. Par exemple, `[1]@[2] = 1::[2]` vaut `true`. En dehors des restrictions décrites ci-dessous, la comparaison de deux valeurs retourne toujours un résultat.

> *Danger 63* (D-O-63) : *Un comparateur prédéfini peut ne pas retourner de résultat.*
> La comparaison de deux valeurs fonctionnelles lève l'exception `Invalid_argument`. La comparaison de valeurs cycliques peut ne pas terminer. ■

> *Danger 64* (D-O-64) : *Possibilité d'observer de valeurs encapsulées d'un type abstrait.*
> Un comparateur permet l'observation de valeurs de types abstraits : une valeur sensible peut être identifiée en procédant par dichotomie ou par essais/erreurs. ■

Pour interdire la comparaison de deux valeurs encapsulées, on peut les envelopper dans un type fonctionnel, comme le montre l'exemple suivant.

```
1  module M :
2    sig
3      type secret
4      val construire : int -> secret
5    end =
6    struct
7      type secret = unit -> int
8      let construire i = fun () ->  i
9    end
10 ;;
11 let x = M.construire 2 and y = M.construire 25;;
12 val x : M.secret = <abstr>
13 val y : M.secret = <abstr>
14 x = y;;
15 Exception: Invalid_argument "equal: functional value".
16 x < y;;
17 Exception: Invalid_argument "equal: functional value".
```

Recommandation 65 (R-O-65) : *Compléter la protection des données sensibles encapsulées dans des modules.*
L'inclusion dans un type fonctionnel permet de répondre au danger D-O-64 mais elle ne fait que contourner le problème de l'autorisation de l'égalité sur les types abstraits. ■

Hachage

La **fonction de hachage** (`Hashtbl.hash`) de type `'a -> int` associe à toute valeur un entier. Deux valeurs différentes ont une probabilité élevée d'être associées à deux entiers différents. Cette propriété permet de définir une version approximative de l'égalité, qui pose les mêmes problèmes que l'égalité et la comparaison prédéfinis

Danger 66 (D-O-66) : *Les fonctions de hachage permettent de contourner l'encapsulation.*
Le hachage permet de contourner l'encapsulation (même si elle est complétée par des types abstraits fonctionnels comme indiqué par la recommandation R-O-65). ■

Recommandation 67 (R-O-67) : *Vérifier l'utilisation des fonctions de hachage.*
Il faut s'assurer que les fonctions de hachage `Hashtbl.hash` et `Hashtbl.hash_param` ne sont pas appliquées à un type abstrait. ■

Partage de valeurs

Des valeurs d'un type structuré représentant différentes structures de données peuvent partager certaines sous-structures. Typiquement, si `l = [1;2]`, alors `l1 = l @ l` et `l2 = 3 :: l` partagent `l`. Ce partage défini dans le texte source par l'utilisation d'un même identificateur de type structuré est implémenté par le partage physique (pas de recopie). OCaml assure le partage des sous-structures identiques.

Intérêt 68 (I-O-68) : *Partage de sous-structures non-mutables.*
Le partage de sous-structures non-mutables permet de diminuer, sans danger, la place mémoire utilisée. Il est totalement transparent pour l'utilisateur. ■

Danger 69 (D-O-69) : *Partage de sous-structures mutables.*
Le partage d'une sous-structure mutable entre plusieurs structures de données implique que toute modification physique de cette sous-structure sera répercutée sur toutes les structures, par effet de bord. Cela peut conduire à des erreurs d'exécution, parfois difficiles à diagnostiquer (voir 1.2). ■

Intérêt 70 (I-O-70) : *Le GC garantit la désallocation correcte des structures.*
Le GC garantit qu'aucune structure partagée ne sera désallouée avant la fin de toutes ses utilisations. ■

1.1.5 Filtrage

Mécanisme de filtrage

OCaml propose le mécanisme de **filtrage** (*pattern matching* en anglais) pour définir les manipulations sur des valeurs de type structuré. Un **filtre** se compose d'un préfixe d'une valeur structurée, les sous-structures absentes étant représentées soit par des paramètres, soit par le symbole `_` appelé « attrape-tout ». Filtrer (ou superposer) une valeur par un filtre consiste à vérifier que le filtre est bien un préfixe de cette valeur. Si c'est le cas, le filtrage réussit et les paramètres du filtre sont liés aux sous-structures correspondantes dans la valeur (`_` ne provoque aucune liaison). Le filtrage permet de décomposer une structure de données en sous-structures, à une profondeur choisie par le développeur. Les filtres peuvent contenir des valeurs de type atomique, des constructeurs de types somme et des constructeurs de types enregistrement. Ils ne peuvent pas contenir plusieurs occurrences d'un même paramètre (sauf `_`) (le filtrage est dit linéaire).

Une évaluation peut se faire par cas suivant la forme d'une valeur, en utilisant la construction syntaxique (`match exp with F1 -> e1 | F2 -> e2 | ...`). Les cas sont décrits par des clauses de la forme `F -> e`, où `F` est un filtre et `e` une expression construite avec les paramètres du filtre (sauf `_`) décrivant le résultat correspondant. L'évaluation d'un `match` provoque le filtrage de la valeur de `exp` par les filtres du `match`, en suivant l'ordre des filtres donné dans le texte. Le premier filtrage réussi provoque l'évaluation de la clause correspondante, en utilisant les liaisons construites par le filtrage.

L'inférence de type attribue à un filtre le type des valeurs qu'il peut filtrer. Tous les filtres d'un même `match` doivent être de même type. Les résultats des

clauses correspondantes doivent être tous de même type.

Le mécanisme de filtrage de OCaml permet de décomposer une valeur structurée en ses sous-structures, sans avoir recours à des manipulations de pointeurs (voir I-O-88).

Intérêt 71 (I-O-71) : *Le filtrage est un mécanisme puissant de manipulation de données structurées.*

Intérêt 72 (I-O-72) : *Le compilateur assure la robustesse des définitions par filtrage.*
Le compilateur vérifie que tous les filtres sont du même type et que l'ensemble des filtres couvre toutes les valeurs de ce type. ■

Recommandation 73 (R-O-73) : *Favoriser la représentation de données par des types structurés pour bénéficier du filtrage.*
Pour pouvoir bénéficier du filtrage, il est préférable d'utiliser des types structurés plutôt que des types non filtrables comme les types objet. ■

Recommandation 74 (R-O-74) : *Utiliser si possible des filtres disjoints.*
La règle de choix du filtre à utiliser élimine toute ambiguïté du filtrage. Cependant, l'utilisation de filtres se recouvrant ne facilite pas la relecture de texte source et sa maintenance. ■

Analyse statique du filtrage

Un filtrage de valeurs de type `t` est dit **exhaustif** si l'ensemble de ses filtres reconnaît toutes les valeurs possibles du type `t`. La vérification d'exhaustivité du filtrage est faite statiquement par le compilateur OCaml, qui émet un avertissement en cas de filtrage non exhaustif.

Intérêt 75 (I-O-75) : *La vérification de l'exhaustivité du filtrage empêche l'oubli de cas.*
Par défaut, le compilateur émet un avertissement en cas de filtrage non exhaustif. ■

Recommandation 76 (R-O-76) : *N'utiliser que des filtrages exhaustifs.*

Il est recommandé d'activer l'option du compilateur qui produit une erreur à la compilation en cas de filtrage non exhaustif et d'éliminer les filtres non exhaustifs (voir [RECOM-OCAML, 2011]). ■

L'ordre des filtres est déterminant en OCaml. Si un filtre F décrit un cas particulier d'un filtre qui le précède dans la liste des filtres, alors F ne sera jamais utilisé.

Intérêt 77 (I-O-77) : *Le compilateur détecte les filtres non utilisés.*
Le compilateur OCaml vérifie qu'un filtre n'est pas complètement masqué par les filtres précédents et émet un avertissement dans le cas contraire. ■

Les filtres non utilisés indiquent souvent une erreur de programmation.

Recommandation 78 (R-O-78) : *Traiter les avertissements du compilateur concernant les filtres non utilisés.*

Filtrage sur les entiers, flottants, tableaux, chaînes, flottants

Le mécanisme de filtrage sur les valeurs de type `int` (resp. `float`) est défini de manière identique à celui sur les types structurés. Un filtre est un littéral entier (resp. flottant), un paramètre, ou `_`. Le filtrage ne peut être exhaustif que si l'un des filtres du `match` filtre toutes les valeurs non filtrées par les filtres précédents donc s'il s'agit d'un paramètre ou de `_`.

Il en est de même pour les tableaux et les chaînes de caractères. Les filtres sur les tableaux peuvent être décrits avec la syntaxe dédiée aux tableaux (`[| |]`) en dénotant explicitement tous les éléments du tableau (sorte d'écriture en compréhension). Les filtres sur les chaînes de caractères peuvent être décrits par des littéraux (de la forme `"abc"`).

Intérêt 79 (I-O-79) : *Le filtrage sur les types atomiques peut être exhaustif.*
L'exhaustivité du filtrage sur les types atomiques ne peut être obtenue qu'en incluant un filtre réduit à un paramètre ou un `_`. Elle est aussi vérifiée automatiquement sur les types atomiques. ■

Filtrage fragile

Un filtrage est dit **fragile** si son dernier filtre est non discriminant c'est-à-dire réduit à un paramètre ou à l'attrape-tout.

Intérêt 80 (I-O-80) : *OCaml détecte les filtres fragiles.*

Dans un filtrage fragile, le filtre non discriminant traite tous les cas non explicités et force donc l'exhaustivité. Un tel filtrage enlève tout l'intérêt de la vérification de l'exhaustivité puisque même si toute valeur est traitée, les traitements ne sont pas différenciés en fonction de la représentation.

Danger 81 (D-O-81) : *Les filtrages fragiles peuvent cacher des problèmes d'exhaustivité des traitements.*

De plus, le filtrage reste exhaustif même si le type sur lequel le filtrage agit évolue au cours du développement (maintenance, nouvelle version). Soit le type `t` avec deux constructeurs de valeurs `A` et `B` : `type t = A | B` et la fonction `f = function A -> traitement_A | _ -> traitement_B` définie avec un filtrage fragile. Si un constructeur de valeurs `C` est ajouté à ce type structuré `type t = A | B | C`, le filtrage de la définition de `f` reste exhaustif alors que le `traitement_B` est appliqué par défaut aux valeurs construites avec le constructeur `C`. Au contraire, si la définition de `f` n'utilise pas de filtrage fragile `f = function A -> traitement_A | B -> traitement_B`, le non traitement du cas `C` est détecté automatiquement à la compilation car le filtrage n'est plus exhaustif.

Un filtrage sur des valeurs des types `bool`, `list` ou `option` n'est pas considéré comme fragile car ces types, bien que structurés, sont prédéfinis et garantis sans évolution possible dans les versions ultérieures.

Recommandation 82 (R-O-82) : *Activer la détection automatique des filtrages fragiles.*
La détection de filtrages fragiles est faite par le compilateur, à condition d'activer l'avertissement correspondant avec l'option `-w +4` du compilateur. ■

Recommandation 83 (R-O-83) : *Justifier toute utilisation d'un filtrage fragile.*
Il est préférable, lorsque cela est possible, de décrire toutes les formes possibles d'un type structuré à l'aide des constructeurs de valeur de ce type. ■

Filtrage des exceptions

Bien que le type `exn` soit un type somme, le filtrage sur les exceptions diffère légèrement du filtrage sur les types somme ordinaire car il ne peut être exhaustif (cf. section 1.2.3).

Filtrage et gardes

Une clause de filtrage peut contenir une **garde** qui est une expression booléenne. Elle s'écrit sous la forme `F when g -> e`. Si le filtrage d'une valeur par le filtre `F` réussit, alors l'expression `g` est évaluée dans l'environnement enrichi avec les liaisons créées par le filtrage. Si `g` vaut `true`, alors `e` est évalué. Sinon, le filtrage passe au filtre suivant.

> *Danger 84* (D-O-84) : *Le compilateur ne peut vérifier l'exhaustivité d'un filtrage gardé.*

L'utilisation d'un filtrage dont tous les filtres sont gardés peut être parfaitement licite, même si cela introduit de la redondance dans le texte source. Mais cela peut indiquer aussi une erreur de programmation.

> *Intérêt 85* (I-O-85) : *Détection des filtrages entièrement gardés.*
> Le compilateur émet un message si un filtrage ne contient que des filtres gardés (avertissement 25). ■

Dés qu'un filtre gardé est essayé, tous les effets de bord qu'ils contient sont évalués. À la fin du filtrage, tous les effets de bord de tous les filtres tentés auront été évalués. Savoir si une garde est évaluée ou non est déjà difficile à déterminer à la relecture mais déterminer quels effets de bord seront effectués à l'exécution est encore plus difficile.

> *Recommandation 86* (R-O-86) : *Justifier l'utilisation de gardes et ne pas y inclure d'effets de bord.*
> Les gardes facilitent l'écriture du texte source. Mais il faut les manipuler avec prudence pour conserver l'exhaustivité et permettre la compréhension du code source, en particulier en n'y mettant pas d'effet de bord. ■

1.1.6 Récapitulatif

	Filtrage exhaustif	Distingabilité des types	Égalité structurelle
Type produit	●	○	●
Type enregistrement	●	●	●
Type `list`	●	○	●
Type `option`	●	○	●
Type somme	●	●	●
Type `exn`	○	●	●
Type objet	○	○	○
Type fonctionnel	○	○	○

TABLE 1.1 – Comparaison des types OCaml vis-à-vis du filtrage, typage et de l'égalité

1.2 Traits impératifs

OCaml possède une extension de son noyau fonctionnel pur offrant des traits impératifs classiques. Certaines de leurs caractéristiques ont déjà été évoquées dans la section consacrée aux traits fonctionnels car certaines constructions syntaxiques peuvent être utilisées dans les deux cadres (pur et impur). Dans cette section, les traits impératifs sont étudiés de manière systématique du point de vue de la sécurité.

Les instructions que l'on trouve usuellement dans les langages impératifs comme la séquence et les boucles sont présentes en OCaml et ne posent pas de problème spécifique pour la sécurité.

1.2.1 Effets de bord

La plupart des fonctions de la bibliothèque standard de OCaml effectuant des effets de bord l'indiquent par leur type, où intervient le type `unit`.

> *Recommandation 87* (R-O-87) : *Identifier les effets de bord des fonctions de la bibliothèque standard.*
> Le style de programmation fonctionnel pur facilite la maîtrise du

développement. Les effets de bord nécessaires aux échanges avec l'extérieur doivent être soigneusement contrôlés. ■

1.2.2 Références et mutables

OCaml propose plusieurs déclinaisons de valeurs mutables : les références, les tableaux, les chaînes de caractères, les enregistrements comportant des champs mutables. L'utilisateur peut combiner ces différents types avec des types non mutables pour représenter des structures de données complexes dont seules certaines sous-structures sont mutables (voir recommandation R-O-29).

Références, variables mutables

Les pointeurs n'existent pas en OCaml. Ils sont remplacés par la notion de **référence**. Le type OCaml `'a ref` est un type prédéfini (cf. 1.1.3), dont les valeurs sont des adresses mémoire, appelées références. Une référence de type `t ref` désigne une zone mémoire où est stockée une valeur de type `t`. OCaml n'autorise aucune arithmétique sur les références.

Intérêt 88 (I-O-88) : *OCaml ne permet pas l'arithmétique de pointeurs.*

OCaml fournit deux opérations pour respectivement accéder au contenu de la référence par l'opérateur de déréférencement noté `!` (de type `'a ref -> 'a`) et modifier la valeur référencée par l'affectation notée `:=` (de type `'a ref -> 'a -> unit`).

La double signification d'un identificateur dans certains langages — adresse-mémoire et contenu — peut conduire à des erreurs de programmation. C'est impossible en OCaml.

Intérêt 89 (I-O-89) : *La dénotation du déréférencement par l'opérateur* `!` *est explicite en OCaml, ce qui facilite la relecture du texte source.*

Intérêt 90 (I-O-90) : *Contrôle par le typage des lectures-écritures de références.*
Ce contrôle s'appuie sur la distinction entre le type d'une référence et celui de son contenu. ■

Il est impossible de créer une référence sans préciser la valeur référencée, ce qui détermine le type de cette référence de manière unique et permanente.

> *Intérêt 91* (I-O-91) : *En OCaml, une variable mutable est initialisée dès sa création.*

L'inférence de type est conservée en présence de valeurs mutables. Quelques restrictions du polymorphisme sont cependant nécessaires afin que les tentatives de référencer des valeurs de type polymorphe soient rejetées (leur acceptation serait une forme de cast). Par exemple, sachant que la liste vide notée `[]` est une valeur polymorphe de type `'a list`, la déclaration `let x = ref []` n'attribuera pas le type `'a list ref` à `x` mais le type `_a list ref`, où `_a` est une variable de type dont la valeur est fixée dès qu'une valeur différente de `[]` est affectée à `x`. Ainsi, l'affectation `x:=[1]` fixe définitivement le type de `x` à `int list ref`. Ce faisant, le typage conserve toutes les propriétés mentionnées dans la section 1.1.3.

> *Intérêt 92* (I-O-92) : *L'inférence et la gestion de types mutables polymorphes ne peut pas conduire à des cast.*
> Les manipulations génériques de valeurs mutables sont décrites par des types polymorphes mais leur utilisation est contrôlée de manière à garantir l'intégrité de la mémoire. ■

> *Intérêt 93* (I-O-93) : *Déréférencement contrôlé.*
> Les références sont des valeurs typées et peuvent être utilisées sans impact sur la sécurité du programme. Le pointeur Null n'existant pas en OCaml, aucune erreur due à son déréférencement n'est possible. Selon les besoins de la spécification, l'absence de valeur plausible peut être traduite par un type `option` et le marqueur de fin d'une structure de donnée par un constructeur d'un type somme (comme le constructeur `[]` du type `list`). ■

Le corps d'une fonction peut contenir des variables libres, mutables. La fermeture correspondante (voir 1.1.2) encapsule donc ces variables mutables. Si la portée de ces variables est restreinte au corps de la fonction, il n'est possible d'y accéder que depuis le corps de la fonction. On peut ainsi facilement implanter une fonction qui a besoin d'une variable rémanente (c'est-à-dire une variable interne à une fonction dont la valeur persiste d'un appel de la fonction à l'autre) sans ajout de mécanisme particulier. De même deux ou plusieurs fonctions peuvent partager une même référence comme le montre l'exemple suivant, sans avoir besoin de rendre cette référence globale :

```
let incr, reset =
  let c = ref 0 in
  (fun () -> c:=!c + 1; !c),
  (fun () -> c:= 0; !c);;
val incr : unit -> int = <fun>
val reset : unit -> int = <fun>
incr ();;
- : int = 1
incr ();;
- : int = 2
reset ();;
- : int = 0
```

Le couple de fonctions **incr** et **reset** partagent la référence **c** encapsulée dans leurs fermetures. Aucune lecture ou affectation de **c** n'est possible, hormis celles effectuées par **incr** et **reset**.

> *Intérêt 94* (I-O-94) : *L'encapsulation dans la fermeture de références locales à une fonction protège ces références contre toute lecture/écriture extérieure.*

> *Recommandation 95* (R-O-95) : *Restreindre au maximum la portée des variables mutables contenant des données sensibles.*
> Cette recommandation complète la recommandation R-O-1. ■

Rappelons que le partage de valeurs est aussi assuré en OCaml pour les références. La modification du contenu d'une variable mutable partagée affecte toutes les expressions partageant cette variable, comme cela est déjà signalé par le danger D-O-69.

Enregistrements à champs mutables

Les champs des enregistrements peuvent également être modifiés par des affectations pourvu qu'ils aient été déclarés mutables (mot-clé **mutable**) lors de la définition de l'enregistrement. Ainsi on peut syntaxiquement faire la différence entre champs mutables et champs non mutables d'un même enregistrement. Une valeur de type enregistrement ne peut être construite que si toutes les valeurs de tous ses champs sont fournies.

Intérêt 96 (I-O-96) : *Vérification statique du caractère mutable.*
Le compilateur vérifie que l'on ne peut pas modifier un champ non déclaré comme mutable. ■

Tableaux

OCaml propose des tableaux, de type `'a array`, dont les éléments sont modifiables. Un valeur de type `array` est définie en donnant sa taille et, soit une fonction qui énumère ses éléments, soit avec la fonction `Array.make` appliquée à une valeur servant à l'initialisation de tous ses éléments. La taille d'un tableau est fixée au moment de sa création au cours de l'exécution et ne peut pas être modifiée par la suite.

Intérêt 97 (I-O-97) : *Les tableaux sont obligatoirement initialisés dès leur création.*

Intérêt 98 (I-O-98) : *Les accès aux tableaux sont vérifiés au cours de l'exécution.*
Toute tentative d'accès en dehors des bornes déclenche la levée de l'exception `Invalid_argument` et il n'est pas possible de changer la taille d'un tableau. ■

Danger 99 (D-O-99) : *L'option* `-unsafe` *du compilateur supprime tous les contrôles de borne.*

Danger 100 (D-O-100) : *Les accès par* `Array.unsafe_*` *ne sont pas vérifiés.*
Ces fonctions ne sont pas documentées dans le manuel de référence et leur utilisation n'est pas conseillée. ■

Recommandation 101 (R-O-101) : *Ne jamais désactiver le contrôle des bornes.*
Ne jamais utiliser l'option `-unsafe` du compilateur. Ne jamais utiliser les fonctions `Array.unsafe_*`. ■

Chaînes de caractères

Les chaînes de caractères (type `string`) proposées par OCaml sont mutables. Il est donc possible de modifier le contenu d'une chaîne.

> *Danger 102* (D-O-102) : *Les chaînes de caractères de OCaml sont mutables.*

Une valeur de type `string` est définie soit par une constante, soit par un appel à la fonction `String.make` sur une valeur servant à l'initialisation de tous ses éléments, soit par un appel à la fonction `String.create`, qui ne fait aucune initialisation.

> *Danger 103* (D-O-103) : *Les chaînes de caractères ne sont pas obligatoirement initialisées dès leur création.*
> L'utilisation de `String.create` peut provoquer une fuite d'information (détaillé dans [MODE-EX, 2011]). ■

> *Recommandation 104* (R-O-104) : *Ne pas utiliser* `String.create`.

> *Intérêt 105* (I-O-105) : *Les accès aux chaînes sont vérifiés dynamiquement.*
> Tout accès en dehors des bornes de la chaîne déclenche la levée de l'exception `Invalid_argument`. ■

> *Danger 106* (D-O-106) : *L'option du compilateur* `-unsafe` *supprime la vérification des bornes.*
> Voir R-O-101. ■

> *Danger 107* (D-O-107) : *Les fonctions* `String.unsafe_*` *suppriment les contrôles d'accès dans les chaînes.*
> Ces fonctions ne sont pas documentées dans le manuel de référence et leur utilisation n'est pas conseillée. ■

> *Recommandation 108* (R-O-108) : *Ne jamais désactiver la vérification des accès aux chaînes.*
> Ne jamais utiliser l'option `-unsafe` du compilateur ou les fonctions `String.unsafe_*`. ■

1.2.3 Exceptions

Les **exceptions** permettent de définir un traitement à effectuer en réponse à une situation anormale. Elles changent donc le flot d'exécution. C'est pourquoi elles sont incluses dans les traits impératifs.

Les exceptions OCaml ont pour type `exn`. Elles sont déclarées par la construction `exception`, sont levées par la fonction `raise` et peuvent être rattrapées par la construction `try ... with ...`. Par exemple, l'exception `Failure "Message"` est levée par la fonction `failwith` définie par `let failwith s = raise(Failure s)`.

Paramètres d'exceptions

Les exceptions peuvent être paramétrées. Les données que contiennent ces paramètres sont les seules informations sur l'environnement lors de la levée d'exception qui seront accessibles au moment où le rattrapage de l'exception est fait.

> *Intérêt 109* (I-O-109) : *Non divulgation de l'environnement de levée d'exception.*
> Les exceptions ne divulguent que les informations choisies par le programmeur. ■

> *Recommandation 110* (R-O-110) : *Contrôler les arguments d'exceptions.*
> Il est recommandé de ne pas fournir des données confidentielles en argument aux exceptions. Cette recommandation s'applique même si ces données confidentielles sont encapsulées dans un type abstrait. ■

Filtrage et exceptions

Le mécanisme de filtrage peut être utilisé sur les exceptions. Il est à noter que le type `exn` des exceptions étant extensible, l'analyse d'exhaustivité ne peut pas s'appliquer au rattrapage d'exception. Il est donc difficile de s'assurer qu'il ne manque pas un cas dans un rattrapage d'exception. Une analyse de redondance du filtrage est cependant effectuée.

Danger 111 (D-O-111) : *Absence d'analyse d'exhaustivité du filtrage sur le type* `exn`.

Rattrapage d'exceptions

Le mécanisme d'exception permet soit le **rattrapage nominal** d'une exception connue `try ... with Exception ... -> ...` soit le **rattrapage exhaustif** par la construction *attrape-tout* `try ... with _ -> ...` (ne provoquant pas de liaison) ou `try ... with e -> ...` (provoquant la liaison de `e` avec l'exception levée).

Le rattrapage exhaustif peut amener à rattraper une exception qui n'avait pas été considérée lors de l'écriture du `try with` (cf. danger D-O-115). Il permet d'assurer, grâce à la liaison de `e`, un traitement de fin d'exécution, avant la nouvelle levée de l'exception liée à `e`.

Recommandation 112 (R-O-112) : *Utiliser de préférence le rattrapage nominal.*
Il est recommandé de ne rattraper que les exceptions que l'on a créées ou découlant de l'utilisation d'une bibliothèque. Il est recommandé de ne pas utiliser la construction *attrape-tout*. ■

Le rattrapage exhaustif revêt toutefois l'intérêt de pouvoir confiner l'évaluation d'une expression et donc de protéger le programme contre toute défaillance. Il est recommandé dans deux situations particulières.

Recommandation 113 (R-O-113) : *Utiliser le rattrapage exhaustif pour se prémunir de défaillances de bibliothèques externes.*
Le rattrapage exhaustif d'exceptions permet de confiner un calcul pour prémunir l'application complète contre toutes défaillances de ce code. ■

La construction rattrapant toutes les exceptions peut être utilisée comme finaliseur, c'est à dire en levant à nouveau l'exception rattrapée : `try .. with e -> ...; raise e`.

Recommandation 114 (R-O-114) : *Utiliser le rattrapage exhaustif comme un finaliseur.*

Chemins d'exécution et exceptions

Le mécanisme de levée et rattrapage d'exceptions est une structure de contrôle non locale car un `try with` rattrape l'exécution d'un `raise` situé loin dans le code. Cette structure crée ainsi des chemins d'exécution difficilement identifiables dans un code.

> *Danger 115* (D-O-115) : *Le flot de contrôle est dérouté par les exceptions.*
> Le mécanisme d'exception complique la compréhension des chemins d'exécution possibles d'un programme. ■

Contrairement à d'autres langages, les informations concernant les exceptions qu'une fonction lève ou rattrape ne sont pas explicites. De plus les exceptions sont transparentes vis-à-vis du typage : les exceptions ont pour type `exn` mais une fonction aura le type `int -> int` qu'elle lève ou non une exception.

> *Danger 116* (D-O-116) : *Les types des fonctions ne fournissent aucune indication sur les exceptions présentes dans leur corps.*

> *Danger 117* (D-O-117) : *Le rattrapage exhaustif multiplie les chemins d'exécution possibles.*
> Cela ne facilite pas la relecture du code. ■

Encapsulation et exceptions

Le type des exceptions `exn` est singulier en OCaml. Son comportement vis-à-vis de l'encapsulation ouvre une possibilité de fuite d'information. En temps normal un type abstrait cache la structure de ses valeurs, qui ne sont accessibles que par les fonctions du module du type abstrait (cf. section 1.1.3). Cependant il est possible de définir une exception avec un argument `exception Exc of t` où `t` est un type abstrait.

Cette exception peut ensuite être utilisée pour contourner l'encapsulation de plusieurs manières :

- Une exception levée mais jamais rattrapée peut provoquer l'affichage (partiel ou complet) de l'argument.
- Si l'exception est passée en argument d'une fonction du module `Printexc` pour en obtenir une représentation textuelle (`Printexc.to_string (Exc v)`).

– En utilisant l'égalité et la comparaison avec une autre exception de même nom. (cf. danger D-O-124).

Voici un exemple d'exposition de la `cle_privee` utilisant le mécanisme des exceptions.

```
1  module Confidentiel :
2    sig
3      type cle_privee
4      val cle : cle_privee
5    end =
6    struct
7      type cle_privee = string
8      let cle = "secret"
9    end;;
10 module Confidentiel : sig type cle_privee val cle : ↵
       cle_privee end
11 module Malveillant =
12   struct
13     exception Exc of Confidentiel.cle_privee
14     let cle_volee =
15       Printexc.to_string (Exc Confidentiel.cle)
16   end;;
17 module Malveillant : sig exception Exc of ↵
       Confidentiel.cle_privee val cle_volee : string end
18 Malveillant.cle_volee;;
19 - : string = "Malveillant.Exc(\"secret\")"
```

Dans le code précédent, le module `Malveillant` peut être caché dans une bibliothèque, par exemple sous la forme d'un foncteur qui prendrait le module `Confidentiel` en argument.

Danger 118 (D-O-118) : *Contournement de l'encapsulation avec une exception.*
Il est possible de contourner l'encapsulation d'un type abstrait en utilisant une exception ayant en argument une valeur de ce type.■

Recommandation 119 (R-O-119) : *Vérifier les arguments d'exceptions de bibliothèques externes.*
Quand on utilise une bibliothèque ne provenant pas d'une source de confiance, il est important d'examiner son code source pour

vérifier que les définitions d'exception ne prennent pas de types abstraits en arguments. ■

1.3 Traits modulaires

Les modules permettent de placer des barrières d'abstraction entre différentes parties du texte source. Celles-ci peuvent servir à refléter l'architecture du système et apporter ainsi des garanties de cloisonnement entre sous-systèmes. Elles peuvent aussi être utilisées pour restreindre la manipulation de données, les confiner et permettre le respect d'invariants. Le système de modules de OCaml s'appuie sur trois notions, les modules et leurs types, les interfaces de module, et les foncteurs.

1.3.1 Modules et types de module

Un module OCaml est une suite d'importations de modules et de déclarations de sous-modules, de types, de valeurs, d'exceptions et de classes. Un fichier `fich.ml` est considéré par défaut comme un module portant le nom `Fich`. Le texte source en cours d'élaboration est donc considéré comme le module courant.

Le système d'inférence de types attribue un type aux modules. Ce type porte le même nom que le module et est défini par la suite des identificateurs associés à leurs types (délimitée par `sig` et `end`).

```
1 module M =
2   struct
3     type t = int
4     let v = 1
5   end;;
6 module M : sig type t = int val v : int end
```

Les identificateurs d'un module sont accessibles grâce à la notation pointée (nom du module.identificateur). L'inférence de types sait gérer la notation pointée et le typage garantit la cohérence des utilisations de ces deux premières notions.

Intérêt 120 (I-O-120) : *Le système de modules de OCaml conserve les garanties du typage.*
La construction `module` permet de bénéficier les avantages du typage et de la compilation séparée. ■

1.3.2 Encapsulation et interfaces de module

La seconde notion fournie par le système de modules de OCaml est celle d'**interface de module**, qui permet de mettre en œuvre l'**encapsulation**. Une interface de module est obtenue à partir du type du module en enlevant les définitions et en enlevant le nom et le type de tous les éléments que l'on ne souhaite pas exporter. On peut en particulier cacher complètement un type en ne mentionnant ni son nom ni sa définition, l'abstraire en ne mentionnant que son nom ou le laisser manifeste en donnant son nom et sa définition. Ce mécanisme d'encapsulation permet également de masquer les abréviations de type.

L'interface par défaut d'un module est son type. Une interface est introduite par la construction syntaxique `module type` comme présenté dans l'exemple suivant, où le type `t` est encapsulé :

```
1  module type MT =
2      sig
3        type t
4        val v : t
5      end;;
6  module type MT = sig type t val v : t end
7  module M : MT =
8    struct
9      type t = int
10     let v = 1
11   end;;
12 module M : MT
```

Intérêt 121 (I-O-121) : *Le typage assure le respect de l'encapsulation des modules.*
L'encapsulation et l'abstraction de types d'un module se font par association d'une interface au module. Le typage garantit le respect de cette barrière d'abstraction, sauf dans quelques cas dûment identifiés détaillés ci-après. ■

Recommandation 122 (R-O-122) : *Utiliser les modules pour protéger les données sensibles.*
L'encapsulation dans un module offre une garantie très forte sur la confidentialité/intégrité des données encapsulées. ■

Recommandation 123 (R-O-123) : *Utiliser les modules pour contrôler les propriétés des représentations de données.*
Le système de modules permet de gérer très finement l'encapsulation de données et l'abstraction des types (voir le manuel de référence pour des exemples). Il permet de tirer parti de la généricité tout en conservant l'encapsulation. Cela permet de maintenir facilement des invariants sur les représentations des données. ■

Contournement de l'encapsulation

L'utilisation d'exceptions permet le contournement de l'encapsulation. Dans l'exemple ci-dessous, une utilisation d'exceptions combinée à de la redéfinition (lignes 1 et 5) permet d'obtenir des informations sur les valeurs du type abstrait `M.t` de l'exemple précédent.

Ainsi les essais des lignes 7–12 permettent de conclure que le type `M.t` est représenté par des entiers machine et que la valeur `M.v` est représentée par l'entier `1`.

```
1  exception Exc of M.t;;
2  exception Exc of M.t
3  let e_secret = Exc (M.v);;
4  val e_secret : exn = Exc <abstr>
5  exception Exc of int;;
6  exception Exc of int
7  e_secret > Exc (0);;
8  - : bool = true
9  e_secret > Exc (100);;
10 - : bool = false
11 e_secret = Exc (1);;
12 - : bool = true
```

Danger 124 (D-O-124) : *L'encapsulation peut être contournée.*
L'encapsulation peut être contournée en utilisant les techniques suivantes :
- par définition d'une exception et utilisation de l'égalité, comme dans l'exemple ci-dessus.
- par observation en utilisant les primitives d'égalité, de comparaison ou de hachage comme décrit dans les dangers D-O-64 et D-O-66.

– par le contournement du typage, comme décrit dans la section 1.1.3. ■

Recommandation 125 (R-O-125) : *Vérifier l'absence de contournements de l'encapsulation.*

Il est possible de rechercher ces contournements par relecture de code, sauf pour le danger D-O-64 pour lequel on peut empêcher le contournement en suivant la recommandation R-O-65. ■

1.3.3 Foncteurs

La troisième notion fournie par le système de modules de OCaml est celle de **foncteur**. Un foncteur (ou encore un module paramétré) est une fonction prenant en paramètre un module et retournant un module. Ce mécanisme permet d'exprimer la généricité d'un développement et donc de factoriser non seulement l'écriture du code mais aussi son analyse et sa maintenance. Un foncteur peut exposer, en nommant dans son résultat, certains éléments du paramètre. L'inférence de types est faite également sur les foncteurs et le typage garantit que l'application d'un foncteur à un module est cohérente. Une interface de foncteur peut être associée à un foncteur de manière à encapsuler ou abstraire certains éléments de tous les modules résultat des applications du foncteur. Cette interface est une fonction de l'interface du paramètre vers l'interface du résultat et peut elle aussi exposer, en les nommant certains éléments du résultat.

Intérêt 126 (I-O-126) : *L'utilisation de foncteurs permet d'exprimer la généricité d'une application.*

Recommandation 127 (R-O-127) : *Utiliser les foncteurs pour factoriser du code.*
Les foncteurs facilitent la structuration des programmes, permettent d'éviter la duplication de code et facilitent donc sa relecture et sa maintenance. ■

1.3.4 Importation, nommage et masquage

Les éléments figurant dans l'interface d'un module (connue du compilateur) peuvent être utilisés en les désignant avec la notation pointée. La directive `open`

permet d'utiliser directement les identificateurs mentionnés dans l'interface du module. Cependant, beaucoup de modules utilisent les mêmes noms : typiquement `type t` est un nom du type que l'on retrouve fréquemment dans les modules. L'utilisation des noms qualifiés permet la documentation et facilite la compréhension du texte source.

Le système de modules de OCaml ne propose pas d'héritage de modules. Cependant, les liaisons d'un module (ou d'un type de module) peuvent être dupliquées dans un autre module (ou un autre type de module) grâce à la construction `include`.

Dès lors que les noms d'un module sont rendus visibles par une utilisation de `open` ou de `include`, ils peuvent être masqués dans le texte source, avec les inconvénients du masquage déjà évoqués (cf. R-O-11).

> *Recommandation 128* (R-O-128) : *Restreindre l'utilisation de* `open`*.* Cela permet d'éviter le masquage d'identificateurs et de ne pas diminuer la lisibilité du code. ■

> *Recommandation 129* (R-O-129) : *Éviter de masquer les noms d'un module.*

1.3.5 Le module Pervasives

Le module `Pervasives` qui implante toutes les fonctions prédéfinies d'OCaml est importé par défaut.

> *Recommandation 130* (R-O-130) : *Ne jamais masquer les définitions de* `Pervasives`*.*

L'exemple suivant montre la définition d'un opérateur d'addition dans un module (`ListArith.+` défini en ligne 3). L'importation de ce module (ligne 7) masque ainsi l'opérateur d'addition de `Pervasives` menant dans notre exemple à de possibles erreurs de typage (ligne 8–10) mais pouvant surtout mener à un détournement d'opérateurs ou de fonctions de la bibliothèque standard. Comme le montre l'inférence du type du module `Extension` (ligne 16), les modules OCaml ne permettent pas la liaison retardée (voir la section 1.4.3 pour une définition de cette notion) : la valeur `un` reste bien typée (l'opérateur `+` de sa définition est bien celui défini dans le module `ListArith`) alors que l'opérateur `+` introduit dans le module `Extension` a retrouvé le type habituel `int -> int -> int`, il masque le `+` de `ListArith`.

```
module ListArith =
  struct
    let (+) x y = x :: y
    let un = 1 + []
  end;;
module ListArith : sig val ( + ) : 'a -> 'a list -> 'a ↵
   list val un : int list end
open ListArith;;
1+1;;
Error: This expression has type int but an expression ↵
   was expected of type
         int list
module Extension =
  struct
    include ListArith
    let (+) x y = Pervasives.(+) x y
  end;;
 module Extension : sig val un : int list val ( + ) : ↵
   int -> int -> int end
```

1.4 Traits objet

OCaml est construit par adjonction de traits objet au langage Caml. La notion de type objet adoptée conduit à une intégration complète des paradigmes fonctionnel, impératif, modulaire et objet dans le langage de types de OCaml. On dispose ainsi d'un typage fort, statique, avec inférence, sous-typage, polymorphisme paramétrique et sa déclinaison en polymorphisme d'inclusion. Cela permet d'assurer la cohérence globale d'un développement utilisant les quatre paradigmes.

Bien que proposant comme eux héritage multiple, liaison retardée (voir la section 1.4.3), le langage objet de OCaml diffère notablement des langages objet les plus répandus, en particulier par la différence existant entre types d'objet et types de classe, par la signification non usuelle du marqueur `private` et par le mécanisme d'encapsulation par les interfaces de classe, assez analogues aux interfaces de module. Ces points seront détaillés dans la suite de cette section. Le confinement de données y sera également étudié.

1.4.1 Objets

Un **objet** en OCaml est un ensemble de champs qui sont soit des variables d'instance, soit des méthodes. Bien que les objets soient des groupes de valeurs un peu analogues à des enregistrements, il faut noter que le filtrage n'est pas autorisé sur les objets.

La création des objets peut se faire à partir de la définition d'une classe en utilisant la construction `new` de **création d'objet**. Un objet peut aussi se définir directement sans faire référence à aucune classe. Sa classe est alors dite **anonyme**.

Méthodes

Les **méthodes** (`method`) sont des valeurs de n'importe quel type, a priori visibles hors de la classe et de ses sous-classes. Les méthodes peuvent être polymorphes, à condition d'écrire explicitement le type polymorphe dans leur définition. La définition d'une méthode peut utiliser directement des variables d'instance et des méthodes de l'objet, pour ces dernières à condition qu'un nom générique de l'objet soit fourni dans sa définition de manière à utiliser la notation `nom_objet#nom_methode` de l'appel de méthode.

> *Intérêt 131* (I-O-131) : *Les objets ne possèdent pas de champs prédéfinis par défaut.*
> Contrairement à certains langages objet, les objets de OCaml ne sont pas obligatoirement issus d'une même classe racine et ne possèdent donc aucun champ imposé par le système de construction des objets. ■

Une méthode peut être réservée à une classe et à ses sous-classes en faisant précéder son nom par le marqueur `private`, dont les différentes utilisations sont expliquées dans la section 1.4.4. Une méthode non marquée `private` est dite publique.

Une méthode peut être déclarée et sa définition peut être différée, en faisant précéder son nom par le marqueur `virtual`, dont le rôle est expliqué plus loin (voir section 1.4.2).

Variables d'instance

Les **variables d'instance** sont introduites par `val` et peuvent être de n'importe quel type (voir section 1.4.4), y compris fonctionnel ou mutable. La valeur

d'une variable d'instance est soit donnée directement soit par l'intermédiaire d'une paramétrisation présentée plus loin. Une variable d'instance n'est visible que par les méthodes de la classe ou de ses sous-classes mentionnant directement son nom (la notation **`nom_objet#nom_methode`** n'est pas utilisable pour les variables d'instance).

> *Intérêt 132* (I-O-132) : *Les variables d'instance permettent l'encapsulation.*
> En particulier, si une variable d'instance est une valeur fonctionnelle, elle ne peut être appelée que dans le corps des méthodes, ce qui localise très précisément son utilisation. ■

> *Danger 133* (D-O-133) : *L'encapsulation par la seule utilisation de variables d'instance est insuffisante.*
> Il faut s'assurer qu'aucune méthode de la classe ou de ses sous-classes ne fait fuir la valeur d'une variable d'instance. Voir la section 1.4.4 traitant le contrôle des accès pour des traitements plus complets. ■

Comparaison entre objets

Un objet n'est égal qu'à lui-même. Le résultat d'une comparaison entre deux objets est imprédictible. Dans l'exemple suivant, les objets créés à partir de la classe `c` sont tous identiques (lignes 6 et 8) mais cela n'est pas révélé par les tests de comparaison (lignes 10 et 12).

```
1  class c = object
2      val x = 1
3      method  b = 2
4    end ;;
5  class c : object val x : int method b : int end
6  let un_obj = new c;;
7  val un_obj : c = <obj>
8  let un_autre = new c;;
9  val un_autre : c = <obj>
10 un_autre = un_obj;;
11 - : bool = false
12 un_autre < un_obj;;
13 - : bool = false
```

Types objet

Le typage des objets est pleinement intégré au mécanisme de typage décrit dans la section 1.1.3 et offre donc les mêmes garanties.

> *Intérêt 134* (I-O-134) : *L'utilisation des objets est contrôlée par le typage.*

Le langage de types de OCaml propose deux notions de type relatifs aux objets.

Les **types objet fermés** sont constitués uniquement de noms associés à des types (noté `<n1 : t1; n2 : t2>`). Un objet a pour type un type fermé `t` si et seulement si il possède des méthodes publiques de même nom et de même type que celles figurant dans `t` et s'il n'en possède pas d'autres publiques. L'exemple suivant définit deux objets `o1` et `o2` (lignes 6 et 8–12) de même type `< m1 : string; m2 : int >` (dont le type `c1` est une abréviation). Une fonction prenant un paramètre de type `< m1 : string; m2 : int >` (ligne 14) pourra être appliquée indifféremment aux objets `o1` et `o2` (lignes 16– 19).

```
1  class c1 = object
2      method m1 = "abc"
3      method m2 = 3
4    end;;
5  class c1 : object method m1 : string method m2 : int end
6  let o1 = new c1;;
7  val o1 : c1 = <obj>
8  let o2 = object
9    val x = 4.3
10   method m1 = "abc"
11   method m2 = 3
12 end;;
13 val o2 : < m1 : string; m2 : int > = <obj>
14 let filtre (o : < m1 : string; m2 : int >) = o#m2>0;;
15 val filtre : < m1 : string; m2 : int > -> bool = <fun>
16 filtre o1;;
17 - : bool = true
18 filtre o2;;
19 - : bool = true
```

Deux objets créés indépendamment (via deux classes différentes par exemple) auront le même type s'ils ont les mêmes méthodes publiques, munies des mêmes

types (comme c'est le cas dans notre exemple). Le typage des objets de OCaml est donc un **typage structurel**. Le type d'un objet diffère du type de sa classe (vu plus loin) mais le nom de la classe est utilisé en tant qu'abréviation de type pour le désigner (voir la section 1.1.3 portant sur les abréviations de types).

> *Danger 135* (D-O-135) : *Pas de distinction grâce aux types entre objets provenant de classes différentes mais ayant des méthodes publiques de même nom et de même type.*
> Dans ce cas, les abréviations des deux types objet fournies par le nom de leur classe respective sont différentes mais les types eux-mêmes sont identiques. ■

La distinction entre famille d'objets ne se faisant pas par leur appartenance à une classe mais par leurs méthodes publiques, il est conseillé de leur donner des noms distinctifs et de se servir de la relation entre les types (cf. section 1.1.3) pour distinguer les classes d'objets à partir du type d'une méthode (voir le discussion portant sur l'encapsulation par les interfaces de module en section 1.4.4). Cela ne peut qu'augmenter la lisibilité du code.

> *Recommandation 136* (R-O-136) : *Utiliser les noms de méthode ou leurs types pour différencier des familles d'objets.*

L'exemple suivant illustre différentes manières de définir des familles d'objets ayant une méthode représentant leur masse. Pour la première classe `c1`, la masse est représentée par une méthode `m` de type `float` alors que pour la deuxième le nom de méthode `masse` est plus explicite et portera donc moins à confusion. Pour la classe `c3`, un type `masse` est défini pour différencier les flottants représentant une masse des autres flottants (voir le paragraphe sur les abréviations et définitions de type en section 1.1.3) et ainsi différencier la famille d'objets par le nom et le type de la méthode `masse`. L'encapsulation par les interfaces de modules (cf. section 1.4.4) permet d'augmenter la différenciation et le contrôle d'une famille d'objets.

```
1 class c1 = object method m = 34.0 end;;
2 class c1 : object method m : float end
3 class c2 = object method masse = 34.0 end;;
4 class c2 : object method masse : float end
5 type masse = Masse of float;;
6 type masse = Masse of float
7 class c3 = object method masse = Masse 34.0 end;;
8 class c3 : object method masse : masse end
```

Un **type objet ouvert** est construit comme un type fermé en ajoutant une variable de type (appelée aussi variable de rangée) représentée par une ellipse `..`. Cette variable de type peut être instanciée par un ensemble quelconque de noms de méthodes associées à leur type. Les types ouverts sont utilisés pour typer des fonctions prenant des objets en paramètre et réalisant des appels de méthode de ces objets dans leur corps. De telles fonctions sont applicables à tout objet possédant les méthodes requises par le corps de la fonction.

> *Intérêt 137* (I-O-137) : *Les types ouverts permettent d'exprimer la généricité d'une fonction sur des objets.*

La fonction `poids` de l'exemple suivant calcul le poids de tout objet ayant un méthode `masse` du type `masse` introduit dans l'exemple précédent.

```
type poids = Poids of float;;
let g = 9.81;;
let poids o =
  match o#masse with
    Masse masse -> Poids (m *. g);;
val poids : < masse : masse; .. > -> poids = <fun>
```

Sous-typage et polymorphisme d'inclusion

Un type objet `t1` est un **sous-type** d'un type objet `t2` si `t1` est égal à `t2`, ou bien si toute méthode m de `t2` figure dans `t1`, avec un type dans `t1` qui est lui-même un sous type de m dans `t2` (autrement dit, `t1` est plus spécialisé que `t2`). Le sous-typage permet d'établir des coercitions explicites qui consistent à oublier des champs. Il n'existe aucune coercition implicite en OCaml.

Les fonctions ayant un paramètre de type ouvert peuvent être appliquées à tout objet dont le type fermé est un sous-type du type ouvert. Ce mécanisme de typage est appelé **polymorphisme d'inclusion** et est parfaitement compatible avec le polymorphisme paramétrique. Il est à la base de l'intégration des approches fonctionnelle et objet dans OCaml.

L'exemple suivant illustre ces différents points. Les fonctions `f` et `g` sont définies sans qu'aucune classe ni objet n'aient été déjà introduits. La fonction `f` est polymorphe par polymorphisme d'inclusion (variable de rangée dénotée par `..` dans son type) et par polymorphisme paramétrique (variable de type `'a` dans son

type). Elle est donc applicable à tout objet dont le type est à la fois un sous-type et une instanciation de son type de départ. La classe `c1` est ensuite introduite et permet, avec l'utilisation de la marque `private` et de l'application de `f` à `o1` de montrer la différence entre le type de la classe (qui contient la méthode privée `p`, ligne 9) et le type fermé de l'objet (qui lui ne la contient pas, ligne 13) créé via cette classe. L'objet `o2` définissant des méthodes de même nom, type et visibilité que la classe `c1` montre explicitement que les types d'objet n'incluent que les méthodes publiques (ligne 22). Les applications de `f` et `g` à l'objet `o3` illustrent le sous-typage (ligne 34).

```
1  let f x = x#p;;
2  val f : < p : 'a; .. > -> 'a = <fun>
3  let g x = (x#m 10) + 1;;
4  val g : < m : int -> int; .. > -> int = <fun>
5  class c1 = object
6      method private p = 1
7      method m x = 2 * x
8    end;;
9  class c1 : object method m : int -> int method private ↵
       p : int end
10 let o1 = new c1;;
11 val o1 : c1 = <obj>
12 f o1;;
13 Error: This expression has type c1 but an expression ↵
       was expected of type < p : 'a; .. >
14        The first object type has no method p
15 g o1;;
16 - : int = 21
17 let o2 =
18   object
19     method private p = 3
20     method m = 4 * x
21   end;;
22 val o2 : < m : int -> int > = <obj>
23 f o2;;
24 Error: This expression has type < m : int > but an ↵
       expression was expected of type < p : 'a; .. >
25        The first object type has no method p
26 g o2;;
27 - : int = 41
```

```
28 let o3 =
29   object
30     method p x = 100 + x
31     method m x = 6 * x
32   end;;
33 val o3 : < m : int -> int; p : int -> int > = <obj>
34 (f o3) (g o3);;
35 - : int = 161
```

Intérêt 138 (I-O-138) : *Types ouverts, types fermés et sous-typage permettent de définir des traitements génériques d'objets.*
Il est ainsi possible de définir un traitement applicable à une famille d'objets partageant des noms de méthodes et en particulier, de définir une fonction s'appliquant à tous les objets d'une classe et de ses sous-classes, même non encore construites. Cela favorise le partage de code et sa maintenance. ■

Danger 139 (D-O-139) : *Le contrôle des applications des fonctions ayant un type ouvert peut être difficile.*
On peut accidentellement appliquer une fonction à un objet dont le type est un sous-type du type de départ de la fonction. ■

Recommandation 140 (R-O-140) : *Ne définir une fonction ayant un type ouvert que si nécessaire.*
C'est la spécification de la fonction qui doit guider sa définition et permettre de déterminer le compromis à faire entre la généricité de la fonction et le danger lié à la variété de ses applications. ■

1.4.2 Classes

Une classe est un générateur d'objets. Sa définition introduit donc des variables d'instance et des méthodes. Sa définition introduit également la définition de son **type**, qui porte le même nom qu'elle. Le type d'une classe est inféré, il est constitué des noms des variables d'instance et des méthodes, accompagnés de leur type et de leurs marqueurs `virtual` ou `private`. La lecture du type de la classe permet donc de contrôler l'existence des marqueurs.

Intérêt 141 (I-O-141) : *Les marques sur les méthodes figurent dans le type de la classe.*

La définition de la classe introduit également le type (fermé) des objets de la classe, qui diffère de celui de la classe par l'absence de la mention des variables d'instance et la seule présence des méthodes publiques. Le nom de la classe est une abréviation pour le type des objets produits par cette classe. Ce choix peut complique la relecture du code puisque le nom de la classe est aussi celui son type. Il est important de bien distinguer entre le type des objets de la classe (ligne 4), le type de la classe (ligne 2) et la classe elle-même (ligne 1).

```
1 class c = object method m = 10 end;;
2 class c : object method m : int end
3 let o = new c;;
4 val o : c = <obj>
```

Danger 142 (D-O-142) : *Risque de confusion entre le type de la classe et le type de ses objets.*

OCaml n'offre ni variables ni méthodes de classe. Un comportement similaire peut être obtenu en utilisant des fonctions prenant en paramètre un objet de la classe (cf. point précédent).

Classes paramétrées

Une **classe** peut être **paramétrée** par des paramètres de valeur ou de type. Les paramètres de valeur servent à déterminer les valeurs initiales des variables d'instance mutables ou sont utilisés dans le corps des méthodes. Dans ce cas, la classe se comporte comme une fonction qui attend des paramètres supplémentaires à la création de l'objet (`let o = new c v`). Les paramètres de la classe ne sont pas accessibles en dehors de la classe.

Intérêt 143 (I-O-143) : *Il est impossible de créer un objet partiellement initialisé.*
Soit la classe utilisée n'est pas paramétrée et ses champs sont tous connus à sa déclaration. Soit la classe est paramétrée et la création ne peut se faire que lorsque la classe a été appliquée à tous ses paramètres de valeur. ■

Les **initialiseurs** sont des méthodes anonymes dont les corps sont exécutés immédiatement après initialisation des variables d'instance et peuvent donc utiliser ces valeurs.

Recommandation 144 (R-O-144) : *Contrôler les initialiseurs.*
Les initialiseurs d'une classe `c` sont exécutés à chaque appel de la construction `new c`. On peut ainsi faire fuir des informations sur ces variables. Comme les initialiseurs sont anonymes, leur présence n'est pas visible dans le type et ils ne sont pas appelés explicitement. ■

Classes et méthodes virtuelles

La marque de **champs virtuels** (`virtual`) permet de déclarer une variable d'instance ou une méthode en donnant uniquement son type. La classe doit alors être marquée virtuelle et il est impossible de fabriquer des objets de cette classe.

Intérêt 145 (I-O-145) : *On ne peut créer des objets d'une classe que si elle est complètement définie.*

1.4.3 Héritage et redéfinition

Héritage multiple

OCaml propose l'**héritage multiple**. Une classe définie à l'aide d'une clause d'héritage possède tous les champs de cette classe. Si une méthode est définie dans plusieurs classes héritées, alors l'héritage n'est possible que si ces méthodes ont le même type. La définition utilisée dans la sous-classe est celle de la classe héritée en dernier. Si une variable d'instance est définie dans plusieurs classes héritées, alors la définition utilisée dans la sous-classe est celle de la classe héritée en dernier, quels que soient les types de la variable d'instance dans les classes héritées.

L'héritage multiple est contrôlé statiquement. Les exemples suivants montre le contrôle de l'héritage. La classe `cd` ne peut pas être construite à cause du conflit de types sur la méthode `f` (ligne 6).

```
1 class c = object method f = 2 method g = 10 end;;
2 class c : object method f : int method g : int end
3 class d = object method f = true end;;
4 class d : object method f : bool end
5 class cd = object inherit c inherit d end;;
6 Error: The method f has type int but is expected to ↵
    have type bool
```

```
class e = object method f = 25 end;;
class e : object method f : int end
class ce = object inherit c inherit e end;;
class ce : object method f : int method g : int end
let o= new ce;;
val o : ce = <obj>
(o#f,o#g);;
- : int * int = (25, 10)
```

Intérêt 146 (I-O-146) : *Héritage contrôlé statiquement.*
Le compilateur vérifie que des variables d'instance et des méthodes de même nom héritées de plusieurs classes ont le même type. ■

Un champ (variable d'instance ou méthode) peut être **redéfini** dans une sous-classe, à condition que son type soit conservé. Une méthode peut également être réintroduite dans une sous-classe et sa nouvelle définition différée en lui associant le marqueur `virtual` : l'ancienne définition du champ devient dès lors inaccessible. De plus, toutes les occurrences du nom de cette méthode dans les méthodes précédemment introduites seront liées à la nouvelle définition : ce mécanisme est appelé **liaison retardée**. Le typage des redéfinitions est fait à la compilation, le lien entre le nom et la valeur est créé à l'exécution. L'exemple suivant illustre la liaison tardive.

```
class c1 = object (s)
  val x = 1
  method m = 1
  method m_x = x
  method m_m = s#m
end;;
class c1 : object val x : int method m : int method ↵
   m_m : int method m_x : int end
class c2 = object
  inherit c1
  val x = 2
  method m = 2
end;;
Warning 13: the instance variable x is overridden.
The behaviour changed in ocaml 3.10 (previous ↵
   behaviour was hiding.)
Warning 7: the method m is overridden.
```

```
16 class c2 : object val x : int method m : int method ↵
     m_m : int method m_x : int end
17 let o1 = new c1 and o2 = new c2;;
18 val o1 : c1 = <obj>
19 val o2 : c2 = <obj>
20 (o1#m_x, o1#m_m);;
21 - : int * int = (1, 1)
22 (o2#m_x, o2#m_m);;
23 - : int * int = (2, 2)
```

Intérêt 147 (I-O-147) : *Redéfinition de méthodes contrôlée par le typage.*

Le mécanisme d'héritage et la redéfinition fournissent des outils très puissants pour construire progressivement une application de grande envergure. La redéfinition permet d'adapter une méthode aux besoins d'un nœud particulier de la hiérarchie. Ils favorisent la réutilisation de code et la lisibilité de l'architecture. Héritage et redéfinition facilitent le développement en permettant le partage de code. Encore faut-il rester mesuré dans leur utilisation et éviter de construire des hiérarchies de classe comportant de très nombreux niveaux d'héritage.

Recommandation 148 (R-O-148) : *Limiter l'héritage à une profondeur raisonnable.*

Comme l'ont montré les exemples, le compilateur OCaml émet des avertissements de compilation lors de la redéfinition d'une variable d'instance (avertissement numéro 13 activé par défaut) ou d'une méthode (avertissement numéro 7 activé par l'option de compilation `-w +7`). Mais, dans les deux cas, il accepte la redéfinition, si elle est correctement typée. Ces avertissements incitent à utiliser les mots-clés `val!` et `method!` pour redéfinir une variable d'instance ou une méthode, ce qui permet de rendre explicite dans la syntaxe une redéfinition.

Recommandation 149 (R-O-149) : *Utiliser le contrôle des redéfinitions par le compilateur.*

Les notions d'héritage et de sous typage ne doivent pas être confondues. Rappelons que le type d'un objet est défini par l'ensemble des types de ses méthodes (visibles, voir section suivante pour ce point) et que le sous-typage est défini par une sorte d'inclusion de type. Si une classe `c` hérite d'une classe `c1`, alors le type des objets de `c` est un sous-type de celui des objets de `c1`.

1.4.4 Contrôle d'accès

Cette partie rassemble un certain nombre de points relatifs au contrôle des accès aux méthodes et variables d'instance des objets.

Méthodes privées

Une méthode `m` peut être marquée `private`, étiquette qui n'a pas la même signification qu'en Java ou C++. Cela signifie que l'expression `s#m` ne peut figurer que dans le corps des méthodes d'une classe ayant nommé `s` l'objet générique de la classe. Elle ne figure pas dans le type de l'objet. Elle reste (par défaut) exposée dans le type de la classe.

Une méthode marquée `private` peut être héritée. Elle perd sa marque si elle est redéfinie ou marquée `virtual` afin de préparer une redéfinition ultérieure. Dans l'exemple suivant, la marque sur `p1` a disparu dans le type de la classe `c2` (ligne 10). La définition de la méthode `p2` reste `private` dans la classe héritée.

```
1  class c1 = object
2    method private p1 = 1
3    method private p2 = 5
4  end;;
5  class c1 : object method private p1 : int method ↵
       private p2 : int end
6  class c2 = object
7    inherit c1
8    method! p1 = 25
9  end;;
10 class c2 : object method p1 : int method private p2 : ↵
       int end
11 let o2 = new c2;;
12 val o2 : c2 = <obj>
13 o2#p1;;
14 - : int = 25
15 o2#p2;;
16 Error: This expression has type c2
17        It has no method p2
```

> *Recommandation 150* (R-O-150) : *Ne pas utiliser le marqueur* `private` *comme mécanisme de sécurité.*

Encapsulation par les interfaces de classe

Rappelons que les classes possèdent un type qui indique quelles sont les variables d'instance et les méthodes déclarées par la classe avec leur type et leurs marqueurs éventuels (`private`, `virtual`). Une **interface de classe** est un type de classe qui ne mentionne que les variables d'instance et les méthodes qu'on souhaite exporter. Par défaut, l'interface d'une classe est son type de classe. Une interface de classe peut être construite en utilisant le mot-clé `class type` et en listant les noms et types des méthodes et variables d'instance qui doivent rester visibles à l'extérieur de la classe. La définition d'une classe peut être contrainte par une interface de classe. Les méthodes qui sont cachées par l'interface doivent être définies avec le mot-clé `private`. Elles ne peuvent pas être marquées `virtual`.

Une méthode ne figurant pas dans l'interface d'une classe n'est pas héritée, comme le montre l'exemple suivant. La définition de la classe `d1` est rejetée car `m1` n'est pas disponible. Cela permet d'ailleurs d'introduire une nouvelle méthode nommée aussi `m1` dans la classe `d2` ayant un type différent. Il ne s'agit donc pas d'une redéfinition du `m1` de `c` mais de l'ajout d'une nouvelle méthode.

```
1  class type t = object
2    method m2 : int
3  end;;
4  class type t = object method m2 : int end
5  class c : t = object (s)
6    method private m1 = 1
7    method m2 = s#m1
8  end;;
9  class c : t
10 class d1 = object (s)
11   inherit c
12   method m3 = s#m1
13 end;;
14 Warning 17: the virtual method m1 is not declared.
15 Error: This class should be virtual. The following ↵
      methods are undefined : m1
16 class d2 = object (s)
17   inherit c
18   method m1 = true
19   method m3 = s#m1
20 end;;
21 class d2 : object method m1 : bool method m2 : int ↵
```

```
  method m3 : bool end
```

Une méthode peut figurer dans l'interface en étant marquée par `private`.

Danger 151 (D-O-151) : *Le marqueur* `private` *ne cache pas la méthode marquée.*
Il faut se rappeler que `private` ne signifie par "masquée". ■

La suppression de son nom dans l'interface de classe cache la méthode au cours de l'héritage de la classe. La marque `private` ne fait que restreindre son utilisation.

Une variable d'instance peut figurer dans une interface de classe. Si elle n'y figure pas, elle est héritée mais n'est pas accessible à une nouvelle méthode. Elle reste en revanche accessible aux méthodes de la classe héritée.

```
1  class type ct = object
2    val x : int
3    method m : int
4  end;;
5  class type ct = object val x : int method m : int end
6  class c1 : ct = object
7    val x = 1
8    val y = 2
9    method m = x + y
10 end;;
11 class c1 : ct
12 class c2 = object
13   inherit c1
14   method m2 = x - y
15 end;;
16 Unbound value y
17 class c3 = object
18   inherit c1
19   method m2 = x + 1
20 end;;
21 class c3 : object val x : int method m : int method m2 ↵
     : int end
22 (new c3)#m;;
23 - : int = 3
24
```

Encapsulation par les interfaces de module

Une classe peut être encapsulée dans un module. Si l'interface de ce module ne la mentionne pas, elle ne pourra pas être héritée. Dans l'exemple suivant, une interface de module est définie et expose un objet `o` (ligne 3) et son type objet (ligne 2). Le module `M` est ensuite défini, il contient une classe (visible dans son type, ligne 10). Le module `X` est obtenu en contraignant les exportations du module `M`. La classe `d1`, qui hérite de `M`, peut être construite. Mais la tentative de construire la classe `d2`, qui hériterait de `X`, échoue.

```
1  module type T = sig
2    type c = < m : int >
3    val o : c
4  end;;
5  module type T = sig type c = < m : int > val o : c end
6  module M = struct
7    class c = object method m = 1 end
8    let o = new c
9  end;;
10 module M : sig class c : object method m : int end val ↵
     o : c end
11 module X = (M : T);;
12 module X : T
13 class d1 = object inherit M.c end;;
14 class d1 : object method m : int end
15 class d2 = object inherit X.c end;;
16 Error: Unbound class X.c
```

1.4.5 Conclusion

> *Intérêt 152* (I-O-152) : *L'utilisation conjointe des modules et des classes fournit une aide importante au développement d'applications de grande envergure.*

En résumé, l'encapsulation peut être réalisée en utilisant à la fois les notions de classe et de module.

1. Définition d'une classe :
   - Les variables d'instance fournissent un premier niveau d'encapsulation, permettant de cacher une donnée, à condition qu'elle ne soit pas par

ailleurs dévoilée par une méthode faisant fuir de l'information sur elle. Cette absence de fuite doit être vérifiée tout au long de la chaîne d'héritage.
- Le marqueur `private` restreint l'utilisation de la méthode à l'objet lui-même mais son existence n'en est pas pour autant rendue invisible à l'héritage.
- Encapsuler une méthode se fait par la définition d'une interface de classe, qui ne la nomme pas. Une méthode ainsi masquée n'est pas héritée.

2. Définition d'un module englobant :
   - Pour empêcher l'héritage d'une classe, il faut l'encapsuler dans un module. Cela permet encore d'exporter un objet de cette classe, si il figure dans l'interface du module.

Cette section montre ensuite que la mise en œuvre de l'encapsulation dans le cadre objet est fort complexe.

Danger 153 (D-O-153) : *Risque de mauvaise compréhension des rôles des marqueurs et des interfaces vis-à-vis de l'encapsulation.*

Recommandation 154 (R-O-154) : *Utiliser le contrôle de visibilité apporté par les interfaces.*
Les interfaces permettent de plus d'exprimer la spécification de la classe et ont donc aussi un rôle de documentation. ■

Recommandation 155 (R-O-155) : *N'utiliser le paradigme objet que pour développer des applications qui le nécessitent vraiment.*
Modules et classes ne se différencient fondamentalement que par l'héritage multiple et la possibilité de liaison retardée offerte par les classes. Si son utilisation n'est pas indispensable pour faciliter le développement d'une application, il est préférable de n'utiliser que les modules. ■

Recommandation 156 (R-O-156) : *Préférer si possible les modules aux classes.*
Bien que les traits objet du langage permettent de contrôler l'accès aux méthodes, ce contrôle est plus facile et plus explicite en utilisant des modules pour encapsuler le code. ■

1.5 Sémantique du langage

Le langage est d'abord décrit par son manuel de référence. De nombreux articles de recherche et des thèses définissent formellement la sémantique des principaux mécanismes du langage et décrivent son schéma de compilation et son ramasse-miettes. Nous indiquons quelques-uns de ces travaux ci-dessous, sans rechercher aucune exhaustivité.

- La sémantique opérationnelle, les systèmes de type, l'inférence de types ont beaucoup été étudiés. Ces travaux concernent en général des sous-langages de OCaml (noyau fonctionnel pur, noyau fonctionnel avec mutables, objets etc.). Aucune sémantique formelle n'existe cependant pour l'ensemble du langage OCaml.
  L'article [Dubois et Ménissier-Morain, 1999] contient une preuve en Coq de la correction et de la complétude de l'algorithme d'inférence de types pour le noyau purement fonctionnel de OCaml, appelé MiniML. L'article [Garrigue, 2004] porte sur le typage des mutables dans un cadre du sous-typage. L'article [Pottier et Rémy, 2005] présente une nouvelle vue de l'inférence de types qui s'appuie sur la production et la résolution de contraintes. La sémantique opérationnelle du sous-ensemble OCaml-Light a été formalisée dans HOL et son système de types a été prouvé correct par rapport à la sémantique [Owens, 2008]. Enfin l'orientation objet et son intégration à un noyau fonctionnel ont été finement étudiées dans [Rémy et Vouillon, 1998, Garrigue et Rémy, 1999, Rémy, 2002]. La sérialisation et la désérialisation des valeurs ont été examinées dans [Henry *et al.*, 2006].
- Le système de modules a été l'objet des publications suivantes [Leroy, 1994, Leroy, 1995, Leroy, 2000].
- La compilation vers du code natif a été formalisée dans [Leroy, 1997]. La ZINC Abstract Machine (ZAM), machine virtuelle de OCaml est présentée dans [Leroy, 1990].
- Le gestionnaire de mémoire est décrit dans les articles [Doligez et Leroy, 1993, Doligez et Gonthier, 1994].

1.6 Compilation

La compilation de OCaml est étudiée en détail dans [MODE-EX, 2011].

Intérêt 157 (I-O-157) : *La compilation apporte des garanties sur la correction du code.*

1.7 Modèles d'exécution et propriétés associées

Le système OCaml propose trois modèles d'exécution du code source : code natif, bytecode et boucle interactive. Ces trois modèles d'exécution sont étudiés en détail dans [MODE-EX, 2011].

Les trois modes d'exécution d'un même code ne diffèrent que par la vitesse d'exécution et la portabilité : le code natif est le plus rapide mais ne fonctionne que sur certaines architectures, le bytecode et le toplevel sont plus lents mais fonctionnent sur toute architecture qui fournit un compilateur C-ANSI.

Intérêt 158 (I-O-158) : *La boucle interactive permet de tester du code rapidement.*

1.8 Support de la programmation concurrente

Le système OCaml fournit une bibliothèque de processus légers pour la programmation concurrente étudiés vis à vis de la sécurité dans [MODE-EX, 2011]. Il est à noter que ces processus (bibliothèque `threads`) ne fonctionnent jamais en parallèle : cette bibliothèque sert à faciliter l'écriture des programmes qui s'expriment naturellement en termes de concurrence, mais elle ne peut pas servir à augmenter les performances de calcul sur les machines parallèles.

OCaml propose deux implémentations de la bibliothèque `threads`, selon les possibilités offertes par le système d'exploitation.

- Si le système d'exploitation fournit une implémentation de processus légers (POSIX 1003.1c ou *Win32 threads*), OCaml l'utilise pour implémenter la bibliothèque `threads`. Elle est alors disponible en bytecode comme en code natif.
- Sous Unix, OCaml propose aussi une implémentation de la bibliothèque `threads` basée sur la machine virtuelle ZAM qui ne nécessite pas de support du système, mais ne fonctionne qu'en mode bytecode.

La bibliothèque `threads` fournit la plupart des fonctionnalités de **POSIX threads**, avec la mémoire partagée, les processus légers, les verrous d'exclusion mutuelle, et les variables de condition.

Elle fournit aussi une interface de plus haut niveau inspirée de Concurrent ML, basée sur la notion d'événements, qui facilite la synchronisation et la communication entre processus légers.

1.9 Interfaçage avec d'autres langages

OCaml dispose d'une interface qui permet d'appeler des fonctions C et vice versa. Cette interface est basée sur les conventions d'appel de code natif du compilateur C. Il est possible d'interfacer d'autres langages, soit directement (s'ils utilisent les conventions d'appel de C), soit indirectement (en passant par leurs interfaces avec C).

L'interfaçage de OCaml avec d'autres langages est étudié en détail dans [MODE-EX, 2011].

1.10 Apport des méthodes formelles et semi-formelles

L'utilisation des méthodes formelles permet de produire une application écrite en OCaml correcte par rapport à sa spécification i.e. dont on a démontré qu'elle satisfait les propriétés énoncées dans sa spécification.

> *Intérêt 159* (I-O-159) : *La compilation effectue des analyses statiques.*
> Le compilateur OCaml effectue par défaut des passes de typage et d'analyse de filtrage qui sont des analyses statiques du texte source. ■

1.10.1 Compilation et analyse statique

Du fait des capacités de vérification du compilateur, des outils externes d'analyse statique n'existent pas pour OCaml comme ils existent pour des langages impératifs (FramaC, Astree et PolySpace pour C) ou objet (ESC Java 2 pour Java et PolySpace pour C++).

Un point fort du langage OCaml, et plus généralement de tous les langages de la famille ML, concerne son système de types et son algorithme d'inférence de types et de typage.

> *Intérêt 160* (I-O-160) : *Le compilateur fait de nombreuses vérifications.*

De nombreux travaux se sont intéressés à l'extension du compilateur par d'autres analyses comme la détection des exceptions non rattrapées, la terminaison des fonctions ou encore le contrôle du flot d'informations.

1.10.2 OCaml et Coq

L'assistant à la preuve Coq permet de produire des programmes OCaml corrects par construction en utilisant le mécanisme d'extraction de cet outil. On peut distinguer deux façons d'extraire du code OCaml avec Coq.

La première consiste à extraire le contenu algorithmique d'une preuve d'une propriété de la forme *pour tout* x *de type* t_1 *il existe un* y *de type* t_2 *tel que* $P(x, y)$. L'extraction fournira un programme qui réalise la propriété démontrée, à savoir une fonction de type $t_1 \rightarrow t_2$ qui à tout x associe un y qui, par construction, vérifie la propriété $P(x, y)$.

La seconde façon d'utiliser l'extraction de Coq consiste à utiliser Coq comme un langage de programmation et donc à écrire un programme qui pourra se traduire en OCaml via l'extraction. L'intérêt d'utiliser Coq est ici que l'on peut démontrer dans Coq des propriétés sur ce programme, la terminaison des fonctions tout particulièrement mais encore bien d'autres propriétés de correction. C'est l'approche suivie par les concepteurs de CompCert [Leroy, 2009], compilateur pour un large sous-ensemble de C, vérifié formellement en Coq. Ainsi le compilateur a été écrit dans le langage de Coq. Il a été prouvé correct au sens où il produit un exécutable qui a la même sémantique que celle du programme source.

1.10.3 OCaml et Focalize

L'atelier de développement [2] Focalize [Ayrault *et al.*, 2008] permet de développer des programmes certifiés. Dans le cadre d'un développement où un fort niveau de confiance est attendu, Focalize permet dès les premières phases du

2. `http:focalize.inria.fr//`

cycle de vie de spécifier les propriétés de sûreté et/ou de sécurité, les exigences fonctionnelles du logiciel attendu, puis de démontrer que les exigences fonctionnelles sont suffisantes pour atteindre les propriétés de sûreté/sécurité requises. Ensuite via son mécanisme de raffinement, Focalize permet de passer progressivement des spécifications à une implantation. Le langage de programmation de l'atelier est très proche du noyau fonctionnel pur de OCaml, étendu par des traits objet tels que l'héritage, la liaison retardée, la redéfinition et par un mécanisme fort d'encapsulation. Les preuves que les propriétés requises par la spécification sont satisfaites par l'implémentation, sont faites avec un outil de preuve semi-automatique, Zenon, qui, s'il réussit, crée une preuve Coq. Le compilateur traduit tout ce qui est opérationnel en code OCaml exécutable, il fournit également un miroir Coq de l'ensemble (spécifications, implantations et preuves) à des fins de vérification à la fois de la cohérence de l'architecture du développement et des preuves qui y sont incluses.

Bibliographie

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

[Ayrault *et al.*, 2008] Ayrault, P., Carlier, M., Delahaye, D., Dubois, C., Doligez, D., Habib, L., Hardin, T., Jaume, M., Morisset, C., Pessaux, F. et Weis, P. (2008). Trusted Software within FoCaL. *In CE&SAR*, pages 142–157, Rennes, France.

[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.

[Doligez et Gonthier, 1994] Doligez, D. et Gonthier, G. (1994). Portable, unobtrusive garbage collection for multiprocessor systems. *In POPL*, pages 70–83.

[Doligez et Leroy, 1993] Doligez, D. et Leroy, X. (1993). A concurrent, generational garbage collector for a multithreaded implementation of ML. *In POPL*, pages 113–123.

[Dubois et Ménissier-Morain, 1999] Dubois, C. et Ménissier-Morain, V. (1999). Certification of a type inference tool for ML : Damas-Milner within Coq. *Journal of Automated Reasoning (JAR)*, 23(3–4):319–346.

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

[Garrigue, 2004] Garrigue, J. (2004). Relaxing the value restriction. *In* Kameyama, Y. et Stuckey, P. J., éditeurs : *FLOPS*, volume 2998 de *Lecture Notes in Computer Science*, pages 196–213. Springer.

[Garrigue et Rémy, 1999] Garrigue, J. et Rémy, D. (1999). Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155(1–2):134–169.

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

[Henry *et al.*, 2006] Henry, G., Mauny, M. et Chailloux, E. (2006). Typer la dé-sérialisation sans sérialiser les types. *In JFLA*, pages 133–146.

[Leroy, 1990] Leroy, X. (1990). The ZINC experiment : an economical implementation of the ML language. Technical report 117, INRIA.

[Leroy, 1994] Leroy, X. (1994). Manifest types, modules, and separate compilation. *In POPL*, pages 109–122.

[Leroy, 1995] Leroy, X. (1995). Applicative functors and fully transparent higher-order modules. *In POPL*, pages 142–153.

[Leroy, 1997] Leroy, X. (1997). The effectiveness of type-based unboxing. *In TIC*.

[Leroy, 2000] Leroy, X. (2000). A modular module system. *Journal of Functional Programming*, 10(3):269–303.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

[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/`.

[MODE-EX, 2011] Modèles d'exécution du 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.

[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.

[Owens, 2008] Owens, S. (2008). A sound semantics for OCamllight. *In* Drossopoulou, S., éditeur : *ESOP*, volume 4960 de *Lecture Notes in Computer Science*, pages 1–15. Springer.

[Pottier et Rémy, 2005] Pottier, F. et Rémy, D. (2005). The essence of ML type inference. *In* Pierce, B. C., éditeur : *Advanced topics in types and programming languages*, chapitre 10, pages 389–489. MIT Press.

[RECOM-OCAML, 2011] Recommandations relatives à l'utilisation du langage OCaml et à l'installation et la configuration des outils associés. Étude de la sécurité intrinsèque des langages fonctionnels (LaFoSec) L3.3.2, ANSSI (2011).

Étude menée par un consortium composé de SafeRiver, CEDRIC, Normation et Oppida.

[Rémy et Vouillon, 1998] RÉMY, D. et VOUILLON, J. (1998). Objective ML : An effective object-oriented extension to ML. *TAPOS*, 4(1):27–50.

[Rémy, 2002] RÉMY, D. (2002). Using, understanding, and unraveling the OCaml language. *In* BARTHE, G., éditeur : *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag.

[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`.

Chapitre 2

Analyse du langage F#

F# [1] est un langage fonctionnel distribué en *open source* par *Microsoft* pour sa plateforme .NET. Il est le premier langage fonctionnel à la ML pour .NET ayant atteint une bonne maturité. Il a été précédé de prototypes comme SML.NET, un compilateur Standard ML (SML) pour .NET.

F# est un langage combinant plusieurs paradigmes de programmation (fonctionnel, objet, impératif). Il est inspiré de OCaml dont il partage un grand nombre de traits. Il s'en distingue toutefois sur plusieurs points dont son mécanisme objet hérité de .NET.

L'étude porte sur la version 2.0 de F# et sa distribution standard, datant de novembre 2010.

Les traits du langages sont analysés dans la section 2.1 pour les traits fonctionnels et le typage, la section 2.2 pour les traits impératifs, la section 2.3 pour les traits modulaires et la section 2.4 pour les traits objet. La section 2.5 porte sur la sémantique du langage. Les sections 2.6 et 2.7 décrivent la compilation et l'exécution de programmes F# tandis que la section 2.8 présente la programmation concurrente en F#. L'interfaçage avec d'autres langages est décrite en section 2.9. Finalement, la section 2.10 présente les mécanismes de sécurité de F# et de la plateforme .NET.

2.1 Traits fonctionnels et typage

F# est en premier lieu un langage fonctionnel statiquement fortement typé. Cette section présente les traits fonctionnels du langage F#, leurs mécanismes et

1. `http://msdn.microsoft.com/en-US/fsharp`

les garanties qu'ils apportent.

Un programme purement fonctionnel F# est une suite de déclarations et d'expressions. Un **environnement** est un ensemble de liaisons d'identificateurs à leurs valeurs. Une expression est construite à partir d'applications de fonctions et est évaluée dans un environnement précisé par la sémantique.

2.1.1 Gestion des noms

Déclaration, portée, initialisation et allocation

Une **déclaration globale** est introduite par la construction `let` et sa portée couvre tout le code source suivant son introduction.

Une **déclaration locale**, introduite par la construction `let x = expr_x in expr`, crée la liaison de `x` à la valeur de `expr_x`, le temps de l'évaluation de l'expression `expr`.

La portée de chaque identificateur est déterminée par la syntaxe du programme (**portée statique**) : l'identificateur ne peut être utilisé que dans un intervalle de code bien défini et facile à identifier par le programmeur. La portée statique est implémentée par la notion de fermeture (cf. section 2.1.2).

La portée statique des identificateurs facilite l'analyse et la maintenance du texte source. Le compilateur garantit que les liaisons respectent la sémantique donnée dans son texte source. En particulier il garantit que toutes les lectures d'un identificateur non-mutable sont incluses dans sa portée.

> *Recommandation 1* (R-F-1) : *La portée de tout identificateur doit être choisie en accord avec son utilisation.*
> Il est recommandé de restreindre la portée des variables pour bénéficier pleinement des garanties apportées par la portée statique. Il est donc souhaitable de n'utiliser des identificateurs globaux que si cela est pleinement justifié et de réduire si possible la portée des variables locales. ■

> *Recommandation 2* (R-F-2) : *Activer la détection automatique des variables locales non utilisées.*
> La portée statique des identificateurs permet la vérification de l'utilisation des variables locales qui peut être activée par l'option `--warnon:1182` du compilateur F#. ■

L'évaluation d'une application d'une fonction (cf. section 2.1.2) ne permet l'accès à aucune liaison de l'environnement d'appel. Par conséquence, lors de

l'appel d'une fonction `f`, définie avant la déclaration d'un identificateur `x`, `f` ne peut pas accéder à la valeur liée à `x`, à moins que celle-ci n'ait été passée en paramètre de `f`.

Dans la portée d'une variable non-mutable `x`, sont incluses toutes les lectures de cette variable (pas d'aliasing possible).

La valeur d'un identificateur `x` ne peut fuir que de l'une des manières suivantes, facilement détectables par lecture du code.

- Appel d'une fonction avec `x` en argument ;
- Affichage de la valeur de `x` ;
- Affectation de la valeur de `x` dans un mutable (cf. section 2.2.1) ;
- Retour de la valeur de `x` à la fin de la portée de `x` ;
- Levée d'une exception avec `x` en argument ;
- Liaison d'un nouvel identificateur avec le contenu de `x` et fuite de la valeur de ce nouvel identificateur.

Si la valeur d'un identificateur ne fuit pas comme décrit précédemment, sa confidentialité est assurée au niveau langage par la portée statique.

> *Recommandation 3* (R-F-3) : *Vérifier l'absence de fuite de la valeur des données confidentielles par relecture du code.*

> *Danger 4* (D-F-4) : *L'utilisation de valeurs mutables par référence diminue la lisibilité apportée par la portée statique.*
> voir section 2.2.1 ■

> *Recommandation 5* (R-F-5) : *Limiter l'utilisation des variables mutables.*

La valeur d'initialisation est la valeur permanente d'un identificateur non mutable. Un identificateur mutable est lié à son adresse, qui n'est pas modifiable et c'est seulement la valeur à cette adresse qui peut évoluer. Une liaison n'est donc pas modifiable (mais elle peut être masquée par un autre identificateur qui a le même nom, voir plus loin). Elle est supprimée à la fin de sa portée. Le GC se charge de la **désallocation** des valeurs qui ne sont plus liées. Les informations de type résultant du typage et la structure des types (cf. section 2.1.3) permettent de déterminer statiquement l'espace à allouer en mémoire pour chaque valeur. La **désallocation** des valeurs devenues inutiles est à la charge du **ramasse-miettes** ou *garbage collector* en anglais (GC). F# ne propose ni `malloc` ni `free` pour manipuler la mémoire car ces manipulations sont réalisées par le GC.

Intérêt 6 (I-F-6) : *Le GC réalise automatiquement les manipulations mémoire.*
Le GC réalise seul l'allocation et la désallocation de la mémoire. Ce mécanisme automatique garantit que l'allocation est correcte vis à vis du type de l'identificateur (en particulier sa taille) et permet un placement des données efficace. Le GC garantit aussi que la désallocation est réalisée une fois et une seule après la dernière utilisation.
Il est ainsi impossible de désallouer prématurément une valeur. Ceci évite la manipulation de valeurs non initialisées ainsi que les erreurs et attaques par *double-free*. ■

Intérêt 7 (I-F-7) : *Déclaration, initialisation et allocation mémoire sont indissociables.*
Il n'est pas possible d'introduire une nouvelle déclaration sans donner la valeur de cet identificateur, y compris lorsqu'il est mutable. Cela évite les oublis d'initialisation. Il n'est pas possible de déclarer un identificateur (même mutable) sans que le GC lui associe un espace mémoire adéquat et lui attribue une valeur initiale. ■

Recommandation 8 (R-F-8) : *L'utilisation d'un type option permet de gérer l'absence de valeur plausible.*
Le type `option` (voir 2.1.3) permet de dénoter l'absence de valeur plausible en fournissant la valeur `None`, les valeurs plausibles étant, elles, dénotées par `Some (..)`. Cela permet par exemple d'éviter un choix arbitraire de valeur initiale, si aucune valeur ne convient, tout en conservant les avantages du typage. ■

Partage de valeurs

Les valeurs non-mutables sont persistantes. La manipulation de valeurs d'un type structuré (cf. section 2.1.3) peut engendrer le partage de certaines sous-structures. Typiquement, si `x = [1; 2]`, alors si `y = x @ x` et `z = 3 :: x`, `y` et `z` partagent la valeur de `x`. Ce partage défini dans le code source par l'utilisation d'un même identificateur est implémenté par le partage physique (sans recopie). F# assure le partage des sous-structures nommées de manière identique.

Intérêt 9 (I-F-9) : *Partage de sous-structures non-mutables.*
Le partage de sous-structures non-mutables permet de diminuer,

sans danger, la place mémoire utilisée. Il est totalement transparent pour l'utilisateur. ■

Danger 10 (D-F-10) : *Partage de sous-structures mutables.*
Le partage d'une sous-structure mutable entre plusieurs structures de données implique que toute modification physique de cette sous-structure sera répercutée sur toutes les structures, par effet de bord. Cela peut conduire à des erreurs d'exécution, parfois difficiles à diagnostiquer (voir section 2.2). ■

Transparence référentielle

Le noyau fonctionnel pur de F# aboutit à du code **référentiellement transparent** pour lequel une expression peut être remplacée par sa valeur sans modifier le comportement du programme. La transparence référentielle permet donc une certaine aisance dans la factorisation et donc la réutilisation de code mais surtout elle facilite grandement l'analyse de code.

Intérêt 11 (I-F-11) : *La transparence référentielle facilite la compréhension et la maintenance du code.*

Danger 12 (D-F-12) : *Les effets de bord cassent la transparence référentielle.*
Les effets de bord par entrées-sorties ou par affectation de valeurs mutables cassent la transparence référentielle. L'exécution du programme est plus difficile à prédire. ■

Recommandation 13 (R-F-13) : *Favoriser la transparence référentielle du code.*
Il est conseillé, en particulier pour les code sensibles, d'éviter ou d'isoler les effets de bord et l'utilisation de données mutables. ■

Surcharge des opérateurs

La **surcharge** des identificateurs globaux et locaux n'est pas autorisée en F#, sauf :
- pour les opérateurs (surcharge décrite dans ce paragraphe) et
- pour les identificateurs de champs de classes (voir section 2.4.1).

L'absence de surcharge des identificateurs globaux et locaux permet à l'inférence de types de rester décidable (cf. section 2.1.3).

Les **opérateurs** sont des éléments lexicaux formés d'un ensemble prédéfini de symboles (+, -, etc.). Le système d'inférence de types essaye d'inférer le type approprié de l'opérateur utilisé pour déterminer quelle fonction appliquer. Ainsi, l'opérateur + correspond à la concaténation de chaînes de caractères (`string -> string -> string`) dans l'expression `"AB" + "CD"` et à l'addition de flottants (`float -> float -> float`) dans l'expression `2.3 + 3.2`.

La **surcharge d'un opérateur** pour un type donné se fait par extension du type (cf. section 2.3.5) en déclarant l'opérateur comme champ statique de ce type. Cette surcharge peut poser des problèmes pour les opérateurs d'égalité et de comparaison. Ce point est discuté en section 2.1.4.

L'exemple suivant illustre la surcharge de l'opérateur + pour un nouveau type `t`. L'opérateur peut ainsi être appliqué à des valeurs de ce type (ligne 6) et toujours être appliqué à des entiers (ligne 6) ou aux valeurs d'autres types pour lesquels l'opérateur est défini. La tentative de surcharge (lignes 10–13) aboutit à un masquage qui n'est pas autorisé[2] pour les identificateurs globaux ou de modules (voir plus loin la section portant sur le masquage).

```
1  type t = A | B with
2      static member (+) (x,y) =
3          match x with
4              | A -> A
5              | B -> y
6  A + B
7  val it : t = A
8  1 + 2
9  val it : int = 3
10 let plus2 (x:int) = x + 2
11 val plus2 : int -> int
12 let plus2 (x:float) = x + 2.0
13 error FS0037: Duplicate definition of value 'plus2'
```

Il est à noter que surcharge ne signifie pas polymorphisme (cf section 2.1.3). Le compilateur effectue donc un choix par défaut quand l'inférence de type ne permet pas de déterminer à partir du contexte la fonction à utiliser, le type ne

2. Il est à noter que ce type de masquage étant autorisé en mode boucle interactive (cf. section 2.7.3), la deuxième définition de `plus2` aurait donc masqué la première.

pouvant être généralisé. Ainsi, une fonction `plus` définie à l'aide de l'opérateur `+` par `let plus x y = x + y` aura comme type `int -> int -> int`.

Il est toutefois possible de repousser ce choix aux appels de la fonction `plus` en la définissant comme une **fonction expansée** ou fonction *inline*. Le type de `plus` pourra prendre en argument des valeurs d'un type possédant une opération `+`. Chaque appel à la fonction `plus` donne lieu à une expansion de sa définition dans le bytecode compilé.

```
let inline plus x y = x + y
val inline plus :
    ^a -> ^b -> ^c when ( ^a or ^b) : (static member ( ↵
    + ) : ^a * ^b -> ^c)
```

Masquage

Une liaison peut être **masquée**, de manière temporaire ou permanente, par une nouvelle déclaration de l'identificateur lié. La liaison masquée n'est plus accessible dans l'environnement courant mais existe toujours en concordance avec sa portée. Les déclarations masquée et masquante sont totalement indépendantes et aucune concordance de type n'est demandée. Le masquage est à distinguer d'une redéfinition de méthode d'une classe car celle-ci conserve le type de la méthode redéfinie et cohabite avec l'ancienne version (voir section 2.4.2). Il n'a rien à voir non plus avec l'affectation. Le masquage n'est accepté en F# qu'au niveau des expressions (cf. intérêt I-F-130).

> *Intérêt 14* (I-F-14) : *Restriction du masquage aux identificateurs locaux.*

Ainsi, dans le code suivant, la valeur du premier identificateur `a` est capturée dans la fermeture de `f` même si la liaison a été masquée par une nouvelle déclaration de `a`.

```
let g () =
    let a = 1
    let f x = a + x
    let a = 10
    f 1
val g : unit -> int
g ()
val it : int = 2
```

Le masquage peut parfois être utile pour éviter l'introduction d'un nouvel identificateur mais il risque d'obscurcir le code source et invalide la transparence référentielle de l'identificateur masqué.

> *Recommandation 15* (R-F-15) : *Éviter le masquage de fonctions importées.*
> Il est préférable d'éviter le masquage de fonctions importées depuis un module ouvert (cf. section 2.3.3). ■

2.1.2 Étude de la fonctionnalité

En F#, une fonction est une valeur de première classe : elle peut être passée en argument d'une autre fonction, retournée en tant que valeur d'une expression, etc. F# propose la pleine fonctionnalité. Une fonction est dite d'**ordre supérieur** si elle prend une fonction en argument ou si son application retourne une valeur fonctionnelle.

Fermetures et applications

La valeur d'une expression fonctionnelle, appelée **fermeture** ou **clôture**, est constituée du corps de la fonction et des liaisons des identificateurs libres dans ce corps présentes au moment de l'évaluation de cette expression. L'application d'une fonction à un paramètre effectif se fait en étendant l'environnement présent dans la fermeture par la liaison du paramètre formel à la valeur du paramètre effectif. Cette méthode d'évaluation des applications est le fondement de la portée statique (voir section 2.1.1).

> *Intérêt 16* (I-F-16) : *Le calcul de fermeture garantit la portée statique.*

Fonctions récursives

Les **fonctions récursives** (ou mutuellement récursives) sont des fonctions dont le corps contient un appel à elles-mêmes. Elle sont indispensables pour décrire les manipulations de données structurées récursives (listes, arbres, etc.) non-mutables. Leur définition, utilisant le filtrage, peut rester très proche de leur spécification.

Intérêt 17 (I-F-17) : *Bonne traçabilité des fonctions récursives manipulant des données complexes.*

Le langage ne garantit pas que l'appel d'une fonction récursive se termine.

Les valeurs récursives non-fonctionnelles sont abordées en section 2.1.7.

L'exécution d'une fonction récursive requiert l'utilisation d'une pile d'exécution. Si la récursion ne termine pas, la pile peut grandir de manière indéfinie et si elle atteint sa limite, il y a débordement de pile et arrêt du programme par la levée de l'exception .NET (cf. section 2.2.3) `StackOverflowException`.

Danger 18 (D-F-18) : *Débordement de la pile d'exécution par appels récursifs.*

Recommandation 19 (R-F-19) : *S'assurer du non-débordement de la pile d'exécution.*
Quel que soit le langage de programmation, il est nécessaire d'estimer le nombre d'appels récursifs engendrés par chaque application d'une fonction récursive, particulièrement lors de la manipulation de grosses structures de données ou de calculs arithmétiques complexes. ■

La taille de la pile est limitée en .NET ce qui évite que le programme ne monopolise la mémoire du système.

Les **appels récursifs terminaux** ne font pas grossir la pile car à chaque appel récursif les paramètres de l'appel précédent ne sont plus utiles.

Intérêt 20 (I-F-20) : *La pile ne grossit pas pour un appel récursif terminal.*

Les fonctions de la bibliothèque standard manipulant les listes et autres structures de données sont récursives terminales sauf mention du contraire dans la documentation.

Recommandation 21 (R-F-21) : *Utiliser les fonctions de la bibliothèque standard pour les manipulations de données structurées récursives.*

Le compilateur F# compile les appels récursifs terminaux en une séquence d'instructions bytecode correspondant au corps de la fonction. Cette séquence se finit soit par un saut inconditionnel (`br.s`) soit par l'instruction (`tail.`) qui après

avoir effacé de la pile les paramètres de l'appel à la méthode courante, effectue de nouveau un appel à cette méthode.

> *Danger 22* (D-F-22) : *Un appel englobé dans un rattrapage d'exception n'est pas terminal.*
> Si on englobe un appel récursif terminal dans un `try with`, l'appel récursif n'est plus terminal car il est nécessaire de dépiler le bloc de gestion d'exceptions. ■

> *Recommandation 23* (R-F-23) : *S'assurer que les appels supposés terminaux le sont effectivement.*

Le compilateur F# et l'éditeur Visual Studio ne permettent pas d'identifier la nature des appels, contrairement à OCaml.

> *Danger 24* (D-F-24) : *L'exception de débordement de pile n'est pas rattrapable.*
> L'exception .NET `StackOverflowException` de débordement de pile n'est pas rattrapable par le mécanisme de rattrapage d'exceptions, contrairement à OCaml. ■

Il est à noter que l'exception `StackOverflowException` peut être lancée par un programme (elle est rattrapable dans ce cas). Il est ainsi possible de simuler le dépassement de pile et ainsi détourner l'attention des mainteneurs.

Une manière d'éviter les appels récursifs non-terminaux consiste à programmer par **continuation** en passant à l'appel récursif la valeur fonctionnelle représentant la suite du calcul plutôt que d'appliquer ce calcul au résultat de l'appel récursif, évitant ainsi les appels non terminaux. Ce style de programmation est facilité par les flux de contrôle (cf. 2.1.6).

Fonctions anonymes

Une expression fonctionnelle peut être utilisée sans être nommée, on la désigne alors sous le vocable de **fonction anonyme**.

> *Intérêt 25* (I-F-25) : *Laisser anonyme une fonction utilisée une seule fois allège le code.*

> *Intérêt 26* (I-F-26) : *L'utilisation d'une fonction anonyme prévient contre tout danger pouvant menacer son nom.*

Il est possible qu'une fonction appelée par son nom ne soit pas celle souhaitée par le programmeur, en cas de redéfinition ou masquage. ■

Intérêt 27 (I-F-27) : *L'utilisation d'une fonction anonyme permet son confinement.*

Malgré ces deux intérêts, l'utilisation de fonctions anonymes ne peut que rester très limité car il est largement préférable de nommer une fonction qui va être utilisée plusieurs fois, pour faciliter la lecture et la maintenance.

Recommandation 28 (R-F-28) : *Limiter l'utilisation de fonctions anonymes.*
Les fonctions anonymes doivent être réservées aux fonctions utilisées une seule fois sinon leur utilisation entraîne la duplication du code. ■

Application partielle

Une fonction F# peut retourner une fonction en résultat. Par exemple, la fonction définie par `let f x y = x + y` a le type `int -> int -> int`, ou de manière plus lisible `int -> (int -> int)`. Cette fonction, dite à deux arguments, peut être **appliquée partiellement** à un argument : par exemple `(f 1)` est une fonction équivalente à `function y -> 1 + y`.

Intérêt 29 (I-F-29) : *Fonctions d'ordre supérieur et applications partielles permettent un partage optimal de texte source.*
Utiliser une fonction d'ordre supérieur permet de bien transcrire la généricité d'une spécification, tout en permettant une spécialisation progressive du texte source par passage successif d'arguments. ■

Dans l'exemple suivant, la fonction `fold_left` possède trois arguments, `op` et `accu` d'une part, qui vont permettre de spécialiser son code et `l` d'autre part, qui est la liste sur laquelle le traitement spécialisé est appliqué. Les fonctions `somme` (respectivement `concat_ABC`) spécialisent la fonction `fold_left` en une fonction qui fait la somme des entiers (resp. la concaténation des chaînes de caractères et d'un préfixe `"ABC"`) de la liste passée en argument.

```
1 let rec fold_left op accu l =
```

```
2     match l with
3         | [] -> accu
4         | a::l -> fold_left op (op accu a) l
5 val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
6 let somme = fold_left (+) 0
7 val somme : (int list -> int)
8 somme [1;2;3]
9 val it : int = 6
10 let concat_ABC = fold_left (+) "ABC"
11 val concat_ABC : (string list -> string)
12 concat_ABC ["1";"2";"3"]
13 val it : string = "ABC123"
```

Si une donnée est utilisée plusieurs fois dans l'exécution d'un traitement, il peut être intéressant de définir ce traitement par une fonction à plusieurs arguments et de nommer l'application partielle de ce traitement à cette donnée. La donnée sera alors encapsulée dans la fermeture et n'aura pas à être présente dans l'environnement courant (comme par exemple la fonction de codage `+` ou le préfixe `"ABC"` qui sont encapsulées dans la fonction `concat_ABC`).

> *Intérêt 30* (I-F-30) : *L'application partielle permet d'encapsuler des données.*

Séparation programme–données

La lecture de programmes fonctionnels peut donner l'impression que des fonctions sont créées dynamiquement selon le chemin d'exécution du programme alors que seules les fermetures sont créées dynamiquement. Le compilateur n'ajoute à la séquence d'instructions codant le corps de la fonction que les instructions d'accès aux valeurs des identificateurs libres dans ce corps. Ces valeurs n'apparaissent donc pas dans le jeu d'instructions produit par la création dynamique d'une fermeture et leurs liaisons aux identificateurs libres sont gérées par le GC.

L'exemple ci-dessous montre différentes formes possibles d'utilisation d'une même expression (`(* corps de f *)` contenant les paramètres formels `x` et `y`) pour créer une fonction `main_1`, en utilisant une fonction anonyme `main_2`, une définition locale `main_3`, une application partielle `main_4` ou un paramètre fonctionnel `main_5`.

```
1 let main_1 (x,y) =
```

```
2   (* corps de f *)
3 let main_2 (u,v) =
4   (fun x y -> (* corps de f *) ) u v
5 let main_3 (u,v) =
6   let f x y = (* corps de f *) in
7     f u v
8 let main_4 (u,v) =
9   let f x y = (* corps de f *) in
10  let g = f u in
11    g v
12 let main_5 (u,v) =
13  let f x y = (* corps de f *) in
14  let g f x y = f x y in
15    g f u v
```

Les différences entre les codes produits pour ces cinq fonctions ne concernent que les instructions d'accès aux valeurs des identificateurs. Par exemple, la valeur `g` dans `main_4` sera construite à partir du code du corps de `f` et de l'instruction d'accès à la valeur effective de `u`.

> *Intérêt 31* (I-F-31) : *Le compilateur garantit la séparation programme–données.*
> Même s'il contient des traits impurs, le code qui s'exécute est produit à la compilation sous la forme de fermetures et ne peut pas s'altérer lui-même. Son intégrité est donc garantie. ■

> *Danger 32* (D-F-32) : *En présence de chargement dynamique de code les garanties du compilateur sont affaiblies.*
> Le chargement dynamique de code modifie ipso-facto le code exécuté. ■

2.1.3 Étude du typage

Cette section présente le langage de types de F# et son typage. Des aspects complémentaires sont présentés plus loin dans les sections appropriées.

Propriétés du typage

Le typage peut être vu comme un mécanisme d'abstraction des identificateurs et des expressions, qui facilite le prototypage et permet de rendre compte de la

spécification du programme. Attribuer un type à une valeur, permet de contrôler les utilisations de cette valeur à l'aide d'un algorithme de typage et de déterminer les espaces d'allocation des données (voir intérêt I-F-6).

F# est un langage **fortement statiquement typé**. Cela signifie que toute expression possède un type et un seul et que celui-ci peut être déterminé sans évaluer l'expression. Le compilateur effectue une **vérification des types** du programme et traduit ensuite les types F# en types .NET dans le bytecode compilé (cf. section 2.6).

> *Intérêt 33* (I-F-33) : *Le typage statique fort apporte des garanties.*
> Le typage apporte plusieurs garanties sur le code source : accès cohérent aux variables, cohérence de la manipulation des données vis-à-vis de leur type, constance des valeurs non-mutables, protection des données de types abstraits, exhaustivité du filtrage. Ces garanties sont présentées dans les sections décrivant les traits en question. ■

> *Intérêt 34* (I-F-34) : *Le compilateur détecte les erreurs de typage.*
> Le compilateur produit un code compilé qui offre ces garanties ou détecte des erreurs de typage. ■

Le **langage de types** de F# est suffisamment expressif pour décrire toutes les structures de données requises par le développement d'algorithmes complexes, et offre ainsi une assistance importante au programmeur.

Dans la suite du texte, on appellera informellement **type structuré** un type défini à partir de types somme et enregistrement (définis dans la suite de cette section).

> *Recommandation 35* (R-F-35) : *Utiliser les types structurés pour représenter les données.*
> Il est recommandé d'utiliser le langage de types pour refléter au mieux la structure des données manipulées par le programme. Cela facilite la traçabilité de la spécification dans le texte source et permet de bénéficier de vérifications automatiques et systématiques. Cela libère de plus le programmeur de la gestion bas niveau de ces structures. ■

Cela va de pair avec le mécanisme de définition de fonctions par filtrage, qui permet de décrire ces traitements selon les différentes formes possibles des données (cf. section 2.1.5).

Le compilateur F# se base sur les informations de types pour automatiquement **instrumenter le code** du programmeur avec des instructions bas niveau de construction et de déconstruction de données.

Comme la construction ou la déconstruction de données ne passe pas par des manipulations de pointeurs définies par le programmeur, toute une classe d'erreur disparaît. En effet, le compilateur effectue une vérification de la cohérence de l'utilisation des données par typage et analyse de filtrage.

Cette section présente les types F#. Certains types directement hérités de .NET doivent être considérés séparément des types spécifiques à F# (cf. section 2.9.1).

Inférence de types

F# propose l'**inférence de types** avant d'effectuer les vérifications de types : le type de toutes les expressions d'un code source d'implémentation (fichier `.fs`) est automatiquement synthétisé par le compilateur. Ceci signifie que le programmeur n'a pas à déclarer tous les types. Le type inféré par le compilateur est correct, au sens où il est conforme aux règles de typage. L'inférence de type est possible même en présence de traits impératifs ou objets.

> *Intérêt 36* (I-F-36) : *L'inférence de types allège l'écriture et la relecture de code.*

Dans le cas où l'inférence de type échoue, le compilateur signale quelles expressions lui posent problème et if suffit alors soit de corriger l'erreur de typage soit, dans des cas bien identifiés, d'ajouter l'information de type nécessaire au compilateur.

Tous les types inférés durant la compilation ne sont par défaut pas présentés au développeur car le volume d'informations produit est trop important. L'option `--sig:sortie.fsi` imprime l'**interface** (liste des identificateurs globaux et leur type) de fichiers `.fs` en entrée dans un fichier `sortie.fsi` (cf. section 2.3.2).

> *Intérêt 37* (I-F-37) : *Tous les types inférés peuvent être présentés au développeur.*
> L'option `--sig:` du compilateur produit la liste des identificateurs globaux liés dans le texte source et de leur type.
> Les environnements de développement (IDE) permettent d'obtenir directement le type inféré de chaque expression. ■

L'inférence de type peut attribuer un type qui ne correspond pas forcément à celui imaginé par le programmeur, soit parce que trop général soit parce que différent de la spécification voulue.

> *Intérêt 38* (I-F-38) : *L'inférence de type révèle des erreurs de programmation.*
> Une différence entre le type inféré par le compilateur et celui attendu par le développeur peut révéler une erreur dans le texte source. En particulier, si le type inféré est plus générique que celui attendu cela démontre que le texte source est plus générique que prévu. ■

> *Recommandation 39* (R-F-39) : *Vérifier que le type inféré correspond au type attendu.*

L'exemple ci-dessous explicite un cas courant d'erreur qui peut être détecté en vérifiant les types inférés.

```
let instruction =
    printf "%s\n" "Hello world"
Hello world
val instruction : unit = ()
let main n =
    instruction
    instruction
    fact (n)
val main : int -> int
main 4
val it : int = 24
```

Dans cet exemple, `instruction` fait un effet de bord d'impression. Le type inféré pour `instruction` est `unit` car l'effet de bord est effectué au moment de la déclaration, donc l'identificateur `instruction` est lié à la valeur `()`. L'évaluation de l'appel `main 4`, évalue deux fois l'expression `instruction` soit `()` ce qui ne fait rien, puis calcule `fact (4)`.

Si l'intention du développeur était d'imprimer deux fois "Hello world" puis d'évaluer la factorielle de `4`, le code de l'exemple est faux et le type inféré pour F#instruction le montre. Pour que le code agisse comme attendu par le développeur, il aurait fallu que le type inféré pour `instruction` soit le type fonctionnel `unit -> unit`.

La version correcte du code est :

```
let instruction () =
    printf "%s\n" "Hello world"
val instruction : unit -> unit
let main n =
    instruction ()
    instruction ()
    fact (n)
val main : int -> int
main 4
Hello world
Hello world
val it : int = 24
```

Types atomiques et types prédéfinis

F# dispose de **types atomiques**, insécables, considérés par défaut comme non-mutables par le langage : octets (`sbyte`, `byte`), entiers (`int`, `uint int16`, `uint16`, `int64`, `uint64`), flottants (`float`, `float32`), décimaux (`decimal`), booléens (`bool`), caractères (`char`) et le **type unit** (`unit`).

Le **type exception** (`exn`) représente les exceptions (cf. section 2.2.3). Une valeur du type exception peut être paramétrée.

Les **chaînes de caractères** (`string`) sont représentées par des tableaux non-mutables de caractères.

> *Intérêt 40* (I-F-40) : *Non-mutabilité des chaînes de caractères.*
> Voir la section 2.2.1 pour une discussion sur les intérêts de la manipulation de valeurs non-mutables. ■

F# n'effectue pas de conversion implicite. Il fournit par contre des fonctions pour faire ces conversions explicitement, le typage vérifiant leurs bonnes utilisations.

> *Intérêt 41* (I-F-41) : *Absence de conversion implicite.*
> L'absence de conversion implicite rend le programme plus robuste car elle évite les erreurs de conversions implicites non voulues. ■

L'attribut `[<Literal>]` permet d'associer à un identificateur une valeur constante, cet identificateur est appelé un **littéral**. Les littéraux sont utilisables dans les motifs de filtrage (cf. section 2.1.5) dont ils améliorent la lisibilité.

Type format

F#, comme OCaml (cf. intérêt I-O-48), propose à l'aide d'un **type format** des afficheurs fortement typés comme `Printf.printf` (de type `TextWriterFormat<'T> -> 'T`). Le premier argument de cette fonction `printf` est une chaîne de caractères spéciale représentant un format c'est à dire un texte contenant des paramètres (indiqués par le symbole `%`). Ces paramètres sont typés, et la bibliothèque se charge de leur transformation en chaînes de caractères.

> *Intérêt 42* (I-F-42) : *Garanties apportées par les afficheurs typés.*
> Les fonctions d'affichage de F# sont fortement typées et évitent ainsi les erreurs de programmation et les vulnérabilités. ■

Les paramètres `%A` et `%+A` sont universels, ils permettent d'afficher des valeurs de n'importe quel type. Le paramètre `%A` fait appel à la méthode `ToString` du type de la valeur passée en paramètre. Dans le cas d'une valeur d'un type abstrait, `ToString` retourne le nom du type au lieu de la valeur (comportement pouvant être modifié par l'attribut `[<StructuredFormatDisplay>]`). Le paramètre `%+A` affiche la représentation interne de la valeur sans respecter les frontières d'abstraction ou les redéfinitions de la méthode `ToString`.

> *Danger 43* (D-F-43) : *Contournement de l'encapsulation par le format* `%+A`.

> *Recommandation 44* (R-F-44) : *Interdire l'utilisation du format* `%+A`.

L'exemple suivant illustre le contournement de l'encapsulation par utilisation du format `%+A`. Le module `Secret` définit deux types abstraits `t1` et `t2` et deux valeurs de ces types. Le deuxième type redéfinit sa méthode `ToString`. Le code suivant est l'interface (fichier `.fsi`) du module `Secret`.

```
1 // secret.fsi
2 module Secret
3 type t1
4 type t2
5 val v1 : t1
6 val v2 : t2
```

Le code suivant est son implémentation (fichier `.fs`).

```
1 // secret.fs
2 module Secret
```

```
3 type t1 = A1 of string
4 type t2 = A2 of string with
5     override this.ToString() = "cache"
6 let v1 = A1 "secret"
7 let v2 = A2 "secret"
```

Le code suivant utilise les valeurs des types abstraits du module `Secret`. Les paramètres `%A` respectent l'abstraction alors que les paramètres `%+A` la contourne en affichant les valeurs encapsulées (lignes 4 et 8).

```
1 printf "%A\n" Secret.v1
2 Secret+t1
3 printf "%+A\n" Secret.v1
4 A1 "secret"
5 printf "%A\n" Secret.v2
6 cache
7 printf "%+A\n" Secret.v2
8 A2 "secret"
```

Types fonctionnels

Les fonctions étant des valeurs de première classe en F#, elles sont représentées par les **types fonctionnels** (`t1 -> t2` où `t1` et `t2` représentent chacun un type).

> *Intérêt 45* (I-F-45) : *Contrôle de la conformité des applications des fonctions à leur type.*
> Le typage fort et l'absence de conversion implicite assurent la cohérence du texte source, grâce à l'inférence de type des fonctions. ■

> *Intérêt 46* (I-F-46) : *Documentation automatique des fonctions par leur type.*
> Le type d'une fonction fournit une première information sur cette fonction. En particulier, les fonctions de type `t1 -> unit` sont souvent des fonctions effectuant des effets de bord. ■

Types enregistrement

Un **type enregistrement** permet de définir une structure de données composée d'un groupe fixé par avance de champs étiquetés.

Un ensemble de valeurs peut aussi être représenté par les types produits (voir plus loin) et par les objets (cf. section 2.4). Voir la description des abréviations de types pour une comparaison entre types enregistrement et types produits. Un type enregistrement F# est compilé en une classe .NET scellée, contrairement aux classes F# qui peuvent être héritées. Les types enregistrement autorisent l'égalité, la comparaison structurelle et le filtrage.

> *Recommandation 47* (R-F-47) : *Préférer l'usage des types enregistrement à celui des classes.*
> Il est recommandé de préférer l'utilisation de types enregistrement plutôt que des classes car ils possèdent certains avantages :
> – L'égalité et la comparaison d'enregistrement sont structurelles (elles sont référentielles pour les classes) ;
> – Le filtrage s'applique aux enregistrements mais pas aux objets ;
> – Les types enregistrement sont des structures figées alors que les classes peuvent être héritées ;
> – Les enregistrements F# deviennent des classes scellées au niveau .NET. ■

Les champs d'un enregistrement sont par défaut non mutables mais peuvent être individuellement déclarés mutables en leur associant un type référence ou en les déclarant mutables. La construction `{ e with ... }` permet de définir un nouvel enregistrement à partir des champs d'un enregistrement `e` précédemment défini. Il est à noter qu'il ne s'agit pas ici d'une modification mais bien d'une copie partielle de l'enregistrement. Ainsi, un champ référence sera copié et pointera vers la même valeur mutable alors qu'un champ mutable verra sa valeur dupliquée au moment de la copie (les champs mutables des deux enregistrements sont donc distincts).

```
1  type t = { n : int; mutable m : int; r : int ref }
2  let e = { n = 1; m = 2; r = ref 3 }
3  val e : t = {n = 1; m = 2; r = {contents = 3;};}
4  let e' = {e with n = 10}
5  val e' : t = {n = 10; m = 2; r = {contents = 3;};}
6  e.m <- 20
7  val it : unit = ()
8  e.r := 30
9  val it : unit = ()
10 e
11 val it : t = {n = 1; m = 20; r = {contents = 30;};}
12 e'
```

```
13 val it : t = {n = 10; m = 2; r = {contents = 30;};}
```

Il est à noter que le type référence (cf. section 2.2.1) est un enregistrement ayant un champ unique mutable appelé `contents`, comme l'illustre l'exemple ci-dessus.

Bien que représentant tous deux des n-uplets de valeurs, les types enregistrement et produit cartésien sont traités différemment par F# car les types enregistrement sont obligatoirement nommés et définis par l'utilisateur. Cela a des conséquences sur l'égalité entre types (voir plus bas la discussion sur la relation entre types).

Types somme

Les **types somme** (ou **types union**) permettent de définir des valeurs ayant plusieurs formes possibles. Chacune des formes est définie à l'aide d'un **constructeur de valeur** qui lui est propre. Chaque constructeur est donc unique et est associé à un seul type. Les constructeurs peuvent être constants ou appliqués à une valeur de n'importe quel type.

Le filtrage permet de **déconstruire une valeur** d'un type sans avoir à utiliser explicitement une fonction sur la donnée (cf. section 2.1.5).

> *Intérêt 48* (I-F-48) : *Les types somme exploitent le filtrage.*
> Les fonctions sur les valeurs de type somme peuvent être définies par filtrage. ■

> *Intérêt 49* (I-F-49) : *Les types somme représentent des données complexes sans manipulation de pointeurs.*
> Aucune manipulation de pointeur n'est nécessaire pour la création et la manipulation de structures de donnée, aussi complexes soient-elles. Leur désallocation est assurée par le GC (voir I-F-6), sans aucune intervention du programmeur. ■

> *Intérêt 50* (I-F-50) : *Contrôle de la construction et déconstruction de valeurs de type somme.*
> Le compilateur vérifie qu'une construction d'une valeur d'un type somme ou sa déconstruction sont effectuées en accord avec la définition du type. Ces vérifications réduisent fortement les erreurs de manipulations de données. ■

Recommandation 51 (R-F-51) : *Utiliser les types somme pour représenter des structures de données linéaires et arborescentes.*
Cette recommandation complète la recommandation R-F-35. Les types somme permettent de représenter directement toutes les structures de données linéaires ou arborescentes, exactement comme elles sont décrites dans les ouvrages d'algorithmique. ■

Types récursifs

Dans une définition d'un type enregistrement ou d'un type somme, le nom du type peut figurer dans sa définition, permettant de créer ainsi un **type récursif**. Par exemple, la déclaration suivante introduit un nouveau type `suite_int`, défini par la donnée d'un constructeur de valeur `Suitevide` définissant une valeur constante et un constructeur de valeur `ConsInt` pouvant être appliqué à tout couple de valeurs de type `int * suite_int` pour créer une valeur de type `suite_int`. Ce type permet de représenter des suites finies d'entiers.

```
1 type suite_int =
2   | Suitevide
3   | ConsInt of int * suite_int
```

Une valeur d'un type récursif est généralement finie. Certaines valeurs peuvent être récursives (voir les valeurs récursives en section 2.1.7).

Types polymorphes ou paramétrés

Un **type polymorphe** (ou **type paramétré**) est un type dont la définition dépend d'autres types, représentés par des paramètres de type (encore appelés variables de type). Ainsi, il est possible de définir des types enregistrement, des types somme, des types associés à des classes, etc. en utilisant des paramètres de type pour désigner le type des sous-structures qui apparaissent génériques dans la spécification de la structure.

Intérêt 52 (I-F-52) : *La généricité des structures de données se traduit dans les types polymorphes.*
Pouvoir exprimer la généricité d'une structure de données évite la duplication de code. Si l'inférence de types attribue un type polymorphe à une expression, cela signifie que cette expression possède un degré de généricité qui peut être voulu ou témoigner d'une erreur de programmation (voir recommandation R-F-39). ■

Contraintes de type

Il est possible d'exprimer des contraintes sur les paramètres des types polymorphes. On peut par exemple requérir que le type propose l'égalité (`'a when 'a : egality`) ou que le type soit un type .NET (`struct`, `enum`, etc.). Certaines de ces contraintes peuvent être inférées par le compilateur (en cas par exemple d'égalité sur les valeurs de type `'a`). Le respect de ces contraintes est vérifié à la compilation.

> *Intérêt 53* (I-F-53) : *Contrôle fin de l'abstraction par les contraintes sur les paramètres de types.*

Types option

Le **type option** (`'a option`) est un type somme prédéfini polymorphe paramétré par un type quelconque `'a`. Il possède deux constructeurs : `Some v` où la valeur `v` a pour type `'a`, et `None`. La forme `Some` marque la présence d'une valeur et la fournit tandis que `None` marque l'absence de valeur.

Dans la plupart des langages la valeur non significative et les valeurs significatives sont du même type. Par exemple `-1` est souvent utilisé pour représenter une valeur non significative de type entier positif ou la chaîne vide pour marquer la valeur par défaut d'une chaîne de caractères. Les valeurs non significatives ne peuvent être repérées de manière sûre ni à la compilation, ni à l'exécution ce qui est la source de nombreuses erreurs.

> *Intérêt 54* (I-F-54) : *Le type option dénote explicitement les valeurs non significatives.*

> *Recommandation 55* (R-F-55) : *Utiliser le type option pour représenter les valeurs non significatives.*
> Ceci permet non seulement de pouvoir différencier dans le code les traitements de ces valeurs, mais aussi de vérifier statiquement qu'elles sont traitées par l'ensemble des traitements (par typage et filtrage 2.1.5). Cette recommandation complète la recommandation R-F-35. ■

Malgré l'existence du type option, la bibliothèque `Unchecked` propose une fonction `defaultof<'a>` qui retourne la valeur par défaut d'un type donné.

Cette fonction a pour seul intérêt de permettre la compatibilité avec .NET.

Recommandation 56 (R-F-56) : *Ne pas utiliser la fonction* `defaultof` *du module* `Unchecked`.
Il faut proscrire l'utilisation de la fonction `defaultof<'a>` de la bibliothèque `Unchecked` sauf pour des besoins de compatibilité avec .NET (cf. D-F-198 et R-F-199). Le type option doit être utilisé comme alternative. ■

De la même manière, il est courant dans certains langages comme C d'utiliser le pointeur `null` pour marquer l'absence. Pour des raisons de compatibilité, F# propose un attribut `[<AllowNullLiteral>]` pour ajouter `null` comme valeur possible d'un type donné.

Recommandation 57 (R-F-57) : *Ne pas utiliser l'attribut* `AllowNullLiteral` *et la valeur* `null`.
Il faut proscrire l'utilisation de l'attribut `[<AllowNullLiteral>]` et de la valeur `null` sauf pour des besoins de compatibilité avec .NET. Le type option doit être utilisé comme alternative. ■

Types abstraits

Un type est dit **abstrait** si une interface (cf. section 2.3) définie par le développeur interdit l'accès à sa représentation et restreint l'accès aux fonctions qui permettent la création et la manipulation des valeurs de ce type. Il n'y a pas de mot-clé marquant la qualité de type abstrait.

La représentation d'un type abstrait n'est pas visible, aucune fonction de construction ou de déconstruction sur ce type ne peut être ajoutée à l'extérieur. De plus, seules les fonctions choisies peuvent être utilisées pour manipuler les valeurs de ce type. Un type abstrait apporte donc une garantie de confidentialité sur les valeurs de ce type.

Intérêt 58 (I-F-58) : *L'utilisation de types abstraits protège la représentation des données.*

Recommandation 59 (R-F-59) : *Utiliser des types abstraits pour masquer la représentation de données sensibles.*

Pour maintenir des propriétés de la représentation d'un ensemble de données d'un type donné, il suffit de fournir des fonctions de construction de ces

valeurs qui maintiennent ces propriétés et de cacher la définition du type. C'est précisément le rôle d'un type abstrait.

> *Recommandation 60* (R-F-60) : *Utiliser des types abstraits pour maintenir des invariants de représentation des données.*

Autres types structurés prédéfinis

F# propose d'autres types structurés tels que le **type produit** (`'a * 'b`) format des tuples, le **type liste** (`'a list`), le **type séquence** (`seq<'a>`) et les **flots de données**.

Les séquences et les listes sont des ensembles énumérés, elles implémentent l'interface `IEnumerable` de .NET (un énumérable .NET). Les listes sont des ensembles finis alors que les séquences peuvent potentiellement être infinies et sont donc évaluées paresseusement.

Listes et séquences peuvent être définies par énumération ou par compréhension : en utilisant des plages `a .. b` ou les opérations `yield` et `yield!` pour générer un ou plusieurs éléments.

L'opérateur de création de plage `(..) a b` crée une séquence d'éléments allant de l'élément `a` à l'élément `b`. Le type de `a` et `b` doit être énumérable.

Type énumération

F# propose des énumérations qui sont un sucre syntaxique pour nommer des valeurs d'un type atomique. Elles s'apparentent syntaxiquement aux types somme mais n'en possèdent pas les propriétés. Il est par exemple possible d'utiliser une valeur d'un type énumération qui n'a pas été nommée.

> *Danger 61* (D-F-61) : *Les énumérations ne sont pas des unions disjointes.*
> Les énumérations ne sont pas des unions disjointes et n'en partagent pas les propriétés. Il n'est pas possible de vérifier l'exhaustivité d'un traitement sur une valeur d'une énumération. ■

Comme indiqué par le danger D-F-198 et la recommandation R-F-199, il est déconseillé d'utiliser le type énumération hors de besoins de compatibilité .NET.

> *Recommandation 62* (R-F-62) : *Préférer les types somme aux types énumération.*

Il est recommandé d'utiliser les types somme associés à des fonctions de correspondance à des valeurs atomiques plutôt que d'utiliser les types énumération. Ceci permet de bénéficier de la vérification d'exhaustivité des traitements. ■

Type délégué

.NET utilise des pointeurs de fonction typés pour l'ordre supérieur. Le type d'un pointeur de fonction est appelé un **type délégué** ou *delegate*. F# permet la définition de ce genre de type pour la compatibilité avec .NET. Comme indiqué par le danger D-F-198 et la recommandation R-F-199, il est déconseillé d'utiliser le type délégué hors de besoins de compatibilité .NET.

Attributs

Le langage F# possède une syntaxe pour ajouter des métadonnées au programme appelées **attributs**. Ces attributs donnent lieu à des vérifications supplémentaires à la compilation, ou à l'exécution. Les attributs permettent d'étendre les possibilités offertes par la réflexion (cf. section 2.4.4).

Les attributs s'appliquent à différent niveaux du langage. Il est parfois utile de préciser en préfixe de l'attribut à quel niveau celui-ci doit s'appliquer pour éviter toute ambiguïté (`[<cible:Attribut>]` où la cible est `assembly`, `field`, `parameter` ou `return`).

Unités de mesure

F# propose en complément du typage une gestion d'**unités de mesure**. Il est ainsi possible d'adjoindre à des entiers et flottants une unité de mesure. Le compilateur vérifie statiquement la correction des utilisations des valeurs ayant une unité de mesure.

Intérêt 63 (I-F-63) : *Contrôle de la conformité des opérations sur valeurs à unités de mesure.*
La gestion des unités de mesure par le compilateur rend la manipulations de valeurs à unité de mesure robuste. ■

Abréviation et définition de types

La déclaration d'un nouveau nom de type (`type t = ...`) peut avoir deux significations distinctes.

- Ce nouveau nom peut être associé à une **expression de type** (combinaison de types atomiques, de noms de types existants, de tuples, de types fonctionnels et de variables de type), on parle alors d'une **abréviation de type**.
- Il peut être associé à une nouvelle construction de type (un type enregistrement ou un type somme), on parle alors de **définition de type** ou de **définition générative** car elle donne naissance à une nouvelle représentation de type.

Une abréviation de type est considérée comme équivalente à l'expression la définissant. Ainsi le typage ne fera pas de différence entre une valeur du type de l'expression de type ou du type de son abréviation.

```
1 let v = (true, 1)
2 val v : bool * int = (true, 1)
3 type t = bool * int
4 let f (x : t) = snd x
5 val f : bool * int -> int
6 f v
7 val it : int = 1
```

Danger 64 (D-F-64) : *Une abréviation de type ne crée pas une nouvelle représentation de type.*

Un type déclaré par une définition générative est en revanche diffère toujours de tous les types préalablement définis. Après sa déclaration, une abréviation de ce nouveau type peut être introduite.

Recommandation 65 (R-F-65) : *Préférer les définitions génératives pour s'assurer de la nouveauté des types.*

F# interdit l'abstraction d'une abréviation de type. Un type abstrait doit ainsi être soit un type enregistrement soit un type somme.

Relations entre types

Les relations entre types permettent de contrôler finement la manipulation des expressions.

Il existe trois relations sur les types : l'égalité entre types, la compatibilité liée à l'instanciation et la compatibilité liée au sous-typage.

L'**égalité de deux expressions de type** se fait sur la structure des expressions : deux types `t1 * t2` et `t3 * t4` sont égaux si `t1` et `t3` sont égaux et `t2` et `t4` le sont également. Autrement dit, deux expressions de type sont égales si et seulement si elles ont la même structure. Deux abréviations de type sont égales si les expressions de type qu'elles dénotent sont égales : les abréviations suivantes de `float`, `type chou = float` et `type carotte = float`, ne permettent pas de différencier les valeurs de type `chou` de celles de type `carotte`.

> *Recommandation 66* (R-F-66) : *Utiliser l'égalité de type pour contrôler finement la construction de valeurs.*

Un type `t1` est une **instanciation** d'un type polymorphe `'a t` si `t1` est le résultat du remplacement de `'a` par un type plus défini. Toute instanciation d'un type peut remplacer ce type dans le typage d'une expression (voir l'intérêt I-F-52). Par exemple `('b list) list` et `int list` sont deux instances du type `'a list`.

Un type `t1` est dit **compatible** avec un type `t2` si `t1` et `t2` sont égaux ou si `t1` est une instance de `t2` ou bien si `t1` est un sous-type de `t2`. Le type `t1` peut dans ce cas remplacer `t2` au cours du typage.

2.1.4 Égalité et comparaison

Le langage F# propose des fonctions génériques testant l'**égalité** et la **comparaison** de valeurs d'un même type. Une **fonction de hachage** calcule une **empreinte** pour une valeur d'un type donné. Ces méthodes sont définies par défaut et peuvent être redéfinies ou désactivées (voir ci-après les paragraphes portant sur les contraintes d'égalité et de comparaison).

Ces fonctions sont implémentées par les méthodes `Equals` et `GetHashCode` de `System.Object` (cf. section 2.4.1) pour l'égalité et le hachage et par la méthode `CompareTo` de l'interface `IComparable` pour la comparaison. Les opérations d'égalité (`=`) et d'inégalité (`<>`), la fonction de hachage (`hash`) et les opérations et fonctions de comparaison (`<`, `<=`, `>`, `>=`, `compare`, `min`, `max`) font appel à ces méthodes.

L'égalité peut être soit **structurelle** (deux valeurs ayant la même structure sont égales) soit **référentielle** (deux valeurs pointant vers le même donnée physique sont égales). L'égalité est structurelle pour les types atomiques, les chaînes de caractères, les tuples, les listes, les options, les enregistrements, et les valeurs de type somme (si leurs types constitutifs permettent aussi l'égalité structurelle). Pour les objets, et les références, l'égalité est référentielle.

La fonction `PhysicalEquality` de la bibliothèque standard `LanguagePrimitives` permet de tester l'égalité référentielle sur des valeurs de tout type. La fonction `GenericLimitedHash` permet le calcul d'une empreinte en évitant les cycles par utilisation d'une limite.

Invariants d'égalité, de hachage et de comparaison

L'égalité, le hachage et la comparaison doivent respecter un certain nombre d'invariants. Ces invariants sont supposés vrais dans les bibliothèques standards utilisant ces opérations et fonctions (`Set`, `Map`, `Collection`, etc.).

> *Intérêt 67* (I-F-67) : *Les implémentations de l'égalité, du hachage et de la comparaison tels que produits par le compilateur pour les types structurés respectent leurs invariants.*

> *Danger 68* (D-F-68) : *Les invariants de l'égalité, du hachage et de la comparaison ne sont pas vérifiés statiquement ou dynamiquement.*

L'égalité telle qu'implémentée par le compilateur respecte les invariants suivants.

- L'égalité doit être réflexive `x = x` sauf dans le cas particulier de l'arithmétique des flottants (l'évaluation de `nan = nan` retourne toujours `false`).
- L'égalité doit être symétrique (les évaluations de `x = y` et `y = x` retournent toujours la même valeur).
- L'égalité doit être transitive (si les évaluations de `x = y` et `y = z` retournent `true`, alors celle de `x = z` doit retourner `true` aussi).
- Tant que `x` et `y` ne sont pas modifiés, l'évaluation de `x = y` doit rester inchangée.
- Dans le cas d'un type `t` acceptant la valeur `null`, l'évaluation de `(x:t) = null` doit retourner `false`.

Le hachage tel qu'implémenté par le compilateur respecte les invariants suivants :

- Si l'évaluation de `x = y` retourne `true`, alors les empreintes `hash x` et `hash y` doivent être égales. Par contre, si deux valeurs ne sont pas égales, leur empreintes ne doivent pas nécessairement être différentes.
- L'empreinte d'une valeur mutable doit changer quand celle-ci mute mais doit rester inchangée quand celle-ci ne mute pas (ceci est particulièrement pertinent pour les objets et leurs états internes).

La comparaison telle qu'implémentée par le compilateur respecte les invariants suivants.

- L'évaluation de `compare x x` doit retourner `0`.
- Si l'évaluation de `compare x y` retourne `0` alors `compare y x` doit aussi retourner `0`.
- Si les évaluations de `compare x y` et `compare y z` retournent `0` alors l'évaluation de `compare x z` doit retourner `0`.
- Si l'évaluation de `compare x y` retourne une valeur non-nulle n alors l'évaluation de `compare y x` doit retourner la valeur $-n$.
- Si les évaluations de `compare x y` et `compare y z` retournent des valeurs non-nulles n et m de même signe, alors l'évaluation de `compare x z` doit retourner une valeur non-nulle p de même signe.

Redéfinitions de l'égalité, du hachage et de la comparaison

Il est possible de définir ou redéfinir[3] ses propres fonctions d'égalité, de hachage et de comparaison (interface .NET `System.IComparable`) :

```
1 [<CustomEquality; CustomComparison>]
2 type T = ...
3     override t.Equals(t') = ...
4     override t.GetHashCode() = ...
5     interface System.IComparable with
6       member t.CompareTo t' = ...
```

Il est conseillé de prendre des précautions lors de ce type de définitions car de nombreuses bibliothèques standard en font usage. Il est aussi préférable de redéfinir `Equals` et `GetHashCode` de manière cohérente. Lors de la redéfinition, il faut s'assurer que les invariants énoncés précédemment sont respectés.

Recommandation 69 (R-F-69) : *S'assurer du respect des invariants de l'égalité, du hachage et de la comparaison lors de leurs redéfi-*

3. La recommandation R-F-76 rappelle qu'il s'agit d'une redéfinition et non d'un masquage ou d'une surcharge.

nitions.

D'autre part, il est conseillé de ne pas lever d'exceptions dans les implémentations de l'égalité, du hachage et de la comparaison et d'éviter d'aboutir à une définition circulaire de `GetHashCode`.

Contraintes d'égalité et de comparaison

Lors de la définition d'un type, il est possible de choisir, via des attributs, le type d'égalité qui s'appliquera sur les valeurs de ce type :
`[<StructuralEquality>]` pour l'égalité structurelle ou `[<ReferenceEquality>]` pour l'égalité référentielle.

Les opérations et fonctions d'égalité et de hachage ne peuvent s'appliquer que sur les types ayant une notion d'égalité définie (contrainte `equality` de .NET). Les opérations et fonctions de comparaison ne peuvent s'appliquer que sur les types ayant une notion de comparaison définie (contrainte `comparison`).

Ces vérifications sont faites statiquement au typage.

Il est possible d'interdire l'égalité et le hachage d'une part, la comparaison d'autre part via les attributs `[<NoEquality>]` et `[<NoComparison>]`. Interdire l'égalité revient à interdire aussi la comparaison.

> *Recommandation 70* (R-F-70) : *Restreindre l'égalité et la comparaison sur les types des données sensibles.*

Un type paramétré par un autre type peut imposer que ce type soit comparable comme c'est le cas pour les `Set` et `Map` de la bibliothèque standard.

La non-égalité et la non-comparaison se propagent. Par exemple tester l'égalité de deux exceptions ayant en argument un type `NoEquality` retourne toujours faux.

Il est à noter que les exceptions (cf. section 2.2.3) n'autorisent pas la comparaison mais autorisent l'égalité.

Égalité, comparaison et encapsulation

L'encapsulation (cf. section 2.3.2) permet de restreindre à des fonctions bien identifiées la manipulation de données sensibles. Cette restriction ne s'applique pas à l'égalité et à la comparaison.

> *Danger 71* (D-F-71) : *Égalité et comparaison pour contourner l'encapsulation.*

Les fonctions d'égalité, de hachage et de comparaison ne sont pas limitées par encapsulation. Ainsi il est possible par défaut de les appliquer à des valeurs d'un type abstrait. Ceci ne permet toutefois pas la création ou la modification de telles valeurs mais simplement leur observation. ■

Les attributs `[<NoEquality; NoComparison>]` vus précédemment permettent d'établir une telle restriction et ainsi de renforcer l'encapsulation.

Recommandation 72 (R-F-72) : *Restreindre l'égalité et la comparaison sur les types abstraits représentant des données sensibles.*
Pour étendre l'encapsulation d'un type `t` en interdisant l'égalité, le hachage et la comparaison de ses valeurs, il est conseillé de cacher sa représentation et de rajouter les contraintes sur l'égalité et la comparaison dans le fichier d'interface `.fsi`.

```
1 [<NoEquality; NoComparison>]
2 type t
```

Ces contraintes ne s'appliquent pas aux fonctions internes du module (fichier implémentation `.fs`). ■

Des versions particulières des fonctions d'égalité, de hachage et de comparaison ne sont pas sujettes à ces contraintes.

Danger 73 (D-F-73) : *Contournement des contraintes sur l'égalité et la comparaison.*
Les fonctions `equals`, `hash` et `compare` de la bibliothèque `Unchecked` et la fonction `GenericHash` de la bibliothèque `LanguagePrimitives` outrepassent les contraintes existantes d'égalité et de comparaison. ■

Recommandation 74 (R-F-74) : *Ne pas utiliser les fonctions contournant les contraintes d'égalité et de comparaison.*
Il est recommandé de vérifier les utilisations des fonctions des bibliothèques `Unchecked` et `LanguagePrimitives` qui contournent les contraintes d'égalité et de comparaison. ■

Surcharge des opérateurs d'égalité et de comparaison

Il est à noter que les opérations d'égalité et de comparaison sont polymorphes contrairement aux autres opérateurs surchargés (cf. section 2.1.1 pour une dis-

cussion sur la surcharge des opérateurs).

Malgré le polymorphisme des opérations d'égalité et de comparaison, le compilateur autorise leur surcharge tout en émettant un avertissement de compilation. Ce type de surcharge aboutit à une ambiguïté. Dans l'exemple deux opérations d'égalité coexistent pour un même type.

```
type t = A | B
  with static member (=) (x,y) = false
warning FS0086: The name '(=)' should not be used as a member name. To define equality semantics for a type, override the 'Object.Equals' member. If defining a static member for use from other CLI languages then use the name 'op_Equality' instead.
t.(=) (A,A)
val it : bool = false
A = A
val it : bool = true
A.Equals(A)
val it : bool = true
```

Danger 75 (D-F-75) : *Ambiguïté résultant de la surcharge des opérations d'égalité et de comparaison.*

Recommandation 76 (R-F-76) : *Ne pas surcharger les opérations d'égalité et de comparaison.*
Un avertissement de compilation est émis en cas de surcharge de ce type, il est possible de les éviter en imposant que les avertissements de compilation soient traités comme des erreurs de compilation (cf. recommandation R-F-170). ■

2.1.5 Filtrage

Le **filtrage** (ou *pattern matching* en anglais) est un mécanisme de transformation et de déconstruction de valeurs. Le filtrage est utilisé par l'expression de filtrage (`match ... with ...`), les définitions (`let ... =`), les fonctions anonymes (`function ... -> ...` et `fun ... -> ...`) et les expressions de rattrapage d'exceptions (cf. section 2.2.3).

Un filtrage est une suite de paires `p -> e` où `p` est un **filtre** et `e` une expression. Les filtres permettent de reconnaître une donnée d'un type et de lier des noms à

des parties de la donnée afin d'évaluer l'expression `e`. Les filtres peuvent contenir des constantes de types atomiques, des littéraux, des filtres actifs (voir ci-après), des constructeurs des types somme et apparentés (option, listes, enregistrements, tableaux, tests de réflexion) et des étiquettes d'exceptions. Les filtres permettent donc de déconstruire les valeurs de types structurés.

Il est possible de combiner les filtres et de ne pas augmenter l'environnement d'évaluation à l'aide d'un nom spécial (`_`) aussi appelé **attrape-tout**.

Le filtrage de F# est linéaire : un même identificateur n'est lié qu'au plus une fois par filtre.

> *Recommandation 77* (R-F-77) : *Favoriser la représentation de données par des types structurés pour bénéficier du filtrage.*
> Pour pouvoir bénéficier au maximum des vérifications d'exhaustivité du filtrage, il est préférable d'utiliser des types structurels (types somme et enregistrement) plutôt que des types non filtrables (classes, type fonction, types .NET). ■

Lorsqu'on manipule de données complexes, les différents filtres d'un filtrage sont souvent loin dans le texte source du programme. Écrire des filtres qui décrivent des parties disjointes du type argument facilite leur lecture. De plus ce genre de filtrage restera exhaustif en cas d'ajout de constructeurs au type filtré.

> *Recommandation 78* (R-F-78) : *Utiliser si possible des filtres disjoints.*
> La règle de choix du filtre à utiliser élimine toute ambiguïté du filtrage. Cependant, l'utilisation de filtres se recouvrant ne facilite pas la relecture de texte source et sa maintenance. ■

Analyse statique du filtrage

Le système de type attribue statiquement des types aux filtres, chaque filtre d'un filtrage devant reconnaître des valeurs d'un même type `t` et chaque expression d'un filtrage devant être d'un même type `t'` pour que le filtrage soit correct et de type `t -> t'`. Un filtrage est dit **exhaustif** si l'ensemble des ses filtres reconnaît toutes les valeurs possibles du type `t`. La vérification d'exhaustivité du filtrage est faite statiquement par le compilateur F#.

> *Intérêt 79* (I-F-79) : *La vérification de l'exhaustivité du filtrage empêche l'oubli de cas.*
> Le typage et la vérification d'exhaustivité du filtrage garantissent

la robustesse et la complétude des transformations de données (transformation de type `t -> t'`) ou les traitements à partir de données (traitement de type `t -> unit`). ■

Le compilateur effectue des vérifications de complétude et de robustesse du filtrage garantissant son exhaustivité et sa non-redondance.

En cas de filtrage non exhaustif, le compilateur émet un avertissement de compilation.

Recommandation 80 (R-F-80) : *N'utiliser que des filtrages exhaustifs.*
Il est recommandé d'activer l'option du compilateur qui produit une erreur à la compilation en cas de filtrage non exhaustif et d'éliminer les filtrages non exhaustifs (cf. recommandation R-F-170 pour que cette vérification émette des erreurs de compilation au lieu de simples avertissements). ■

L'ordre des filtres est déterminant en F#. En effet, le filtrage d'une valeur aboutit à l'évaluation de la première expression pour laquelle le filtre est acceptant. Le compilateur F# vérifie qu'un filtre ne masque pas complètement un filtre suivant. Ceci améliore la robustesse du filtrage présenté par l'intérêt I-F-79.

Intérêt 81 (I-F-81) : *Le compilateur détecte les filtres non utilisés.*
F# vérifie qu'un filtre ne masque pas complètement un filtre suivant et émet un avertissement si besoin. ■

Les filtres non utilisés indiquent souvent une erreur de programmation.

Recommandation 82 (R-F-82) : *Traiter les avertissements du compilateur concernant les filtres non utilisés.*

F# ne signale pas les filtres fragiles, contrairement à OCaml (cf. intérêt I-O-80).

Filtrage sur les entiers, flottants, caractères, tableaux, chaînes

Les types atomiques (entiers, flottants, caractères) et certains types prédéfinis (tableaux, chaînes de caractères, séquences) couvrent des domaines de valeurs énumérables. Le mécanisme de filtrage sur ces types est défini de manière identique à celui sur les types structurés. Un filtre est soit une valeur, soit un littéral, soit un paramètre, soit le symbole _. Le filtrage ne peut être exhaustif que si l'un

des filtres admet toutes les valeurs non filtrées par les filtres précédents donc s'il s'agit d'un paramètre ou de l'attrape-tout (_).

Filtrage des exceptions

Malgré la ressemblance syntaxique entre constructeurs de types et étiquettes d'exceptions, l'exhaustivité du filtrage de ces dernières n'est pas et ne peut pas être vérifiée par le compilateur (cf. section 2.2.3).

Danger 83 (D-F-83) : *Non exhaustivité des filtrages d'exceptions.*

Filtrage et gardes

Un filtre peut avoir une **garde** (`p when b -> e`) qui est une expression booléenne s'évaluant dans le même environnement que l'expression `e` du filtre. Sa véracité dépend de la donnée particulière qui est passée en argument du filtre. Il est donc impossible de déterminer statiquement la valeur de la garde et donc l'exhaustivité d'un filtrage ne contenant que des filtres avec gardes. Le compilateur émet un avertissement de compilation dans ce cas.

Recommandation 84 (R-F-84) : *Justifier l'utilisation de gardes.*
Les gardes facilitent l'écriture du texte source. Mais il faut les manipuler avec prudence pour conserver l'exhaustivité. ■

Filtres actifs

F# propose une notation élégante pour étendre le filtrage avec des filtres représentant les cas de résultats d'un calcul, ces filtres sont appelés **filtres actifs** ou *active pattern*. Ils permettent de filtrer des valeurs non structurelles tout en bénéficiant de l'expressivité du filtrage de valeurs structurelles. Ainsi un filtre actif sur des valeurs d'un type non-structurel `ns` aura un type de la forme `ns -> s` et un filtrage vers une valeur de type `t` utilisant ce filtre actif aura pour type `ns -> t`.

Le type intermédiaire `s` généré par F# dépend de la sorte de filtres actifs utilisé : `'a option` pour les filtres actifs partiels, `Choice<'a,'b,...>`, pour les filtres actifs à plusieurs cas, sachant qu'ici les types `'a`, `'b`, ... sont instanciés par le type `unit` pour les cas simples et par le type de chaque paramètre pour les filtres actifs paramétrés.

Intérêt 85 (I-F-85) : *Les filtres actifs facilitent la relecture et la généricité du code.*
Les filtres actifs permettent d'éviter l'utilisation de gardes et offrent un mécanisme d'abstraction évitant la définition d'un type dédié à un traitement intermédiaire. Les filtres actifs offrent les mêmes garanties de complétude et de robustesse (cf. intérêts I-F-79 et R-F-78) que le filtrage nominal. ■

Danger 86 (D-F-86) : *Imprévisibilité des effets de bords dans les filtres actifs.*
Inclure des effets de bords dans des filtres actifs rend le comportement d'un programme difficilement prévisible car le compilateur optimise les filtres actifs par mémoisation. ■

Recommandation 87 (R-F-87) : *Ne pas faire d'effets de bords dans la définition de filtres actifs.*

L'exemple suivant définit trois intervalles d'entiers sous la forme de filtres actifs (ligne 1) et permet d'effectuer du filtrage suivant ces intervalles (lignes 10–12). La vérification d'exhaustivité s'applique aux filtres actifs (comme le montre l'avertissement de compilation de la ligne 19).

```
1  let (|Intervalle1|Intervalle2|Intervalle3|) i =
2      if i < 0 then
3          Intervalle1
4      elif 100 > ithen
5          Intervalle2
6      else
7          Intervalle3
8  val ( |Intervalle1|Intervalle2|Intervalle3| ) : int -> ↵
       Choice<unit, unit, unit>
9  let f = function
10     | Intervalle1
11     | Intervalle3 -> true
12     | Intervalle2 -> false
13 val f : int -> bool
14 f 50
15 val it : bool = false
16 let g = function
17     | (Intervalle2,Intervalle2) -> true
```

```
18      | (Intervalle1,Intervalle3) -> false
19  warning FS0025: Incomplete pattern matches on this ↵
        expression.
20  val g : int * int -> bool
```

2.1.6 Flux de contrôle

F# propose une syntaxe dédiée pour les **flux de contrôle** encore appelés manipulations monadiques, expressions de calcul ou *computation expressions* et *workflows* en anglais. La définition d'un flux de contrôle permet d'étendre le langage avec une nouvelle structure d'expression représentant un processus de calcul spécialisé. Le compilateur vérifie que les valeurs mutables par valeur ne sont pas utilisées dans les flux de contrôle car celles-ci risqueraient de perturber le comportement attendu du flux.

> *Intérêt 88* (I-F-88) : *Les flux de contrôle facilitent la programmation par continuation.*

Dans l'exemple suivant, un contrôle de flux `total` est défini pour exprimer les calculs de manière totale.

```
1  type ('a,'b) total = OK of 'a | KO of 'b
2  type TotalWorkflow () =
3      member t.Bind (x,cont) =
4          match x with
5              | OK x -> cont x
6              | KO e -> KO e
7      member t.Return x = x
8  let total = TotalWorkflow ()
9  val total : TotalWorkflow
```

Ce flux de contrôle peut ainsi être utilisé dans la suite du programme en utilisant la construction `total { ... }` et les mots clés `let!` et `return`.

```
1  let concatenation x y =
2      match x,y with
3          | "",_ | _,"" -> KO "chaine vide"
4          | _ -> OK (x + y)
5  val concatenation : string -> string -> ↵
     (string,string) total
```

```
let total_concatenation_4 s1 s2 s3 s4 =
    total {
        let! x = concatenation s1 s2
        let! y = concatenation s3 s4
        return concatenation x y
    }
val total_concatenation_4 : string -> string -> string ↵
  -> string -> (string,string) total
printf "%A\n" (total_concatenation_4 "A" "B" "C" "D")
OK "ABCD"
printf "%A\n" (total_concatenation_4 "A" "B" "" "D")
KO "chaine vide"
```

Ce même exemple aurait requis l'imbrication de tests du résultat de chaque concaténation pour prendre en compte le cas de la chaîne vide. Le flux de contrôle permet de gérer de manière générique le traitement du cas particulier (ici `KO` avec le champ `Bind`) pour permettre de se concentrer sur le processus général du calcul par la suite (fonction `concatenation`).

Un certain nombre de mots clés peuvent ainsi être définis pour un flux de contrôle. Ce mécanisme permet d'ajouter des processus de calcul complexe sans complexifier la syntaxe.

> *Intérêt 89* (I-F-89) : *Extension robuste du langage avec les flux de contrôle.*

2.1.7 Stratégies d'évaluation

Évaluation stricte

L'évaluation est **stricte** en F#. L'évaluation des expression booléennes est minimale (i.e. l'expression `f ()` ne sera pas évaluée à l'évaluation des expressions `true || f()` et `false && f()`). Voir la section 2.4 pour l'évaluation des composantes des classes et objets.

Évaluation paresseuse

Le langage propose toutefois l'**évaluation paresseuse** par l'utilisation de structures de données particulières (séquences `seq<'a>`, flots `System.IO.Stream`)

et par la définition d'expressions paresseuses (`lazy ...`) qui sont évaluées par appel à leur champ `Force`.

```
let f x = lazy (printf "%A\n" "Hello"; 41 + x)
val f : int -> Lazy<int>
(f 1)
val it : Lazy<int> = <unevaluated>
(f 1).Force()
"Hello"
val it : int = 42
(f 1).Force()
"Hello"
val it : int = 42
let e = f 1
val e : Lazy<int> = <unevaluated>
e.Force()
"Hello"
val it : int = 42
e.Force()
val it : int = 42
```

Danger 90 (D-F-90) : *Complexité du mélange de modes d'évaluation.*
Le mélange de modes d'évaluation expose le développeur aux complications liées à l'ordre d'évaluation en particulier si le code fait des effets de bords. ■

Recommandation 91 (R-F-91) : *Ne pas mélanger évaluation paresseuse et effets de bords.*

Évaluation de valeurs récursives

Contrairement aux types récursifs et aux fonctions récursives, les **valeurs mutuellement récursives** (non fonctionnelles) posent le problème de leur initialisation. En OCaml, seule une forme restreinte[4] est autorisée. Les autres cas d'utilisation requièrent de faire appel aux valeurs mutables ou aux expressions paresseuses.

4. Le déclaration d'une valeur récursive ne peut pas contenir d'appels de fonctions.

Les valeurs mutuellement récursives étant courantes dans les bibliothèques d'interfaces graphiques (GUI), F# permet leur plus large utilisation. Le compilateur F# transforme de manière transparente les valeurs mutuellement récursives en expressions évaluées paresseusement et instrumente le programme pour vérifier dynamiquement que la valeur en question a bien été initialisée. L'avertissement de compilation 40 informe de ce comportement.

Ce traitement peut toutefois entraîner des comportements différents selon l'ordre de déclaration des valeurs récursives. De plus, l'avertissement 40 n'est pas émis dans certains cas.

```
let rec f () = 1 + x
and x = f ()
val f : unit -> int
val x : int = 1
let rec x = f ()
and f () = 1 + x
warning FS0040: This and other recursive references to the object(s) being defined will be checked for initialization-soundness at runtime through the use of a delayed reference. This is because you are defining one or more recursive objects, rather than recursive functions. This warning may be suppressed by using '#nowarn "40"' or '--nowarn:40'.
System.InvalidOperationException: a lazy value was accessed during its own initialization
  at System.Lazy`1[System.Int32].get_Value ()
  at FSI_0007.f ()
  at FSI_0007+clo@14-3.Invoke (Microsoft.FSharp.Core.Unit arg00@)
  at System.Lazy`1[System.Int32].get_Value ()
Stopped due to error
let rec a = 42
and b = a
and c = b
val a : int = 42
val b : int = 42
val c : int = 42
let rec a = 42
and c = b
and b = a
```

```
23 warning FS0040
24 val a : int = 42
25 val c : int = 42
26 val b : int = 42
27 let rec b = a
28 and a = 42
29 and c = b
30 warning FS0040
31 val b : int = 42
32 val a : int = 42
33 val c : int = 42
34 let rec b = a
35 and c = b
36 and a = 42
37 warning FS0040
38 val b : int = 42
39 val c : int = 42
40 val a : int = 42
41 let rec c = b
42 and a = 42
43 and b = a
44 warning FS0040
45 val c : int = 0
46 val a : int = 42
47 val b : int = 0
48 let rec c = b
49 and b = a
50 and a = 42
51 warning FS0040
52 val c : int = 42
53 val b : int = 42
54 val a : int = 42
```

Danger 92 (D-F-92) : *Imprévisibilité de la compilation et de l'exécution des valeurs mutuellement récursives.*

Recommandation 93 (R-F-93) : *Éviter l'utilisation des valeurs mutuellement récursives hors de l'utilisation de GUI ou de graphes.*

2.1.8 Récapitulatif

	Filtrage exhaustif	Définition générative	Égalité structurelle
Type produit	●	○	●
Type liste	●	○	●
Type enregistrement	●	●	●
Type somme	●	○	●
Type option	●	●	●
Classes	○	●	○
Type fonctionnel	○	○	○
Type des exceptions	○	●	○

TABLE 2.1 – Comparaison des types F# vis-à-vis du filtrage, typage et de l'égalité

2.2 Traits impératifs

Le langage F# est multi-paradigme et permet de combiner programmation fonctionnelle, impérative et orientée objet.

Certains traits impératifs sont fournis par F# et peuvent bénéficier de certaines garanties du compilateur, d'autres sont directement issus de l'interface avec la plateforme .NET (cf. section 2.9.1).

2.2.1 Références et mutables

Par défaut en F#, toute valeur est non-mutable. Ceci est un trait fondamentale de la programmation fonctionnelle qui rend les programmes plus prévisibles en restreignant les effets de bords.

> *Intérêt 94* (I-F-94) : *Non-mutabilité par défaut des valeurs.*
> Les valeurs, définies par `let` ou par `member` (cf. intérêt I-F-143), sont par défaut non-mutables. Ce choix apporte une garantie d'intégrité des données au niveau du langage : une fois définie, une donnée ne peut pas être changée au sein du code F#. Ce choix oriente aussi le programmeur vers un style de programmation fonctionnel plus pur rendant ainsi le comportement du programme plus prévisible. ■

La manipulation de valeurs mutables est toutefois utile pour représenter l'état d'un système et est indispensable quand il s'agit de s'interfacer avec certaines API. F# propose donc des types de valeurs mutables.

Contrairement aux langages bas niveau, la manipulation d'adresses est impossible en F#. Ainsi seule la valeur pointée est modifiable et pas la valeur référence.

> *Intérêt 95* (I-F-95) : *Absence de manipulation de pointeurs.*
> L'absence de manipulation de pointeurs garantit l'intégrité de la mémoire gérée par tout programme F#. ■

Il existe deux formes de mutables, les mutables par référence `ref` et les mutables par valeur `let mutable`, un mutable par référence étant construit à partir d'un mutable par valeur. Les mutables par valeur sont enregistrés dans la pile alors que les mutables par référence sont enregistrés comme des champs dans le tas. Dans les deux cas, une valeur initiale est requise.

> *Intérêt 96* (I-F-96) : *Une variable mutable est initialisée dès sa création.*

Le type `ref` des **mutables par référence** est un enregistrement possédant un champ mutable `contents` : `type 'a ref = { mutable contents : 'a }`. La fonction `ref` crée une référence à partir d'une valeur (`'a -> 'a ref`). L'accès et la modification du champ mutable se font par les opérateurs d'accès `!` et d'affectation `:=`[5]. Une référence est une valeur pointant vers le champ mutable. Il est donc possible d'avoir plusieurs **alias** d'une même référence : deux identificateurs pointant vers la même valeur. Modifier le champ mutable de l'un revient à modifier celui de l'autre. L'enregistrement est enregistré dans le tas.

```
let x = ref 1
val x : int ref = {contents = 2;}
let y = x
val y : int ref = {contents = 2;}
x := 2
!y
val it : int = 2
```

Les **mutables par valeur** sont des valeurs directement modifiables. On y accède directement par leur identificateur et on les modifies par l'opération

5. La bibliothèque standard F# définit aussi la fonction d'incrémentation `incr` et de décrémentation `decr` de type `int ref -> unit`.

d'affectation <-. Il n'est pas possible de créer un alias d'une valeur mutable comme illustré par cet exemple.

```
let mutable x = 1
val mutable x : int = 1
let y = x
val y : int = 1
x <- 2
y
val it : int = 1
```

Le compilateur F# effectue une vérification pour empêcher la capture de valeurs mutables par des fermetures (niveau expression). Elle est toutefois permise au niveau des modules et des classes (cf. section 2.4.1).

> *Intérêt 97* (I-F-97) : *Accès locaux aux valeurs mutables.*
> La vérification d'absence de capture de valeurs mutables par fermetures restreint les accès et modifications de valeurs mutables au contexte local (module et objet) rendant plus compréhensible le code et obligeant de passer par des fonctions d'accès ou des références si besoin. ■

Ainsi, dans l'exemple suivant définissant un compteur, la valeur mutable par valeur `compteur` n'est modifiable qu'en utilisant l'accesseur `c` (cf. section 2.4.1 pour plus de détails sur les accesseurs).

```
type Compteur () =
    static let mutable compteur = 0
    static member c
      with get () = compteur<-compteur+1; compteur
    static member reset () = compteur <- 0
Compteur.c
val it : int = 1
Compteur.c
val it : int = 2
Compteur.reset ()
val it : unit = ()
Compteur.c
val it : int = 1
```

> *Recommandation 98* (R-F-98) : *Préférer les mutables par valeur aux mutables par référence.*

Parce que F# impose que les modifications des mutables par valeur soient localisées à la portée de l'identificateur du mutable, il est recommandé de les préférer aux mutables par référence dont il est plus difficile de vérifier les accès en modification. ■

Les variables mutables par valeur permettent en F# de cantonner les affectations de variables mutables à leurs portées. Il est recommandé de choisir adéquatement la portée de ces variables (cf. recommandation R-F-1).

A part ces deux formes de mutables, les seules structures de données mutables de la bibliothèque standard de F# sont les tableaux `array` et les types .NET tels que `System.Array` ou `System.Collections.Generic.List`. Les noms qualifiés des types commencent par `System` pour les types .NET et par `Microsoft.FSharp` pour les types F#.

Intérêt 99 (I-F-99) : *Identification des valeurs mutables.*
Le langage F# facilite par sa syntaxe et son système de type la localisation des valeurs mutables d'un programme. Les valeurs mutables sont restreintes aux identificateurs déclarés mutables (`mutable`), aux références (`ref`), aux tableaux (type `array`) et aux valeurs de types .NET. ■

Recommandation 100 (R-F-100) : *Limiter l'usage de valeurs mutables.*
Il est recommandé de limiter autant que possible l'usage de valeurs mutables. Ainsi il est conseillé d'utiliser les chaînes de caractères ou les listes de la bibliothèque F# plutôt ceux de .NET ; de préférer transmettre un état par passage d'argument plutôt que d'utiliser une variable mutable ; d'utiliser la récursivité plutôt que l'itération avec affectation d'une valeur mutable. ■

2.2.2 Boucles

F# propose trois types de **boucles**.
- L'itérateur `while ... do` ;
- Le compteur `for ... to ... do` par exemple pour parcourir les entiers de 1 à 10 : `for i = 1 to 10 do ...` ;
- L'énumérateur `for ... in ... do` pour parcourir les éléments de l'ensemble énumérable : `for i in ensemble_enumerable do ...`, il est à noter qu'ici `i` peut être remplacé par n'importe quel filtre.

Intérêt 101 (I-F-101) : *Compteurs de boucles non-mutables.*
Il est à noter que dans le corps des deux boucles `for`, la variable de parcours est non-mutable. Cette contrainte apporte une robustesse au code en le rendant plus prévisible (cf. section 2.2.1). ■

L'utilisation d'une boucle `while` s'accompagne souvent d'une manipulation de variables mutables et d'effets de bords. Pour ne pas aboutir à des comportements imprévisibles, il est recommandé de limiter la portée de cette variable locale en utilisant la construction `let mutable` (cf. section 2.2.1). De manière générale, il est recommandé de préférer aux boucles `while` l'utilisation de fonctions récursives ou l'utilisation des fonctions de parcours des structures de listes ou de séquences de la bibliothèque standard.

Recommandation 102 (R-F-102) : *Limiter l'utilisation de la boucle* `while`.

En F#, il n'existe pas de commandes de sortie de boucle comme `break` ou `continue` de C. Ce type de commandes de sortie de boucle complexifie la compréhension des chemins d'exécution possibles d'un programme. Le comportement voulu est pourtant exprimable d'une manière facilitant la relecture de code par l'utilisation de combinaisons de structures de contrôle de flot.

Intérêt 103 (I-F-103) : *Absence de commandes de sortie de boucles.*
L'absence de commandes de sortie de boucle en F# rend plus robuste les programmes en facilitant l'identification des flots d'exécution du programme.
Il est toutefois possible de dérouter les flots d'exécution en utilisant le mécanisme d'exception (cf. section 2.2.3 et danger D-F-105). ■

2.2.3 Exceptions

Les **exceptions** permettent de définir un traitement à effectuer en réponse à une situation anormale. Il existe deux types d'exceptions en F#, les exceptions F# de type `exn` et les exceptions .NET.

Les exceptions F# sont similaires aux exceptions OCaml (cf. 1.2.3) : elles ont pour type `exn`, sont déclarées par la construction `exception`, sont levées par la fonction `raise` et peuvent être rattrapées.

Recommandation 104 (R-F-104) : *Contrôler les arguments d'exceptions.*
Il est recommandé de ne pas fournir des données confidentielles en argument aux exceptions. ■

La fonction `failwith` (de type `string -> 'a`) permet de lever l'exception `Failure m` avec un message d'erreur `m` passé en argument. La fonction `failwithf` (de type `Printf.StringFormat<'a,'b> -> 'a`) est une version de `failwith` avec mise en forme du message.

Les exceptions .NET sont organisées par classes d'exceptions héritant de la classe `System.Exception`. Elles se répartissent en exceptions levées par le système run-time .NET CLR (`System.SystemException`) et en exceptions levées par l'application `ApplicationException`. Les exceptions F# héritent directement de `System.Exception`. Cette classe définit un certain nombre de champs qui fournissent des renseignements comme le message d'erreur d'exécution à afficher (champ `Message`) ou la pile d'appels qui a mené à l'exception (champ `StackTrace`). Voir la section 2.7.1 pour une discussion sur les problèmes de divulgation que peut poser la pile d'appels.

Exceptions et chemins d'exécution

Comme dans les autres langages de programmation, les exceptions permettent en F# de détourner le flot du programme. Les problèmes potentiels sont les mêmes qu'en OCaml (cf. 1.2.3).

Danger 105 (D-F-105) : *Flot de contrôle dérouté par les exceptions.*
Le mécanisme d'exception complique la compréhension des chemins d'exécution possibles d'un programme. ■

Recommandation 106 (R-F-106) : *Ne rattraper que les exceptions identifiées.*
Il est recommandé de ne rattraper que les exceptions que l'on a créées ou découlant de l'utilisation d'une bibliothèque.

Par conséquence, il est recommandé d'utiliser une filtrage nominal des exceptions plutôt que la construction attrape-tout. ■

Il est à noter que l'exception `StackOverflowException` n'est pas rattrapable quand elle est levée par le système (cf. danger D-F-24).

Exceptions, filtrage et égalité

Les constructions `try ... with` et `try ... finally` permettent de rattraper une exception. Ce rattrapage est effectué par filtrage. Comme en OCaml, il n'est pas possible de vérifier la couverture de ce filtrage (cf. danger D-F-83). Les mêmes recommandations s'appliquent quant à l'utilisation de l'attrape-tout.

> *Recommandation 107* (R-F-107) : *N'utiliser l'attrape-tout que comme un finaliseur.*
> Il est conseillé de n'utiliser la construction rattrapant toutes les exception que comme un finaliseur : c'est à dire en levant à nouveau l'exception rattrapée par l'attrape-tout `try .. with e -> ...; raise e`. ■

> *Recommandation 108* (R-F-108) : *Éviter d'utiliser les exceptions pour les cas de fonctionnement corrects.*
> Il est préférable d'utiliser les types option ou les flux de contrôle pour traiter les comportements exceptionnels mais corrects. ■

Contrairement à OCaml, F# n'autorise pas la comparaison d'exceptions (correspondant à la propriété `StructuralComparison`, cf. section 2.1.4). Le test d'égalité est en revanche possible mais, contrairement à OCaml, retournera toujours `false` pour deux valeurs d'exceptions provenant de déclarations différentes (même si ces deux déclarations ont en commun le nom de constructeur et la structure de l'exception).

```
1  exception E
2  let e = E
3  val e : exn = E
4  let e' = E
5  val e' : exn = E
6  e = e'
7  val it : bool = true
8  exception E
9  let e'' = E
10 val e'' : exn = E
11 e = e''
12 val it : bool = false
```

> *Intérêt 109* (I-F-109) : *Impossibilité de contournement de l'encapsulation au moyen des exceptions.*

Contrairement à OCaml (cf. danger D-O-124), le langage F# rend impossible de contourner l'encapsulation par l'utilisation du test d'égalité sur les exceptions. ■

Assertions

Les **assertions** (`assert`) permettent d'inclure dans le code des tests pour vérifier à l'exécution une hypothèse sous la forme d'une expression booléenne. Si celle-ci s'avère être fausse, une exception est levée. Cette construction est à utiliser en mode débogue (cf. 2.6.4), étant désactivée en mode normal.

Danger 110 (D-F-110) : *Assertions désactivées en mode normal.*

2.2.4 Entrées-sorties

Gestion de ressources

F# propose deux constructions similaires au `let ... in` pour assurer la libération des ressources système au moment où elles ne sont plus utilisées. Ces constructions `use` et `using` permettent d'associer la demande d'accès à une ressource (l'interface `IDisposable`), les accès à cette ressource et la relâche de la ressource, cette dernière étant implicite dans le code (par appel au champ `IDisposable.Dispose`).

Recommandation 111 (R-F-111) : *Construction* `use` *pour la gestion des ressources.*
Il est recommandé d'utiliser les constructions `use` et `using` car elles garantissent une libération au plus tôt de la ressource. ■

Ce mécanisme peut être utilisé afin qu'un traitement de finalisation soit effectué à la libération d'un objet d'une classe donnée (ce mécanisme est différent de la méthode par défaut `Finalize` qui permet de définir un traitement de finalisation à la désallocation par le GC, cf. section 2.7.1).

```
1 type Classe () =
2   member o.champ () = printf "%s\n" "Appel"
3   interface System.IDisposable with
4     member x.Dispose() = printf "%s\n" "Liberation"
5 type Classe =
6   class
```

```
7       interface System.IDisposable
8       new : unit -> Classe
9       member champ : unit -> unit
10    end
11  printf "%s\n" "Debut"
12  do
13    use objet = new Classe ()
14    objet.champ ()
15  printf "%s\n" "Fin"
16  Debut
17  Appel
18  Liberation
19  Fin
```

Sérialisation

La sérialisation et la désérialisation de valeurs pour la communication ou l'enregistrement s'effectuent par les fonctions `Serialize` et `Deserialize` de la bibliothèque `System.IO` de .NET.

> *Danger 112* (D-F-112) : *Contournement de l'encapsulation par sérialisation.*
> Par défaut, toute structure de donnée est sérialisable quelle que soit sa visibilité. ■

Il est possible de désactiver la sérialisation automatique par l'adjonction de l'attribut `[<AutoSerializable(false)>]`, une tentative de sérialisation d'une valeur d'un type ayant cet attribut lèvera, à l'exécution, l'exception `System.Runtime.Serialization.SerializationException`.

> *Recommandation 113* (R-F-113) : *Désactiver la sérialisation des types de données sensibles.*

Il est possible d'exclure de la sérialisation certains champs avec l'attribut `[<NonSerialized>]`. Ceci a pour effet de sérialiser la valeur par défaut de leur type au lieu de leur valeur effective. Une valeur est sérialisée sous forme binaire pour être enregistrée dans un fichier ou transmise sur un canal de communication. Sa valeur et son type sont enregistrés. De plus, des attributs permettent de définir une sérialisation XML.

Intérêt 114 (I-F-114) : *Contrôle fin de la sérialisation.*

La désérialisation se fait donc en deux étapes : l'appel à la fonction `Deserialize` qui retourne une valeur de type `obj` et la coercition descendante (voir les détails sur la coercition descendante en section 2.4.4) de cette valeur en son type attendu. Le type fourni pour la coercition doit correspondre au type sérialisé. La désérialisation ne casse donc pas les protections du typage.

Intérêt 115 (I-F-115) : *Les données sont sérialisées avec leur type.*

La sérialisation et la désérialisation même typée ne protègent pas l'intégrité de la donnée mais seulement son type. Il est recommandé de s'assurer de l'origine de la valeur à désérialiser à l'aide par exemple de signatures cryptographiques de la donnée sérialisée.

Recommandation 116 (R-F-116) : *Vérifier l'origine des valeurs à désérialiser.*

2.3 Traits modulaires

2.3.1 Espaces de noms et modules

F# propose deux méthodes d'organisation du code d'un développement. La première se base sur le système de modules, la seconde sur le mécanisme d'espaces de noms. Ils permettent de refléter l'architecture du système.

Les **espaces de noms** (`namespace`) permettent d'organiser le programme en entités abstraites transversales : plusieurs fichiers peuvent définir un unique espace de nom. Un fichier peut définir plusieurs espaces de nom non-imbriqués : il n'existe pas de sous-espaces de noms comme il existe des sous-modules. Le compilateur interdit aux espaces de noms de contenir des valeurs ; ils ne peuvent contenir que des déclarations de types et des définitions de modules.

Les **modules** (`module`) permettent de placer des barrières d'abstraction entre différentes parties du texte source. Celles-ci apportent des garanties de cloisonnement entre sous-systèmes. Elles peuvent aussi être utilisées pour restreindre la manipulation de données, les confiner et permettre le respect d'invariants. Un fichier peut être déclaré comme étant un module : le fichier doit débuter avec une déclaration de nom de module `module M` (sans signe =). Par ailleurs, l'expression `module M = ...` est une autre façon de définir un module. Une définition de

module peut contenir des définitions de sous-modules, des déclarations de types, des déclarations d'exceptions, des abréviations de modules (alias d'un module existant), des définitions d'identificateurs et des actions (`do exp` où `exp` est une expression de type `unit`). La définition d'un module est de manière générale faite de manière monolithique mais il est toutefois possible d'étendre avec de nouveaux identificateurs un module d'un ficher depuis un autre fichier (voir section 2.3.5).

Les identificateurs d'un module sont accessibles grâce à la notation pointée : `Conteneur.identificateur` où `Conteneur` est le nom d'un module pouvant lui aussi être pointé et commencer par l'espace de nom dans lequel le module est défini.

Les espaces de noms sont compilés en espaces de noms .NET alors que les modules sont compilés en classes .NET.

2.3.2 Interfaces de modules et encapsulation

Le système de modules de F# permet d'associer une **interface de module** (contenue dans un fichier `.fsi`) à une **implémentation de module** (contenue dans un fichier `.fs`). Une interface de module est une spécification donnant l'ensemble des éléments définis par le module ainsi que leurs types et attributs tels que proposés pour un usage externe. Elle permet de les séparer de l'implémentation et de documenter la vue publique d'une bibliothèque.

Les interfaces de modules mettent en œuvre l'**encapsulation**. Une interface de module est un type de module que l'on peut obtenir en compilant un fichier `fichier.fs` avec l'option `--sig:fichier.fsi`. Pour ne pas exporter des identificateurs ou des représentations de types, il suffit de les retirer du fichier d'interface. Ces identificateurs et ces représentations ne sont plus accessibles en dehors du fichier.

> *Intérêt 117* (I-F-117) : *Le typage assure le respect de l'encapsulation des modules.*
> L'encapsulation et l'abstraction de types d'un module se font par association d'une interface au module. ■

> *Recommandation 118* (R-F-118) : *Utilisation des fichiers d'interface.*
> Il est recommandé de fournir systématiquement un fichier d'interface car celui-ci représente la vision externe du module alors que l'implémentation représente les mécanismes internes. Ceci permet au programmeur de séparer la spécification de l'usage du module

de son implémentation et donc de s'assurer qu'une mise à jour de l'implémentation ne changera pas l'interface. ■

Recommandation 119 (R-F-119) : *Utiliser les modules pour protéger les données sensibles.*
Le système de modules permet de gérer très finement l'encapsulation de données et l'abstraction des types (voir le manuel de référence pour des exemples). Il permet de tirer parti de la généricité tout en conservant l'encapsulation. ■

Danger 120 (D-F-120) : *Compilation des interfaces.*
Il est à noter que par défaut le compilateur F# ne prend pas en compte la présence d'une fichier `m.fsi` lors de la compilation du fichier `m.fs` correspondant. ■

Recommandation 121 (R-F-121) : *S'assurer que les fichiers d'interface soient pris en compte à la compilation.*
Il est indispensable de vérifier que les fichiers d'interface et d'implémentation sont bien fournis au compilateur et que chaque fichier d'interface est fourni avant le fichier d'implémentation correspondant (`fsharpc m.fsi m.fs`). Il en va de même avec Visual Studio où les deux fichiers doivent être indiqués dans le bon ordre dans la liste des fichiers du projet. ■

2.3.3 Importation de modules

Le mécanisme d'**importation de modules** par le mot clé `open` permet de ne pas répéter le **nom qualifié** des valeurs à référencer quand elles proviennent d'un même module.

En cas d'importations multiples, F# détermine la valeur à référencer en prenant en compte en priorité l'importation la plus récente. Ainsi, si la fonction `f` est définie à la fois dans les modules `A` et `B`, le programme suivant effectuera un appel à la fonction `B.f`.

```
1  open A
2  open B
3  f ()
```

Danger 122 (D-F-122) : *Ambiguïté de l'importation de modules.*
Le compilateur ne signale pas l'ambiguïté qui existe lors de l'utilisation d'un identificateur existant dans plusieurs modules importés. ■

Recommandation 123 (R-F-123) : *Utilisation des noms qualifiés.*
Pour réduire l'ambiguïté que peut induire l'importation de modules, il est recommandé d'utiliser de préférence les noms qualifiés de valeurs dans le code source.
L'attribut `[<RequireQualifiedAccess>]` permet d'indiquer lors de la définition d'un module que les accès à ses valeurs devront se faire par des appels qualifiés. Une erreur de compilation se produira en cas d'ouverture d'un module possédant cet attribut. ■

De la même manière, l'importation de sous-modules peut être ambiguë. Ainsi, si le module `A` contient un sous-module `B` définissant une fonction `f`, la fonction appelée par l'exemple précédent sera bien celle de ce sous module. Le compilateur détermine le module à importer à partir des dernières importations dans le code ou par l'option `-r` du compilateur qui permet de faire référence à une bibliothèque.

L'ordre d'importation des modules détermine le code référencé par le programme. Un module importé peut ainsi engendrer des importations non contrôlées. Par exemple, si un module `M` est importé avant un module de la bibliothèque standard tel que `List`, il peut très bien définir lui-même un module `List` qui sera importé en place de celui de la bibliothèque standard. Le comportement du programme pouvant ainsi être grandement altéré.

```
1  open M
2  open List
3
```

Danger 124 (D-F-124) : *Importation non contrôlée de modules.*

Recommandation 125 (R-F-125) : *Chemins complets d'importation.*
Il est conseillé d'indiquer les chemins complets d'importation de modules (e.g. `Microsoft.FSharp.Collections`) et d'utiliser les noms qualifiés (e.g. `List.map`). ■

Recommandation 126 (R-F-126) : *Ne pas utiliser les espaces de noms existants.*

En conjonction avec la recommandation précédente, il est conseillé de vérifier que les espaces de noms (`namespace`) sont bien réservés. Il est par exemple important de vérifier que le nom `Microsoft` n'est pas utilisé comme nom de module car cela pourrait engendrer des importations inopportunes. ■

Importation par défaut

Les modules de la bibliothèque standard sont importés par défaut.

`Microsoft.FSharp.Core` : types atomiques prédéfinis F# ;

`Microsoft.FSharp.Core.Operators` : opérateurs sur les types atomiques prédéfinis ;

`Microsoft.FSharp.Collections` : modules F# pour la manipulation de structures de données telles que les listes (module `List`) et les tableaux (module `Array`) ;

`Microsoft.FSharp.Control` : flux de contrôle prédéfinis ;

`Microsoft.FSharp.Text` : fonctions d'entrées-sorties et d'affichage (comme `printf`).

Danger 127 (D-F-127) : *Importation par défaut.*
Toute importation par défaut portant un risque potentiel, il est recommandé d'être conscient de leur existence. ■

Recommandation 128 (R-F-128) : *Considérer les noms de modules de la bibliothèque standard comme réservés.*

Importation automatique

L'attribut `AutoOpen` permet d'activer l'importation automatique d'un module dès l'importation de l'espace de nom auquel il appartient. Ce mécanisme permet de pouvoir structurer un espace de nom en modules sans imposer de les ouvrir tous indépendamment.

Recommandation 129 (R-F-129) : *Vérifier quels identificateurs seront importés par les directives* `open` *et les attributs* `AutoOpen`.

Dans l'exemple suivant, le module `M` sera importé automatiquement en cas d'importation de `Espace.De.Nom`.

```
namespace Espace.De.Nom
[<AutoOpen>]
module M =
    ...
```

2.3.4 Champs de modules

Les définitions de **champs de modules** ne peuvent pas être dupliquées ; ceci évite ainsi leur masquage. Ainsi le compilateur n'accepte pas les codes de la forme suivante.

```
module M =
    let f (x:int) = ...
    ...
    let f (x:int) = ...
```

Intérêt 130 (I-F-130) : *Lisibilité du code par non masquage de champs de modules.*
La contrainte de non masquage de valeurs *toplevel* de modules rend le code plus lisible. ■

Le masquage de déclaration prédéfinies est toutefois possible. Il est par exemple possible de masquer l'opération surchargée + en la redéfinissant (`let (+) x y = x`). Les utilisations ultérieures de l'opération + feront appel à cette définition polymorphe plutôt que la version standard adéquate.

Danger 131 (D-F-131) : *Masquage d'opérations et de fonctions standards.*

Recommandation 132 (R-F-132) : *Ne pas masquer les opérations et fonctions standards.*

Champs non-fonctionnels de modules

L'initialisation des champs non fonctionnels de modules ou **valeurs *toplevel*** est faite par la machine virtuelle .NET au moment où une valeur de ce toplevel est requise. Il peut être parfois difficile de déterminer à quel moment cette

initialisation aura lieu. L'initialisation des valeurs d'un paquetage est effectuée au premier appel d'une fonction dont la fermeture inclut un appel à une valeur toplevel.

L'initialisation n'est donc faite ni à l'importation ni lors d'un appel d'une fonction ne faisant pas appel à des valeurs d'initialisation.

> *Danger 133* (D-F-133) : *Évaluation de valeurs de modules.*
> Les expressions d'initialisation pouvant contenir des effets de bords, le comportement d'un programme peut être grandement altéré si ces expressions sont ou non évaluées et suivant le moment de cette évaluation. ■

> *Recommandation 134* (R-F-134) : *Vérifier les valeurs d'initialisation.*
> Il est recommandé de vérifier l'utilité des valeurs d'initialisation d'un module et de vérifier qu'elle seront bien initialisées au moment voulu par exemple en définissant une fonction explicite d'initialisation. ■

Un attribut `[<EntryPoint>]` permet d'indiquer au compilateur quelle fonction joue le rôle de point d'entrée du programme. Le compilateur vérifie qu'il n'y a qu'un point d'entrée, que la fonction est bien la dernière du module d'où elle provient et que la fonction en question a pour type `string list -> int` (fonction prenant une liste d'options et retournant une valeur de sortie).

2.3.5 Extension de type, module et classe

F# permet l'**extension de types**, c'est à dire l'ajout d'un champ après définition. Celle-ci s'applique aussi aux modules et aux classes qui sont considérés comme des types particuliers. Dans la suite de cette section, la notion d'extension de type englobe l'extension de classe et l'extension de module.

L'ajout d'un champ peut donc se faire après la définition, soit dans la même unité de compilation (module, source ou paquetage) — on parle alors d'*extension intrinsèque* — soit hors de l'unité de compilation — on parle alors d'*extension optionnelle*.

Voici un exemple de deux extensions. La première (lignes 8–9) est intrinsèque car elle étend avec la méthode `nb_paire_A` le type `T` défini dans la même unité de compilation. La seconde (lignes 22–27) est optionnelle car elle étend un type d'une bibliothèque externe, ici le type des listes de la bibliothèque standard.

```
type T =
    | A of T
    | B
let rec nb_paire_A = function
    | B | A B -> 0
    | A (A t) -> (nb_paire_A t) + 1
val nb_paire_A : T -> int
type T with
    member t.nb_paire_A = nb_paire_A(t)
type T =
  | A of T
  | B
  with
    member nb_paire_A : int
  end
let t = A (A (A (A (A B))))
val t : T
t.nb_paire_A
val it : int = 3
let l = [1; 2; 3; 4; 5]
val l : int list
type List<'T> with
    member l.nb_paire =
        let rec aux acc = function
            | [] | [_] -> acc
            | _ :: _ :: t -> aux (acc + 1) t
        aux 0 l
type List<'T> with
  member nb_paire : int
l.nb_paire
val it : int = 2
```

De la même manière que lors d'une définition en une fois, le compilateur ne permet pas les redéfinitions de méthodes (duplication de méthode) pour les extensions intrinsèques.

> *Intérêt 135* (I-F-135) : *Les extensions ne masquent pas les champs d'origine.*
> Les champs ajoutés par extension intrinsèque ou optionnelle ne peuvent pas masquer des champs d'origine (provenant de la pre-

mière définition du type ou d'une extension intrinsèque). ■

Il est à noter que les champs privés ou internes peuvent être définis à nouveau, par extension, passé leur visibilité. Cela n'implique toutefois pas de masquage ou redéfinition mais rajoute de l'ambiguïté.

Danger 136 (D-F-136) : *Les extensions optionnelles peuvent être masquées.*
Les définitions provenant d'une extension optionnelle peuvent être masquées par une définition dans une extension optionnelle plus récente. ■

Recommandation 137 (R-F-137) : *Limiter l'usage des extensions optionnelles.*
Les champs définis dans une définition de type ou dans une extension intrinsèque ne pouvant pas être masqués, le programmeur est assuré du code qui sera exécuté à l'appel du champ. Comme un champ défini par une extension optionnelle peut être redéfini, il est recommandé de ne pas utiliser ce type d'extension pour les traitements sensibles. ■

Danger 138 (D-F-138) : *L'extension s'applique aux valeurs du type avant extension.*
Il est important de noter que les champs définis par extension d'un type sont accessibles à toute valeur de ce type une fois l'extension définie, même si la valeur a été introduite avant l'extension. ■

Il est à noter que les extensions optionnelles ne sont pas visibles par réflexion ni en dehors de F#. Il n'est pas possible de faire appel aux champs étendus par extension optionnelle depuis un code C#.

Danger 139 (D-F-139) : *Le scellement n'interdit pas l'extension.*
Il est important de noter que le scellement (cf. section 2.4.2) n'empêche pas l'extension du type mais uniquement son héritage. ■

Danger 140 (D-F-140) : *L'extension de bibliothèques prédéfinies est possible.*
Comme il est possible d'étendre un type à partir de son nom qualifié, l'attribut `RequireQualifiedAccess` (cf. recommandation R-F-123) ne permet pas d'empêcher l'extension d'un type donné. ■

Recommandation 141 (R-F-141) : *Contrôle d'accès pour empêcher l'extension.*
Déclarer un type `private` ou `internal` permet de se prémunir contre son extension. Ce mécanisme est beaucoup plus restrictif que le scellement, le type et ses instances auront par conséquence une visibilité restreinte. ■

2.4 Traits objet

Contrairement à ses traits fonctionnels similaires à ceux de OCaml, les traits objet de F# différent fortement de ceux de OCaml présentés en section 1.4. L'approche objet de F# est basée sur celle de la plateforme .NET et en possède les mêmes traits : objets, classes, héritage simple d'implémentation, héritage multiple d'interfaces, réflexion. Certains traits peuvent toutefois être plus restrictifs ou plus expressifs en F# que dans d'autres langages .NET comme C#.

La recommandation R-F-47 donne les avantages de l'utilisation des types enregistrement par rapport aux classes.

2.4.1 Objets et classes

Un **objet** est une valeur symbolique possédant deux types de **champs** : les **champs d'état** ou d'instance qui représentent l'état interne de l'objet (ils peuvent être mutables) et les **champs d'accès** ou méthodes qui sont des fonctions permettant d'interagir avec l'objet.

Chaque objet est une instance d'une classe donnée. Une **classe** permet de mutualiser la définition des champs d'objets. Une classe n'est pas une valeur mais un créateur d'objets. Les **champs de classe** sont des champs partagés par l'ensemble des objets instances de la classe, ils sont accessibles hors d'une instance particulière. Les classes sont des types pour le langage F#, elles sont définies par le mot-clé `type`.

Les champs d'état ne sont accessibles que depuis la définition de la classe ou de l'objet. Il est possible de contrôler la portée des champs d'accès.

Les objets et classes de F# peuvent donc posséder des champs de plusieurs sortes suivant quatre caractères combinables identifiés syntaxiquement et décrits par la table 2.2.

Caractère	Mot-clé	Description
Portée	*(par défaut)*	Le champ est associé à un objet.
	`static`	Le champ est associé à une classe.
État/accès	`let`	Le champ représente un état de l'objet ou de la classe. La définition du champ est évaluée à la déclaration de la classe pour les champs de classe ou bien à la création de l'objet.
	`member`	Le champ représente un accès à l'objet ou à la classe. La définition du champ est évaluée à chaque appel.
	`abstract` `default` `override`	Le champ représente un accès à l'objet redéfinissable par héritage. Les mots-clés correspondent à la déclaration de type du champ, la définition de sa valeur par défaut et sa redéfinition.
Mutabilité	*(par défaut)*	Le champ n'est pas mutable.
	`mutable`	Le champ est mutable. Seuls les champs d'état (`let`) peuvent être déclarés mutables.
Contrôle d'accès	*(par défaut)*	Le champ est globalement accessible (le marqueur `public` peut aussi être utilisé).
	`private`	Le champ est accessible uniquement depuis la définition de la classe. Les champs d'état `let` sont toujours privés.
	`internal`	Le champ est accessible uniquement depuis le paquetage.

TABLE 2.2 – Mots-clés pour la déclaration de champs de classes et d'objets suivant quatre caractères combinables

Intérêt 142 (I-F-142) : *Syntaxe explicite pour distinguer les caractères de champs.*
La syntaxe de F# distingue strictement les caractères des champs d'objet ou de classe augmentant ainsi la lisibilité du programme. Le fait que la syntaxe soit axée sur la sémantique permet un meilleur contrôle du code par le programmeur ou le relecteur. ■

Intérêt 143 (I-F-143) : *Non-mutabilité par défaut des champs d'objets.*
Contrairement à C#, les champs sont par défaut non-mutables. Ce choix permet une meilleure localisation des valeurs mutables dans les programmes F# (cf. section 2.2.1 et intérêt I-F-99). ■

Surcharge des champs

Comme pour les opérateurs (cf. section 2.1.1), les méthodes d'un objet ou d'une classe peuvent être surchargées. Le compilateur n'autorise toutefois pas la définition des méthodes de mêmes noms avec des arguments de même type.

Intérêt 144 (I-F-144) : *Déterminisme des appels de méthodes surchargées.*
Le compilateur affiche une erreur de compilation si, à l'appel d'une méthode, il n'arrive pas à déterminer quelle définition doit être appelée d'après les informations de type qu'il a inféré. ■

Le compilateur F# accepte les définitions surchargées mêmes si elles risquent d'aboutir à des appels ambigus. Dans l'exemple suivant, le champ `surcharge` est surchargé, chaque surcharge ayant un type différent : `int -> int` et `int -> int -> int`. Le compilateur ne rejette pas ce programme même si il ne pourra déterminer la définition à utiliser pour les appels au champ `surcharge` (`(new A()).surcharge 1 1` ou `(new A()).surcharge 1`).

```
type A () =
    member a.surcharge x = 1 + x
    member a.surcharge x y = x + y
type A =
  class
    new : unit -> A
    member surcharge : x:int -> int
    member surcharge : x:int -> y:int -> int
```

```
9 end
```

Recommandation 145 (R-F-145) : *Regrouper les paramètres de champs en un paramètre produit.*
Il est recommandé de définir les champs avec un paramètre produit plutôt que plusieurs paramètres pour éviter l'ambiguïté de la surcharge de champs. ■

Constructeurs d'objets

Un champ de classes spécial, le **constructeur d'objets** (`new`) permet l'instanciation d'un nouvel objet de la classe. Il est défini implicitement par le paramétrage de la définition de classe ou explicitement comme tout autre champ. Il peut être surchargé. Le mot clé `new` en F# ne doit pas être confondu avec une allocation mémoire dans d'autres langages comme C#.

Classes polymorphes

F# permet la définition de **classes polymorphes** ou génériques. Une variable de type est passée en paramètre à la définition de la classe : `type Classe_generique<'a> () = ...` (cf. section 2.1.3 sur les types paramétrés et leurs contraintes associées).

Champs par défaut

En .NET, tout objet du langage dérive de la classe .NET `System.Object`. Cette classe définit deux méthodes statiques (`Equals` et `ReferenceEquals`) et les méthodes d'instance `ToString`, `GetHashCode`, `Equals`, `GetType`, `MemberwiseClone` et `Finalize`.

Danger 146 (D-F-146) : *Existence de champs de base pour chaque valeur du langage.*

Certaines de ces méthodes ne sont pas redéfinissables comme `GetType` mais d'autres le sont (la redéfinition de méthodes est décrite plus loin en section 2.4.2). La redéfinition de méthodes comme `Equals` ou `GetHashCode` peut avoir un impact important puisque de nombreuses opérations telles que l'égalité leur font appel implicitement. Par défaut pour les objets, l'égalité est référentielle. Voir la section 2.1.4 pour plus de détails sur l'égalité et la comparaison en F#.

Recommandation 147 (R-F-147) : *Prendre garde en cas de redéfinition de champs de base.*
Les champs de base sont largement utilisés, il est recommandé une grande attention et le respect de leurs propriétés attendues en cas de besoin de redéfinition. ■

Accès et modification de champs

F# fournit une syntaxe particulière pour les accès et modification de champs d'objets. Une syntaxe simplifiée (`champ` et `champ <- ...`) est associée aux membres proposant des accesseurs : `member c.champ with get() = ... and set(v) = ...`. De la même manière, il est possible de définir un champ spécial `Item` pour la création d'index `a.[i]` (avec sous-champ `get` pour l'accès et `set` pour la modification : `a.[i]<-...`).

Intérêt 148 (I-F-148) : *Encapsulation des champs d'état.*
Les champs d'état sont encapsulés et ne se donc pas visibles hors de la définition de la classe. ■

Intérêt 149 (I-F-149) : *Séparation entre variable d'état d'un objet et ses fonctions d'accès et de modification.*
La syntaxe simplifiée pour les définitions des fonctions spéciales d'accès, de modification facilite leur utilisation et donc le contrôle qu'elles apportent sur la manipulation de l'état interne de l'objet. ■

2.4.2 Héritage et interfaçage

F# propose deux mécanismes pour définir une hiérarchie de classes : l'héritage simple et l'interfaçage.

Héritage simple, champs abstraits et redéfinition

Une classe peut hériter des champs d'une autre classe. Il s'agit de l'**héritage simple** (`inherit`). F# n'autorise pas l'héritage multiple.

F# autorise la **redéfinition de champs** (`override`) lors de l'héritage. Cette redéfinition n'est possible que pour les champs abstraits.

Intérêt 150 (I-F-150) : *Redéfinition explicite de champs.*
La redéfinition de champs se fait de manière explicite par le mot-clé

`override`. Seuls les champs déclarés explicitement par le mot-clé `abstract` sont redéfinissables. Il n'est pas possible de redéfinir les champs définis avec le mot-clé `member`. ■

Un **champ abstrait** (**abstract**) est un champ d'objet et d'accès dont seul le nom et le type sont fournis. Instancier une classe dont un champ ne serait pas défini poserait problème à l'appel de ce champ, le compilateur impose donc aux classes de fournir une définition par défaut (`default`).

Intérêt 151 (I-F-151) : *Le compilateur s'assure que tous les champs d'objets ont une définition.*

Il est toutefois possible de définir une classe possédant des champs abstraits sans définition par défaut. Elles sont qualifiées d'**a**bstraites et doivent être définies avec l'attribut `[<AbstractClass>]`. Il n'est pas possible de les instancier.

Intérêt 152 (I-F-152) : *Le compilateur interdit l'instanciation des classes abstraites.*

Les classes abstraites permettent de définir une hiérarchie de classes en n'autorisant que l'instanciation des classes les plus spécialisées.

Il est à noter que les mots-clés `default` et `override` sont équivalents même si le premier est habituellement utilisé après une déclaration de champs abstraits et le second lors d'une redéfinition accompagnant un héritage.

Type classe et interfaçage

F# propose un mécanisme d'interface de classes. Un **type classe** (ou interface) peut être défini, il s'agit d'un ensemble de noms de champs et de leurs types. Une classe peut alors s'interfacer avec un type classe `IT` en fournissant une définition à chaque champ déclaré par `IT`. Un tel **interfaçage** se note `interface IT with ...`. Un objet d'une classe ainsi définie pourra être utilisé en lieu et place des valeurs de type `IT`. Ce mécanisme d'interface permet de définir des fonctions s'appliquant à des objets de classes n'ayant pas forcément de relations d'héritage entre elles.

Intérêt 153 (I-F-153) : *Le compilateur vérifie les interfaces de classes.*

Typage des traits objet

Le typage des traits objet de F# est nominal (contrairement à celui de OCaml qui est structurel). Ceci implique qu'il est impératif de déclarer une interface pour pouvoir définir une fonction s'appliquant sur des objets proposant des champs d'accès d'un certain type.

En F# le typage vérifie statiquement qu'à chaque appel de champ d'un objet correspondra bien un code à exécuter, ce qui est appelé *early binding* en anglais. Les mécanismes de redéfinition font qu'il n'est pas possible de déterminer statiquement quel code parmi les différentes définitions d'un même champ correspondra à chaque appel. En F# et en .NET, cette détermination est faite dynamiquement et s'appelle *dynamic dispatch* en anglais et liaison retardée en français (à ne pas confondre avec le *late binding* qui consiste à ne déterminer que dynamiquement qu'un champ existe bien pour un appel donné).

En F#, l'existence d'un code exécutable est vérifié statiquement pour chaque appel de méthode, le choix du code à exécuter est fait dynamiquement mais il est toutefois possible avec les mécanismes de réflexion de .NET d'effectuer des appels dynamiques de champs, *late binding* en anglais (cf. section 2.4.4).

> *Intérêt 154* (I-F-154) : *Le compilateur garantit qu'à chaque appel de champ correspondra bien un code.*

Classes scellées

Il est possible d'empêcher l'héritage d'une classe donnée en la déclarant **scellée** avec l'attribut `[<Sealed>]`.

> *Intérêt 155* (I-F-155) : *Le compilateur vérifie statiquement que les classes scellées ne sont pas héritées.*

> *Intérêt 156* (I-F-156) : *Contrôle de l'héritage.*
> Les classes scellées permettent d'éviter certaines utilisations malveillantes de l'héritage (usurpation, *spoofing*) : hériter d'une classe pour en obtenir ses accès normalement contrôlés, ou redéfinir ses méthodes pour en éliminer les contrôles. ■

Contrairement à C# (et plus généralement .NET), seules les classes et non les méthodes peuvent être scellées en F#. D'une certaine manière, les champs d'objet non redéfinissables `member` de F# peuvent être vus comme des méthodes

scellées, mais elles le sont tout au long de la hiérarchie alors que les méthodes scellées de C# changent de nature et deviennent non-redéfinissables. Le contrôle de l'héritage en F# est donc moins précis qu'en C#. Il est possible de masquer tout champ hérité, la nouvelle définition masquant la précédente qui reste toutefois accessible par le mot clé `base`.

2.4.3 Contrôle d'accès

Les champs d'accès des objets sont par défaut accessibles sans restriction. F# propose deux marqueurs d'**accessibilité** (aussi applicables aux autres expressions de type, aux définitions et aux constructeurs) pour contrôler leur accessibilité.

Le marqueur `internal` restreint l'accès au champ à un **paquetage** ou *assembly* (cible de compilation prenant la forme d'un fichier `.dll` ou d'un exécutable `.exe`). Le marqueur `private` restreint l'accès aux champs à la définition du type. Il est possible d'étendre la visibilité des champs internes à des paquetages identifiés (cf. section 2.7.2) en utilisant l'attribut `InternalsVisibleTo`. Il n'existe pas de `protected` mais les `protected` de .NET sont respectés.

Les marqueurs d'accessibilité peuvent être appliqués à l'ensemble des éléments du langages (types, identificateurs, modules, etc.). Il est à noter qu'il n'est pas possible de modifier l'accessibilité des abréviations de types (`type t = int`), il faut pour cela définir un nouveau type (`type t = T of int`).

> *Recommandation 157* (R-F-157) : *Utilisation d'interfaces explicites.*
> Pour rendre les marqueurs d'accessibilité plus lisibles, il est conseillé de fournir un fichier d'interface. Ce fichier d'interface ne contenant que les types et les marqueurs des éléments définis dans le fichier d'implémentation, il permet d'en avoir une vue globale. ■

> *Danger 158* (D-F-158) : *Contournement des marqueurs d'accessibilité par réflexion.*
> Les champs à accessibilité restreinte restent visibles par les opérations de réflexion de .NET. En utilisant les opérations de réflexion de .NET il est possible d'accéder à tous les champs d'un objet sans restriction (cf. section 2.10.1 portant sur .NET CAS). ■

> *Danger 159* (D-F-159) : *Accessibilité partagée par les objets d'une même classe.*
> Comme c'est le cas en Java et C++, et contrairement à OCaml,

une méthode privée d'un objet est accessible à un autre objet de la même classe (*siblings*). ■

Intérêt 160 (I-F-160) : *L'héritage préserve l'accessibilité.*

Comme indiqué précédemment, les définitions de types, classes et modules peuvent aussi avoir un marqueur d'accessibilité. Le compilateur s'assure par typage que les valeurs d'un type auront une accessibilité au moins aussi forte que celle de leur type.

L'exemple suivant l'illustre avec d'abord la tentative de création d'une valeur publique d'un type privé puis avec la tentative de faire fuir la valeur avec un champ du type.

```
1 type private Priv () =
2     member p.soi = p
3 let p1 = new Priv ()
4 error FS0410: The type 'Priv' is less accessible than ↵
    the value, member or type 'val p1 : Priv' it is ↵
    used in
5 let private p2 = new Priv ()
6 val private p2 : Priv
7 let p_fuite = p2.soi
8 error FS0410: The type 'Priv' is less accessible than ↵
    the value, member or type 'val p_fuite : Priv' it ↵
    is used in
```

Intérêt 161 (I-F-161) : *Contrôle de la concordance des accessibilités de valeurs avec celle de leur type.*

Pour imposer un contrôle des accès au niveau bytecode il est possible d'utiliser .NET CAS (cf. section 2.10.1).

2.4.4 Réflexion

Le langage F# et les bibliothèques .NET proposent plusieurs degrés de réflexion allant de l'observation non-intrusive à la manipulation à l'exécution.

Coercitions

F# offre un mécanisme de **coercition** explicite. D'une part, on peut attribuer le type de la classe à un objet de l'une de ses sous-classes (coercition ascendante ou *upcast* notée `:>`). Dans ce cas, le contrôle de la coercition peut être fait statiquement. D'autre part, il est possible d'attribuer à un objet d'une classe le type de l'une de ses sous-classes (coercition descendante ou *downcast* notée `:?>`). Pour ce dernier cas, le contrôle ne peut être que dynamique, cette coercition ne pouvant être validée qu'à l'exécution grâce aux informations de typage obtenues à l'exécution. L'exception `System.InvalidCastException` est levée si la coercition descendante est invalide. F# possède une opération de test dynamique de coercition (`:?`) permettant de tester dynamiquement par filtrage le type d'un objet.

> *Recommandation 162* (R-F-162) : *Tester les coercitions dynamiques pour les rendre plus robustes.*

Une notation spécifique pour les annotations de type permet d'effectuer une coercition implicite à l'appel d'une fonction : `let f (x : #SomeType) = ...` (`#SomeType` est un raccourci de `'a when 'a :> SomeType`).

Réflexion .NET

Certaines fonctions de **réflexion** importées par défaut permettent d'obtenir des informations sur la représentation interne des types : `typeof<'a>` renvoie le type .NET et `sizeof<'a>` retourne l'espace pris dans le tas.

Des bibliothèques .NET permettent de consulter plus largement la représentation interne des types et valeurs à l'exécution. Comme indiqué par le danger D-F-158, les marqueurs de contrôle d'accès F# sont valables au niveau du langage et du bytecode mais ne sont pas pris en compte par les fonctions de réflexion.

Il existe deux bibliothèques de réflexion. La première, `System.Reflection`, est un bibliothèque .NET et présente les types et les valeurs tels qu'ils apparaissent dans le bytecode c'est à dire sous la forme compilée par F#. La seconde, `Microsoft.FSharp.Reflection`, est spécifique à F# et effectue une sorte de décompilation des types et valeurs bytecode sous leur forme initiale F#.

Instanciation dynamique

Des fonctions proposées dans ces bibliothèque de réflexion permettent aussi de créer dynamiquement de nouveaux types et d'**instancier dynamiquement** de nouvelles valeurs.

> *Danger 163* (D-F-163) : *Les garanties du typage ne s'appliquent pas aux instanciations dynamiques.*

> *Recommandation 164* (R-F-164) : *Justifier toute utilisation d'instanciations dynamiques.*

Chargement dynamique

La bibliothèque `System.Reflection` permet de charger dynamiquement du bytecode (classe `Assembly`) et la bibliothèque `System.Activator` permet d'instancier le bytecode.

> *Recommandation 165* (R-F-165) : *Vérifier dynamiquement que le bytecode chargé a bien le type attendu.*

> *Recommandation 166* (R-F-166) : *Justifier tout chargement dynamique de bytecode.*

La plateforme .NET vérifie que le bytecode est bien compatible. Il est possible d'effectuer des contrôles supplémentaires sur le bytecode au chargement (cf. section 2.7.2 et 2.10.1).

Expressions de code

F# offre la possibilité de manipuler des valeurs sous la forme d'un Arbre de Syntaxe Abstraite (AST) F#. Ces valeurs sont appelées **expressions de code** ou *quotation*. Les expressions `<@ e @>` et `<@@ e @@>` ont pour valeur l'AST typé de l'expression F# `e` et pour type `Quotations.Expr<'T>` et `Quotations.Expr` où `'T` est le type de `e`.

L'attribut `[<ReflectedDefinition>]` placé devant une définition d'un identificateur `i` permet d'obtenir son expression de code en plus de le définir.

Ce mécanisme permet de manipuler et d'évaluer différemment des expressions F# tout en bénéficiant du typage statique (l'expression `e` est typée).

Intérêt 167 (I-F-167) : *Manipulation d'expressions de code typées.*

2.5 Sémantique du langage

La spécification du langage F# est décrite par le document [Syme, 2010]. F# partage une partie de la sémantique des traits fonctionnels et impératifs de OCaml (cf. section 1.5). Les extensions principales apportées par F# vis-à-vis de cette sémantique ont donné lieu à des publications : sur les données mutuellement récursives [Syme, 2006], sur les unités de mesure [Kennedy, 1997], sur les filtres actifs [Syme *et al.*, 2007b], sur le GC et la programmation réactive [Petricek et Syme, 2010] ou sur la programmation asynchrone [Syme *et al.*, 2011]. Le développement initial de F# fut entrepris par Don Syme en parallèle à l'ajout de types génériques au langage de *bytecode* de .NET [Yu *et al.*, 2004].

2.6 Compilation

2.6.1 Vérifications et options de compilation

Le compilateur F# effectue un grand nombre de vérifications statiques sur le code source avant de compiler le programme en bytecode F#.

Intérêt 168 (I-F-168) : *Analyses statique de code source.*
Les vérifications proposées par le compilateur F# sont statiques, elles ne requièrent donc pas d'exécution du programme et apportent des garanties pour toutes les exécutions du programme. De plus ces vérifications sont basées sur des théories qui ont permis de démontrer leurs décidabilité, complétude et correction (cf. section 2.5). ■

Avertissements de compilation

Ces vérifications peuvent être systématiques ou optionnelles. Elles donnent lieu selon le cas à des erreurs ou des avertissements de compilation. Certaines vérifications comme le non débordement de tableaux sont dynamiques et donnent lieu à des exceptions à l'exécution.

Recommandation 169 (R-F-169) : *Activer toutes les vérifications du compilateur.*
Il est recommandé d'activer toutes les vérifications que le compilateur puisse faire en levant le niveau d'avertissements au maximum (option `--warn:4` du compilateur). ■

Recommandation 170 (R-F-170) : *Demander au compilateur de considérer tout avertissement comme une erreur.*
Il est aussi recommandé de demander au compilateur par son option `--warnaserror+` d'émettre une erreur de compilation au lieu de simples avertissements. ■

Ces recommandations sont essentielles pour le processus de développement mais non indispensables en cas de diffusion de sources.

Recommandation 171 (R-F-171) : *Ne pas désactiver les vérifications dynamiques de dépassements.*
Il est recommandé de ne pas désactiver les vérifications dynamiques de dépassement qui sont actives par défaut. L'option `--checked+` du compilateur les activent explicitement. ■

L'affichage des avertissements de compilation peut être déterminé par des options de compilation mais aussi directement par des directives dans le code source (cf. section 2.6.3). La directive `#nowarn "25"` par exemple désactive l'avertissement de compilation concernant les filtrages incomplets. Ces directives permettent de désactiver ponctuellement des vérifications pour un fichier source sans les désactiver pour l'ensemble des fichiers.

Intérêt 172 (I-F-172) : *Contrôler finement la désactivation d'avertissement par la directive* `nowarn`.

Il n'existe pas d'option du compilateur permettant de désactiver ces directives. Il n'existe pas non plus de directive activant des avertissements.

Danger 173 (D-F-173) : *Directives de compilation dans le code source.*
Des directives de compilation dans le code source F# permettent de modifier certaines options de compilation comme le choix des avertissements de compilation à émettre. ■

Recommandation 174 (R-F-174) : *Justifier les désactivations de vérifications du compilateur.*

Danger 175 (D-F-175) : *Contournement des vérifications du compilateur par les fonctions de la bibliothèque* `Unchecked`.
Les fonctions de la bibliothèque `Unchecked` ne sont pas sujettes à certaines vérifications du compilateur (cf. danger D-F-73). ■

Recommandation 176 (R-F-176) : *Ne pas utiliser les fonctions de la bibliothèque* `Unchecked`.

Chargements de bibliothèques

L'approche par défaut de la plateforme .NET est celle du chargement dynamique de bibliothèques de bytecode. La bibliothèque est chargée à l'exécution. Il est toutefois possible de forcer le chargement statique d'une bibliothèque à la compilation avec l'option `--static-link`. Le fichier `.dll` passé en argument sera ainsi lié à la création de l'exécutable `.exe`. Il est possible de créer un exécutable incluant toutes les bibliothèques qu'il requiert en utilisant l'option `--standalone`. Tous les fichiers `.dll` dont dépend le programme sont ainsi liés.

Recommandation 177 (R-F-177) : *Assurer l'intégrité du programme par chargement statique.*
Il est recommandé de lier et charger statiquement les bibliothèques d'une application sensible pour éviter la possibilité de chargement de versions malicieuses des bibliothèques à l'exécution. Cela pose toutefois la question de la mise à jour des bibliothèques qui ont été incluses dans l'application. ■

2.6.2 Production de bytecode

Le **compilateur** F#, `fsharpc`, produit du bytecode Common Intermediate Language (.NET CIL) pour la plateforme .NET.

Propriétés du bytecode produit

Le bytecode .NET CIL est orienté objet. Les objets du bytecode sont typés, ce qui offre des garanties de sûreté à l'exécution. Le compilateur traduit des

constructions haut niveau F# et les types F# en classes, méthodes et instructions .NET CIL.

Un type structurel F# est compilé en classe .NET CIL abstraite, ses constructeurs sont compilés en sous classes de cette classe abstraite. Un module est compilé en une classe .NET CIL abstraite scellée dont toutes les méthodes sont statiques. Les fonctions masquées d'un module sont compilées en méthodes dont l'accès est restreint au paquetage. Une classe F# est directement compilée en une classe .NET CIL.

Une valeur non-mutable associée à un identificateur `ident` est compilée en une méthode .NET CIL `get_ident`. Une valeur mutable associée à un identificateur `ident` est compilée en deux méthodes .NET CIL `get_ident` et `set_ident`.

L'utilisation des mécanismes d'encapsulation (cf. section 2.3.2) et de contrôle d'accès (cf. section 2.4.3) a pour effet de contrôler l'utilisation du code en rajoutant des marqueurs .NET CIL d'accessibilité (typiquement `assembly` ou `private` en lieu et place de `public`).

> *Intérêt 178* (I-F-178) : *Transposition de propriétés du source en propriétés du bytecode.*
> Certaines propriétés garanties dans le code source (non-mutabilité, encapsulation, nombre de constructeurs fixe, restriction d'accès) sont ainsi transposées dans le bytecode. ■

Certains attributs donnent au programmeur la possibilité de choisir la construction bytecode voulue. Par exemple, l'attribut `[<StructAttribute>]` permet de choisir de compiler un type classe F# en un `struct` .NET dont l'accès est directement fait dans la pile. Ceci peut augmenter l'efficacité pour des objets de petite taille. L'allocation dans la pile et non dans le tas implique que l'égalité est une égalité de valeurs et non de références comme pour les objets.

Il est à noter que les types atomiques `int`, `char`, etc. sont des types `struct`.

Lisibilité du bytecode

La représentation interne de types des fonctions et des valeurs sont lisibles dans le bytecode (fichiers `.dll` ou `.exe`). L'attribut `[<CompiledName(...)>]` permet de contrôler le nom d'une valeur dans le bytecode compilé.

> *Danger 179* (D-F-179) : *Lisibilité du bytecode produit.*
> Le bytecode produit par F#, comme tout bytecode .NET, est un code lisible ; .NET ne propose ni chiffrement ni offuscation de by-

tecode. Les protections apportées par le langage F# et le bytecode .NET portent sur l'exécution et non les fichiers. ■

Recommandation 180 (R-F-180) : *Proscrire les données sensibles du code source.*
Il est recommandé de ne pas inclure de données sensibles dans le code source F#. ■

Il existe de nombreux produits pour le chiffrement ou l'offuscation de bytecode .NET.

Décompilation

Il existe de nombreux outils permettant de décompiler un fichier `.exe` ou `.dll` en code source .NET CIL ou en codes de plus haut niveau. L'outil `Reflector` permet ainsi la décompilation vers du C#, VB.NET et F#. Les outils `monodis` de Mono et `Ildasm` (Intermediate Language Disassembler) de *Microsoft* permettent la décompilation en code source .NET CIL.

La bibliothèque `Cecil` de Mono offre des possibilités avancées de réflexion de bytecode .NET CIL.

2.6.3 Directives

Le système de directives de F# est un système de macros limité à des directives de compilation, de spécialisation de code et de localisation. Il n'y a pas de `#define` comme en C, les directives ne permettent donc pas de remplacer un élément syntaxique par du code source.

Intérêt 181 (I-F-181) : *Mécanisme de directives de compilation limité.*
Le mécanisme de directives de F# étant limité, il facilite la compréhension de code en ne modifiant pas la syntaxe du langage. ■

La directive `#light` permet en outre d'activer ou de désactiver la syntaxe allégée F#.

Spécialisation de code

Il est possible de spécialiser le code selon des variables passées en option de compilation : `#if ... #else #endif` pour tester une valeur définie par l'option de compilation `-define`.

Localisation

En mode débogue, le compilateur F# mémorise les localisations (numéro de ligne, répertoire et nom du fichier) dans le code source de chaque expression. Ces informations sont utilisées lors de l'affichage d'erreurs de compilation ou d'exécution. Il est possible de modifier cette information d'origine du code avec la directive `#line`. Cette directive est particulièrement utile pour les générateurs de code. Les identificateurs `__LINE__`, `__SOURCE_DIRECTORY__` et `__SOURCE_FILE__` sont des identificateurs ayant comme valeur une chaîne de caractères représentant les informations de localisation courante.

> *Intérêt 182* (I-F-182) : *Traçabilité de code par localisation.*
> Le mécanisme de localisation donne la possibilité de tracer l'origine de codes sources générés. ■

> *Danger 183* (D-F-183) : *Falsification de la localisation de codes sources.*
> Des informations de localisation factices peuvent donc être placé dans le code source de bibliothèques malveillantes. Il rendrait inutile une vérification de l'origine de codes par utilisation de la variables `__SOURCE_DIRECTORY__`. ■

2.6.4 Mode débogue

F# propose une compilation en mode débogue (option `--debug+`). Les assertions (cf. section2.2.3) dans le code F# sont activées en mode débogue.

> *Intérêt 184* (I-F-184) : *Assertions activées en mode débogue.*

Ces assertions ne sont pas activées en mode normal de production (cf. danger D-F-110).

> *Recommandation 185* (R-F-185) : *Préférer les exceptions au mécanisme des assertions pour les vérifications dynamiques.*

Les informations de localisation ne sont incluses dans le code compilé qu'en mode débogue.

2.7 Modèles d'exécution et propriétés associées

Les programmes développés en F# ont vocation à être exécutés sur la plateforme logicielle .NET via la production de bytecode. Le modèle d'exécution pour F# est donc par exécution de bytecode sur machine virtuelle (cf. section 2.7.1). La distribution F# propose aussi une boucle interactive et une exécution en mode script (cf. section 2.7.3). Des méthodes de compilations alternatives permettent d'exécuter un programme F# en mode natif (cf. section 2.7.4).

2.7.1 Exécution sur machine virtuelle

Le sujet de ce document n'étant pas la plateforme .NET, il n'abordera pas dans le détail les mécanismes et les propriétés de Common Language Runtime (.NET CLR).

Le modèle d'exécution de programmes F# possède les caractéristiques de l'exécution sur machine virtuelle .NET CLR :

- Un code .NET CIL est exécutable sur toute machine possédant une plateforme logicielle implémentant les spécifications dites Common Language Infrastructure (.NET CLI). Cette plateforme est composée d'une machine virtuelle, appelée .NET CLR, et d'une bibliothèque standard, appelée Base Class Library (.NET BCL). L'implémentation principale se nomme Microsoft .NET Framework sous Windows. Il existe une plateforme alternative de développement et d'exécution, appelée Mono, de Novell sous Linux et des plateformes alternatives d'exécution : .NET Compact Framework (pour les mobiles et l'embarqué), XNA (pour la X-Box, Zune et Windows Phone 7) et le client Microsoft Silverlight (pour le Web).
- Un code .NET CIL compilé est indépendant de sa plateforme d'origine et donc exécutable sous une autre. Une option permet toutefois d'indiquer à la compilation la plateforme cible ; elle ne spécialise pas le code mais indique quelle plateforme est ciblée pour anticiper les cas de dépendances relativement à du code natif.
- Les langages de programmation de la plateforme .NET (C#, VB.NET, F#, etc) peuvent partager leurs bytecodes .NET CIL et donc être conjointement utilisés dans un même développement (cf. section 2.9.1).

Par défaut, la plateforme .NET effectue une **compilation à la volée** ou Just-In-Time (JIT) du bytecode en code natif à l'exécution. Ceci permet de trouver

un juste milieu entre portabilité et efficacité.

Gestion de la mémoire

La machine virtuelle .NET CLR possède un GC qui se charge de la désallocation. Pour s'assurer de la libération d'une valeur au plus tôt lors de l'exécution d'un programme il est préférable de limiter les possibilités que des références à cette valeur subsistent après traitement, en réduisant la portée des identificateurs (cf. section 2.1.1).

La méthode `Finalize` permet de programmer des actions de finalisation à effectuer avant qu'une valeur ne soit libérée par le GC.

Vérifications de bytecode

Le bytecode .NET CIL est typé. Au chargement du bytecode, la .NET CLR effectue une **vérification dynamique des types** garantissant ainsi que les accès mémoire se font en respectant les types et les droits d'accès.

> *Intérêt 186* (I-F-186) : *Garanties apportées par le typage de bytecode.*
> Les vérifications de typage du bytecode .NET CIL par la .NET CLR apportent des garanties d'accès cohérent aux objets et à leurs méthodes vis-à-vis de leurs types et de leurs accessibilités. ■

Pile d'appels

.NET CLR fournit des informations sur l'environnement courant d'exécution via la classe `System.Environment`. La pile d'appels `StackTrace` contient ainsi la séquence courante d'appels de fonctions. En cas de levée d'exception `e`, la propriété `e.StackTrace` contient cette pile d'appels. Si l'exception n'est pas rattrapée, la pile d'appels est affichée pour aider le débogue avec le message décrivant l'exception. Elle contient les noms qualifiés des fonctions appelées avec le type de leurs arguments et, si l'option de débogue `--debug+` a été passée à la compilation, les localisations des définitions de ces fonctions (nom du fichier et ligne pour chaque appel). La pile d'appels n'inclut pas les valeurs manipulées par le programme mais inclut le chemin d'exécution de la levée de l'exécution à son point de rattrapage (qui est le point de départ du programme s'il n'y a pas de rattrapage explicite). Par ce canal caché, des valeurs du programme pourraient être déduites de ce chemin d'exécution.

Danger 187 (D-F-187) : *Divulgation du chemin d'exécution par levée d'exception.*
Le chemin d'exécution divulgué par défaut à la levée d'exception peut permettre de déduire des informations sensibles sur l'état du système. ■

Recommandation 188 (R-F-188) : *Masquer la pile d'appels des exceptions de modules sensibles.*
Il est préférable de masquer la pile d'appels à la définition des exceptions qui seront levées par du code sensible pour cacher cette information.

```
1 exception Foo of string
2   with
3     override e.StackTrace = ""
```

La pile d'appels ne sera ni transmise en propriété d'exception ni affichée en cas de non rattrapage. ■

À tout moment, un code peut consulter la propriété `StackTrace` de la classe `System.Environment` pour consulter la pile d'appels qui a mené à son exécution. En cas de levée d'une exception `e` et de son rattrapage, les propriétés `StackTrace` de `e` et de `System.Environment` ne contiennent pas les mêmes piles d'appels.

```
1 let f x =
2   try ...  with
3   | e ->
4     printf "%A" e.StackTrace
5           (* appels depuis la levee d'exception *)
6     printf "%A" System.Environment.StackTrace
7           (* appels de f *)
```

`e.StackTrace` contient des appels consécutifs depuis la levée de l'exception jusqu'à son rattrapage (cette pile pouvant être masquée comme indiqué par la recommandation R-F-188) tandis que `System.Environment.StackTrace` contient la pile d'appels de la fonction contenant la construction de rattrapage.

Il est à noter qu'une fonction ayant un paramètre de type fonction permet à la fonction ainsi passée en argument d'accéder à la pile d'appels à partir du moment où elle est elle-même appelée.

```
1 module Sensible =
2   let verif k = ...
```

```
3   let appel_oui f =
4     ... ; f () ; ...
5   let appel_non f =
6     ... ; f () ; ...
7   let appel_public k f =
8     if verif k
9     then
10      appel_oui f
11    else
12      appel_non f
13 module Malveillant =
14   Sensible.appel_public un_k (fun () -> printf "%A" ↵
      System.Environment.StackTrace)
```

Dans l'exemple précédent, la pile d'appels affichée par le module `Malveillant` dépend de l'évaluation de `verif un_k` alors que ce résultat n'était pas a priori divulgué.

> *Danger 189* (D-F-189) : *Accès à l'environnement d'exécution des fonctions d'ordre supérieur.*
> L'environnement d'exécution d'une fonction d'ordre supérieur est accessible au paramètre fonctionnel au moment de son appel. ■

La seule information pertinente rendue disponible par l'environnement d'appel (`System.Environment`) est la pile d'appels (`StackTrace`). Cette information n'est pas exploitable par une fonction passée en paramètre si l'exécution de celle-ci se fait toujours avec la même pile.

> *Recommandation 190* (R-F-190) : *Éviter les fonctions d'ordre supérieur pour les modules sensibles.*

2.7.2 Bibliothèques partagées

La plateforme .NET offre un mécanisme de partage de bibliothèques appelé Global Assembly Cache de .NET (.NET GAC). Cette base de bibliothèques associe à une bibliothèque `.dll` un identificateur unique appelé *strong-name* composé du nom de bytecode, d'un numéro de version, d'un paramètre régional et d'une clé publique. La bibliothèque contient un hash de la clé publique. L'outil `gacutil` permet, entre autres actions sur la base, de vérifier que le hash de la bibliothèque correspond bien à la clé. La clé privée, correspondant à la clé publique, est à

fournir à chaque mise à jour de la bibliothèque. La paire clé publique/privée est initialement créée avec l'utilitaire **`sn`**.

Le .NET GAC offre un mécanisme de signature de bibliothèques .NET qui n'a pas été étudié dans ce document. En effet, l'analyse de la plateforme .NET sort du cadre de cette étude.

2.7.3 Boucle interactive et scripts

La **boucle interactive** **`fsharpi`** s'exécute sur la machine virtuelle .NET CLR. Chaque phrase F# entrée est lue, typée, compilée et évaluée dans le contexte formé par les phrases précédemment entrées. Le type inféré et le résultat de l'évaluation sont affichés. Ce mode d'exécution est compatible avec le mode habituel par compilation et exécution. Il est destiné à l'apprentissage et au débogue de code.

> *Intérêt 191* (I-F-191) : *Compatibilité des modes par compilation et par boucle interactive.*
> La compatibilité entre les modes d'exécution par compilation et par boucle interactive fournit une assistance au développement. ■

> *Recommandation 192* (R-F-192) : *Ne pas utiliser la boucle interactive hors de la phase de développement.*
> Il est recommandé de ne pas utiliser la boucle interactive hors de la phase de développement. ■

Certaines directives supplémentaires sont proposées pour la boucle interactive comme par exemple la directive de référencement (**`#r`**) pour référencer un nouveau paquetage ou la directive de chargement de bytecode (**`#l`**).

Les **scripts** F# (extensions de fichiers **`.fsx`** ou **`.fsscript`**) correspondent à des programme source F# pouvant contenir les directives de la boucle interactive. Ils peuvent être compilés par la compilateur **`fsharpc`**. Le comportement du programme obtenu est celui de la boucle interactive c'est à dire que les effets de bord des modules sont exécutés au lancement du programme.

2.7.4 Exécutions natives

Exécution native .NET

Il est possible de compiler statiquement du bytecode .NET CIL en code natif, compilation Ahead-Of-Time (AOT), en utilisant le compilateur Native Image

Generator (.NET NGEN) de *Microsoft* (le code natif ainsi compilé requiert la présence du Framework .NET à l'exécution) ou l'option `--aot` de Mono. Le code natif ainsi généré est spécialisé pour une architecture donnée.

Exécution native via la compatibilité avec OCaml

Il est à noter qu'un sous-ensemble du langage est compatible avec OCaml. L'option de compilation `--mlcompatibility` permet d'indiquer à F# que la syntaxe utilisée est celle de OCaml (les différence entre les syntaxes sont principalement l'indentation et quelques mots clés). Il est donc possible de compiler un même code source avec F# et OCaml. Ceci permet donc une exécution suivant un des mécanismes offerts par ce langage (voir section 1.7). Cette possibilité n'est toutefois envisageable que pour des programmes ou des bibliothèques ne dépendant pas de bibliothèques spécifiques à F# ou .NET. D'autre part, les codes compilés par les deux compilateurs ne sont pas compatibles.

Intérêt 193 (I-F-193) : *Compilation native de source F# via OCaml.*

2.8 Support de la programmation concurrente

Le langage F# et la plateforme .NET proposent des constructions et des bibliothèques dédiées aux programmations asynchrone, parallèle et concurrente par partage de mémoire et par passage de messages.

Programmation asynchrone

Le flux de contrôle (cf. section 2.1.6) prédéfini `asyn` permet d'exprimer simplement des opérations asynchrones. Une opération asynchrone est représentée par une valeur de type `Async<'T>` où le résultat de l'opération est une valeur de type `'T`. Des fonctions de la bibliothèque `Async` permettent de lancer et contrôler l'exécution d'une opération asynchrone. La fonction `Async.StartWithContinuations` permet de lancer et de gérer une opération asynchrone. Son type est le suivant :

```
1 Async.StartWithContinuations :
2   Async<'T> *
3   ('T -> unit) *
4   (exn -> unit) *
5   (System.OperationCanceledException -> unit) ->
```

```
6     unit
```

Ses paramètres sont respectivement le flux à exécuter, la fonction à appeler quand le flux se termine correctement, la fonction à appeler quand le flux se termine anormalement, la fonction à appeler si le flux est annulé). Cette fonction est une combinaison de `Async.Catch`, `Async.TryCancelled`, `Async.Start`.

Le contrôle de flux `async` fait appel à la bibliothèque `System.Threading` qui est une implémentation d'un système de *Threads* pour .NET. Le modèle asynchrone de cette bibliothèque .NET, appelé Asynchronous Programming Model (.NET APM), impose une décomposition des opérations asynchrones avec un mécanisme de fonctions de rappel (*callbacks*) pouvant s'avérer complexe et engendrer des comportements inattendus si des étapes du mécanisme sont omises. Le contrôle de flux `async` encapsule ces appels aux fonctions de la bibliothèque asynchrone .NET (via les opérateurs de flux comme `let!`).

> *Intérêt 194* (I-F-194) : *Programmation asynchrone sûre par l'encapsulation dans le contrôle de flux* `async`.

> *Recommandation 195* (R-F-195) : *Préférer l'utilisation des flux* `async` *plutôt que l'utilisation directe de la bibliothèque asynchrone .NET.*

Programmation parallèle

Le parallélisme est proposé par la plateforme .NET par la bibliothèque `System.Threading.Tasks`. Cette extension s'illustre dans les boucles parallèles `Parallel.For` et `Parallel.ForEach`. F# propose une version parallèle de sa bibliothèque de manipulation de tableaux `Array.Parallel`.

> *Intérêt 196* (I-F-196) : *Un texte source ne comportant pas de traits mutables est plus facilement parallélisable. .*
> Les traits fonctionnels et l'immutabilité rendent la décomposition de fonctions et donc leur parallélisation plus simple. ■

Partage de mémoire et structures de données concurrentes

Les communications entre *threads* concurrents peuvent s'opérer par partage de mémoire. Pour éviter le problème des *race conditions*, F# fournit une fonction de verrou `lock` pour les références.

Son utilisation requiert toutefois de prendre garde au problème d'interblocage (*deadlock*). Un ensemble de structures de données concurrentes est proposé par la bibliothèque `System.Collections.Concurrent` pour résoudre ces deux problèmes : pile concurrente `ConcurrentStack<'T>`, file d'attente concurrente `ConcurrentQueue<'T>`, collection concurrente `ConcurrentBag<'T>` et dictionnaire concurrent `ConcurrentDictionary<'T1,'T2>`..

> *Intérêt 197* (I-F-197) : *Le contrôle de la mutabilité rend la programmation concurrente plus robuste.*
> Le contrôle de la mutabilité en F# (cf. intérêt I-F-94) rend plus sûr le partage de mémoire entre threads concurrentes. ■

Partage par passage de messages

Les communications entre Threads concurrentes peuvent s'opérer par passage de messages comme proposé par la bibliothèque `FSharp.Control.Mailbox`.

2.9 Interfaçage avec d'autres langages

2.9.1 Interopérabilité avec .NET

En tant que langage de la plateforme .NET, F# offre un haut niveau de compatibilité avec les autres langages de la plateforme. Toute bibliothèque `.dll` est directement accessible en F#.

Pour faciliter cette interopérabilité, de nombreuses constructions spécifiques à .NET sont présentes dans le langage F# (valeur `null`, réflexion, etc). Ces constructions sont intégrées au système de type de F# mais en réduisent souvent les bonnes propriétés (voir les détails dans les sections précédentes relatives à F#).

> *Danger 198* (D-F-198) : *Atténuation des garanties apportées par le compilateur par l'utilisation des constructions héritées de .NET.*
> Certaines constructions héritées de .NET et permettant la compatibilité avec la plateforme ne permettent pas le même niveau de vérification que pour les constructions spécifiques à F#. Elles réduisent les garanties apportées par le compilateur F#. ■

Recommandation 199 (R-F-199) : *N'utiliser les constructions héritées de .NET qu'en cas de besoin d'interfaçage avec .NET.*

Des outils tels que **`Gendarme`** de Mono ou **`FxCop`** de *Microsoft* permettent de vérifier la bonne interopérabilité du bytecode produit. Bien que le code produit par le compilateur F# soit directement exécutable par une machine virtuelle .NET, ce type de vérification peut mettre au jour des décalages vis-à-vis des recommandations de la .NET Library Design Guidelines qui peuvent affecter la compatibilité avec d'autres langages de la plateforme comme par exemple la syntaxe des différents identificateurs.

2.10 Mécanismes et bibliothèques de sécurité

La bibliothèque standard du langage F# ne possède pas de modules dédiés à la sécurité. En revanche, F# permet d'utiliser les bibliothèques de sécurité de la plateforme .NET (voir section 2.10.1). .NET et Mono fournissent des outils de sécurité pour la .NET CLR et le bytecode .NET. Un projet de recherche étend le système de types de F# pour vérifier des propriétés de sécurité sur du code source F# (voir section 2.10.2).

2.10.1 Bibliothèques de sécurité de .NET

L'espace de nom **`System.Security`** de la plateforme .NET organise les bibliothèques de sécurité de la .NET CLR.

Ces bibliothèques proposent des mécanismes de contrôle d'accès (**`AccessControl`**, **`Permissions`**, **`Policy`**, **`Principal`**, **`RightsManagement`**), d'authentification (**`Authentication`**) et de cryptographie (**`Cryptography`**).

Code Access Security (.NET CAS) regroupe les bibliothèques permettant d'identifier un bytecode, de définir une politique d'accès, des permissions pour certaines actions et de vérifier qu'un bytecode a les permissions requises pour chaque action privilégiée. Ces permissions sont exprimées à l'aide d'attributs (cf. section 2.1.3) dans le code source F# que le compilateur transpose dans le bytecode. .NET CAS est l'équivalent du **`SecurityManager`** de Java.

2.10.2 Apport des méthodes formelles

Le projet de recherche F7 de *Microsoft Research* a développé un analyseur statique pour F#. F7 étend le système de types de F# avec des pré- et post-conditions. Le système de type produit des conditions de preuve qui peuvent être vérifiées par un système de preuve. La première application de cet outil a été faite pour la vérification de protocoles cryptographiques implémentés en F# [Bhargavan *et al.*, 2010b] [Bhargavan *et al.*, 2010a] [Bengtson *et al.*, 2008]. Un prototype est disponible en ligne sous une licence interdisant tout usage commercial (Microsoft Research License Agreement).

Le projet de recherche F*[6] [Swamy *et al.*, 2011] [Borgström *et al.*, 2011] [Swamy *et al.*, 2010] [Chen *et al.*, 2010] a pris le relais de F7. F* est un langage fonctionnel avec types dépendants et son compilateur pour la plateforme .NET. Le langage permet ainsi d'exprimer des propriétés et de les vérifier. Même si F* est basé sur F#, les textes sources des deux langages ne sont pas compatibles car leurs langages de type diffèrent. Leurs codes compilés .NET sont toutefois interopérables.

6. `http://research.microsoft.com/en-us/projects/fstar/`

Bibliographie

[Bengtson *et al.*, 2008] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D. et Maffeis, S. (2008). Refinement types for secure implementations. *In CSF*, pages 17–32. IEEE Computer Society.

[Bhargavan *et al.*, 2010a] Bhargavan, K., Fournet, C. et Gordon, A. D. (2010a). Modular verification of security protocol code by typing. *In* Hermenegildo, M. V. et Palsberg, J., éditeurs : *POPL*, pages 445–456. ACM.

[Bhargavan *et al.*, 2010b] Bhargavan, K., Fournet, C. et Guts, N. (2010b). Typechecking higher-order security libraries. *In* Ueda, K., éditeur : *APLAS*, volume 6461 de *Lecture Notes in Computer Science*, pages 47–62. Springer.

[Borgström *et al.*, 2011] Borgström, J., Chen, J. et Swamy, N. (2011). Verifying stateful programs with substructural state and hoare types. *In* Jhala, R. et Swierstra, W., éditeurs : *PLPV*, pages 15–26. ACM.

[Chen *et al.*, 2010] Chen, J., Chugh, R. et Swamy, N. (2010). Type-preserving compilation of end-to-end verification of security enforcement. *In* Zorn, B. G. et Aiken, A., éditeurs : *PLDI*, pages 412–423. ACM.

[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.

[Kennedy, 1997] Kennedy, A. (1997). Relational parametricity and units of measure. *In POPL*, pages 442–455.

[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.

[Petricek et Syme, 2010] Petricek, T. et Syme, D. (2010). Collecting hollywood's garbage : avoiding space-leaks in composite events. *In* Vitek, J. et Lea, D., éditeurs : *ISMM*, pages 53–62. ACM.

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

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

[Swamy *et al.*, 2010] SWAMY, N., CHEN, J. et CHUGH, R. (2010). Enforcing stateful authorization and information flow policies in fine. *In* GORDON, A. D., éditeur : *ESOP*, volume 6012 de *Lecture Notes in Computer Science*, pages 529–549. Springer.

[Swamy *et al.*, 2011] SWAMY, N., CHEN, J., FOURNET, C., STRUB, P.-Y., BHARAGAVAN, K. et YANG, J. (2011). Secure distributed programming with value-dependent types. *In ICFP*. ACM. To appear.

[Syme, 2006] SYME, D. (2006). Initializing mutually referential abstract objects : The value recursion challenge. *ENTCS, Proceedings of the ACM-SIGPLAN Workshop on ML (2005)*, 148(2):3–25.

[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.*, 2007a] SYME, D., GRANICZ, A. et CISTERNINO, A. (2007a). *Expert F#*. Apress.

[Syme *et al.*, 2007b] SYME, D., NEVEROV, G. et MARGETSON, J. (2007b). Extensible pattern matching via a lightweight language extension. *In* HINZE, R. et RAMSEY, N., éditeurs : *ICFP*, pages 29–40. ACM.

[Syme *et al.*, 2011] SYME, D., PETRICEK, T. et LOMOV, D. (2011). The F# asynchronous programming model. *In PADL*. To appear.

[team, 2010] TEAM, F. (2010). Draft F# component design guidelines. Microsoft Research and the Microsoft Developer Division. Disponible en ligne `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/fsharp-component-design-guidelines.pdf`.

[Thai et Lam, 2003] THAI, T. L. et LAM, H. (2003). *.Net Framework Essentials*. O'Reilly Media, third édition.

[Yu *et al.*, 2004] YU, D., KENNEDY, A. et SYME, D. (2004). Formalization of generics for the .NET Common Language Runtime. *In* JONES, N. D. et LEROY, X., éditeurs : *POPL*, pages 39–51. ACM.

Chapitre 3

Analyse du langage Scala

Le langage Scala a été créé par Martin Odersky, chercheur du Laboratoire de Méthodes de Programmation (LAMP) de l'École Polytechnique Fédérale de Lausanne (EPFL). Ses travaux sur l'étude théorique des systèmes de types pour les langages objet ont conduit à la création du compilateur Generic Java (GJ) dont la version `javac` d'Oracle actuelle est une évolution, ainsi qu'à l'ajout des **types paramétrés** (`Generics`) à Java [Bracha *et al.*, 1998]. Scala est construit pour être interopérable avec Java de manière complètement transparente, afin que toute classe Java soit directement utilisable dans un code source Scala et réciproquement.

Ce chapitre suppose une connaissance du langage Java et se positionne comme une suite de l'étude JavaSec : JavaSec permet de comprendre l'environnement d'exécution de Scala, et ce chapitre permet d'étudier en détail le langage. Il est organisé en deux grandes sections.

La première s'attache à l'étude de la sécurité de l'environnement de développement Scala : sa sémantique, le cycle de développement d'un programme dans ce langage de sa compilation à son exécution, les outils externes qui peuvent être utilisés pour améliorer la sécurité d'un programme écrit dans ce langage.

La seconde section s'intéresse aux constructions du langage qui peuvent avoir un impact sur la sécurité des programmes qui les utilisent. Cette seconde partie nécessite une connaissance a priori de Scala, en particulier de sa syntaxe, des constructions usuelles du langage et de son vocabulaire propre.

3.1 Environnement de développement

Le développement et l'exécution de programmes Scala se font dans un cadre général conçu pour offrir de nombreux éléments qui concourent à la sécurité :

- la définition du langage, fondée sur de nombreux travaux de recherche, traitée dans la section 3.1.1 ;
- la bibliothèque standard du langage, traitée dans la section 3.1.2 ;
- les modes de compilation du texte source, traités dans la section 3.1.2, et les options que le compilateur propose afin d'éviter certains types d'erreurs, traités dans la section 3.1.4 ;
- l'environnement d'exécution, et la manière dont il interagit avec le système d'exploitation, traité dans la section 3.1.5 ;
- les possibilités d'interaction avec d'autres langages, traitées dans la section 3.1.6 ;
- les outils externes qui peuvent être utilisés pour garantir ou renforcer certaines propriétés de sécurité du langage, traités dans la section 3.1.7.

L'interopérabilité avec Java repose en partie sur le fait que Scala est compilé en bytecode Java normalisé, et donc que l'environnement d'exécution de Scala est le même que l'environnement d'exécution Java. Du point de vue de la sécurité, cette interopérabilité et ce partage de l'environnement d'exécution implique que l'ensemble des fonctionnalités du langage est à étudier en tenant compte de leur traduction en bytecode Java et des caractéristiques propres de l'environnement d'exécution du bytecode Java. Cet environnement a largement été étudié dans le cadre de l'étude JavaSec [JavaSec langage, 2009, JavaSec exécution, 2009] : nous nous attacherons ici à pointer les différences et spécificités du langage et du compilateur Scala par rapport à Java, ainsi que les constructions spécifiques utilisées dans le bytecode Java produit, sans revenir sur les détails de l'étude déjà réalisée, qui pourra être consultée en complément de celle-ci.

Afin de cerner précisément les limites respectives de chaque étude, nous présentons avec la figure 3.1 l'environnement d'exécution standard de Scala et les domaines couverts d'une part par ce chapitre et d'autre part par l'étude JavaSec.

Dans la suite de ce chapitre, l'environnement d'exécution du bytecode Java issu de sources Scala est défini comme étant l'environnement standard d'exécution de bytecode Java, à savoir la machine virtuelle JVM HotSpot d'Oracle dans sa version 1.6.25. Les versions des composants propres à Scala étudiées seront celles liées à la version 2.8.1 du langage Scala.

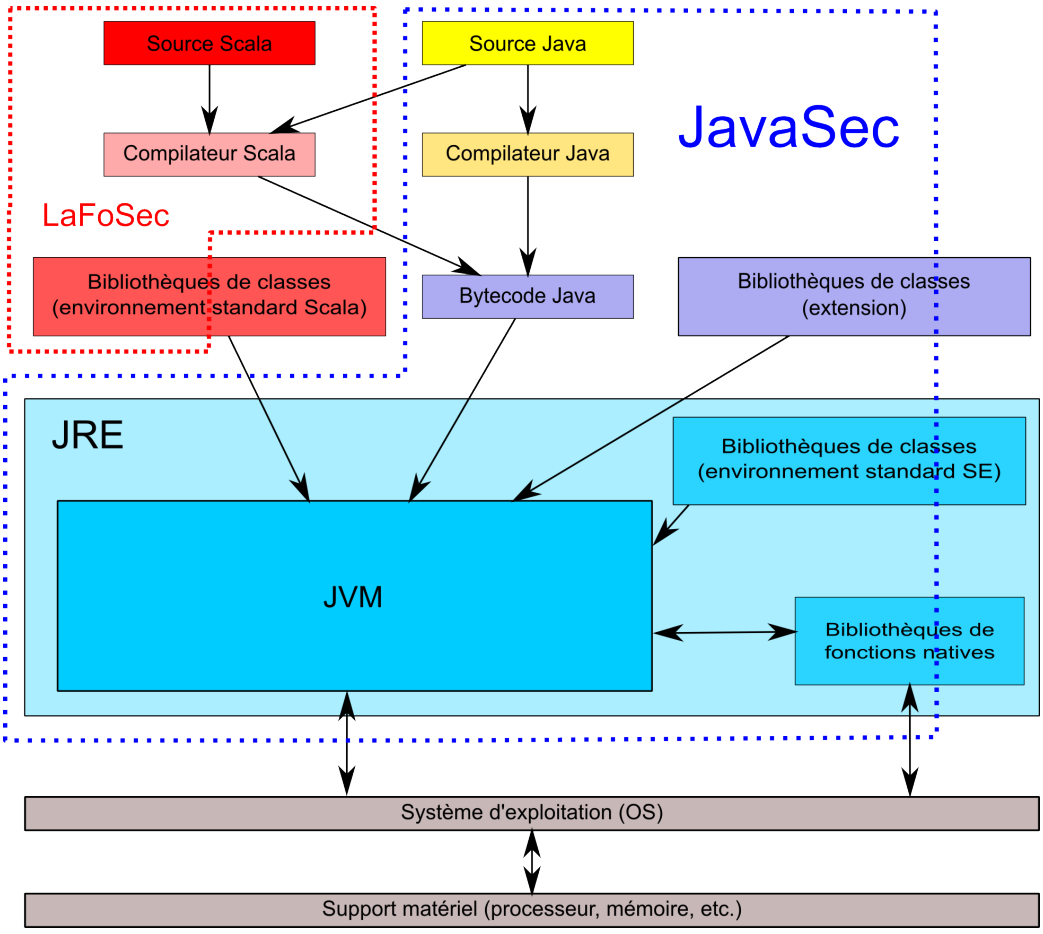

FIGURE 3.1 – Environnement d'exécution de Scala

Architecture de l'environnement Scala (exécuté sur Java Standard Edition), avec les limites relatives des études JavaSec et LaFoSec

3.1.1 Fondements et sémantique du langage

Scala est l'aboutissement d'une suite de travaux réalisée par Martin Odersky sur Java : la création du compilateur GJ dont la version `javac` d'Oracle actuelle est une évolution, et l'ajout des **types paramétrés** (`Generics`) à Java [Bracha *et al.*, 1998]. Il est inspiré d'un premier langage fonctionnel compilé en bytecode Java : langage PIZZA [Odersky *et al.*, 2000], Son typage est issu des travaux sur les langages objets du même auteur.

La principale référence sur la syntaxe et la sémantique du langage Scala est sa spécification [Odersky, 2010]. Elle est plus simple et précise que celle de Java, mais contient encore quelques imprécisions.

> *Danger 1* (D-S-1) : *Sémantique non stabilisée.*
> La sémantique de certaines fonctionnalités, comme celle du filtrage, reste imprécise et peut évoluer dans les nouvelles versions du compilateur. ■

La spécification est complétée par divers travaux de recherche portant sur la correction de propriétés du langage :

- l'unification du système de types qui englobe les traits objets et fonctionnels du langage [Odersky *et al.*, 2003, Altherr et Cremet, 2005] [Cremet *et al.*, 2006, Cremet, 2006] dans un premier temps, puis dans un second temps l'ajout des types paramétrés et la vérification de correction du nouveau système de type [Moors *et al.*, 2008, Dragos et Odersky, 2009, Moors, 2009]
- l'étude du filtrage et de sa correction sur un langage objet [Emir *et al.*, 2006, Emir *et al.*, 2007, Emir, 2007]
- l'étude des similarités entre les Type Classe et les fonctionnalités d'arguments implicites de Scala [Oliveira *et al.*, 2010, Odersky et Moors, 2009]

La programmation concurrente et parallèle au niveau du langage est le domaine étudié le plus récemment. Il est vu plus en détail dans le paragraphe 3.1.5.

3.1.2 Bibliothèque standard

Le langage Scala est associé à une bibliothèque standard qui est liée à tous les programmes Scala. Ce composant est l'équivalent de la bibliothèque standard Java pour le langage Scala : tout comme elle, son Application Programming Interface (API) est publique et documentée par l'équipe de développement de Scala pour chaque version du langage. Il n'existe qu'une implémentation de la

bibliothèque standard, qui évolue conjointement avec les différentes versions du langage. Elle est implémentée entièrement en Scala et les classes sont regroupées en packages suivant le rôle qu'elles jouent vis-à-vis du langage. Ces rôles sont eux-mêmes regroupés en deux grandes catégories : d'une part, les classes nécessaires au fonctionnement du langage, et d'autre part les classes utilitaires, les API et environnements de développement communément utilisés dans les programmes Scala.

Les classes nécessaires au fonctionnement du langage sont aussi celles qui sont utilisées dans la spécification du langage Scala [Odersky, 2010]. Elles seront vues dans la section sur les détails du langage.

Les classes utilitaires qui couvrent les besoins courants des programmes (collection d'objets, représentation XML, gestion des entrées/sorties, etc.) peuvent être considérées comme des bibliothèques externes, et ne sont pas indispensables au fonctionnement de programmes Scala même si elles sont généralement utilisées. Elles ne seront pas étudiées plus précisément dans ce document.

La bibliothèque standard Java

Du fait de l'interopérabilité avec le langage Java et de l'utilisation du même environnement d'exécution que ce dernier, les programmes Scala peuvent utiliser de manière transparente les bibliothèques Java, et en particulier la bibliothèque standard du langage. La bibliothèque standard Scala repose en partie sur la bibliothèque standard Java.

Ce composant a été vu dans l'étude JavaSec (voir [JavaSec langage, 2009, *3.2 L'environnement d'exécution standard (Java SE 6)*] et [JavaSec recommandations, 2009, *5 Recommandations relatives à la bibliothèque standard*]) et ne sera pas revu ici. Dans la suite de ce document, lorsque nous ferons référence à ce composant, nous nous limiterons à sa version Java Standard Edition (JSE).

3.1.3 Modèles d'exécution et propriétés associées

Scala propose trois modèles d'exécution, qui permettent d'exécuter du code source Scala :

- la compilation de texte source en bytecode Java par le compilateur `scalac` puis l'exécution du résultat sur une JVM ;
- l'utilisation interactive d'une boucle de haut niveau ;
- l'exécution de scripts Scala.

Du point de vue des outils et des phases impliqués, ces trois modes ne sont que marginalement différents. Les outils utilisés dans chacun des cas partagent eux-mêmes la quasi-totalité de leur code. Le paragraphe suivant présente ces points communs, les suivants s'attachent à détailler les spécificités de chacun des modes.

Principes généraux.

Dans chacun des cas, le texte source Scala est compilé en bytecode Java standard. Le modèle d'exécution de Scala est donc le même que celui de Java dès que l'on a atteint ce niveau et il est traité dans la section 3.1.5.

Compilation de texte source

Cette méthode est l'approche classique de compilation de texte source en mode non interactif : le compilateur Scala est appelé par la commande **`scalac`**. Les arguments de cette commande permettent de définir les sources à compiler, les **dépendances de compilation** (`classpath`) à utiliser, les options du compilateur, etc. Dans son principe de fonctionnement, elle est identique à la compilation de sources Java avec la commande **`javac`**.

Les détails de cette méthode sont vus dans la section 3.1.4.

Boucle interactive

Le second mode d'exécution de code source Scala est l'utilisation de la boucle interactive, ou Read-Eval-Print Loop (REPL).

Ce mode de fonctionnement est appelé par la commande *scala*. Une fois la boucle lancée, tout le langage Scala est disponible, suivant les mêmes contraintes de typage que dans un fichier source. Chaque commande donne lieu à une évaluation dans le contexte des commandes déjà évaluées au cours de la session.

Ce mode n'est en aucun cas de l'interprétation. Chaque ligne du texte source est compilée dans le contexte de l'environnement de la boucle interactive et des commandes déjà compilées. Le bytecode Java résultant de la compilation est exécuté sur la JVM utilisée par le REPL. Le résultat est affiché immédiatement.

Il existe néanmoins deux différences majeures entre ce mode et le mode de compilation de texte source standard :

Le traitement des exceptions Toutes les exceptions sont récupérées au niveau du REPL. Elles ne terminent donc pas le programme.

La gestion par ligne Un espace de noms propre est attribué à chaque ligne de la boucle interactive, contrairement à ce qui se passe pour un fichier source de plusieurs lignes, qui est associé à un seul espace de noms. Ainsi, les constructions du langage qui ont des prérequis sur le fichier contenant le code source, comme les classes annotées `sealed` (voir paragraphe 3.2.1) ou les objets compagnons (voir paragraphe 3.2.1) auront des comportements différents.

Danger 2 (D-S-2) : *Différence d'exécution entre REPL et compilation.*

Recommandation 3 (R-S-3) : *Ne pas utiliser la boucle interactive pour valider un développement.*
La boucle interactive doit être réservée à des tests rapides (validation de l'utilisation d'une méthode, d'un point de syntaxe, etc). ■

Scripts Scala

Ce dernier mode consiste à passer en argument de la commande `scala` un fichier source Scala afin d'exécuter son contenu.

Ce mode de fonctionnement est en réalité identique à l'utilisation de la boucle interactive, pour laquelle la seule *ligne* exécutée est la ligne qui contient l'ensemble du fichier source passé en argument.

3.1.4 Compilation

Cette section présente les principales spécificités du compilateur ou de l'environnement Scala qui sont importantes au regard de la sécurité des applications. Il n'existe aujourd'hui qu'une seule version du compilateur, implémentée par l'équipe de développement de Scala. Elle évolue conjointement avec les différentes versions du langage.

Le compilateur Scala est chargé de transformer le texte source Scala en bytecode Java. De ce fait, les intérêts, dangers et recommandations émis sur le bytecode Java dans le rapport JavaSec, en particulier dans [JavaSec langage, 2009, *5 Caractéristiques et propriétés du bytecode Java*], peuvent être repris.

Projet mixte Java et Scala

Il est possible de créer des projets mixtes Java et Scala. Ces projets peuvent contenir du texte source Java et Scala dont les définitions sont mutuellement dépendantes. Par exemple, une interface Java peut être étendue par une classe Scala qui est elle-même étendue par une classe Java.

Dans ce contexte, le compilateur Scala analyse les classes Java afin de construire son arbre de syntaxe abstraite, puis les sources Scala sont compilées en bytecode Java, puis le compilateur Java est utilisé pour compiler les sources Java en ajoutant le résultat de la compilation des sources Scala aux dépendances incluses à la compilation. Cette compilation mixte n'induit pas de problèmes de sécurité spécifiques et ne sera pas plus détaillée. Tous les intérêts, dangers et recommandations émis sur les sources Java dans le rapport JavaSec, en particulier [JavaSec langage, 2009, *4 Caractéristiques t propriétés du langage Java*], peuvent s'appliquer aux projets mixtes.

Production de bytecode Java

Pour chaque classe, le compilateur Scala ajoute des méta-informations, essentiellement liées aux types, dans le code Scala compilé en bytecode Java. Ces méta-informations sont ajoutées sous la forme d'une annotation Java nommée `ScalaSignature` [Dubochet, 2010] [1].

Cette annotation est ignorée par les outils d'analyse de bytecode Java standard. Les informations qu'elle contient sont utilisées par des outils comme le débogueur qui ont besoin d'accéder à l'ensemble des informations de typage, alors que celles-ci ont été effacées du bytecode Java par le mécanisme d'**effacement des types** (`type erasure`) [Langer, 2010]. Le compilateur les utilise afin de valider le typage des classes pour lesquelles le texte source est inaccessible, comme pour les classes contenues dans des Java ARchive (JAR) externes issus de Scala.

> *Intérêt 4* (I-S-4) : *Conservation des informations de type dans le bytecode Java.*
> Elles sont réutilisées par le mécanisme de compilation séparée. ■

> *Recommandation 5* (R-S-5) : *Utilisation exclusive de Scala.*
> L'utilisation exclusive de textes source Scala permet de profiter pleinement des informations de typage. ■

1. Le contenu de l'annotation est prévu pour être utilisé dans l'API d'introspection Scala une fois que celle-ci sera disponible.

Lorsque le bytecode Java ne contient pas les informations de typage (texte source issu de Java), le compilateur tente de les synthétiser. Il attribue un type existentiel (voir 3.2.2) aux éléments dont il ne peut synthétiser le type. Ainsi, le système de types de Scala reste cohérent même si le bytecode Java issu d'autres langages que Scala est utilisé.

Intérêt 6 (I-S-6) : *La cohérence du système de types est assurée.*

Plugins

Le compilateur Scala a une architecture modulaire qui accepte l'ajout de plugins. Ces plugins ont accès aux différentes phases de compilation, ainsi qu'aux différentes représentations intermédiaires des programmes. Ils sont un moyen de choix pour valider ou contraindre des règles de programmation, par exemple pour interdire d'utiliser certaines constructions du langage jugées dangereuses, ou pour imposer des standards de mise en forme du texte source.

Leur intérêt pour la sécurité est donc important : les plugins du compilateur peuvent être un moyen de mettre en oeuvre les choix de politiques de sécurité définies.

Cependant, pour l'instant, il existe peu de plugins génériques, et il n'existe pas d'index qui regroupe les plugins existants : il faudra la plupart du temps développer ses propres plugins.

Recommandation 7 (R-S-7) : *Validation des plugins utilisés.*
L'utilisation d'un plugin doit être un choix conscient et validé pour le projet. En particulier, il faut analyser le rôle du plugin et son impact sur le bytecode Java produit. ■

Option de compilation

Le compilateur Scala possède de nombreuses options de compilation, dont certaines peuvent être utilisées afin de valider des propriétés de sécurité du programme. Il existe trois niveaux d'option du compilateur :

- les options de base, données via la commande `scalac -help` ;
- les options avancées, données via la commande `scalac -X`
- les options privées à l'intention des développeurs du compilateur ou de plugins, données via la commande `scalac -Y`.

Les paragraphes suivants décrivent pour chaque niveau les options qui peuvent avoir un intérêt au regard des propriétés de sécurité.

Les options standards du compilateur

Certaines options standard du compilateur permettent d'améliorer les messages d'erreur, d'ajouter des marqueurs de débogue, ou encore de modifier le profil d'optimisation du code compilé.

Les principales options liées au débogue sont :

- `-nowarn` : cette option supprime les avertissements de compilation. Elle ne doit donc pas être utilisée, car aucun avertissement ne doit être ignoré a priori. ;
- `-uniqid` : cette option permet de générer des identifiants uniques pour l'ensemble des éléments du programme qui peuvent être utilisés pour le débogue ;
- `-explaintypes` : cette option permet de détailler les erreurs de types ;
- `-g:<g>` : cette option permet de modifier le niveau des informations de débogue générées ;
- `-print` : cette option permet d'afficher le programme, une fois toutes les différentes phases de typage et de compilation effectuées, avec l'ensemble du sucre syntaxique remplacé, l'ensemble des types et des classes générées explicités. Cette option est extrêmement utile pour voir le code source que le compilateur Scala va effectivement utiliser afin de générer le bytecode Java.

Recommandation 8 (R-S-8) : *Ne pas utiliser l'option* `-nowarn`.

D'autres options peuvent être intéressantes au regard de la sûreté :

- `-unchecked` : cette option permet d'émettre des avertissements de compilation sur les potentiels problèmes de typage dus à l'effacement de types de la JVM ;
- `-optimise` : cette option permet d'activer l'optimisation de code du compilateur. L'utilisation de cette option peut éventuellement changer le comportement de l'application dans certains cas limites. En pratique, il semble que la plupart des optimisations sont également faites dynamiquement par la JVM HotSpot. Les conclusions de l'étude JavaSec, en particulier [JavaSec exécution, 2009, *4.4 Exécution de bytecode*] sont donc valides pour ce point.

Les options avancées du compilateur

Plusieurs options avancées permettent d'avoir plus d'informations sur les actions précises du compilateur, en particulier sur les phases de compilation et l'état du code dans ces phases.

Cette information est particulièrement intéressante lorsqu'il est nécessaire de s'assurer de l'état de la représentation interne du programme à certains niveaux, en particulier dans le cadre de développement de plugins. Les options les plus directement liées à des considérations de sécurité sont décrites dans la suite de la section.

Un couple d'options permet d'avoir des messages lors de l'utilisation de constructions qui ont subi une modification de sémantique entre les versions `2.7.x` et `2.8.x` de Scala :

- `-Xmigration` émet un message sur les structures dont la sémantique a évolué ;
- `-Xwarninit` émet un message sur les codes d'initialisation de variables ou d'objets dont la sémantique a pu évoluer.

`-Xfatal-warnings` permet de transformer les messages d'avertissement en échec de compilation.

Le couple d'options suivant permet de supprimer à la compilation un certain nombre de méthodes initialement présentes dans le texte source du programme :

- `-Xdisable-assertions` supprime les méthodes `assert`, comme il est possible de le faire en Java.
- `-Xelide-below` supprime les méthodes marquées avec l'annotation `@elidable`, jusqu'au niveau indiqué en paramètre de l'annotation.

> *Danger 9* (D-S-9) : *Suppression de validation de contrainte de sécurité.*
> Ces deux méthodes permettent de supprimer des appels de code à la compilation. Il faut s'assurer que des méthodes ainsi annotées ne sont pas utilisées pour valider des contraintes de sécurité qui ne seraient donc plus exécutées. ■

Enfin, il existe deux options qui vérifient des conditions sur l'accès à des objets ou à des champs non correctement initialisés dans le cadre d'héritage multiple par trait :

- `-Xcheck-null` émet un avertissement lors de la sélection d'une référence `null` ;
- `-Xcheckinit` ajoute une vérification à l'exécution du programme sur l'ac-

cesseur du champ. Un champ non initialisé lors de l'accès lance une exception.

Danger 10 (D-S-10) : *Détection incomplète des champs nuls ou non initialisés.*
`-Xcheck-null` et `-Xchekinit` ne détectent pas tous les cas de non initialisation ou d'utilisation de `null`, mais uniquement ceux qui sont liés à la séquence d'initialisation de variable lors de l'héritage par traits. ■

Les options privées du compilateur

Ces options sont pour la plupart expérimentales et ne sont pas garanties d'exister d'une version à l'autre du compilateur.

Recommandation 11 (R-S-11) : *Ne pas utiliser les options privées du compilateur.*

3.1.5 Exécution

Une fois le code Scala compilé en bytecode Java, le bytecode Java est exécuté par une JVM, dont l'implémentation HotSpot d'Oracle est l'implémentation de référence.

Pour rappel, la JVM assure les fonctions principales suivantes :
- la vérification et l'exécution du bytecode Java sur la plate-forme native ;
- la gestion de la mémoire via le **ramasse-miettes** (garbage collector) ;
- la gestion des différents threads d'exécution ;
- le chargement, l'édition de liens et l'initialisation des classes de l'application et de la bibliothèque standard ;
- l'interfaçage avec certaines fonctionnalités offertes par le système d'exploitation ;
- la gestion des erreurs et des exceptions Java ;
- l'implémentation de certaines méthodes utilisées par la bibliothèque standard pour des raisons d'optimisation, de facilité d'implémentation ou de sûreté ;
- la vérification des profils de sécurité du `Security Manager`

Ce composant a été vu dans l'étude JavaSec [JavaSec langage, 2009] et [JavaSec exécution, 2009] et les conclusions de cette étude sont valides dans

le cadre de la plateforme Scala. Seules les différences entre Java et Scala seront abordées dans la suite de ce chapitre.

Gestion de la mémoire

Scala n'apporte pas de modification sur la gestion de la mémoire par rapport à Java, et l'ensemble des conclusions des rapports de l'étude JavaSec [JavaSec langage, 2009] et plus particulièrement [JavaSec exécution, 2009, *4.6 Gestion de la mémoire*] sont valides pour Scala.

Référence et copie d'objet

Scala n'apporte pas de modification par rapport à Java et la JVM en ce qui concerne les références d'objets et le fonctionnement de méthode `clone`. Les conclusions de l'étude JavaSec [JavaSec langage, 2009, *4.1.9 Références et copies d'objets*] restent valides, en particulier l'utilisation de l'interface `Cloneable` reste déconseillée au profit de méthodes ad hoc.

Sérialisation

Scala ne propose pas de mécanisme propre de sérialisation. Les mécanismes natifs de sérialisation de Java sont disponibles, avec les mêmes contraintes et risques que ceux abordés dans l'étude JavaSec [JavaSec langage, 2009, *4.1.11 Sérialisation*] et [JavaSec recommandations, 2009, *4.3 Sérialisation*]. En particulier, il n'est pas possible de désérialiser une fonction Scala qui aurait été sérialisée sans que la classe dont est issue la fonction n'ait été chargée dans la JVM.

En Scala, les classes sérialisables sont définies via une annotation. Cependant, cette annotation est transformée par le compilateur en l'interface Java `Serializable` habituelle.

> *Danger 12* (D-S-12) : *La sérialisation native de Scala est identique à celle de Java.*
> La sérialisation de classes Scala repose sur le mécanisme de sérialisation de bytecode Java par la JVM. Elle possède donc les même limites et dangers que celles de Java qui sont détaillées dans l'étude JavaSec [JavaSec langage, 2009, *4.1.11 Sérialisation*] et [JavaSec recommandations, 2009, *4.3 Sérialisation*]. ■

Intérêt 13 (I-S-13) : *Pas de blocs d'initialisation statique.*
Contrairement à Java, Scala ne donne pas la possibilité de définir des blocs d'initialisation statique, qui en Java sont exécutés lors du chargement d'une classe. ■

Introspection

À ce jour, Scala ne possède pas d'API d'introspection propre. Cependant, les programmes Scala ont entièrement accès aux capacités d'introspection de Java : l'ensemble des problématiques abordées dans JavaSec [JavaSec langage, 2009, *4.1.12 Programmation réflexive*] à ce sujet est donc valide pour Scala.

Scala apporte tout de même certaines spécificités dans ce domaine qui sont décrites dans les paragraphes suivants.

SecurityManager

L'introspection est utilisée par le typage structurel de Scala. Or un **gestionnaire de sécurité** (`SecurityManager`) peut interdire l'accès aux champs par introspection. Scala ne fournit pas de profil standard de `SecurityManager` et il n'existe pas de liste exhaustive de l'utilisation des types structurels au sein même de la bibliothèque standard, ce qui implique que sans analyse poussée et maintenue dans le temps de l'utilisation par la bibliothèque standard de Scala des types structurels, un `SecurityManager` devra autoriser l'introspection pour l'ensemble de la bibliothèque standard.

Danger 14 (D-S-14) : *Le* `SecurityManager` *doit autoriser l'introspection.*
L'introspection est nécessaire au bon fonctionnement du typage structurel de Scala. ■

Manifeste de classe

Scala propose une fonctionnalité qui permet d'accéder, lors de l'exécution du programme, aux informations contenues dans l'annotation `ScalaSignature` générée à la compilation : les **manifestes de classe**[2].

En Scala, un `Manifest` est une classe qui contient des méta-informations sur une classe, dont les informations sur les paramètres de type des types paramétrés

2. Attention, cette fonctionnalité n'a rien à voir avec les `manifest` Java utilisés pour la définition et la signature numérique de JAR

qui ne sont pas maintenus au niveau du bytecode Java. Ainsi, ces informations peuvent être utilisées à l'exécution du programme, ce qui permet par exemple de construire des tableaux dont le type n'est pas connu à la compilation sans erreur de typage, de construire de nouvelles instances d'un type connu uniquement à l'exécution, ou encore de valider la cohérence de types connus uniquement à l'exécution.

Programmation concurrente

Le modèle de thread de Scala est celui de la JVM, l'ensemble des points traités dans l'étude JavaSec [JavaSec langage, 2009, *6.1 Programmation concurrente*] est donc valide.

La version 2.8.1 de la distribution Scala propose le modèle de programmation par acteurs popularisé par Erlang [Haller et Odersky, 2008].

La version de la bibliothèque standard (2.9) contient une implémentation d'une mémoire logicielle transactionnelle (Software Transactionnal Memory), et des implémentations de l'API `Collection` sur lesquelles les opérations s'exécutent automatiquement en parallèle [Prokopec *et al.*, 2010].

3.1.6 Interopérabilité avec d'autres langages

Scala est directement interopérable avec Java, et indirectement avec tous les langages compilés en bytecode Java, dans les mêmes proportions et avec les mêmes contraintes que l'interopérabilité entre Java et ces langages.

3.1.7 Outils externes liés à la sécurité

A ce jour, il ne semble pas exister d'outils qui portent sur l'analyse de code source Scala d'un point de vue de la sécurité. Néanmoins, l'ensemble des outils de ce genre qui travaillent au niveau du bytecode Java fonctionnement également pour des programmes écrits en Scala.

L'analyse JavaSec [JavaSec langage, 2009], [JavaSec exécution, 2009], et [JavaSec recommandations, 2009, *4.7 6.1 Programmation concurrente*] présente certains des outils, les domaines fonctionnels suivants étant couverts :

- minimiseur de bytecode, offuscation (Proguard, etc) ;
- décompilation, qui se fera vers du code source Java ;
- outils d'analyse statique (Findbugs, etc) ;
- outils de couverture de tests unitaires (Cobertura, etc.) ;

– outils d'analyse de performance, d'utilisation mémoire (jconsole, Jprofiler, Yourkit).

3.2 Analyse du langage

Cette seconde section s'intéresse plus précisément aux constructions propres du langage qui peuvent avoir un intérêt au regard de la sécurité des programmes développés en Scala. Dans cette section, le langage est considéré comme connu.

3.2.1 Notion objet en Scala et bytecode Java

Cette première section introduit l'ensemble des éléments du langage d'un point de vue objet, et pointe en particulier les différences et analogies vis-à-vis de Java. En effet, comme Scala est compilé en bytecode Java et qui est complètement compatible avec Java, les différences entre les langages sont généralement implémentées via des contournements au niveau du bytecode Java, ce qui peut donner lieu à des cas limite au comportement inattendu, ou à des différences sémantiques entre le programme source et le programme exécuté par la JVM. Ces cas limite sont présentés dans la suite de la section.

Classes et objets

Scala est un **langage objet pur** : les programmes sont uniquement organisés autour d'objets qui interagissent les uns avec les autres afin de réaliser le traitement décrit par le programme.

Il n'existe pas de méthodes ou de champs globaux, non rattachés à un objet, comme on peut le voir en Java avec les membres `static`.

> *Intérêt 15* (I-S-15) : *Pas de champs globaux au programme.*
> Les risques et recommandations à propos des champs globaux de Java sont détaillés dans l'étude JavaSec [JavaSec langage, 2009, *4.1.3 Membres statiques*] et [JavaSec recommandations, 2009, *Identifiant 6 : Utilisation des champs static*]. ■

Les classes de Scala sont introduites à partir de quatre mots-clés :

– `class` et `abstract class` introduisent des **classes** qui autorisent un héritage simple et possèdent un constructeur principal dont les arguments paramètrent la classe (voir paragraphe sur les constructeurs). Une classe

(`class`) peut créer des objets via le mot-clé `new`. Les **classes abstraites** (`abstract class`) peuvent laisser tout ou partie de leurs membres abstraits (i.e sans implémentation concrète), l'implémentation étant alors faite par une classe fille.
- `trait` (ou `abstract trait`, le `abstract` étant implicite) introduit un **trait** : c'est une classe abstraite qui autorise une forme simplifiée d'héritage multiple par linéarisation de la hiérarchie. Contrairement au cas précédent, les `traits` n'ont pas de constructeur principal.
- `object` introduit une **classe monoinstanciée** qui est à la fois un type et une valeur du programme : l'environnement d'exécution prend en charge la création paresseuse de l'unique objet issu d'une telle classe. Dans le cas où ce mot-clé est utilisé dans le même fichier qu'une classe du même nom, la classe monoinstanciée est dite **compagnon** de la classe, ce qui implique une relation particulière avec les objets issus de la classe.

Interopérabilité entre les langages Scala et Java. Le langage Scala est pensé de façon à être nativement intéropérable avec le langage Java. Ainsi, toutes les constructions du langage Java sont directement utilisables en Scala : une classe Scala peut étendre une classe ou une interface Java, les membres statiques de classes Java ou les valeurs d'énumération sont accessibles dans le code Scala, etc.

> *Danger 16* (D-S-16) : *Annotation Java paramétrée par une classe.* Il existe une limitation lorsque Scala étend du code Java : il n'est pas possible de passer un objet Scala en argument d'une annotation paramétrée par une classe Java. Il est donc impossible d'utiliser du code Java qui implique une annotation qui attend comme paramètre une classe dont certaines méthodes sont `static`. ■

L'utilisation de classes Scala en Java comporte quelques limites :
- un trait Scala ne peut être étendu simplement par une classe Java que s'il est totalement abstrait. Il est alors vu comme une interface Java ;
- pour chaque objet, le compilateur Scala génère deux classes bytecode Java : pour un object `MonObject`, les classes `MonObjet` et `MonObjet$` sont produite en bytecode Java. Or, la classe qui contient les méthodes statiques est `MonObjet$`, et c'est ce nom qui doit être utilisé dans le code source Java ;

D'autres limites sont détaillées dans [Spiewak, 2009].

> *Danger 17* (D-S-17) : *Utilisation de code Scala depuis Java.*

Toutes les constructions Scala ne peuvent pas facilement être utilisées dans du code source Java. ■

Recommandation 18 (R-S-18) : *Construire des API d'intéropérabilité Java.*
Dans le cas où des bibliothèques Scala doivent être utilisées dans du code source Java, il faut créer des API spécifiques qui se limitent aux constructions simplement utilisables depuis Java. ■

Identificateurs du langage Scala. Scala est très permissif sur les identificateurs autorisés pour les éléments du langage (noms de classe, de méthode, de champ) qui peuvent être entre autres des symboles d'opérateurs.

Danger 19 (D-S-19) : *L'utilisation de symboles UTF-8 dans les noms de méthode peut rendre la lecture du code source incompréhensible.*
Cette très grande liberté peut mener à des abus avec l'utilisation d'identificateurs qui comporteraient des caractères UTF-8 peu discernables les uns des autres, ou la surcharge d'opérateurs usuels avec une sémantique non appropriée. ■

Membres de classes

En Scala, les membres d'une classe sont :
- des méthodes (`def`) ;
- des champs mutables (`var`) ou non mutables (`val`) ;
- des définitions d'autres classes, objets ou traits. Ce point sera vu dans la section 3.2.5 ;
- des types. Ce point sera vu dans la section 3.2.2.

Accès uniforme. La traduction des `val` ou `var` en bytecode Java est faite par la création d'un champ privé et `final` du même nom que le champ défini dans le code source ainsi que de méthodes **accesseurs**, en lecture uniquement pour `val`, et lecture et écriture pour `var`. Dans le cas de champs abstraits, seuls les accesseurs sont générés.

Cette traduction à l'aide de champs et d'accesseurs a des conséquences importantes sur l'héritage et la redéfinition des champs, ainsi que leur initialisation

dans ce contexte, comme il sera vu dans les sections 3.2.1 "Héritage" et 3.2.1 "Initialisation de valeurs dans les traits".

Méthodes

Les méthodes sont introduites par le mot-clé `def`. Le corps d'une méthode est évalué à chacun de ses appels. La valeur de retour de la méthode est la valeur du dernier bloc d'instructions de la méthode. Le type de retour peut être inféré par le compilateur à partir de cette information.

```
1 def echo(s:String) : Unit = {
2   println("echo: " + s)
3 }
4 echo: (s: String)Unit
5
6 echo("1")
7 echo: 1
```

Une méthode est abstraite si elle est déclarée dans un trait ou une classe abstraite sans que son corps soit défini.

> *Intérêt 20* (I-S-20) : *Le compilateur interdit l'instanciation de classes comportant une méthode abstraite.*

En Scala, tous les opérateurs sont des méthodes comme les autres définies sur les classes. Ils peuvent donc être surchargés et redéfinis suivant les règles standard qui s'appliquent aux méthodes.

Les méthodes de type `Unit`, appelées `procedure`, peuvent être définies avec une syntaxe particulière qui omet le type de retour et le `=`. Ainsi, la méthode `echo` de l'exemple précédent peut être réécrite :

```
1 def echo2(s:String) { println("echo: " + s) }
2 echo2: (s: String)Unit
```

> *Danger 21* (D-S-21) : *Difficulté de maintenance des procédures.*
> Si le code de la procédure est changé, et que son type de retour n'est plus `Unit`, l'utilisation de la syntaxe `procedure` peu marquée visuellement induira l'ajout d'une instruction vide par le compilateur pour conserver le type `Unit`. Cela empêchera donc la vérification du typage des appels à cette méthodes. ■

Recommandation 22 (R-S-22) : *Ne pas utiliser la syntaxe spécifique pour les procédures.*

Surcharge, paramètres nommés et valeur par défaut. La surcharge de méthodes est disponible en Scala pour des raisons de compatibilité avec Java.

Cependant, Scala propose une alternative à la surcharge lorsqu'elle est utilisée pour définir des valeurs par défaut pour certains paramètres d'une méthode. Cette alternative repose sur les paramètres de fonctions nommés et la possibilité de donner une valeur par défaut à ces paramètres directement dans le type fonctionnel de la méthode.

Considérons l'exemple suivant qui définit une méthode affichant un cercle en donnant son origine et son rayon en Java :

```
1  /* code Java */
2  public void printCircle(
3      int xOrig, int yOrig, int radius
4  ) {
5    System.out.println(String.format(
6      "Cercle en \%s:\%s de rayon\%s",xOrig,yOrig,radius)
7    );
8  }
9
10 //a default circle on 0/0 with radius 1
11 public void printCircle() { printCircle(0,0,1); }
12
13 //a default circle on 0/0 with radius 1
14 public void printCircle(int radius) {
15   printCircle(0,0,radius);
16 }
17
18 //a default circle with radius 1
19 public void printCircle(int xOrig, int yOrig) {
20   printCircle(xOrig,yOrig);
21 }
22
23 // on ne peut pas definir une methode avec uniquement ↵
     xOrig, car cette
24 // cette methode serait en conflit avec la seconde.
25 //utilisation:
```

```
26
27 //en donnant l'origine
28 printCircle(1,1)
29
30 //en donnant tous les parametres
31 printCircle(1,1,1)
32
33 //en ne donnant que le rayon
34 printCircle(1)
```

Et une version équivalente Scala qui permet automatiquement de choisir quel seront les paramètres spécifiés :

```
1  def printCircle(
2      xOrig : Int = 0
3    , yOrig : Int = 0
4    , radius: Int = 1
5  ): Unit = {
6    println("Cercle en \%s:\%s de rayon \%s".format(
7      xOrig,yOrig,radius
8    ))
9  }
10
11 //en donnant l'origine
12 printCircle(xOrig=1 , yOrig=1)
13 //ou
14 printCircle(1, 1)
15
16 //en donnant tous les parametres
17 printCircle(xOrig=1 , yOrig=1, radius=1)
18 //ou
19 printCircle(1,1,1)
20
21 //en ne donnant que le rayon
22 printCircle(radius=1)
23
24 //en donnant l'abscisse et le rayon
25 printCircle(xOrig=1, radius=1)
```

Intérêt 23 (I-S-23) : *Simplifier et améliorer la robustesse des API avec les valeurs par défaut.*

Les valeurs par défaut permettent de définir des API simples et robustes : les valeurs par défaut sont clairement spécifiées dans la déclaration de type, et le nom des paramètres peut être explicité lors de l'utilisation de la méthode. ■

Recommandation 24 (R-S-24) : *Éviter d'utiliser la surcharge.*
Ne pas utiliser la surcharge pour les cas non couvert par les valeurs par défaut des paramètres mais donner des noms de méthodes différents, qui explicitent le but chacune. ■

Paramètres passés par nom. Scala autorise le passage de paramètre par nom avec la syntaxe `=> parameterName : TYPE`.

Intérêt 25 (I-S-25) : *Le développeur peut utiliser l'évaluation par nom.*
Le passage par nom diffère l'évaluation du paramètre à son utilisation effective. Cette propriété peut entraîner un gain de performance important. ■

Cette fonctionnalité est implémentée en Scala via la génération d'une classe supplémentaire en bytecode Java et par la création d'une instance de cette classe par appel à la méthode.

Il faut donc être conscient qu'un grand nombre de classes peut être généré lors de l'utilisation de méthodes dont les arguments sont passés par nom.

Paramètres répétés. Tout comme en Java, un paramètre répété est un paramètre à arité variable. Le paramètre répété est défini en annotant son type avec le symbole `*` et est obligatoirement le dernier paramètre de la méthode.

Contrairement à Java, le passage d'une collection comme argument du paramètre répété n'est validé par le compilateur que si elle est explicitement annotée avec le type spécial `:_*`.

Intérêt 26 (I-S-26) : *Non ambiguïté de la sémantique d'appel des méthodes avec paramètres répétés en Scala.*
Cette annotation explicite élimine les ambiguïtés qui pourraient naître dans le cas de méthodes surchargées. ■

Le code suivant Java ne compile pas car la méthode "doit" est considérée comme dupliquée :

```java
/* code Java */
public class Test {
  public static void main(String[] args) throws ↵
   Throwable {
    String[] a = new String[]{"a", "b"};
    doit(a);
    //impossible d'appeler la methode avec varags
  }

  //Duplicate method doit(String...) in type Test
  public static void doit(String... x) {
    System.out.println("strings");
  }
  //Duplicate method doit(String...) in type Test
  public static void doit(String[] x) {
    System.out.println("array");
  }
}
```

Alors que le code suivant Scala est licite, et on sait clairement quelle méthode sera appelée dans quel cas :

```scala
object OverloadingVarargs {

  def doit(x:String*) = println("varargs")

  def doit(x:Array[String]) =
    System.out.println("array")

  def main(args: Array[String]): Unit = {
    val a = Array("a", "b")
    doit(a:_*) // "varargs"
    doit(a)    // "array"
  }

}
```

Champ val

Les champs introduits par le mot-clé `val` sont des champs non mutables. Ils peuvent être seulement déclarés dans une classe abstraite ou un trait. S'ils sont définis, ils sont évalués à la création des objets. Les `val` peuvent être de type quelconque.

> *Intérêt 27* (I-S-27) : *Une tentative de modification d'un champ* `val` *engendre une erreur de compilation.*

> *Danger 28* (D-S-28) : *Les champs* `val` *n'existent pas au niveau du bytecode Java.*
> Le concept de valeur non mutable n'est pas connu au niveau du bytecode Java. Une `val` peut donc voir sa valeur changer par manipulation du bytecode Java, réflexion, ou tout autre moyen qui modifie lors de l'exécution la valeur du champ et sur lequel le compilateur ne peut pas intervenir. ■

Modificateur lazy Un champ `val` peut être annoté par le modificateur `lazy`. Dans ce cas, son initialisation est paresseuse : le bloc d'initialisation n'est évalué que lors du premier accès à la valeur, et seulement si un accès est fait au cours du programme.

```
1 lazy val v : Int = { println("Hello") ; 42 }
2 v: Int = <lazy>
3
4 v
5 Hello
6 res0: Int = 42
7
8 v
9 res1: Int = 42
```

Champ var

Le mot-clé `var` introduit un champ mutable, c'est-à-dire un champ dont la valeur peut être modifiée après l'initialisation.

S'il n'est pas abstrait, un champ `var` doit être initialisé. Il peut être initialisé avec la valeur par défaut de son type (`0` pour les numériques, `false` pour les

booléens, `null` pour les types descendants de `AnyRef` et `()` pour le type `Unit`) en utilisant la syntaxe joker `var identificateur : TYPE = _`.

> *Danger 29* (D-S-29) : *L'initialisation des champs* `var` *descendant de* `AnyRef` *est faite par défaut à la valeur* `null`.
> L'initialisation par défaut pour un objet est faite à la valeur `null`, ce qui peut entraîner des levées d'exception à l'exécution. ■

> *Recommandation 30* (R-S-30) : *Toujours effectuer une initialisation explicite des champs* `var`.
> Il est déconseillé d'utiliser l'initialisation par défaut. Il est conseillé de toujours initialiser explicitement un champ `var`. ■

Modificateurs de membres

Tout membre de classe peut recevoir des modificateurs qui sont des mots-clés du langage. Ils modifient la sémantique de l'élément original.

En Scala, les modificateurs applicables aux `def`, `val`, `var` sont :

- `final` : ce modificateur a la même sémantique que son équivalent Java (voir ci-après).
- `implicit` : ce modificateur est abordé dans la section 3.2.2.
- `private` et `protected` : ces modificateurs sont abordés dans la section 3.2.1 "Modificateurs de visibilité".
- `override` : voir ci-dessous.

La bibliothèque standard Scala définit également des annotations de membres qui sont les équivalents de mots-clés et modificateurs Java correspondants :

- transcient
- volatile
- strictfp
- throws

Les détails de la sémantique de ces mots-clés ainsi que les implications en terme de sécurité peuvent être consultés dans l'étude JavaSec [JavaSec langage, 2009].

Override. Toute redéfinition d'un membre concret doit être précédée du mot-clé `override`. Ce mot-clé est facultatif pour les membres abstraits.

> *Recommandation 31* (R-S-31) : *Toujours utiliser le mot-clé override.*

Il est conseillé de toujours utiliser le mot-clé override, même dans le cas de redéfinition de membres abstraits. Dans le cas d'une redéfinition souhaitée, l'utilisation du mot-clé override permet de faire vérifier au compilateur qu'un membre avec un type fonctionnel adéquat existe bien dans une classe parente. ■

Intérêt 32 (I-S-32) : *Invalidation d'une redéfinition non souhaitée par l'utilisation d'override.*
Dans le cas d'une redéfinition non voulue, l'absence du mot-clé override entraîne un message d'erreur du compilateur. ■

Modificateurs de visibilité

Scala possède uniquement deux modificateurs de visibilité de champs : `private` et `protected`. Lorsqu'aucun modificateur n'est précisé, le champ est public.

Comme en Java, `private` limite la visibilité aux objets issus d'une même classe, et `protected` aux objets et sous-classes issues de la classe.

Les deux modificateurs peuvent être paramétrés par un chemin pour restreindre leur visibilité aux membres de ce chemin au plus. Par exemple, dans le code :

```
1 class C {
2   private[this] val v = "secret"
3 }
```

La valeur `v` ne sera accessible qu'à la seule instance qui la possède, et non pas comme en Java, à l'ensemble des instances de cette classe. De même, dans le code :

```
1 package api.public.implementation
2
3 protected[api.public] class DefaultImpl { }
```

La classe DefaultImpl sera visible et pourra être étendue par tous les membres de api.public.

Attention, Scala est compilé en bytecode Java qui n'a pas connaissance de ces modificateurs de visibilité. En fait, seuls deux modificateurs sont conservés au niveau du bytecode Java : `private` pour les items annotés avec le modificateur `private` ou `private[chemin]` ; `public` pour *tous* les autres modificateurs.

Danger 33 (D-S-33) : *Les visibilités de Scala ne sont pas forcément conservées en bytecode Java.*
Du code à visibilité restreinte dans Scala peut être vu comme `public` depuis un autre langage tel que Java. ■

Recommandation 34 (R-S-34) : *Limiter les visibilités à public ou private.*

- ne pas utiliser le modificateur `protected` ;
- toujours garder à l'esprit que le bytecode Java ne contiendra que deux modificateurs : public et private ;
- en ce qui concerne la définition d'API dans le cadre de projets polyglottes, il est conseillé de n'exposer aux autres langages que des API Scala à base de traits totalement abstraits spécifiquement définies dans ce but. ■

Modificateur de classes

La définition d'une classe peut être accompagnée de modificateurs. Ce sont des mots-clés qui modifient la sémantique de la classe ou des instances qui en sont issues, et qui peuvent avoir un impact au regard de la sécurité.

Pour les classes, les modificateurs applicables sont :

- `private` et `protected` : aussi appelés **modificateurs de visibilité**, ils sont traités en détail dans la section 3.2.1 "Modificateurs de visibilité".
- `final` : ce modificateur a exactement la même sémantique que son homologue Java : il permet de clore la hiérarchie au niveau de la classe qui l'utilise. Une classe abstraite ne peut pas recevoir ce modificateur. L'étude JavaSec [JavaSec langage, 2009, *4.1.4 Les champs final*] présente les propriétés de ce modificateur et son intérêt au regard de la sécurité.
- `sealed` : ce modificateur permet de restreindre les possibilités d'héritage de l'élément, et est détaillé dans le paragraphe suivant.
- `case` : ce modificateur est traité dans la section 3.2.3

Scala définit également une liste d'annotations dans la bibliothèque standard qui sont applicables à des classes. Ces annotations sont l'équivalent de modificateurs ou d'interfaces spécifiques du langage Java :

- `@clonable` : correspond à l'interface `clonable` de Java ;
- `@native` : correspond au modificateur `native` de Java ;

– `@serializable` et `@SerialVersionUID` : correspondent à l'interface `serializable` de Java ;

Ils ont exactement la même sémantique et les mêmes contraintes de sécurité que leur équivalent Java et sont traduits par le compilateur Scala, au niveau du bytecode Java, par l'extension des interfaces Java correspondantes. Leur étude a été faite en détail dans JavaSec [JavaSec langage, 2009].

Modificateur Sealed. Scala introduit le modificateur `sealed` (scellé) qui limite les possibilités d'héritage d'une classe : une classe qui porte ce modificateur est close, mais ouverte localement pour les classes qui sont définies dans le même fichier source qu'elle-même. Cependant, les sous-classes d'une classe scellée peuvent être étendues sans limitations.

> *Intérêt 35* (I-S-35) : *Contrôle des capacités d'héritage d'une classe.*
> Ce modificateur permet de définir une arborescence scellée, qui ne sera pas extensible par les utilisateurs de l'arborescence. Cet idiome est particulièrement intéressant pour définir un ensemble de cas fini, dont toutes les extensions doivent être contrôlées, comme dans le cas du filtrage (voir section 3.2.3) ■

> *Recommandation 36* (R-S-36) : *Utiliser sealed et final pour clore un arbre de classes.*
> Utiliser `sealed` sur une classe parente et `final` sur ses enfants pour empêcher toute extension de la hiérarchie de classes ainsi formée. ■

> *Danger 37* (D-S-37) : *Contrôle effectué uniquement sur le texte source, par le compilateur Scala.*
> Ce modificateur n'est reconnu que par le compilateur Scala. Une classe Java peut étendre une classe Scala portant le modificateur `sealed`, contredisant ainsi la sémantique souhaitée pour la classe. ■

Héritage

En ce qui concerne l'héritage, Scala reprend les grandes caractéristiques définies dans Java. Ceci est nécessaire pour que l'interopérabilité entre les deux langages soit optimale. Les principales caractéristiques partagées entre les deux langages sont :

- partage des membres qui ont une visibilité autre que privée ;
- redéfinition des méthodes de même type fonctionnel ;
- résolution dynamique de la méthode à appeler ;

Le masquage des champs se comporte différemment du fait de l'implémentation de ceux-ci par des accesseurs et des champs privés.

Les différents dangers et recommandations de l'étude JavaSec [JavaSec langage, 2009, *4.1.7 Héritage*] s'appliquent donc à l'héritage en Scala.

Linéarisation des traits. La principale différence entre Scala et Java est l'existence de `traits`, qui autorisent une forme simplifiée d'héritage multiple. L'héritage multiple est introduit par le mot-clé `with`. L'héritage peut être déclaré lors de la définition de la classe, ou lors de son instanciation. L'exemple suivant montre différents cas.

```
1  class A { val a = 1 }
2  defined class A
3
4  trait T1 { val t1 = 1 }
5  defined trait T1
6
7  trait T2 { val t2 = 1 }
8  defined trait T2
9
10 class B extends A with T1 with T2
11 defined class B
12
13 val b = new B
14 b: B = B@1d626a4
15
16 println(b.a + " " + b.t1 + " " + b.t2)
17 1 1 1
18
19 val att = new A with T1 with T2
20 att: A with T1 with T2 = $anon$1@68097d */
21
22 println(att.a + " " + att.t1 + " " + att.t2)
23 1 1 1
```

Pour `att` comme pour `b`, les champs `a, t1, t2` sont accessibles.

Dans cet héritage multiple, la hiérarchie est linéarisée afin d'éviter les problèmes d'ambiguïté des redéfinitions de membres. Le processus de linéarisation est défini formellement dans le chapitre 5.1.2 des spécifications du langage Scala.

Une redéfinition de méthode est considérée suivant cette forme linéarisée, suivant les conditions de correspondance de type fonctionnel, que l'on retrouve en Java.

> *Danger 38* (D-S-38) : *L'héritage multiple complexifie le compréhension du code.*
> L'héritage multiple et la linéarisation rendent encore plus difficile que dans le cadre de l'héritage simple la compréhension de la version de la méthode qui sera effectivement appelée. ■

Ce problème peut être particulièrement subtil à repérer, comme ce fut le cas pour le bogue `https://lampsvn.epfl.ch/trac/scala/ticket/4279` de la bibliothèque standard de Scala. Ce bogue entraînait des performances moins bonnes qu'escomptées dans le cadre de l'utilisation d'une fonctionnalité et aurait pu créer des problèmes de disponibilité pour les applications qui l'utilisaient.

> *Recommandation 39* (R-S-39) : *Eviter la redéfinition de champs.*
> La redéfinition de champs concrets est à proscrire, sauf si la spécification l'exige. Une analyse de code source sur l'utilisation du mot-clé `override` peut être utilisée. ■

> *Recommandation 40* (R-S-40) : *Limiter l'utilisation de l'héritage multiple.*
> Utiliser si possible l'héritage multiple uniquement pour composer des traits dont les membres sont disjoints. ■

Redéfinition

Redéfinition de méthode En Scala, la redéfinition de méthode fonctionne suivant les mêmes principes qu'en Java : une méthode d'un type fonctionnel donné peut être redéfinie dans une classe fille. La méthode qui est effectivement appelée lors de l'exécution est choisie suivant les règles de résolution d'appels virtuels habituelles ([JavaSec langage, 2009, *4.1.7 Héritage*]).

Redéfinition de champs La notion de masquage de champs n'existe pas dans le cadre de l'héritage, ce qui est normal puisqu'en Scala les champs sont en

réalité composés d'un champ privé à l'objet et de méthodes accesseurs en lecture et écriture (voir 3.2.1, "Accès uniforme").

Par contre, les méthodes accesseurs en lecture et écriture peuvent être redéfinies, directement (et l'on se retrouve dans le cadre du paragraphe précédent) ou par redéfinition du champs. Les règles exactes de redéfinition sont complexes et varient suivant le type de champs (variable ou valeur), s'il est abstrait ou concret, et s'il est à initialisation paresseuse (`lazy`) ou non. Elles sont exposées dans la spécification du langage [Odersky, 2010], "5.1.3 Class Members" et "5.1.4 Overriding".

> *Danger 41* (D-S-41) : *Les règles d'initialisation des champs dans les classes composées de traits peuvent entraîner des NullPointerException.*
> On peut donc avoir :
> – une classe parente qui définit une valeur, et qui a donc un accesseur et un champ privé.
> – une classe fille qui redéfinit cette valeur de la classe parente, ce qui revient à avoir dans la classe fille un nouveau champ privé, et un accesseur pour ce champ.
>
> Dans certains cas, un des champs privés ne sera pas encore initialisé lorsqu'il sera lu, ce qui peut mener à des déréférencements de pointeur `null`. Un exemple est présenté dans la section 3.2.1 "Initialisation de valeurs dans les traits". ■

> *Recommandation 42* (R-S-42) : *Ne pas redéfinir les valeurs et variables.*
> Une valeur ou une variable ne doit pas être redéfinie, et devrait en générale être marquée `final`. Si un membre doit être redéfini, il faut utiliser une méthode dont la valeur de retour est constante.■

Constructeurs

En Scala, le constructeur initialise les `var` et les `val` et exécute le code inséré dans la classe. Ses paramètres sont les paramètres de la classe. Ils définissent des champs de l'objet. Leur visibilité est définie dans le constructeur, tout comme leur mutabilité.

Dans le code suivant, le dernier `println` fait partie du constructeur :

```
1 class ConstructorEx(pName:String) {
```

```
2   //pName est un parametre du constructeur.
3
4   //valeur accessible en lecture seule
5   //la methode et l'affectation sont faites
6   //lors de l'instanciation de la classe
7   val name = normalisation(pName)
8
9   //methode privee de la classe
10  private def normalisation(s:String) : String = {
11    s match {
12      case null | "" => "defaultName"
13      case n => n.capitalize
14    }
15  }
16
17  //ceci fait aussi partie du constructeur et sera ecrit
18  //a chaque instanciation de cette classe.
19  println(name)
20 }
```

Danger 43 (D-S-43) : *Risque de dispersion du constructeur dans l'ensemble de la classe.*

Recommandation 44 (R-S-44) : *Imposer des conventions relatives à l'écriture des constructeurs.*
Définir des règles de développement strictes sur la zone dédiée au constructeur dans la classe. ■

Modificateur de visibilité des constructeurs. Dans le cas d'un constructeur, les modificateurs de visibilité sont positionnés entre le nom de la classe et le type fonctionnel du constructeur :

```
1 class C private (name:String) { }
2 defined class C
```

Appel des constructeurs parents. Comme en Java, l'ensemble des constructeurs des classes parentes est appelé lors de la création d'un objet. Dans le cas

de l'héritage multiple des `traits`, les constructeurs sont appelés suivant l'ordre qui résulte de la linéarisation de la hiérarchie définie dans le paragraphe 3.2.1.

```
1  trait T1 {  println("Constructeur de T1") }
2  defined trait T1
3
4  trait T2 {  println("Constructeur de T2") }
5  defined trait T2
6
7  class C extends T1 with T2 {
8    println("Constructeur de C")
9  }
10 defined class C
11
12 new C
13 Constructeur de T1
14 Constructeur de T2
15 Constructeur de C
16 res0: C = C@1acecf3
```

La section 3.2.1 "Initialisation de valeurs dans les traits" traite des problématiques d'initialisation de champs engendrées par cette propriété.

Initialisation de valeurs dans les traits

Les traits peuvent contenir des membres concrets, qui comme pour une classe, sont initialisés par le constructeur du trait. L'ensemble du corps du trait est considéré comme définissant le corps du constructeur (voir section 3.2.1) "Constructeurs". Cependant, contrairement à une classe, un trait ne peut avoir que le constructeur par défaut sans paramètre. Cette limitation se contourne facilement en utilisant des membres abstraits à la place de paramètre. Ces membres seront concrétisés par la classe concrète qui étendra le trait, et ainsi le trait obtiendra les valeurs des paramètres voulus.

L'exemple suivant utilise cette technique pour les membres `v` et `d` :

```
1 trait T {
2   val v : String
3   def d : String
4   println("Constructeur de T; v: " + v + "; d: " + d)
5 }
```

Cependant, l'ordre d'appel des constructeurs vu précédemment implique qu'une valeur définie avec le mot clé `val` et redéfinie avec ce même mot-clé ne sera pas initialisée dans le constructeur du trait.

```
1  class C1 extends T {
2    override val v =  "is set"
3    override val d =  "is set"
4    println("Constructeur de C1; v: " + v + "; d: " + d)
5  }
6
7  new C1
8  Constructeur de T; v: null; d: null
9  Constructeur de C1; v: is set; d: is set
```

Danger 45 (D-S-45) : *Les champs redéfinis dans une sous-classe ne sont pas initialisés lors de l'exécution du constructeur de la classe parente.*
Un champ non mutable `val` ou un accesseur en lecture défini dans un trait, puis redéfini dans une sous-classe aura la valeur `null` lors de l'exécution du constructeur du trait, ce qui peut engendrer des `NullPointerException`. ■

Pour que l'initialisation ait lieu au bon moment, il faut redéfinir la valeur en ajoutant le modificateur `lazy`, qui a pour but d'initialiser la valeur à son premier accès, soit dans notre cas *avant* son initialisation normale.

Ce comportement est mis en évidence dans l'exemple suivant :

```
1  class C2 extends T {
2    override lazy val v =  "is set"
3    override def d =  "is set"
4    //on aurait aussi pu utiliser avec le meme resultat:
5    //override lazy val d =  "is set"
6    println("Constructeur de C2; v: " + v + "; d: " + d)
7  }
8
9  new C2
10 Constructeur de T; v: is set; d: is set
11 Constructeur de C2; v: is set; d: is set
```

Comme il a été vu dans la section 3.2.1 "Redéfinition de champs", un champ immutable concret qui n'est pas marqué avec le modificateur `lazy` ne peut pas

être redéfini par un champ immutable portant ce modificateur. Dans le cas d'une valeur concrète, il faut donc utiliser `lazy val` aussi dans le trait.

```
1  trait T {
2    val v  = "dans T"
3    def d  = "dans T"
4    lazy val l = "dans T"
5    println("Const. de T; v: "+v+"; d: "+d+"; l: "+l)
6  }
7
8  class C1 extends T {
9    override val v = "dans C1"
10   override def d = "dans C1"
11   //pour d, on aurait aussi pu utiliser
12   //avec le meme resultat:
13   //override lazy val d =  "dans C1"
14   override lazy val l = "dans C1"
15   println("Const. de C1; v: "+v+"; d: "+d+"; l: "+l)
16 }
17
18
19 new C1
20 Const. de T; v: null; d: dans C1; l: dans C1
21 Const. de C1; v: dans C1; d: dans C1; l: dans C1
```

Cet exemple met en oeuvre plusieurs effets qui se combinent pour arriver au résultat surprenant ligne 20 (valeurs null et "dans C1"). Voici une explication ligne par ligne des définitions des classes `T` et `C1` :

1. ligne 1 : définition du trait parent `T`. Les lignes 2 à 5 représentent son constructeur :
   - ligne 2 : définition d'une valeur concrète, et donc d'un champ privé à l'objet `v` et d'une méthode accesseur, `v()`. L'initialisation du champ se fait dans le constructeur de T.
   - ligne 3 : définition d'une méthode dont la valeur de retour est constante ;
   - définition d'une valeur à initialisation paresseuse (donc lors du premier accès en lecture) et comme précédemment d'un champ privé à l'objet : `l` et d'une méthode accesseur `l()` ;
2. ligne 8 : définition d'une classe `C1` fille de `T`. Les lignes 8 à 15 représentent son constructeur :

- ligne 9 : redéfinition de la valeur `v` et donc création d'un champ privé à la classe C1, `v`, différent de celui présent pour le parent et redéfinition de la méthode accesseur, `v()` ;
- ligne 10 : redéfinition de la méthode `d` ;
- ligne 11 : redéfinition de la valeur `l` et donc création d'un champ privé à l'objet C1, `l`, différent de celui présent dans le parent, et redéfinition de la méthode accesseur, `l()` ;

Et voici ce qui se passe lors de l'instanciation de C1 ligne 19 :

1. `C1` hérite de `T` : le constructeur de `T` est appelé en premier :
   (a) la ligne 2 initialise le champ privé `T.v` à `"dant T"` ;
   (b) la ligne 4 initialise le champ privé `T.l` à `"dant T"` ;
   (c) la ligne 5 est exécutée ce qui implique :
   - la lecture de `v`, donc l'appel à l'accesseur `v()`, qui est surchargé dans `C1`. La résolution dynamique appelle donc l'accesseur `C1.v()`, qui lui-même retourne la valeur du champ `C1.v`, qui n'a pas encore été initialisé puisque le constructeur de `C1` n'a pas encore été exécuté : la valeur retournée est `null` :
   - l'appel à la méthode `d`, qui est surchargée dans `C1` et donc comme précédemment, c'est la méthode `C1.d` qui est appelée : la valeur retournée est `"dans C1"` ;
   - la lecture de `l`, donc comme pour `v`, l'appel à l'accesseur `l()` résolu dynamiquement en l'appel de `C1.l()`. Mais comme `l` est définie comme étant `lazy`, son initialisation se fait lors du premier accès : habituellement, cela veut dire plus tard que l'équivalent `non lazy`, mais dans ce cas précis, c'est plus tôt : `C1.l` est initialisée à sa valeur `"dans C1"`, qui est retournée.
2. puis le constructeur de `C1` est appelé :
   (a) la ligne 9 initialise `C1.v()` à la valeur `"dans C1"` ;
   (b) la ligne 14 ne fait rien puisque la valeur `C1.l()` a déjà été initialisée ;
   (c) la ligne 15 est exécutée, ce qui implique :
   - la lecture de `v`, donc l'appel à l'accesseur résolu dynamiquement `C1.v()` qui retourne la valeur de `C1.v`, correctement initialisée à `"dans C1"` ;
   - l'appel à la méthode `d` qui comme précédemment est résolu en `C1.d` : la valeur retournée est `"dans C1"` ;

– la lecture de `l`, donc l'appel à l'accesseur résolu dynamiquement `C1.l()` qui retourne la valeur de `C1.l` correctement initialisée à `"dans C1"` ;

Danger 46 (D-S-46) : *Risque de levée de* `NullPointerException` *dûe à des champs non initialisés dans les constructeurs de classe.*

Recommandation 47 (R-S-47) : *Vérifier l'initialisation de tous les champs par les constructeurs de classe.*
Plusieurs solutions existent pour cela :
– ne pas utiliser `val` dans les traits pour les attributs non privés ;
– utiliser `final val` pour les valeurs concrètes qui ne doivent pas être redéfinies dans les classes qui héritent du trait ;
– par défaut, il est conseillé d'utiliser des `def` dans les traits même pour les valeurs concrètes qui peuvent être redéfinies (la JVM optimisera ce cas à l'exécution). ■

Trait Application

La bibliothèque standard propose un trait `Application` qui peut être étendu par un objet afin de créer une classe exécutable. Le trait définit la méthode `main(args:Array[String]):Unit` équivalente à la méthode `public static void main(String[] args)` de Java, et qui est reconnue comme un point d'entrée du programme une fois compilée en bytecode Java.

Ce trait permet ainsi d'écrire le code suivant :

```
1 object MyScalaApp extends Application {
2   println("Hello world!")
3 }
```

À la place de :

```
1 object MyScalaApp {
2   def main(args:Array[String]) : Unit = {
3     println("Hello world!")
4   }
5 }
```

Dans les deux cas, la phrase `Hello world!` sera écrite sur l'entrée standard.

Comme vu précédemment, le corps de l'objet est son constructeur. Toutes les instructions qui y sont codées sont donc exécutées lors de l'instanciation de

cet objet. Or cet objet possède la méthode Java magique `main` qui le désigne comme une classe exécutable, et l'objet est instancié par la JVM à cette fin, ce qui déclenche le constructeur. L'ensemble de l'application est in fine exécuté dans le constructeur de l'objet qui possède la méthode `main`.

Danger 48 (D-S-48) : *Le trait Application peut mener à des performances dégradées et des interblocages.*
Les risques pour la sécurité sont essentiellement de deux types :
- la gestion de la concurrence est incorrecte : la norme Java (JSR 133) définit le constructeur comme étant mono-threadé. Utiliser le constructeur pour exécuter l'ensemble d'une application multi-threadée risque d'aboutir à des situations d'interblocage ;
- les performances sont mauvaises, et le risque de déni de services importants : les constructeurs ne sont pas optimisés par la JVM comme le reste du code source.

L'ensemble de ces risques est levé à partir de la version 2.9 du langage : l'implémentation du trait Application a été modifiée pour les résoudre. ■

Recommandation 49 (R-S-49) : *Ne pas utiliser le trait Application.*
Pour les versions de Scala inférieures à la version 2.9, ne pas utiliser le `trait Application` mais implémenter la méthode main dans un objet. ■

Égalité d'objets et `hashCode`

En Scala, tout comme en Java, tous les objets possèdent des méthodes qui permettent de définir l'égalité entre objets. Ces méthodes sont héritées de l'objet Java racine `Object` et sont les méthodes `equals` et `hashCode`.

En Scala, et contrairement à Java, la méthode (==) est équivalente à la méthode `equals`. Le test d'égalité de référence entre objets se fait avec la méthode `eq`.

En Scala, tout comme en Java, la méthode `equals` doit respecter les propriétés suivantes[3] :

3. Ces propriétés ne sont pas définies dans la spécification du langage, mais dans le commentaire de la méthode `equals` pour la classe `Object` de la bibliothèque standard de Java. Elles sont également rationalisées dans [Bloch, 2008].

- elle doit être réflexive, symétrique et transitive ;
- elle doit être consistante dans le temps ;
- pour tout objet `x` non `null`, `x.equals(null)` doit retourner faux.

> *Danger 50* (D-S-50) : *La redéfinitions de equals doit respecter des règles strictes.*
> Les bibliothèques standard de Java et de Scala sont construites sous l'hypothèse de la validité des propriétés de `equals` définies dans les commentaires de cette méthode, pour la classe Java `Object`. Si celles-ci ne sont pas valides, alors le programme perd toute sémantique. ■

Ces propriétés, ainsi que des exemples des conséquences de leur non-respect sont détaillées dans [Bloch, 2008], Item 8 : *Obey the general contract when overriding equals*.

> *Recommandation 51* (R-S-51) : *Pour toute les classes, vérifier la sémantique de la méthode equals.*
> La méthode `equals` est très particulière au niveau de la JVM, et son implémentation doit être faite avec précaution : cette méthode doit être surveillée comme une faille potentielle dans le programme, pour toutes les classes, qu'elle soit redéfinie ou justement qu'elle ne le soit pas. ■

Tout comme en Java, les méthodes `equals` et `hashCode` sont liées par un contrat :

- si l'une d'elles est redéfinie, l'autre doit l'être aussi ;
- deux objets égaux doivent avoir le même `hashCode` ; la réciproque est fausse, et il est courant que deux objets différents aient des `hashCode` égaux.

> *Danger 52* (D-S-52) : *Un hashcode n'est pas un identifiant unique d'un objet.*
> Il ne faut pas considérer le résultat de `hashCode` comme un identifiant de l'objet. ■

> *Danger 53* (D-S-53) : *La redéfinition de* `hashCode` *doit respecter des règles strictes.*
> Si le contrat entre les méthodes `equals` et `hashCode` n'est pas respecté, alors les hypothèses sous lesquelles sont construites les

bibliothèques standard de Java et de Scala ainsi que d'autres propriétés de ces langages sont invalides et toutes les valeurs du programme peuvent être fausses. ■

Dans l'exemple suivant, le contrat de `hashCode` n'est pas respecté : la méthode `equals` a été redéfinie, mais `hashCode` ne l'a pas été donc c'est la méthode de la classe Java `Object` qui est utilisée (`hashCode` basé sur la référence de l'objet).

```
1  class Id(val x:Int) {
2    override def equals(other:Any) = other match {
3      case that:Id => this.x == that.x
4      case _ => false
5    }
6  }
7
8  val id1 = new Id(1)
9  val users = collection.mutable.HashMap(id1 -> "User1")
10
11 //ici, c'est un nouvel objet Id qui est utilise:
12 users.get(new Id(1) )
13 None
14
15 //et pourtant:
16 id1 == new Id(1)
17 true
```

On se retrouve avec un objet `id1` qui est égal aux autres instances de `Id` pour lesquelles `x == 1`, mais les préconditions sur lesquelles l'implémentation de `HashMap` se fonde ne tiennent plus : ses méthodes renvoient des valeurs fausses.

L'exemple précédent se comporte comme attendu si la méthode `hashCode` est redéfinie de manière à ce que deux objets qui sont égaux aient le même `hashCode`.

Cette caractéristique de Java, et donc de Scala, est très largement documentée : [Bloch, 2008].

Recommandation 54 (R-S-54) : *Il faut respecter les règles qui lient equals et* `hashCode`.
Le contrat entre `equals` et `hashCode` doit être respecté, tous les moyens pour atteindre cet objectif sont bons : analyse statique, revue de code, etc. ■

Danger 55 (D-S-55) : *L'utilisation de données mutables pour la définition des méthodes* `equals` *et* `hashCode` *est source d'erreur.*
La définition des méthodes `equals` et `hashCode` en fonction de données mutables est source de bogues et donc de failles de sécurité car les classes de la bibliothèque standard de Java prennent comme hypothèse que la valeur de `hashCode` d'un objet est stable dans le temps et que l'égalité entre objets mutables est une notion dont la sémantique est complexe à définir. ■

Voici un exemple des problèmes qui peuvent résulter de la non stabilité :

```
1  class Data {
2    var data = 1
3    override def hashCode = data
4  }
5  defined class Data
6
7  val data1 = new Data()
8  data1: Data = Data@1
9
10 //un ensemble qui contient des donnees sensibles
11 val set = collection.mutable.HashSet(data1)
12 set: scala.collection.mutable.HashSet[Data] = ↵
       Set(Data@1)
13
14 if(set.contains(data1)) {
15   println("contient des donnees sensibles")
16 } else {
17   println("pas de donnees sensibles")
18 }
19 contient des donnees sensibles
20
21 //maintenant, on met a jour les donnees sensibles
22 data1.data = 2
23
24 //le set pense qu'il ne contient plus data1
25 if(set.contains(data1)) {
26   println("contient des donnees sensibles")
27 } else {
28   println("pas de donnees sensibles")
29 }
```

```
30 pas de donnees sensibles
31
32 //et pourtant, l'objet data1 est toujours dans le set,
33 //et est accessible par iteration sur tous les objets:
34 set.foreach { d =>
35   println("donnees sensibles: " + d.data)
36 }
37 donnees sensibles: 2
```

Recommandation 56 (R-S-56) : *Utiliser des données non mutables pour définir* `equals` *et* `hashCode`.
Il est conseillé de redéfinir `equals` et `hashCode` uniquement en fonction de données non mutables, par exemple en ajoutant un champ "identifiant" à l'objet. ■

L'ensemble de ces contraintes et des problèmes qui leurs sont liées sont exacerbées lorsque l'on considère que dans le cas général, l'ensemble des classes filles ne sont pas connues lors de la compilation (c'est l'hypothèse du monde ouvert de Java). Il faut donc prévoir dans les définitions de ces méthodes le cas des classes filles, et savoir définir a priori la sémantique qui sera associée à ces cas : est-ce qu'une classe fille est considérée comme égale à une classe parente, est-ce que des classes filles appartenant à des branches différentes de l'arbre d'héritage peuvent être égales, etc.

Danger 57 (D-S-57) : *L'hypothèse d'un monde ouvert complexifie la définition de la méthode equals.*

Recommandation 58 (R-S-58) : *N'utiliser equals que pour l'identification d'instances, et définir d'autre méthodes pour les tests d'égalité métier.*

3.2.2 Typage, paradigme objet et fonctionnel

Ce chapitre décrit les grandes caractéristiques du système de types de Scala. Scala a été construit comme l'union de deux grands paradigmes de programmation : orienté objet et impératif d'une part, fonctionnel d'autre part. Les classes jouent donc un rôle prépondérant dans le système de types, qui doit tenir compte

de notions objets telles que l'héritage : comme en Java, une hiérarchie de classes définit la hiérarchie équivalente de types.

Propriétés du typage

L'influence du côté fonctionnel tend à pousser les limites du système de types avec des fonctionnalités telles que les types paramétrés polymorphes, tout en apportant une cohérence à l'ensemble, cohérence qui fait parfois défaut à des langages qui ne sont qu'objet.

Ainsi, Scala est un langage **fortement statiquement typé**[4]. Cela signifie que toute expression possède un type et un seul et que celui-ci peut être déterminé sans évaluer l'expression. Le compilateur effectue une **vérification des types** du programme.

Intérêt 59 (I-S-59) : *Le typage statique fort apporte des garanties.*
Le typage apporte plusieurs garanties sur le code source : accès cohérent aux variables, cohérence de la manipulation des données vis-à-vis de leur type, constance des valeurs non-mutables, protection des données de types abstraits, exhaustivité du filtrage. ■

Intérêt 60 (I-S-60) : *Le compilateur détecte les erreurs de typage.*
Le compilateur produit un code compilé qui offre ces garanties ou détecte des erreurs de typage. ■

Intérêt 61 (I-S-61) : *Expressivité apportée par le typage.*
L'abstraction fournie par les types permet d'exprimer certains aspects de la sémantique du programme. ■

Recommandation 62 (R-S-62) : *Utiliser les types pour représenter les propriétés sur les données.*
Il est recommandé d'utiliser le langage de types pour refléter au mieux la structure des données manipulées par le programme. Cela facilite la traçabilité de la spécification dans le texte source et permet de bénéficier de vérifications automatiques et systématiques. Cela libère de plus le programmeur de la gestion bas niveau de ces structures. ■

4. Le genre de types qui s'applique à chaque élément est défini dans le chapitre 3 des spécifications du langage Scala, et cette partie n'introduit que les types liés à des valeurs.

Inférence de types

Scala propose l'**inférence de types** avant d'effectuer les vérifications de types : le type des expressions d'un code source peuvent être synthétisés par le compilateur. Les types des paramètres des méthodes, constructeurs de classes inclus, ne peuvent pas être inférés et doivent toujours être spécifiés.

Ceci signifie que le programmeur n'a pas à déclarer tous les types. Le type inféré par le compilateur est correct, au sens où il est conforme aux règles de typage.

> *Intérêt 63* (I-S-63) : *L'inférence de types allège l'écriture et la relecture de code.*

Dans le cas où l'inférence de type échoue, le compilateur signale quelles expressions lui posent problème et il suffit alors soit de corriger l'erreur de typage soit d'ajouter l'information de type nécessaire au compilateur.

Tous les types inférés durant la compilation ne sont par défaut pas présentés au développeur car le volume d'informations produit est trop important. Les options `-print` et `-explaintype` vues dans la section 3.1.4 permettent d'afficher l'ensemble des informations sur les types. Les environnements de développement (IDE) permettent également d'obtenir directement le type inféré pour chaque expression.

> *Intérêt 64* (I-S-64) : *Tous les types inférés peuvent être présentés au développeur.*

L'inférence de type peut attribuer un type qui ne correspond pas forcément à celui imaginé par le programmeur, soit parce que trop général soit parce que différent de la spécification voulue.

> *Intérêt 65* (I-S-65) : *L'inférence de type révèle des erreurs de programmation.*
> Une différence entre le type inféré par le compilateur et celui attendu par le développeur peut révéler une erreur dans le texte source. En particulier, si le type inféré est plus générique que celui attendu cela démontre que le texte source est plus générique que prévu. ■

> *Recommandation 66* (R-S-66) : *Vérifier que le type inféré correspond au type attendu.*

L'exemple suivant montre le fonctionnement de l'inférence de type Scala : le type inféré pour `b` est la borne supérieure des types `string` et `int`, c'est-à-dire `Any` (voir ci-après).

```
1 val a = println("hello")
2 hello
3 a: Unit = ()
4
5 val b = {
6   if(true) { "foo"
7   } else   {  10 }
8 }
9 b: Any = foo
```

La suite du chapitre détaille les propriétés avancées du système de types de Scala.

Hiérarchie de classes

Les classes et l'héritage entre classes sont au centre du système de types de Scala. Comme le langage se veut totalement compatible avec Java, en particulier en autorisant les classes Java à étendre les classes Scala et réciproquement, il reprend les grandes caractéristiques de la hiérarchie de classes Java.

Les types des classes de Scala sont bornés par un ensemble de types racine prédéfinis dans la hiérarchie de classes et représentent donc un treillis, présenté dans la figure 3.2.

Bornes supérieures : types Any, AnyRef et AnyVal.

Scala prédéfinit un type **top** nommé `Any` dont toutes les autres classes sont un sous-type. Cette classe englobe à la fois les types des valeurs représentées par des objets (références) et celui des valeurs spécifiques de la JVM (types primitifs, etc).

Cette classe est abstraite et ne peut pas être étendue par d'autres classes que par les deux sous-classes racines suivantes :

- la classe `AnyRef`. C'est la classe parente de toutes les valeurs représentées par des objets sur la JVM. C'est un alias de la classe Java `java.lang.Object`, et l'ensemble des classes Java s'intègre donc naturellement dans la hiérarchie Scala à partir de ce point ;

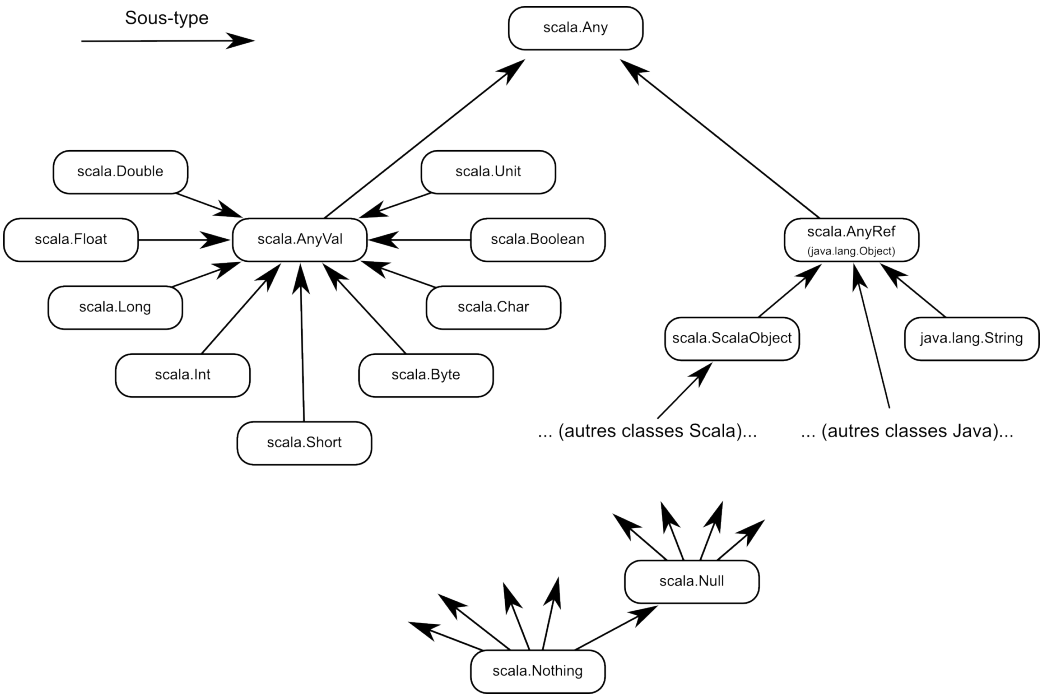

FIGURE 3.2 – Classes racines de Scala

Classes racines de Scala

- la classe `AnyVal`. Cette classe a un nombre prédéterminé de sous-types. C'est la classe parente des objets Scala qui ne se traduisent pas par des objets sur la JVM, et qui sont donc des types spécifiques au niveau du bytecode Java, à savoir :
  - les types primitifs de la JVM ;
  - le type qui correspond à la méthode `void`.

Bornes inférieures : types `Nothing` et `Null`. Contrairement à Java, Scala définit également un type **bottom**. C'est la classe `Nothing`, qui est un sous-type de tous les autres types, et donc des sous-types de `AnyVal`. `Nothing` n'est habité par aucun élément.

`Nothing` a un sur-type direct parmi les classes racines : `Null`. Ce type est habité par un unique élément : `null`, qui est compatible avec tous les sous-types de `AnyRef`. La valeur `null` correspond au `null` du bytecode Java.

Type Unit. Il existe un type `Unit` pour les éléments qui sont strictement impurs, c'est-à-dire qui ne réalisent que des effets de bord, comme les exceptions ou les entrées/sorties. Le type `Unit` possède une unique valeur, `()`. Ce type correspond à la notion de valeur de retour `void` de Java, comme le montre l'exemple ci-dessous.

```
1 val x = println("Hello, World!")
2 Hello, World!
3 x: Unit = ()
```

Danger 67 (D-S-67) : *Le typage d'un bloc annoté par le type Unit réussit toujours.*
Si la dernière instruction d'un bloc de type `Unit` n'est pas de type `Unit`, alors le compilateur introduit implicitement une instruction `()` comme dernière instruction du bloc, sans lever d'erreur de typage. ■

```
1 val x : Unit = { 42 }
2 x: Unit = ()
```

Recommandation 68 (R-S-68) : *Utiliser au minimum les blocs de type Unit.*
Le type `Unit` devrait être évité au profit de valeurs et de types

qui apportent de l'information sur le but du bloc. Dans ce cas, une incompatibilité de type entre le type déclaré du bloc et le type de la dernière instruction donnera une erreur de compilation et l'erreur sera détectée. ■

Ce type n'existe pas réellement en Java. En particulier, Java ne connaît `void` que comme type de retour d'une méthode, et non comme type d'un paramètre. Dans le cas où `Unit` est utilisé comme type d'un paramètre de classe ou de méthode, le compilateur Scala utilise au niveau bytecode Java le type `scala.runtime.BoxedUnit`, qui est une classe.

Ce choix d'implémentation peut entraîner des différences sémantiques entre le code source et le bytecode Java produit, en particulier lors de l'initialisation de champ `var` : par défaut, la JVM initialise les objets à `null`, alors que `Unit` devrait être initialisé avec son unique valeur, `()`. Encore plus étonnant, assigner une valeur, même `null` à ce champ, rétablit le comportement attendu puisque cette fois, la logique de `scala.runtime.BoxedUnit` est utilisée, comme le montre l'exemple suivant :

```
1  val arr = new Array[Unit](1)
2  arr: Array[Unit] = Array(null)
3
4  arr(0).toString
5  java.lang.NullPointerException
6  at ...
7
8  arr(0) = null
9
10 scala> arr(0).toString
11 res2: java.lang.String = ()
```

Danger 69 (D-S-69) : *Les types Unit et AnyVal n'existent pas au niveau bytecode Java, ce qui peut entraîner des incohérences par rapport au code source.*
Les cas limites (en particulier l'initialisation) de valeurs de type `Unit` peut entrainer des différences sémantiques entre code source et bytecode Java généré, ce type n'existant pas à ce niveau. Ceci est aussi valide pour `AnyVal`. ■

Recommandation 70 (R-S-70) : *Les champs mutables doivent toujours être initialisés explicitement.*

> Toujours initialiser explicitement la valeur des variables avant leur utilisation, et ne jamais utiliser l'initialisation par défaut. ■

Types primitifs. Scala n'a pas de types primitifs. A la place de types primitifs, Scala définit des classes qui étendent`AnyVal` : `scala.Boolean`, `scala.Byte`, `scala.Short`, `scala.Char`, `scala.Int`, `scala.Long`, `scala.Float`, `scala.Double`.

Ces types sont différents des types Java primitifs (`boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`) et des objets Java qui leur sont associés (`java.lang.Boolean`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Character`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`).

Cette approche permet d'éviter les problèmes de sécurité liés au mélange de types primitifs et d'objets, et au fonctionnement de l'**auto-boxing** de la JVM.

L'`auto-boxing` permet de laisser à la JVM la liberté de transformer des valeurs de types primitifs en leur équivalent objet, et réciproquement, en fonction des besoins de typage ou d'optimisation.

L'un des problèmes de l'autoboxing est qu'en Java, les objets qui correspondent aux types primitifs peuvent recevoir la valeur `null`, et que l'**unboxing** de `null` lève une `NullPointerException`, comme le montre l'exemple Java suivant :

```
1 /* exemple Java */
2
3 Integer i = null;
4 if(i == 0) { System.out.println("Hello") ; }
5 java.lang.NullPointerException
```

Le cas direct est assez évident, mais il le devient moins lorsque plusieurs étapes de **boxing/unboxing** se succèdent :

```
1 /* exemple Java */
2
3 Integer value = false ? 0 : null;
4
5 //ceci ecrit correctement: Value is: null
6 System.out.println("Value is: " + value );
7 Value is: null
8
```

```
9  Integer value = false ? 1 : ( false ? 0 : null );
10
11 //ceci leve une exception NullPointerException
12 System.out.println("Value is: " + value );
13 NullPointerException
```

On pourrait penser que dans les deux cas, `value` aurait la valeur `null`, ce qui est une valeur correcte pour le type Java `Integer`. Cependant, dans le second cas, le type attendu par le comparateur ternaire pour son membre droit est le type primitif `int`, or il reçoit un type objet `Integer` (qui vaut `null`), ce qui entraîne un **unboxing** automatique, qui lève l'exception.

En Scala, les types qui correspondent aux types primitifs ne peuvent pas recevoir la valeur `null` et cette propriété est vérifiée par le typage :

```
1  val i:Int = null
2  <console>:5: error: type mismatch;
3   found   : Null(null)
4   required: Int
5         val i:Int = null
6                     ^
7
8
9  1 eq null //comparaison de reference
10 <console>:6: error: type mismatch;
11  found   : Int
12  required: ?{val eq(x\$1: ?>: Null(null) <: Any): ?}
13 Note: primitive types are not implicitly converted to ↵
     AnyRef.
14 You can safely force boxing by casting ↵
     x.asInstanceOf[AnyRef].
15        1 eq null
16        ^
17
18 1 == null
19 <console>:6: warning: comparing values of types Int ↵
     and Null using `==' will
20 always yield false 1 == null
21                    ^
22 res1: Boolean = false
```

> *Intérêt 71* (I-S-71) : *Le typage de Scala prévient la plupart des cas de déréférencement de pointeur* `null` *qui concernent les types numériques.*
> Dans la plupart des cas, Scala empêche, par construction, les problèmes de `NullPointerException` dus à l'auto-boxing. ■

De plus, le fait que Scala définisse ses propres types pour représenter les nombres et `Unit` permet au compilateur de clairement séparer les types Scala de ceux issus de bibliothèques Java.

> *Intérêt 72* (I-S-72) : *Les propriétés de cohérence sur les types numériques sont valides aussi pour les bibliothèques dont seul le bytecode est connu.*
> Le système de coercition de types de Scala permet de valider la cohérence des types primitifs ou boxés issus de bibliothèques Java lorsqu'ils sont utilisés en Scala. ■

Malgré ces efforts, il existe des cas pour lesquels un pointeur `null` peut être rencontré. Le problème est dû à la combinaison de l'effacement de type de la JVM et de la valeur d'initialisation des `Object` à la valeur `null`, comme le montre l'exemple suivant :

```
1  class C[T] { var x: T = _ }
2  defined class C
3
4  val c = new C[Int]
5  c: C[Int] = C@1279c8
6
7  c.x.toString
8  java.lang.NullPointerException
9    at ...
10
11 c.x = 0
12
13 c.x.toString
14 res7: java.lang.String = 0
```

Ici, le bytecode Java généré pour `C` est généré pour un objet à cause de l'effacement de type, et le constructeur initialise donc `x` à `null`. Comme précédemment pour `Unit`, l'assignation de n'importe quelle valeur à `x` rétablit la cohérence de la valeur.

Danger 73 (D-S-73) : *Le typage ne permet pas d'éviter les déréférencements de pointeur* `null` *pour les variables numériques non initialisées explicitement.*
L'effacement de type de la JVM couplé à des variables non initialisées explicitement peut entraîner des déréférencements de pointeurs `null` pour les types descendant de `AnyVal`.

Recommandation 74 (R-S-74) : *Toujours initialiser explicitement les variables mutables.*

Spécialisation. L'auto-boxing peut également poser des problèmes de performances, et donc de disponibilités. L'approche de Scala propose une réponse via la spécialisation des types pour les types primitifs : dans le cas où une classe ou une méthode est paramétrée par un type et que ce type est annoté avec `@specialized`, le compilateur Scala génère au niveau du bytecode Java toutes les combinaisons possibles de spécialisation du type.

Soit par exemple le code suivant :

```
1 class C[@specialized T](x:T)
2 defined class C
```

Une fois compilé, ce code génère dix classes en bytecode Java : une pour le cas où T est une classe descendant d'`Object`, une pour chaque type primitif de Java, et une pour le cas `Unit`.

La même logique de compilation est appliquée aux méthodes paramétrées par un type : les différentes versions surchargées de la méthode sont générées dans le bytecode Java.

Lors de l'exécution, grâce à la résolution dynamique, c'est la version la plus intéressante de la classe ou de la méthode qui sera utilisée.

La spécialisation a également pour avantage de contrer les problèmes de divergence de sémantique entre code source Scala et bytecode Java vus dans les paragraphes précédents.

Ainsi, l'exemple suivant n'entraîne pas de déréférencement de `null`[5] :

```
1 class C[@specialized T] { var x: T = _ }
2 defined class C
3
```

5. Un bug du REPL de Scala 2.8.1 rend impossible l'exécution du code. Il faut donc soit utiliser une version plus récente du REPL, soit compiler le code via la commande `scalac`

```
4 val c = new C[Int]
5 c: C[Int] = C@1279c8
6
7 c.x.toString
8 res0: 0
```

Intérêt 75 (I-S-75) : *La spécialisation des types numériques peut améliorer la robustesse et l'efficacité des programmes.*
La spécialisation :
- améliore de manière conséquente les performances de programmes qui utilisent beaucoup les types primitifs.
- évite les différences de sémantique entre code source et bytecode Java dans certains cas limites, en particulier lors d'initialisation de champs. ■

Cependant, le spécialisation coûte cher en place. Dans le cas de méthodes avec plusieurs paramètres, l'ensemble de toutes les combinaisons parmi dix cas doit être généré.

Recommandation 76 (R-S-76) : *Utiliser la spécialisation lorsque son coût n'est pas trop élevé.*
Utiliser la spécialisation dans les cas où peu de types paramétrés sont présents. ■

Fonctions anonymes et fermeture

Scala possède des fonctions anonymes et des fermetures. Dans les deux cas, ces concepts sont implémentés au moyen de classes synthétisées par le compilateur, dont les champs sont les paramètres de la fonction. Chaque appel correspond à la création d'un objet à partir de la classe synthétisée.

Danger 77 (D-S-77) : *Chaque fonction anonyme et chaque fermeture génère une classe au niveau du bytecode Java.*
L'utilisation extensive de fermetures ou de classes anonymes peut générer un très grand nombre de classes bytecode Java. Ceci peut facilement entrainer des dénis de service dus à l'utilisation de l'ensemble de l'espace de mémoire `PermSpace` de la JVM, qui est par défaut mal dimensionné pour cette utilisation. ■

Recommandation 78 (R-S-78) : *Adapter les profils mémoire de la JVM pour chaque programme.*
Adapter la configuration du paramètre `-XX:PermSize` au nombre de classes générées au niveau du bytecode Java par un programme Scala. ■

De vrai objets sont donc manipulés pour traiter les fonctions, avec l'ensemble des possibilités offertes par Scala ou la JVM (introspection, sérialisation, etc) De plus, une fermeture capture l'environnement d'exécution lors de sa définition. Mais l'étendue de l'environnement n'est pas spécifié dans les spécifications du langage et peut être très large.

Danger 79 (D-S-79) : *Les paramètres capturés dans une fermeture sont accessibles par introspection.*
Il est donc possible de faire de l'introspection sur les objets qui représentent une utilisation de fermeture : certains paramètres censés être secrets peuvent être accessibles. ■

Recommandation 80 (R-S-80) : *Ne valider l'utilisation d'une fermeture qu'après avoir vérifié son contenu par introspection.*
Lors de l'utilisation de fermetures, vérifier par introspection que l'objet ne contient pas de données sensibles. ■

Fonctions récursives

Tout comme Java, Scala permet de définir des **fonctions récursives** (ou mutuellement récursives).

Cependant, la JVM de référence n'optimise pas les appels **récursifs terminaux** qui font grossir la pile d'appel. Le compilateur Scala reconnaît les appels récursif terminaux et génère du bytecode Java optimisé (une séquence d'instructions bytecode Java correspondant au corps de la fonction qui se termine lorsque c'est nécessaire par un saut au début des instructions).

Intérêt 81 (I-S-81) : *La pile ne grossit pas pour un appel récursif terminal.*

Le compilateur ne peut optimiser une fonction récursive que si les conditions suivantes sont réunies :

– la méthode ne peut pas être redéfinie (elle est marquée `final` ou `private`, ou elle est locale à une autre méthode)

- elle est récursive simple (le compilateur ne peut pas optimiser de méthodes mutuellement récursives)
- l'appel récursif est terminal.

Danger 82 (D-S-82) : *Non optimisation de certaines méthodes récursives.*

Danger 83 (D-S-83) : *Un appel terminal englobé par un rattrapage d'exception n'est pas terminal.*
Si on englobe un appel récursif terminal par un `try...catch`, l'appel récursif n'est plus terminal car il est nécessaire de dépiler le bloc de gestion d'exceptions. ■

Afin de s'assurer qu'une méthode récursive sera bien optimisée par le compilateur, Scala propose l'annotation `@tailrec` : une méthode qui possède cette annotation sans qu'elle ne vérifie l'ensemble des critères entraîne une erreur de compilation.

Recommandation 84 (R-S-84) : *Utiliser l'annotation @tailrec sur les méthodes récursives terminales pour valider leur optimisation.*

Types paramétrés et types virtuels abstraits

Scala possède deux notions de classes paramétrées par un type :
- les classes dont la déclaration est paramétrée par un type, compatible avec les `generics` Java ;
- les classes qui ont comme membre un type laissé abstrait.

Il a été démontré que les types abstraits sont plus généraux que les types paramétrés, et qu'il existe une transformation simple des classes paramétrées par des types vers des classes possédant des types abstraits (la réciproque est fausse, voir [Odersky et Zenger, 2005], section "Modeling generics with abstract types").

Intérêt 85 (I-S-85) : *Scala permet de choisir le paramétrage le plus simple pour le problème à traiter.*
Les classes paramétrées sont semblables aux `Generics` de Java, alors que les types virtuels abstraits ressemblent aux types abstraits des langages `ML`. ■

Intérêt 86 (I-S-86) : *La généricité des structures de données se traduit dans les types polymorphes.*
Pouvoir exprimer la généricité d'une structure de données évite la duplication de code. ■

Scala permet également de définir des contraintes sur les bornes et la variance des paramètres de type et des types abstraits.

Intérêt 87 (I-S-87) : *Contrôle fin de l'abstraction par les contraintes sur les paramètres de types.*

Bornes de type. Les paramètres des types peuvent être bornés par un type supérieur et/ou un type inférieur.

Le compilateur vérifie la cohérence des types utilisés comme bornes. En particulier, dans le cas où les deux bornes sont spécifiées, le compilateur s'assure que la borne `SUP` est bien un sur-type de la borne `INF`.

Variance. Les types paramétrés peuvent être annotés par une information sur la variance de leurs paramètres. La variance permet de définir la relation de sous-typage entre types paramétrés.

Soient une hiérarchie de types `class B extends A` (donc où `B` est un sous-type de `A`) et un type paramétré `T[X]`.

Scala définit alors trois types de variance :

- non-variant, aussi nommé monovariant ou invariant : dans ce cas, `T[A]` et `T[B]` ne sont pas comparables (ils représentent deux types différents).
- covariant : dans ce cas, `T[A]` est un sur-type de `T[B]` (`T[B] <: T[A]`) ;
- contravariant : dans ce cas, `T[A]` est un sous-type de `T[B]` (`T[A] <: T[B]`) ;

De plus, le compilateur vérifie que les informations de variances sont cohérentes avec les positions d'utilisation du type dans les différentes méthodes de la classe. Ceci permet d'assurer à la compilation qu'aucune valeur d'un type incohérent avec les paramètres ne sera utilisée.

Ce point est une différence majeure entre Scala et Java dans le cas de tableaux. En Java, les tableaux sont covariants : un tableau d'entiers est un sous-type d'un tableau d'`Object`. Ceci introduit une incohérence vis-à-vis de la réassignation de valeurs dans les tableaux, ce qui a pour effet de casser la sûreté du système de types et impose des vérifications coûteuses lors de l'exécution (voir [JavaSec exécution, 2009] : `ArrayStoreException` et `http://www.c2.com/cgi/wiki?JavaArraysBreakTypeSafety`).

L'exemple Java suivant expose le problème :

```
1 public class Test {
2   public static void main(String[] args) {
3     String[] strings = new String[1];
4     Object[] objects = strings;
5
6     //RUN TIME error: java.lang.ArrayStoreException
7     objects[0] = new Integer(1);
8   }
9 }
```

En Scala, ce problème ne peut pas se poser par construction : les tableaux sont monovariants, comme l'impose la vérification de variance sur les méthodes qu'ils proposent. Dans l'exemple précédent, le tableau de String n'est pas compatible avec le tableau d'Object :

```
1 val strings : Array[String] = new Array[String](1)
2 strings: Array[String] = Array(null)
3
4 val objects : Array[Any] = strings
5 <console>:6: error: type mismatch;
6  found   : Array[String]
7  required: Array[Any]
8        val objects : Array[Any] = strings
9                                  ^
```

Type propre

Les classes Scala possèdent une notion de **type propre** (`self type`) : ce type doit être vérifié par toute classe fille de la classe qui définit le type propre. Ce système permet de contraindre les possibilités d'héritage, en particulier dans le cas de l'héritage multiple des `traits`.

Dans l'exemple suivant, la valeur `error` ne compile pas car le type propre de `T` n'est pas vérifié :

```
1 class C
2 defined class C
3
4 trait T { typepropre:C => }
5 defined trait T
```

```
6
7  val ok = new C with T
8  ok: C with T = $anon$1@1aeeb406
9
10 class D
11 defined class D
12
13 val error = new D with T
14 <console>:10: error: illegal inheritance;
15  self-type D with T does not conform to T's selftype T ↵
      with C
16        val error = new D with T
17                                ^
```

Recommandation 88 (R-S-88) : *Utiliser les types propres pour contraindre les possibilités d'héritage.*

Type existentiel

Scala permet de définir des types existentiels. Ces types sont essentiellement utilisés afin d'assurer la cohérence du système de types Scala avec les types jokers des `generics` Java.

Recommandation 89 (R-S-89) : *Ne pas utiliser les types existentiels.*
Préférer aux types existentiels, dont la sémantique est complexe, l'utilisation de types abstraits ou de types paramétrés. ■

Type dépendant du chemin

En Scala, les objets, les classes et les traits sont des types qui dépendent du **chemin** de l'élément dans lequel ils sont déclarés. La notion de chemin est définie dans le chapitre 3.1 des spécifications du langage Scala, et peut être vue comme la liste de l'ensemble des packages, classes, traits ou valeurs qu'il faut traverser pour arriver à l'élément en question (voir la section 3.2.5 pour les possibilités d'imbrication de définitions en Scala).

Cette notion permet de définir des types qui dépendent de valeurs du langage (par exemple lorsqu'une classe est définie dans objet, c'est à dire qu'un type est

défini dans une valeur), et renforce encore l'expressivité du système de types de Scala.

Mot-clé implicit

Le mot-clé `implicit` introduit deux notions différentes en Scala :
- le passage d'argument trouvé dans l'environnement d'exécution d'une méthode de manière implicite ;
- la définition par l'utilisateur et l'utilisation de méthode de coercition entre types, utilisées implicitement par le compilateur lorsqu'une expression qui n'était pas compatible avec le système de types peut le devenir.

Dans les deux cas, des règles strictes d'application de l'ajout des éléments sont définies dans la spécification du langage.

Aucun problème direct de sécurité n'a été trouvé pour ces fonctionnalités, mais leur existence et leur utilisation peuvent rendre le code complexe à comprendre.

> *Recommandation 90* (R-S-90) : *Éviter d'utiliser les méthodes et paramètres implicites.*
> Dans le cas général, il est déconseillé d'utiliser les méthodes ou paramètres implicites. ■

Polymorphisme ad hoc

L'utilisation conjointe des types paramétrés et des arguments implicites permet de définir un polymorphisme ad hoc. Ce type de polymorphisme permet de spécialiser un type en fonction de son contexte d'utilisation, tout en assurant que la cohérence des types est vérifiée à la compilation.

L'exemple suivant montre comment cette propriété peut être utilisée pour se servir de la vérification de type comme d'un vérificateur de théorème du programme.

```
1 trait Color
2 defined trait Color
3
4 trait Color
5 defined trait Color
6 class Red extends Color
7 defined class Red
```

```
class Blue extends Color
defined class Blue


//predicat:
class CalmColor[T <: Color]
defined class CalmColor

//theoreme:
def doWithCalmColor[C <: Color](color: C)(implicit ↵
    cc:CalmColor[C]) = {
    println(color.toString)
}
doWithCalmColor: [C <: Color](color: C)(implicit cc: ↵
    CalmColor[C])Unit

//on ajoute l'axiome suivant a l'environnement
implicit val blueIsCalm = new CalmColor[Blue]
blueIsCalm: CalmColor[Blue] = CalmColor@76fb1b

//theoreme non prouve
doWithCalmColor(new Red)
<console>:14: error: could not find implicit value for ↵
    evidence parameter of
type CalmColor[Red] doWithCalmColor(new Red)
                               ^

//theoreme prouve
scala> doWithCalmColor(new Blue)
Blue()
```

Cette méthode est en particulier utilisée dans la bibliothèque standard Scala, dans l'API collection pour que lorsque les éléments d'une collection sont transformés au moyen d'une méthode d'ordre supérieur appliquée à la collection, le type de la collection retournée soit le plus précis compatible avec les éléments [Odersky et Moors, 2009].

```
BitSet(1,2,3) map (x => x + 1)
res0: collection.immutable.BitSet = BitSet(2, 3, 4)

BitSet(1,2,3) map (x => "'" + x.toString + "'")
```

```
res1: collection.immutable.Set[java.lang.String] = ↵
    Set('1', '2', '3')
```

3.2.3 Filtrage

Comme la plupart des langages fonctionnels, Scala propose du filtrage linéaire, avec un cas joker noté `'_'`, des gardes, et des alternatives de cas introduites par le caractère `'|'`.

Comme pour les autres traits fonctionnels de Scala, le filtrage est pleinement intégré avec le paradigme objet.

Cette intégration s'effectue à plusieurs niveaux :

- l'ensemble de valeurs peuvent être filtrées et leur type polymorphe retrouvé par filtrage ;
- les types somme, qui permettent une analyse de complétude du filtrage, sont définis à partir de hiérarchie de classes et des classes spéciales, les `case classes` ;
- la logique du test de filtrage peut être ajoutée à n'importe quelle classe, même si elle n'est pas une `case class`.

Ces points sont détaillés dans les sections suivantes.

Filtrage objet et type polymorphe

Le filtrage peut être effectué sur n'importe quel objet, et donc sur toutes les valeurs du langage. Si un objet testé par filtrage ne correspond à aucun des filtres, alors une exception est levée à l'exécution. Le filtrage permet également de filtrer un objet suivant son type polymorphe réel et remplace le test Java habituel (`instance of`) suivi d'une **coercition descendante**, comme le montre l'exemple suivant :

```
class A
class B(val x:Int) extends A

val a:A = new B(1)

//a est de type A, on ne peut pas acceder a 'x'
//sans passer par du downcast:

//methode dangereuse: peut lever une ClassCastException
//si l'on n'a pas reellement une instance de B
```

```
11 println("x: " + (a.asInstanceOf[B]).x)
12
13 //methode avec test explicite de type
14 if(a.isInstanceOf[B]) {
15   println("x: " + (a.asInstanceOf[B]).x)
16 } else println("objet d'un type bornes")
17
18 //methode preferee
19 a match {
20   case b:B => println("x: " + b.x)
21   case _   => println("objet d'un type bornes")
22 }
```

L'exemple idiomatique est la redéfinition de méthode `equals` (voir la section 3.2.1, et [Odersky *et al.*, 2008, chapitre 28]).

```
1  final class Test(val value:String) {
2    override val hashCode = 7 + value.hashCode
3
4    override def equals(other:Any) : Boolean = {
5      other match {
6        case that:Test => this.value == that.value
7        case _         => false
8      }
9    }
10
11 }
```

Recommandation 91 (R-S-91) : *Utiliser le filtrage à la place du transtypage direct.*
En Scala, il est conseillé d'utiliser le filtrage pour effectuer des tests de typage et les actions liées de transtypages. ■

Décomposition

Le filtrage permet également de décomposer les valeurs filtrées de manière récursive.

Par exemple, il est possible de déconstruire une liste par filtrage, afin d'étudier les cas de liste vide, de liste à un élément, et les autres cas :

```
1  def filtre(l:List[String]) = l match {
2    case Nil          => println("Liste vide")
3    case "foo" :: Nil => println("Liste foo!")
4    case h :: tail    => println("Liste dont le premier
5                           element est: " + h) }
```

Scala impose des restrictions sur les objets qui peuvent être décomposés par filtrage : soit la classe dont est issu l'objet est une `case class`, soit il existe un extracteur qui correspond au filtre demandé.

Case class.

Une `case class` a plusieurs propriétés :

- les paramètres du constructeur sont utilisables pour la déconstruction,
- ils sont utilisés pour générer automatiquement une méthode `equals`, `hashCode` et une méthode de copie `copy`,
- les paramètres sont implicitement publics et non-mutables, sauf si annotés explicitement comme privés ou mutables.

> *Intérêt 92* (I-S-92) : *Les méthodes* `equals` *et* `hashCode` *sont implémentées automatiquement pour les case classes.*
> Les `case class` sont un moyen efficace de créer des données non mutables complexes qui respectent les contraintes sur les méthodes `equals` et `hashcode`. ■

De plus, la manière idiomatique pour créer un type somme en Scala est la définition d'une hiérarchie de classes dont les noeuds sont des classes abstraites et les feuilles sont des `case classes`.

```
1 sealed trait Cas
2 final case object Cas1 extends Cas
3 sealed trait SousCas   extends Cas
4 final case class Cas2(s:String)    extends SousCas
5 final case class Cas3(i:Int,j:Int) extends SousCas
```

> *Intérêt 93* (I-S-93) : *L'exhaustivité du filtrage est vérifiée sur les classes sealed.*
> Lorsque que le type de l'objet filtré est issu d'une classe annotée avec `sealed`, le compilateur vérifie la complétude du filtrage à la compilation. ■

L'annotation `@unchecked` permet de désactiver la vérification d'exhaustivité du filtrage.

> *Recommandation 94* (R-S-94) : *Ne pas utiliser l'annotation* `@unchecked`.
> Ne jamais utiliser l'annotation `@unchecked`. ■

> *Danger 95* (D-S-95) : *L'exhaustivité du filtrage n'est vérifiée que sur des classes dont le code source est écrit en Scala.*
> La vérification d'exhaustivité ne porte pas sur les classes qui sont définies dans un autre langage que Scala. ■

Par exemple, si une classe `SousCasJava` qui étend `SousCas` est définie en Java, ce qui est possible malgré le modificateur `sealed`, alors le code suivant va lever une exception de cas non filtré sans que le compilateur n'ait soulevé le problème, ni au niveau du typage (SousCas est bien de type Cas), ni au niveau de la complétude du filtrage :

```
1
2 scala> (new SousCasJava:Cas) match {
3   case Cas1      => "ok"
4   case Cas2(x)   => "ok"
5   case Cas3(x,y) => "ok"
6 }
7 scala.MatchError: test.SousCasJava@17b2712
```

Extracteurs. Scala permet de déconstruire les classes autres que les `case class` au moyen des méthodes spéciales `unapply` et `unapplySeq`.

Par exemple, il est possible de créer une règle de filtrage pour les chaînes de caractères qui représentent un email afin d'extraire d'une part le nom et d'autre part le nom d'hôte :

```
1 object EMail {
2   def unapply(str: String): Option[(String, String)] = {
3     val parts = str split "@"
4     if(parts.length == 2) Some( parts(0), parts(1) )
5     else None
6   }
7 }
8 defined module EMail
```

```
9
10 "bob@test.com" match {
11   case EMail(name,host) =>
12     println("name: " + name + " ; host: " + host)
13   case _ =>
14     println("not an email")
15 }
16 name: bob ; host: test.com
```

La méthode `unapplySeq` permet de filtrer des séquences d'objets suivant la même logique.

3.2.4 Alternative à l'utilisation de `null`

La gestion de la référence `null` est un problème majeur dans toute application, qui peut entraîner de nombreux trous de sécurité. Afin d'être compatible avec Java, Scala autorise cette valeur.

> *Danger 96* (D-S-96) : *Le déréférencement de pointeur null est autorisé par le langage mais conduit à des erreurs.*
> Les programmes Scala doivent gérer le déréférencement de pointeur `null` comme les programmes Java. ■

Cependant, Scala propose une alternative via la type `Option`, filtrable en `None` ou `Some(value)`. Cette hiérarchie de classe permet de définir le retour possible d'une valeur sans utiliser la référence `null`. De plus, un type `Option` a une structure monadique et est donc chaînable dans les boucles `for`, comme le montre l'exemple suivant :

```
1 val map = Map(1 -> "a", 2 -> "b")
2 map: collection.immutable.Map[Int,java.lang.String] = ↵
    Map((1,a), (2,b))
3
4 for {
5   val_1 <- map.get(1)  //Map.get : Int => Option[String]
6 } yield {
7   val_1  // ici, val_1 est de type String
8 }
9 res0: Option[java.lang.String] = Some(a)
10
11 for {
```

```
12   val_1 <- map.get(1)
13   val_3 <- map.get(3)
14 } yield {
15   val_1 + val_3
16 }
17 res1: Option[java.lang.String] = None
```

Recommandation 97 (R-S-97) : *À là place de la valeur* `null`*, utiliser le type* `Option` *afin de dénoter une valeur optionnelle.*
Ne jamais utiliser `null` comme valeur de retour d'une fonction.
Pour les valeurs de retour facultatives, utiliser le type `Option` ■

3.2.5 Unité de compilation et espace de noms

En Scala, un même fichier source peut contenir un nombre quelconque de définitions de classes, traits, objets et packages. Ces éléments peuvent être imbriqués sans limites de profondeur avec la restriction qu'un `package` ne peut être contenu que dans un autre `package`.

Tout comme Java, Scala possède une notion d'espace de noms qui permet d'utiliser le même nom pour différents éléments d'un programme sans qu'il y ait de problèmes d'ambiguïté. On définit le chemin de l'élément par la concaténation de l'ensemble des éléments imbriqués depuis le `package` de plus haut niveau jusqu'à l'élément considéré.

```
1  package p1 {
2    package p2 {
3      class C2 {}
4    }
5    object O1 {
6      val name = "name"
7      object O2 {
8        val name = "name"
9        class C1 {}
10       trait T1 {}
11     }
12   }
13 }
```

Dans le code précédent, le chemin du trait T1 est p1.O1.O2.T1. Le code utilise deux fois le nom `name`, mais dans deux espaces de noms différents, et donc avec deux chemins différents.

Si l'on compile ce fichier, qui peut se trouver sous n'importe quel répertoire du code projet, l'arborescence de répertoires suivante est obtenue :

```
1 .
2 `-- p1
3     |-- O1.class
4     |-- O1\$.class
5     |-- O1\$O2\$C1.class
6     |-- O1\$O2\$.class
7     |-- O1\$O2\$T1.class
8     `-- p2
9         `-- C2.class
```

L'arborescence qui résulte de la compilation reprend l'organisation Java, les répertoires copiant l'organisation des packages.

Un fichier bytecode Java class est généré pour chaque élément du fichier comme ce serait le cas pour des classes internes en Java.

Aucun danger de sécurité particulier n'est à soulever, mais les libertés d'organisation fournies par Scala peuvent mener à un code source confus, et donc favoriser l'apparition de bogues.

Certains outils tiers prévus pour Java qui travaillent à la fois sur le code source et le bytecode Java sont prédéfinis pour chercher les sources dans les répertoires correspondant aux noms de packages et pourraient donner des résultats erronés. C'est en particulier le cas pour les IDE, des outils de couverture de test, d'analyse statique, etc.

> *Recommandation 98* (R-S-98) : *Conserver la convention issue de Java entre nom de package et répertoire du code source.*
>
> – conserver la convention "nom de package" = "arborescence de répertoires qui mène au code source de la classe".
> – l'imbrication de définitions devrait être évitée dans le cas général, sauf pour atteindre des vérifications de types spécifiques (voir chapitre 3.2.2). ■

Import et masquage de noms

Scala permet d'importer dans l'espace de nom courant n'importe quelles valeurs ou définitions. Toute valeur importée dans un espace de nom pour laquelle un identificateur de même nom existe déjà masque l'ancienne valeur. Cette fonctionnalité, combinée aux règles d'imbrications précédentes, peut mener à du code peu lisible.

Néanmoins, le compilateur détecte les références ambiguës à un nom dans les cas où un nom est importé plusieurs fois via des imports jocker. Le nom souhaité peut toujours être accédé en utilisant son nom qualifié. Le compilateur ne signale pas d'erreur d'ambiguïté si le nom est importé explicitement par son nom qualifié.

Intérêt 99 (I-S-99) : *Ambiguïté signalée par le compilateur lors de l'importation joker de modules.*
Le compilateur signale l'ambiguïté qui existe lors de l'utilisation d'un identificateur existant dans plusieurs espace de noms et importé suivant le même procédé dans l'espace courant (importation joker, par le nom qualifié, etc). ■

Danger 100 (D-S-100) : *Non ambiguïté de l'importation de noms explicite.*
Le compilateur ne signale pas d'ambiguïté lors de l'utilisation d'un identificateur existant dans plusieurs espace de noms si une importation exactement est faite en utilisant le nom qualifié. Dans ce cas, les autres valeurs sont masquées sans qu'un message d'information ne soit généré par le compilateur. ■

Danger 101 (D-S-101) : *Il n'existe pas d'option du compilateur qui permette de générer des messages pour l'ensemble des masquage de valeur.*

Recommandation 102 (R-S-102) : *Utilisation des noms qualifiés.*
Pour réduire l'ambiguïté que peut induire l'importation de noms, il est recommandé d'utiliser de préférence les noms qualifiés de valeurs dans le code source. ■

Recommandation 103 (R-S-103) : *Chemins complets d'importation.*

Il est conseillé d'indiquer les chemins complets d'importation de classes (e.g. `scala.collection.List`) et d'utiliser les noms qualifiés (e.g. `List.map`). ■

L'exemple suivant démontre différents cas de masquages et d'importations :

```
1  package P {
2    object X { val x = 1; val y = 2 }
3  }
4  package Q {
5    object X { val x = true; val y = "" }
6  }
7
8  package P {          // 'X' lie par la clause de package
9    import Console._  // 'println' lie par l'import joker
10   object A {
11     println("L4: "+X) // 'X' est 'P.X' ici
12     object B {
13       import Q._
14       //println("L7: "+X) // 'X' est ambigue ici
15       println("L7: "+Q.X) // 'X' est accessible par ↵
     son nom qualifie
16       import Q.X._        // 'x' et 'y' lie par ↵
     l'import jocker
17       println("L8: "+x) // 'x' est 'Q.X.x'
18       object C {
19         // 'x' lie par la definition locale
20         val x = 3
21         println("L12: "+x) // 'x' est la constante '3'
22         { import Q.X._
23        // la reference a 'x' est ambigue ici, car x ↵
     est defini dans C
24        // println("L14: "+x)
25           // 'y' lie par l'import explicite
26           import Q.X.y
27           // 'y' est 'Q.X.y'
28           println("L16: "+y)
29           // 'x' lie par la definition locale
30         { val x = "abc"
31           // 'x' et 'y' importe par l'import jocker
32           import P.X._
```

```
33          // la reference a 'y' est ambigue
34          // println("L19: "+y)
35             // 'x' est "abc" ici
36             println("L20: "+x)
37  }}}}}}
```

3.2.6 Exceptions

Scala possède des exceptions compatibles avec les exceptions Java.

Déclaration. En Scala, les exceptions n'entrent jamais dans la déclaration du type des méthodes. En reprenant le vocabulaire de Java, toutes les exceptions sont **non vérifiées** (`unchecked`). Ceci est vrai même pour les exceptions Java qui ne sont pas `unchecked`, comme le montre l'exemple suivant :

```
1  class NoEx {
2    def foo:Unit =
3      throw new Exception("no exception in functional ↵
     type")
4  }
```

Il est possible d'annoter une méthode avec l'annotation `@throws` pour signaler aux utilisateurs de la méthode qu'elle peut lancer une exception du type passé en paramètre de l'annotation. Cette annotation n'a pas de conséquence au niveau du typage de la méthode, mais elle indique au compilateur Scala de générer un type fonctionnel qui contient la clause `throws` dans le bytecode Java.

Le code :

```
1  class WithEx {
2    @throws(classOf[Exception])
3    def foo:Unit = throw new Exception("exception is ↵
     part of functional type")
4  }
```

génère le bytecode :

```
1  public class WithEx extends java.lang.Object ↵
     implements scala.ScalaObject{
2      public void foo() throws java.lang.Exception;
3      public WithEx();
4  }
```

alors que l'exemple précédent génère le bytecode :

```
1 public class NoEx extends java.lang.Object implements ↵
     scala.ScalaObject{
2     public void foo();
3     public NoEx();
4 }
```

Danger 104 (D-S-104) : *Contrairement à Java, les exceptions Scala ne sont pas vérifiées.*
L'intérêt des exceptions vérifiées reste un sujet de controverse, et dans les années récentes, l'opinion générale semble se détourner de celles-ci. Néanmoins, cette différence d'approche entre Scala et le langage Java qui reste le langage de référence de la plateforme peut être considérée comme un risque dans le cadre de projets polyglottes. Les risques directs pour la sécurité restent les mêmes que ceux pour les exceptions (vérifiées ou non) de Java. Il est en particulier recommandé de filtrer les informations contenues dans les traces d'exception (cf. [JavaSec recommandations, 2009, fiche 11]). ■

Recommandation 105 (R-S-105) : *Utiliser l'annotation* `@throws` *afin de documenter les exceptions possibles.*
Il est conseillé d'utiliser l'annotation `@throws` pour toute méthode qui peut lever une exception qui serait une exception vérifiée en Java. ■

Rattrapage des exceptions. Le rattrapage des exceptions se fait à grâce à un bloc `try/catch` couplé au **filtrage** (voir section 3.2.3).

```
1   try {
2    // ...
3    } catch {
4      //exception la plus precise en premier
5      case ioe: IOException => ...
6      case e: Exception => ...
7    }
```

Attention, si le type d'une exception n'est pas précisé dans le filtrage, toutes les exceptions seront rattrapées par le cas `e`.

```
1   //code dangereux
2   try {
3    // ...
4    } catch {
5      //rattrape Throwable, donc ThreadDeath, et d'autres
6      case e => ...
7    }
```

Danger 106 (D-S-106) : *Si le type d'une exception rattrapée n'est pas précisé, toutes les exceptions sont rattrapées.*
Ceci revient à rattraper `Throwable`, ce qui peut entraîner de nombreux bogues et failles : mauvaise gestion de interruption de *thread*, terminaison non correcte de la JVM en cas d'erreur système, etc.■

De plus, en Scala, plusieurs structures de contrôle de flot de données sont implémentées grâce à des exceptions. C'est en particulier le cas de `return` (voir section 3.2.7) et de `break`, comme le montre l'exemple ci-dessous :

```
1  breakable {
2    for (i <- 0 to 2) {
3      try {
4        println(i)
5        if ( i > 0 ) break
6      } catch {
7        case e => println("break caught")
8      }
9    }
10 }
11 0
12 1
13 break caught
14 2
15 break caught
```

Recommandation 107 (R-S-107) : *La borne supérieure du type des exceptions rattrapées doit toujours être précisée.*
Toujours préciser la borne supérieure du type de l'exception à rattraper, sans quoi des exceptions système ou de contrôle de flot peuvent être rattrapées sans le vouloir. ■

3.2.7 Retour non local

En Scala, le mot-clé `return` n'a pas la signification qui lui est généralement attribuée dans les langages à la *C*, et donc en Java (voir [Odersky, 2010, section 6.20]).

le `return` est un retour non local : il doit être utilisé à l'intérieur d'une méthode ou d'une fonction *nommée* (ou d'une fonction anonyme utilisée dans cette fonction), et sort de la fonction nommée.

S'il est utilisé, l'inférence de type n'est pas disponible pour la valeur de retour de la fonction.

Il est implémenté au moyen d'une exception. Cette exception est rattrapée automatiquement au niveau de la fonction nommée qui déclare le `return`, sauf si cette fonction est déjà terminée lorsque le code qui déclenche le `return` s'exécute. Dans ce dernier cas, l'exception n'est pas rattrapée et se propage dans la pile d'appels, comme le montre l'exemple suivant :

```
1  class Lazy(body : => Int) {
2    lazy val value = body
3  }
4  defined class Lazy
5
6  def miscreateLazy() : Lazy = {
7    new Lazy( return new Lazy(1) )
8  }
9  miscreateLazy: ()Lazy
10
11 val lz = miscreateLazy()
12 lz: Lazy = Lazy@54cbb9
13
14 lz.value
15 scala.runtime.NonLocalReturnControl
```

Danger 108 (D-S-108) : *En Scala,* `return` *est implémenté avec une exception.*
`return` est implémenté avec une exception : il peut être rattrapé dans un `catch` qui ne filtre pas correctement sur le type des exceptions. ■

Danger 109 (D-S-109) : *L'exception lancée par return peut ne pas être rattrapée.*

Un `return` peut produire une exception non rattrapée à l'exécution. ■

Danger 110 (D-S-110) : *Le return Scala n'est pas local ce qui complexifie la compréhension du flot de branchement.*
`return` peut produire un flot de branchement qui n'était pas celui attendu lors de l'utilisation de fermeture. ■

Recommandation 111 (R-S-111) : *Ne jamais utiliser return.*
Ne pas utiliser le mot-clé `return`, à moins d'être sûr de la sémantique du code produit, qui n'est en général pas celle attendue. En particulier, `return` ne doit pas être utilisé pour sortir tôt d'une suite de tests conditionnels. ■

Bibliographie

[Altherr et Cremet, 2005] Altherr, P. et Cremet, V. (2005). Inner classes and virtual types. Rapport technique IC/2005/013, EPFL.

[Bloch, 2008] Bloch, J. (2008). *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 édition.

[Bracha *et al.*, 1998] Bracha, G., Odersky, M., Stoutamire, D. et Wadler, P. (1998). Making the future safe for the past : Adding genericity to the java programming language. *In Proc. OPPSLA'98*.

[Chafi *et al.*, 2010] Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A., Hanrahan, P., Odersky, M. et Olukotun, K. (2010). Language Virtualization for Heterogeneous Parallel Computing. Rapport technique, EPFL.

[Cremet, 2006] Cremet, V. (2006). *Foundations for Scala : Semantics and Proof of Virtual Types*. Thèse de doctorat, EPFL. No. 3556, supervised by Martin Odersky.

[Cremet *et al.*, 2006] Cremet, V., Garillot, F., Lenglet, S. et Odersky, M. (2006). A core calculus for scala type checking. *In* [Kralovic et Urzyczyn, 2006], pages 1–23.

[Dragos et Odersky, 2009] Dragos, I. et Odersky, M. (2009). Compiling generics through user-directed type specialization. *In Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 42–47, New York, NY, USA. ACM.

[Dubochet, 2010] Dubochet, G. (2010). Storage of pickled scala signatures in class files. Rapport technique, EPFL.

[Dubochet, 2011] Dubochet, G. (2011). *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. Thèse de doctorat, EPFL, Lausanne.

[Emir, 2007] Emir, B. (2007). *Object-oriented pattern matching*. Thèse de doctorat, EPFL, Lausanne.

[Emir *et al.*, 2007] Emir, B., Ma, Q. et Odersky, M. (2007). Translation correctness for first-order object-oriented pattern matching. *In Proceedings of the 5th Asian conference on Programming languages and systems*, APLAS'07, pages 54–70, Berlin, Heidelberg. Springer-Verlag.

[Emir *et al.*, 2006] EMIR, B., ODERSKY, M. et WILLIAMS, J. (2006). Matching Objects with Patterns. Rapport technique, EPFL.

[Haller et Odersky, 2008] HALLER, P. et ODERSKY, M. (2008). Scala actors : Unifying thread-based and event-based programming. *Theoretical Computer Science*.

[JavaSec exécution, 2009] Rapport d'étude sur les modèles d'exécution de Java. Rapport d'étude JavaSec JAVASEC_NTE_003, version : 1.2, ANSSI (2009). Étude menée par un consortium composé de SILICOM, AMOSSYS et de l'INRIA, dans le cadre formel d'un marché du SGDSN. Disponible en ligne à l'adresse `http://www.ssi.gouv.fr/IMG/pdf/JavaSec-Execution.pdf`.

[JavaSec langage, 2009] Rapport d'étude sur le langage Java. Rapport d'étude JavaSec JAVASEC_NTE_001, version : 1.3, ANSSI (2009). Étude menée par un consortium composé de SILICOM, AMOSSYS et de l'INRIA, dans le cadre formel d'un marché du SGDSN. Disponible en ligne à l'adresse `http://www.ssi.gouv.fr/IMG/pdf/JavaSec-Langage.pdf`.

[JavaSec recommandations, 2009] Guide de règles et de recommandations relatives au développement d'applications de sécurité en Java. Rapport d'étude JavaSec JAVASEC_NTE_002, version : 1.3, ANSSI (2009). Étude menée par un consortium composé de SILICOM, AMOSSYS et de l'INRIA, dans le cadre formel d'un marché du SGDSN. Disponible en ligne à l'adresse `http://www.ssi.gouv.fr/IMG/pdf/JavaSec-Recommandations.pdf`.

[Kralovic et Urzyczyn, 2006] KRALOVIC, R. et URZYCZYN, P., éditeurs (2006). *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 de *Lecture Notes in Computer Science*. Springer.

[Langer, 2010] LANGER, A. (2010). Java generics FAQs.

[Moors, 2009] MOORS, A. (2009). *Type Constructor Polymorphism for Scala : Theory and Practice*. Thèse de doctorat, Informatics Section, Department of Computer Science, Faculty of Engineering. Joosen, Wouter and Piessens, Frank (supervisors).

[Moors *et al.*, 2008] MOORS, A., PIESSENS, F. et ODERSKY, M. (2008). Generics of a higher kind. *In Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 423–438, New York, NY, USA. ACM.

[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.*, 2003] ODERSKY, M., CREMET, V., RÖCKL, C. et ZENGER, M. (2003). A nominal theory of objects with dependent types. *In Proc. ECOOP'03*, Springer LNCS.

[Odersky et Moors, 2009] ODERSKY, M. et MOORS, A. (2009). Fighting bit rot with types (experience report : Scala collections). *In* KANNAN, R. et KUMAR, K. N., éditeurs : *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 de *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Odersky *et al.*, 2000] ODERSKY, M., RUNNE, E. et WADLER, P. (2000). Two ways to bake your pizza – translating parameterised types into java. *In Generic Programming – International Seminar on Generic Programming Dagstuhl Castle*, volume 1766 de *Springer Lecture Notes in Computer Science*, pages 114–133.

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

[Odersky et Zenger, 2005] ODERSKY, M. et ZENGER, M. (2005). Scalable Component Abstractions. *In Proceedings of OOPSLA 2005*.

[Oliveira *et al.*, 2010] OLIVEIRA, B. C. d. S., MOORS, A. et ODERSKY, M. (2010). Type Classes as Objects and Implicits. *In Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 341–360. ACM.

[Prokopec *et al.*, 2010] PROKOPEC, A., ROMPF, T., BAGWELL, P. et ODERSKY, M. (2010). A Generic Parallel Collection Framework. Rapport technique, EPFL.

[Scala Team, 2011] Scala Team (2011). Domain-Optimised Parallelisation by Polymorphic Language Embeddings and Rewritings.

[Spiewak, 2009] SPIEWAK, D. (2009). Interop between java and scala.

[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/`.

Table des intérêts

OCaml

F#

Scala

Table des dangers

OCaml

F#

Scala

Table des recommandations

OCaml

F#

Scala

Acronymes

ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information
AOT	Compilation Ahead-Of-Time
API	Application Programming Interface
AST	Arbre de Syntaxe Abstraite
ECMA	European Computer Manufacturers Association
EPFL	École Polytechnique Fédérale de Lausanne
GJ	Generic Java
GUI	Graphical User Interface
IDE	Integrated Development Environment
INRIA	Institut National de Recherche en Informatique et en Automatique
JAR	Java ARchive
JIT	Compilation Just-In-Time
JSE	Java Standard Edition
JVM	Java Virtual Machine
LAMP	Laboratoire de Méthodes de Programmation de l'EPFL
.NET APM	Asynchronous Programming Model
.NET BCL	Base Class Library de .NET
.NET CAS	Code Access Security 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 GAC	Global Assembly Cache de .NET

.NET NGEN	Native Image Generator
REPL	Read-Eval-Print Loop
SGDSN	Secrétariat Général de la Défense et de la Sécurité Nationale
SML	Standard ML
STM	Software Transactionnal Memory
ZAM	ZINC Abstract Machine (machine virtuelle de OCaml)

Index