MÉTHODES STATIQUES POUR LA PROGRAMMATION FONCTIONNELLE DE SIMULATEURS D'ÉCONOMIES

Static methods for the functional programming of economics simulators

ANTOINE KASZCZYC

CHRISTOPHE FOUQUERÉ, DIRECTEUR
PIERRE BOUDES, ENCADRANT

MARCO PEDICINI, RAPPORTEUR
DANIELE VARACCA, RAPPORTEUR
SYLVIE BOLDO, EXAMINATRICE

18 DÉCEMBRE 2019
LABORATOIRE D'INFORMATIQUE DE PARIS NORD
ÉCOLE DOCTORALE GALILÉE
UNIVERSITÉ SORBONNE PARIS NORD

Table des matières

Résumé de la thèse

La première partie de la thèse a pour sujet les effets de bord dans les programmes fonctionnels, et leur simulation par les monades. Les propositions ont une portée générale : elles peuvent s'appliquer peu importe le thème du programme. Cependant, elles ont été inspirées par un contexte de programmation de simulateur d'économies. Ce type de programme utilise intensément les effets de bord et les répétitions de fonctions. Les propositions de la thèse sont plus pertinentes pour les programmes qui présentent ces caractéristiques. Comme bien d'autres, elles sont pensées dans un objectif d'organisation stricte du programme. Elles ne sont pas adaptées pour réaliser un programme "vite fait bien fait", mais plutôt un programme dans lequel des parties critiques ont été sécurisées et n'ont plus besoin d'attention.

Le chapitre 1 introduit la programmation fonctionnelle, et des schémas d'exécution reflétant la consommation mémoire lors de l'exécution. Il introduit les monades, et les transformateurs de monades, en tant que mécanismes permettant d'automatiser la simulation des aspects mutables dans un contexte immutable.

Le chapitre 2 aborde la gestion des effets de bord : la simulation d'une variable mutable par la monade $State$, et l'encadrement de sa valeur par un prédicat. Le chapitre commence par définir précisément ce que signifie le respect du prédicat, notamment par la solution bien connue du type abstrait. Ensuite, nous faisons observer que le prédicat est plus expressif lorsqu'il concerne un type fonctionnel $(A \rightarrow A)$ plutôt que simplement A. La monade $State$ fournit un support adéquat pour implanter le prédicat dans le programme, puisqu'elle exprime une variable mutable par des compositions de fonctions. Dans une simulation, la monade $State$ abstraite avec prédicat permet d'exprimer des mécanismes auxiliaires mais ubiquitaires, tels la génération de nombre pseudo-aléatoire, ou la gestion de comptes bancaires. Ce sont deux concepts, la simulation d'effets de bord, et la garantie de prédicat, qui se résolvent par une même structure : la monade $State$ abstraite.

Le chapitre 3 concerne un autre aspect de la sécurité du programme : la consommation mémoire. Celle-ci est bornée par les composants physiques de l'ordinateur qui exécute le programme. Si l'exécution du programme nécessite trop d'espace mémoire, elle est simplement arrêtée. Le chapitre présente les deux types d'erreurs mémoire rencontrés en programmation fonctionnelle : $Stack\_Overflow$ et Out_Of_Memory. L'erreur $Stack\_Overflow$ est causée par les fonctions récursives. Le remède est l'optimisation de l'appel terminal. Cependant il est plus difficilement applicable en présence de monades, qui remplacent l'application standard par l'application $bind$, sur laquelle le programmeur a moins de contrôle. L'exemple emblématique est la monade $State$, qui crée des compositions de fonctions dont l'évaluation n'est pas optimisée. De plus, les compositions de fonctions doivent être stockées et risquent de générer une erreur Out_Of_Memory.
Le chapitre présente ensuite les solutions existantes, dont la meilleure définit une fonction de récursion spécialisée à chaque monade. Les solutions existantes ne résolvent pas la situation où la récursion n'est pas "contrôlée", c'est à dire où le programmeur fournit seulement la fonction à répéter. Dans ce contexte, les solutions provoquent un risque d'erreur Out_Of_Memory, car elles stockent des opérations en suspension. Nous proposons comme solution le principe des "concrets monadiques". Il s'agit de créer un contexte à l'intérieur duquel toutes les informations sont connues, spécifiquement la valeur auxiliaire de la monade $State$. Il est alors possible d'effectuer des récursions non contrôlées sans suspendre des opérations. Le contexte se charge de l'attente de la valeur auxiliaire initiale.
Le chapitre termine en présentant le contexte du $Read\ Eval\ Print\ Loop$ $(REPL)$. Il consiste à extraire (lire) la valeur monadique entre chaque $bind$. Cela pose problème avec les solutions existantes, car il faut

à chaque fois recalculer les suspensions. Les concrets monadiques se montrent alors adéquats, puisqu'ils n'utilisent pas les suspensions.

Le chapitre 4 réunit un ensemble d'observations résultantes de l'écriture d'un petit simulateur fonctionnel. Les observations concernent la bibliothèque de monades réalisée à partir des chapitres précédents, la difficulté du mécanisme de *lift*, l'utilisation de mutabilité dans des endroits isolés et critiques. Le chapitre termine par une tentative de comparaison entre programmation avec variables mutables versus variables immutables.

La deuxième partie de la thèse et son chapitre 5 a pour sujet les systèmes d'acteurs et leur typage, qui sont des systèmes prévus pour exprimer des programmes concurrents. Ils sont envisagés pour réaliser des simulateurs basés agents, de part les similudes entre agents et acteurs. Nous présentons *Akka*, une bibliothèque implémentant ce modèle, et nous voyons que dans sa version basique elle n'applique pas de vérification sur les types des messages transmis, ce qui peut mener le programme vers un état non prévu. Nous exprimons alors les acteurs dans un langage simplifié, auquel nous montrons comment ajouter du typage simple (à la manière du lambda calcul simplement typé). En fin de chapitre, nous présentons quelques exemples illustrant pourquoi ce typage doit être augmenté pour retrouver l'expressivité initiale.

La deuxième partie se conclut par le chapitre 6 qui rend compte d'un travail effectué dans la réalisation d'une plateforme de lancement de simulations reposant sur le modèle acteurs. La plateforme permet de lancer plusieurs simulations de façon simultanée, potentiellement sur différentes machines, et possède une interface client affichant les résultats au fur et à mesure.

Mots-clés

— fonctionnel
— monades
— simulateurs
— mémoire

Abstract

The first part of the phd concerns side effects in functional programs, and their simulation by monads. The proposed solutions have a general goal : they are useful no matter the subject of the program. However, they have been inspired by the context of programming economics simulators. Indeed this kind of program use intensively side effects and repetition of functions calls. The proposed solutions are more useful for program with these characteristics. Like many others, they are suited for a strict organization of the program. They are not meant to build a "quick and well done" program, rather a program whose critical parts have been made safe and does not need any more attention.

The chapter 1 introduces functional programming, with execution schemes representing memory consumption. It introduce monads, and monads transformers, as mechanisms allowing to automatize the simulation of mutable aspect in an immutable context.

Chapter 2 address handling of side effects : the simulation of a mutable variable by the $State$ monad, the framing of its value by a predicate. The chapter starts by defining precisely what means the respect of a predicate, in particular by the well known solution of abstract type. Then, we observe that the predicate is more expressive when it concerns a functional type $(A \rightarrow A)$ rather than simply A. The $State$ monad gives a adequate structure to implant the predicate in the program, since it simulates a mutable variable by function composition. In an economics simulator, the abstract $State$ monad with a predicate can express similar but ibiquitous mechanisms, as pseudo-random generation, or handling of bank accounts. These are two concepts, simulation of side effects and respect of a predicate, that are resolved by the abstract $State$ monad.

Chapter 3 address another aspect of program safety : memory consumption. It is boundered by physical components of the computer. If the execution of the program necessits too much memory space, the program is simply stopped. The chapter presents the two type of memory error in functional programing : $Stack\_Overflow$ and Out_Of_Memory. The error $Stack\_Overflow$ is caused by recursive functions. The solution is tail-call optimization. However it is more difficult to apply in the presence of monads, which replace the standard application by application $bind$, on which the oprogrammer has less control. The emblematic example is the $State$ monad, which creates composition of functions non-optimized by tail-call. Moreover, composition of functions has to be stored and can generate an error Out_Of_Memory.
The chapter then introduces the existing solutions, and the best one defines a function of recursion specialized for each monad. However none of the solution resolve the case where the recursion is "not controlled", ie when the programmer just gives a function to be repeted. In this context, the existing solutions present a risk of Out_Of_Memory, because they store functions in suspension. We present our solution : "monadic concretes". The idea is to create a context inside of which all the informations are known, for example the auxiliary variable of $State$. It is then possible to perform recursive uncontrolled recursion without suspending operations. The context is responsible for waiting the informations.
The chapter ends by presenting the context of $Read\ Eval\ Print\ Loop$ $(REPL)$. It consists in extracting (reading) the monadic value between every $bind$. This present a problem with existing solutions, since every tome they have to recompute the suspensions. Monadic concretes then shows themselves adequate, since they do not use suspensions.

Chapter 4 gather a set of observations resulting from the writing of a small functional economis simulator. They concerns the monads library realized from the previous chapter, the problem of *lift*, the use of muta-

bility in isolated and time critical places. The chapter ends by an attempt of comparing programing with mutability versus immutability.

The second part of the phd and its chapter 5 address the actors systems and their typing. Theses are systems meant to express concurrent programs. They are envisaged to realize agent based simulators, by the similarities between agents and actors. We present *Akka*, an existing library for the actors model, and we observe that in its basic version, it does not type the communication message between actors. This can lead the system into an unplanned state. We then express the actors systems in a simple language, and we show how to add simple typing (à la simply typed lambda calculus). The chapter ends by presenting some examples showing why this typing must be augmented to restore the inital expressivity.

The second part concludes with chapter 6 who accounts the work done in the realization of a porgram of simulations launching, based on actors model. The program permits to launch several simulations concurrently, potentially on different computers, and present a GUI displaying the results in real time.

Keywords

— functional
— monads
— simulators
— memory

Introduction de la thèse

Le point de départ de ce travail est l'écriture de simulateurs agents d'économies par la programmation fonctionnelle. Le contexte d'application est plus large et couvre la programmation fonctionnelle avec monades.
Les simulateurs agents sont des programmes qui font jouer et interagir un ensemble d'agents. Leur but est d'obtenir des scénarios complexes endogènes à la multiplicité des agents et des interactions. Ils sont par exemple utilisés en économie, pour exprimer des systèmes dans lesquels l'argent ne disparaît pas entre chaque période. Citons Pascal Seppecher, créateur du simulateur d'économies *Jamel* [Sep14] :

> La théorie monétaire de la production, parce qu'elle s'attache à explorer les mécanismes du flux et du reflux de la monnaie dans le système économique, constitue un point d'appui essentiel pour une approche de la macroéconomie basée sur les interactions entre les agents. Cependant cette théorie s'appuie sur une représentation abstraite du temps et de l'enchaînement des interactions individuelles. Elle débouche sur des modèles qui ne rendent pas compte de façon satisfaisante de catégories aussi essentielles que le profit et l'intérêt. Les techniques de modélisation multi-agents permettent de s'affranchir des limites des modèles analytiques traditionnels. Nous décrivons la construction d'un modèle d'économie monétaire réellement dynamique, peuplé d'agents multiples, autonomes et hétérogènes en interaction directe et décentralisée, tout en respectant rigoureusement la cohérence des stocks et des flux. Par la simulation, nous observons l'émergence de dynamiques macroéconomiques complexes qui montrent que ce modèle, en donnant un contenu concret aux principes essentiels de la théorie monétaire de la production, permet d'en dépasser les contradictions.

Nous abordons les simulateurs agents d'économies de l'angle de leur programmation. Pascal Seppecher utilise le langage *Java* pour écrire Jamel, dont le paradigme est la *programmation orientée objet*. Il s'agit de définir des *classes*, ensembles d'*attributs* et *méthodes*. Une classe est utilisée pour créer des *objets* appartenant à celle-ci. Pour un simulateur d'économies, nous pouvons par exemple définir une classe *Ménage* et une classe *Entreprise*.
Notons que la citation insiste sur le respect de la "cohérence des stocks et des flux". Il s'agit d'une propriété que doivent respecter les comptes bancaires de la simulation d'économies, et qui peut notamment être cassée lors de transferts bancaires. La programmation objet permet d'enforcer une propriété grâce au contrôle qu'un objet a sur ses attributs. Nous pouvons aussi utiliser la méthode "assertive" qui consiste à appeler régulièrement une procédure qui vérifie le respect de la propriété et arrête le programme en cas d'échec.
Nous prenons un autre parti : suivre le paradigme *fonctionnel* dans l'écriture du simulateur agents d'économies. Ce paradigme dispose de caractéristiques propres :

- les structures de données sont immutables, en conséquence aucune procédure ne peut altérer les données utilisées par une autre procédure. Ceci renforce la confiance dans la correction : nous savons que nos données courantes ne peuvent être corrompues par l'appel à une fonction, quelle qu'elle soit.
- la structure de base est la *fonction*, qui est un "élement de première classe". Manipuler des fonctions permet de manipuler des morceaux de code source. L'immutabilité des données simplifie la manipulation des fonctions, ce qui permet de manipuler les codes sources, par exemple pour montrer que deux codes sont équivalents.

En suivant le paradigme fonctionnel, nous espérons écrire sans grande difficulté le simulateur d'économies, et contrôler plus facilement la propriété de cohérence des stocks et des flux. En ce sens, nous avons présenté à la conférence *Artificial Economics* l'article *Monetary Economics Simulation : Stock-Flow Consistent Invariance, Monadic Style* [BKP15].

La caractéristique principale du paradigme fonctionnel à l'écriture du simulateur est l'utilisation intensive de la méthode des monades. L'absence de structures mutables empêche le mécanisme de clôtures mutables, qui permet aux procédures d'avoir un effet qui dépasse le contexte d'appel. C'est par exemple utile pour implémenter un compteur.
En fonctionnel, le seul effet d'une fonction est sa valeur de retour, qui doit donc être utilisée pour exprimer tous les effets. Il s'ensuit un encombrement des valeurs d'entrée et de sortie des fonctions. Dans ce cas nous utilisons une monade afin de nettoyer les entrées/sorties en en rendant certaines implicites, et en automatisant leur transmission. Ce mécanisme est appelé *monade State*.
Nous profitons du caractère centralisateur de la monade *State* pour y associer la vérification de la cohérence des stocks et des flux (chapitre 2).

Le simulateur d'économies est une itération d'une procédure appelée période. Une période est elle-même constituée de phases. Par exemple, lors de la phase des salaires chaque ménage est le destinataire d'un virement sur son compte bancaire. Ainsi le programme du simulateur contient un grand nombre de petites actions répétées qui dépasse aisément le million. Remarquons que nous pouvons en plus "calibrer" le simulateur, c'est à dire le lancer plusieurs fois en modifiant légèrement les paramètres de départ.
Les ordinateurs sont faits pour répéter, et bien que nous leur demandions toujours plus, un programme de simulateur ne devrait pas poser de problème. Dans le pire des cas, il prend du temps à exécuter, en fonction du nombre d'agents, de périodes, et des performances de la machine utilisée. Cependant, bien que les monades soient définis comme une structure "utilisateur", elles sont particulières car elles sont présentes à chaque étape du programme, c'est à dire à chaque application de fonction. Elles ont un impact fort sur l'exécution du programme : elles peuvent le retarder pour attendre une entrée supplémentaire (monade *State*), multiplier le nombre d'exécutions (monade *List*), etc. Utiliser des monades dans un langage revient à modifier le langage initial en lui ajoutant des nouvelles stuctures de contrôle. En témoigne le mécanisme de *run* dans *Freer Monads, More Extensible Effects* d'Oleg Kiselyov et Hiromi Ishii [KI15], qui consiste à écrire un interpréteur dans le programme même.
Non seulement une monade a un fort impact sur l'exécution et sur la valeur représentée, mais elle a tendance à être rendue abstraite. La raison est que toutes les monades partagent une interface commune, ainsi il est tentant de rendre un code polymorphique, c'est à dire applicable à différentes monades.
Ces deux caractéristiques ont pour conséquence de cacher des performances en temps et en espace qui peuvent se révéler problématiques, notamment dans le cas des simulateurs d'économies, où il y a un grand nombre de petites actions. C'est pourquoi nous consacrons le chapitre 3 à la résolution des problèmes de temps et d'espace, puisqu'ils sont indispensables à l'utilisation pratique du programme de simulateur. Le problème principal est celui de *Stack Overflow*, où trop d'appels de fonctions imbriqués sont effectués. Le deuxième problème survient dans des solutions existantes au premier, où l'espace de la *pile* a été échangé par l'espace mémoire de travail, lui non plus n'est pas infini (erreur *Out_Of_Memory*). Des tests d'exécution ont été réalisés pour plusieurs monades, voir en annexe C. Nous préconisons une solution existante mais peu répandue dans les bibliothèques de monades *OCaml*, qui consiste à définir des fonctions de répétitions optimisées, répétant des fonctions de type `'a -> ('a monad)`.
Nous observons que ces fonctions nécessitent d'avoir le contrôle de la récursion, ce qui n'est pas toujours le cas. Aussi, selon la monade utilisée, elles ne permettent pas d'obtenir une information complète sur la valeur monadique à chaque étape. Nous proposons alors une structure nouvelle : les concrets monadiques. Ces derniers s'appliquent à éviter d'accumuler les suspensions de fonctions, en appliquant la suspension à un bloc complet, à l'intérieur duquel les opérations sont efficaces. Les concrets monadiques constituent la contribution première de cette thèse.
En fin de partie, nous rassemblons en chapitre 4 des informations sur la réalisation d'un simulateur jouet d'économies en langage fonctionnel *OCaml*, notamment sur la bibliothèque de monades contenant les deux chapitres précédents.

Dans la première partie du document, nous nous concentrons sur l'exécution d'un simulateur fonctionnel par un programme sur une machine. Dans la seconde partie, nous abordons le paradigme des *programmes*

concurrents. Cela consiste à transformer le programme en un ensemble de programmes, chacun étant exécuté sur une machine propre, et communicant avec les autres par le réseau. Cela peut augmenter la vitesse d'exécution, lorsqu'il est possible d'effectuer certaines actions en parallèle. Cela peut aussi être rendu indispensable par une taille trop large de simulation, qui ne tiendrait pas dans la mémoire de travail d'une seule machine.
Les difficultés de programmation dans ce paradigme sont bien connues. Nous conservons en un sens notre point de vue fonctionnel en utilisant le *modèle acteurs*. Nous y retrouvons le principe de réduire les possibilités d'exécution et l'expressivité des procédures. Un acteur est un *thread* avec un fonctionnement défini : il possède un buffer de stockage de messages entrants, et une zone mémoire privée. Il traite un seul message entrant à la fois, et ne donne jamais aux autres acteurs un accès direct à sa zone mémoire. La communication se fait intégralement par transmission désynchronisée de message. C'est pourquoi le modèle offre une grande liberté de placement des acteurs sur les machines.
Notre travail consiste à renforcer la sûreté des acteurs en complétant le système de vérification des types, qui était très souple sur les messages entre acteurs. Le chapitre 5 présente un langage de lambda calcul standard en y ajoutant des primitives du modèle acteur, et montre comment définir un système de types qui vérifie aussi les envois de message. Nous avons aussi travaillé sur l'élaboration d'une plateforme de lancement paramétré de simulations, voir le chapitre 6.

Première partie

Simulation avec monades

Chapitre 1

Introduction à la programmation fonctionnelle avec monades

Dans ce chapitre introductif à la partie sur les monades, nous présentons le langage de programmation et ses caractéristiques sur lesquels reposent les problèmes observés et leurs solutions. Nous introduisons le contexte qui a permis ces observations : la programmation de simulateur d'économies.

Le paradigme de programmation utilisé dans cette partie est la programmation fonctionnelle, en particulier en *appel par valeur*, mais il y aura aussi quelques remarques concernant l'appel *par nécessité*. Nous allons présenter des schémas d'exécutions qui témoignent de l'état de la mémoire à chaque étape de l'exécution d'un programme, de façon simplifiée. Nous introduirons le mécanisme de pile d'appel, qui sert à retenir des informations du contexte lorsqu'un appel à une fonction commence à s'évaluer. Cela nous permettra de présenter l'erreur *Stack_Overflow*, qui survient lorsque la pile est trop grande pour l'espace mémoire disponible. Nous expliquerons alors l'optimisation dite *d'appel terminal*, qui permet d'utiliser un espace constant de pile dans les récursions bien écrites.

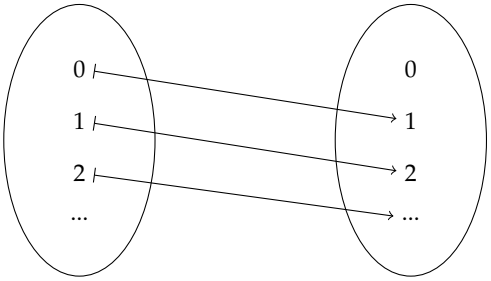
Ensuite, nous présenterons la technique la plus répandue actuellement pour simuler la mutation dans un programme fonctionnel : les monades. Nous les étudierons du point de vue du programmeur. Nous comparerons les différentes versions d'un même programme : sans et avec monade, avec monade et du sucre syntaxique. Nous étudierons la modularité des monades, en présentant les transformateurs de monades.

Enfin, nous aborderons la programmation de simulateur d'économies avec un langage fonctionnel. Notons ici qu'en général les langages utilisés pour programmer des simulateurs utilisent directement mutation, c'est le cas par exemple avec *Java*.

1.a Introduction à la programmation fonctionnelle

La programmation "fonctionnelle" fait référence à la "fonction" : "activité qui tend à un but déterminé"[1]. Il s'agit de voir un programme comme une fonction mathématique, une relation entre un domaine et un co-domaine. Voici un schéma du programme (ou de la fonction) `(fun x -> x + 1)` avec les entiers naturels comme domaine et co-domaine :

1. Wiktionnaire `https://fr.wiktionary.org/wiki/fonction`



domaine : $\mathbb{N}$ co-domaine : $\mathbb{N}$

Une fonction a une "entrée" : elle attend un argument (`x`) dans le type du domaine. Elle a une "sortie" : elle donne une valeur de retour (`x + 1`) dans le type du co-domaine. La fonction totale doit couvrir l'ensemble du domaine, mais pas nécessairement couvrir le co-domaine.

La programmation fonctionnelle est fondée sur l'utilisation intensive de fonctions. Outre l'avantage inhérent au découpage d'un problème principal en sous-problèmes, cela vise aussi à augmenter la réutilisation de code source dans le programme. Une fonction, étant relativement indépendante de son contexte, et présentant une interface très simple (une entrée, une sortie), se prête bien à la réutilisation.

Nous allons maintenant développer les caractéristiques des fonctions dans un langage similaire au langage fonctionnel OCaml. Dans notre langage, nous pouvons déclarer et associer une expression à un nom, par exemple nous associons au nom `v` l'entier `10` , et au nom `f` la fonction `(fun x -> x + 1)` :

```
let v = 10
let f = (fun x -> x + 1)
```

Le programme est exécuté de haut en bas, chaque nom est évalué en gardant en mémoire les noms précédents. Dans l'exemple suivant `b` vaut `7` et la ligne `3` provoque une erreur d'exécution car `d` n'existe pas encore :

```
1 let a = 4
2 let b = a + 3
3 let c = d + 2
4 let d = 5
```

Nous utiliserons parfois le mot "variable" pour désigner un nom. Ce n'est pas anodin car dans les langages non fonctionnels, il est possible de remplacer l'expression déjà associée à un nom par une autre, ce qu'on appelle la *mutabilité* :

```
let a = 4
a := 5 ;
```

Dans les langages fonctionnels, la mutation est interdite ou encadrée. Les noms "standard" sont immutables, leurs expressions ne peuvent être modifiées. Cependant, il est possible de "surcharger" un nom, cela consiste à déclarer un nom déjà déclaré. Dans l'exemple suivant `x` vaut `20` et y vaut `30` à la fin du calcul :

```
let a = 2
let x = a * 10
let a = 3
let y = a * 10
```

Une fois données ces précisions sur les variables, nous pouvons aborder l'application d'une fonction. Cela signifie fournir une entrée et lancer l'exécution du code de la fonction pour obtenir la sortie associée. L'application est une expression que nous pouvons associer à un nom :

```
let f = (fun x -> x + 1)
let r = (f 5)
```

L'exécution du programme se déroule toujours de haut en bas, mais notons que lors de l'évaluation de `f`, l'expression `(fun x -> x + 1)` est laissée telle quelle : il s'agit en effet d'une fonction, qui associe à chaque entrée une sortie. Une fonction est donc une façon de retarder l'évaluation d'un morceau de code, en attendant qu'il reçoive la valeur de son entrée.
L'exécution du programme avance à l'évaluation de `r` qui contient l'application de `f` à `5`. Le code de la fonction `f` est alors chargé, en associant le nom `x` à `5`. Une fois ce code évalué à `6`, l'exécution reprend là où elle en était au moment de l'appel. Ainsi, le nom `r` reçoit l'entier `6`.

Nous utiliserons parfois les mots "appel d'une fonction" pour désigner l'application d'une fonction.

Une fonction peut faire intervenir d'autres noms que celui de son argument :

```
let v = 10
let f = (fun x -> x + v)
let r = (f 5)
```

La variable `v` est dite "libre" dans la fonction `f`, parce qu'elle n'est pas "liée" à un argument, contrairement à la variable `x`. L'ensemble des variables libres de la fonction `f` est appelé "la clôture de `f`".

Dans l'exemple précédent `v` vaut `10`, ainsi `r` vaut `15`. Mais que se passe-t-il dans l'exemple suivant ?

```
1 let v = 10
2 let f = (fun x -> x + v)
3 let v = 20
4 let r = (f 5)
```

Est-ce que `r` vaut `15`, ou bien `25` ?
Si `r` vaut `15`, nous sommes en présence de "liaison statique" : la définition de `v` dans la fonction `f` est pour toujours celle en vigueur au moment de la définition de `f`, ligne `2`.
Si `r` vaut `25`, nous sommes au contraire en présence de "liaison dynamique" : la valeur de `v` est celle en vigueur au moment de l'appel de la fonction `f`, ligne `4`.
Nous choisirons la liaison statique pour notre langage puisque c'est le choix le plus répandu aujourd'hui.

Remarquons qu'avec des noms mutables et de la liaison statique, `r` vaut `25` dans l'exemple suivant :

```
let v = 10
let f = (fun x -> x + v)
v := 20 ;
let r = (f 5)
```

La présence des deux propriétés de liaison statique et d'immutabilité des noms implique que la sortie d'une fonction est entièrement déterminée par son entrée. Ainsi, deux appels à la même fonction avec le même argument ne peuvent être distingués. Autrement dit, une fonction ne peut pas "modifier son environnement", par exemple en changeant la valeur d'une variable dans sa clôture. Nous pouvons même remplacer dans le code l'application d'une fonction à son argument par la sortie résultat de cette application[2]. Cette propriété est nommée la "transparence référentielle".

Plus précisément, une expression `e` est "transparente référentiellement" si, pour tout programme `p`, toute occurrence de `e` dans `p` peut être remplacée par le résultat de l'évaluation de `e`, sans affecter le sens du programme `p`. Il est ainsi possible de raisonner sur les programmes en substituant des expressions transparentes par des expressions transparentes de même valeur. Une fonction `f` est dite "pure" si pour toute expression transparente `e`, l'expression `(f e)` est transparente.

2. À condition que les variables libres de la fonction soient calculables au moment du remplacement. Autrement, si l'on souhaite remplacer l'application par le code de la fonction ("inlining"), il faut prendre garde à ne pas casser la liaison statique.

Le mot "valeur" signifie une expression évaluée. Par exemple, l'expression `((fun x -> x + 1) 3)` n'est pas une valeur, mais l'entier `4` en est une. Dans ce langage, nous utilisons la "forme normale de tête", ce qui signifie que toute expression qui n'est pas une application est une valeur. Notons qu'une fonction, comme l'expression `(fun x -> (g x))`, est bien une valeur, car l'application est contenue dans le corps de la fonction.

Il y a plusieurs possibilités d'évaluation. Par exemple dans l'exemple suivant, quelle expression exacte est donnée en entrée de la fonction `f` ?

```
let f = (fun x -> x + 1)
let g = (fun x -> x * x)
let r = (f (g 10))
```

Si l'entrée est `100`, nous sommes en présence d'évaluation "par valeur" : les arguments sont évalués avant d'être donnés en entrée aux fonctions.
Au contraire si l'entrée est `(g 10)`, nous sommes en présence d'évaluation "par nom" : les arguments sont donnés tels quels en entrée des fonctions.
Dans les deux techniques, la valeur finale de `r` sera toujours la même : `101`. Dans l'évaluation par nom, l'exécution n'évalue que ce qui est nécessaire. Définir un nom ne justifie pas l'évaluation de son expression, donner ce nom en entrée d'un appel de fonction non plus. Dans l'expression suivante, `(print (x + y))`, la commande `print` affiche une valeur à l'écran. L'évaluation de son argument est donc nécessaire, ce qui implique que les évaluations de `x` et de `y` sont nécessaires, afin de pouvoir évaluer la somme `+`, etc.
Nous allons appliquer l'évaluation par valeur car elle rend l'exécution plus évidente, et facilite la maîtrise de la consommation temps et espace du programme[3].

Voici un exemple de programme et un schéma de son exécution qui illustrent les principes présentés précédemment. Pour gérer la surcharge, au lieu de différencier deux variables de même nom avec des numéros (ce qui rendrait la lecture difficile), nous allons appliquer ces deux techniques :

1. les clôtures contiennent le nom de la variable accompagné de la valeur en vigueur à la définition de la fonction

2. pour chaque appel de fonction, nous créons un environnement qui est supprimé à la fin de l'appel. Cela supprime les surcharges de variables effectuées par le code de la fonction.

```
1 let c = 8
2 let f x =
3     let c = (x + c) in
4     let r = (c + c) in
5     r
6 let a = (f 3)
7 let c = 12
8 let b = (f 4)
```

Code 1.a.1 – *Exemple de programme fonctionnel*

3. Dans son livre "Purely functional data structures"[Oka99], Chris Okasaki écrit que l'appel par besoin (appel par nom avec mémorisation de la valeur calculée) est difficile à analyser, et que selon lui la bonne solution est l'appel par valeur avec quelques appels par besoin aux endroits critiques (simulés par variable mutable).

ligne	environnement courant	pile
init	{}	∅
1.	{ c=8 }	∅
2. - 5.	{ c=8 f=fun{ c=8 } }	∅
début 6.	{ c=8 f=fun{ c=8 } a=? }	∅
2.	{ x=3 c=8 }	retour 6. { c=8 f=fun{ c=8 } a=val_retour }
3.	{ x=3 c=11 }	retour 6. { c=8 f=fun{ c=8 } a=val_retour }
4.	{ x=3 c=11 r=22 }	retour 6. { c=8 f=fun{ c=8 } a=val_retour }
5.	{ x=3 c=11 r=22 val_retour=22 }	retour 6. { c=8 f=fun{ c=8 } a=val_retour }
fin 6.	{ c=8 f=fun{ c=8 } a=22 }	∅
7.	{ c=7 f=fun{ c=8 } a=22 }	∅
début 8.	{ c=7 f=fun{ c=8 } a=22 b=? }	∅
2.	{ x=4 c=8 }	retour 8. { c=7 f=fun{ c=8 } a=22 b=val_retour }
3.	{ x=4 c=12 }	retour 8. { c=7 f=fun{ c=8 } a=22 b=val_retour }
4.	{ x=4 c=12 r=24 }	retour 8. { c=7 f=fun{ c=8 } a=22 b=val_retour }
5.	{ x=4 c=12 r=24 val_retour=24 }	retour 8. { c=7 f=fun{ c=8 } a=22 b=val_retour }
fin 8.	{ c=7 f=fun{ c=8 } a=22 b=24 }	∅

Tableau 1.a.1 – *Schéma d'exécution de Code 1.a.1*

Notons que nous ne pouvons pas exprimer strictement les fonctions récursives avec ce schéma, puisqu'une fonction se contient elle-même dans sa clôture. Il faudrait pour cela utiliser un lien indirect entre une fonction et sa clôture. Il existe des machines abstraites d'exécution plus complètes, comme la *SECD*. Nous ne nous préoccuperons pas de ce niveau de détail.

Pour terminer, ajoutons à notre schéma d'exécution la représentation de variables mutables, appelées références en `OCaml`. Nous aurons quelques occasions de les utiliser dans ce document.

```
let v = ref 2          (* 'ref' pour créer une référence *)
let f x =
    v := !v + 1 ;      (* '!' pour obtenir la valeur de la référence *)
    (x + !v)
v := 3 ;               (* ':=' pour modifier la valeur de la référence *)
let r = (f 1)
```

Code 1.a.2 – *Exemple de programme fonctionnel avec mutation*

Nous ajoutons une colonne spéciale pour les références. Dans la clôture nous ne stockons pas la valeur de la référence mais le numéro de la référence. Ainsi la valeur de `v` n'est pas `2` mais `r1` . À chaque création de référence un nouveau numéro est généré.

ligne	environnement courant	références	pile
init	{}	{}	∅
1.	{ v=r1 }	{ r1=2 }	∅
2. - 4.	{ v=r1 f={ v=r1 } }	{ r1=2 }	∅
5.	{ v=r1 f={ v=r1 } }	{ r1=3 }	∅
début 6.	{ v=r1 f={ v=r1 } r=? }	{ r1=3 }	∅
2.	{ x=1 v=r1 }	{ r1=3 }	retour 6. { v=r1 f={ v=r1 } r=val_retour }
3.	{ x=1 v=r1 }	{ r1=4 }	retour 6. { v=r1 f={ v=r1 } r=val_retour }
3.	{ x=1 v=r1 val_retour=5 }	{ r1=4 }	retour 6. { v=r1 f={ v=r1 } r=val_retour }
fin 6.	{ v=r1 f={ v=r1 } r=4 }	{ r1=4 }	∅

Tableau 1.a.2 – *Schéma d'exécution de Code 1.a.2*

Une fonction qui contient une référence externe dans sa clôture est dite "à effet de bord". Une telle fonction casse la propriété de transparence référentielle. Objectivement, l'absence de cette propriété ne facilite pas l'utilisation de la fonction dans le programme, puisqu'il faut systématiquement prendre en compte l'état de la "colonne références" avant et après l'appel de la fonction. Cependant, l'absence de références implique

aussi des aménagements qui ne sont pas négligeables, comme nous le verrons dans l'introduction aux monades.

1.b Simuler des effets mutables : introduction aux monades

Un des avantages de la mutabilité est la capacité à interagir avec des variables "distantes". Il suffit d'avoir la variable dans sa clôture. En programmation objet, il est naturel que les champs d'un objet soient mutables, ce qui donne le même pouvoir : il suffit d'avoir l'objet dans sa clôture pour demander la mutation d'un de ses champs. Ce même objet peut lui-même déclencher la mutation d'un autre objet, et ainsi de suite modifier des variables très éloignées de l'action initiale.

La programmation fonctionnelle ne donne pas ce pouvoir : les valeurs des variables libres dans les clôtures sont définitives. Par exemple, il est impossible d'avoir un compteur global dans sa clôture, nous ne pourrions l'incrémenter, et il serait constant.

Prenons pour exemple un générateur pseudo-aléatoire. Celui-ci fonctionne avec une "graine" initiale. À chaque fois qu'un nombre pseudo-aléatoire est demandé, le générateur consomme la graine, fournit un nombre pseudo-aléatoire, et définit une nouvelle valeur de graine.
En programmation objet, il suffit qu'une fonction possède une référence à l'objet du générateur pour appeler une méthode de génération de nombres. Le générateur modifie alors sa graine par mutation et donne à l'appelant le nombre pseudo-aléatoire généré.
En programmation fonctionnelle, il faut fournir la graine en entrée et en sortie : en entrée pour que la fonction reçoive la graine la plus récente, et en sortie pour que la fonction rende la nouvelle graine si elle a généré des nombres. Il suffit que le générateur soit utilisé par plusieurs fonctions "distantes" dans le programme pour qu'il faille "transporter" la graine partout [4].
En conclusion, la programmation fonctionnelle pure pose un problème pour écrire des algorithmes qui font usage de variables "auxiliaires" mutables et globales.

Le langage OCaml n'est pas "pur" puisqu'il permet d'utiliser des variables mutables avec les références. Cependant ceci implique la perte de la propriété de transparence référentielle. En fait, il est possible de simuler des variables mutables avec des variables immutables, grâce aux monades [5].

Les trois composants d'une monade (d'un point de vue programmation)

Une monade est un triplet composé d'un type paramétré, d'une fonction d'entrée `return` et d'une fonction d'application `bind` :

```
type 'a monad = ...

let return : 'a -> ('a monad) = ...

let bind : ('a -> 'b monad) -> ('a monad -> 'b monad) = ...
```

Le but d'une monade est d'ajouter un "concept", un "mécanisme", un "effet", un "environnement" à une fonction. Dans ce document nous utiliserons le plus souvent le terme "effet".

Exemple de la monade *Writer*

Commençons les explications par un exemple.
Nous souhaitons enregistrer des traces (des "logs") au cours de l'exécution de notre programme. Une fonction doit pouvoir enregistrer "j'ai fait ceci", "j'ai fait cela". Pour rappel, nous nous interdisons d'utiliser une variable globale mutable. Nous devons alors donner en entrée et en sortie la variable qui stocke les logs,

4. Philip Wadler écrit dans "Monads for functional programming" [Wad95] : "The essence of an algorithm can become buried under the plumbing required to carry data from its point of creation to its point of use."
5. Sur le site ncatlab.org [Nca], il est écrit : "Monads provide one way to embed imperative programming in functional programming".

ce qui est encombrant et répétitif. La monade *Writer* permet de faire abstraction de ces opérations. Voici sa définition (avec le type `string list` pour les logs à enregistrer) :

```
type 'a monad = 'a * (string list)

let return a = (a, [])

let bind f ma =
    let (a, logA) = ma in
    let (b, logB) = (f a) in
    (b, logA @ logB)
```

Code 1.b.1 – *Définition de la monade* Writer

Le type `('a monad)` ajoute le type des logs `(string list)` à tout type `'a`. Une valeur "monadique" pour *Writer* est une paire de valeurs : la "principale" de type `'a` et "l'auxilliaire" de type `(string list)`.
La fonction `return` permet de construire une valeur monadique à partir d'une valeur simple, en lui ajoutant un effet neutre, qui n'a pas d'impact. Dans le cas de *Writer*, `return` ajoute une liste de logs vide.
La fonction `bind` applique une fonction `f` à codomaine monadique, à une valeur `ma` monadique. Dans le cas de *Writer*, `bind` concatène les logs de `ma` avec ceux de retour de la fonction `f`.

En composant la fonction `return` après une fonction `f` de type `('a -> 'b)`, nous "élevons" `f` au type `('a -> 'b monad)`. Nous appelerons aussi ceci un "lift". Ce mécanisme est nécessaire afin de composer des fonctions à effets avec des fonctions sans effets, ces dernières recevant alors des effets neutres.

```
let lift : ('a -> 'b) -> ('a -> 'b monad) =
    fun f ->
    (fun a -> return (f a))
```

Code 1.b.2 – *Définition de la fonction* `lift`

La figure 1.b.1 montre l'utilisation de `lift` pour rendre le co-domaine d'une fonction monadique, ce qui permet d'utiliser `bind`. Les types βM et γM sont respectivement le domaine et le codomaine monadiques de `bind (lift g)`.

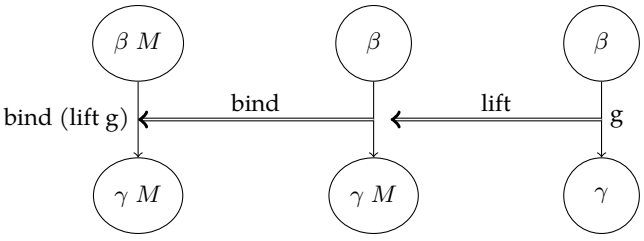


Figure 1.b.1 – *Schema d'utilisation de* `bind` *et* `lift`

Comparaison de codes : avec et sans monade *Writer*, et sucre syntaxique

Nous allons comparer plusieurs versions du même exemple : une version mutable, une version immutable sans monade *Writer*, une version immutable avec monade *Writer*, puis une dernière immutable avec monade *Writer* et sucre syntaxique `perform`.
L'exemple consiste en deux fonctions effectuant des opérations arithmétiques et écrivant des messages à chaque fois.

1. La version mutable

```
let logs : (string list) ref = ref []   (* variable mutable *)
let write : string -> unit =
    fun newLog ->
    logs := (! logs) @ [newLog]    (* modifier la valeur de la variable mutable *)

let carre : int -> int = fun n -> (write "au carré !") ; (n * n)

let modulo2 : int -> int = fun n -> (write "modulo deux !") ; (n mod 2)

List.iter print_endline (! logs) (* affiche : ""  *)

let programme : int =
    write "début du programme !" ;
    let resCarre = (carre 555) in
    let resModulo = (modulo2 555) in
    write "programme terminé !" ;
    (resCarre + resModulo)

List.iter print_endline (! logs)  (* affiche : "début du programme !
                                              au carré !
                                              modulo deux !
                                              programme terminé !"  *)
```

2. La version immutable, sans monade *Writer*

   Dans cette version, les logs sont donnés explicitement en sortie des fonctions, et la fonction `programme` concatène les logs. Le résultat est lisible car l'exemple est court, mais il permet de comprendre qu'il est difficile d'écrire un programme complet de cette façon. Le style est critiquable puisqu'il implique une gestion manuelle des logs :

```
let carre : int -> (int * string list) = fun n -> (n * n, [ "au carré !" ])
let modulo2 : int -> (int * string list) = fun n -> (n mod 2, [ "modulo deux !" ])

let programme : (int * string list) =
    let logs = [ "début du programme !" ] in
    let (resCarre, logsCarre) = (carre 555) in
    let (resModulo, logsModulo) = (modulo2 555) in
    let logs = logs @ logsCarre @ logsModulo @ [ "programme terminé !" ] in
    let (resCarre + resModulo, logs)
```

3. La version immutable avec monade *Writer*

   Dans la troisième version, la gestion des logs est concentrée dans la fonction `bind`. Nous utilisons la fonction infixe `(ma >>= f)` qui est un alias pour `(bind f ma)`. L'aspect du programme va changer puisque nous ne pouvons plus stocker simplement les valeurs de `resCarre` et `resModulo` avec un `let`. Il nous faut nous placer dans le contexte d'une valeur qui possède des logs, et cela se fait à l'aide de `bind` et d'une fonction. La valeur `resCarre` est obtenue dans l'argument d'une fonction donnée à `bind` :

```
let (>>=) ma f = (bind f ma)   (*  (bind f ma) peut maintenant s'écrire (ma >>= f)  *)
let write : string -> (unit monad) = fun newLog -> ((), [newLog])

let carre : int -> (int monad) = fun n -> write "au carré !" >>= (fun () -> return (n *
↪  n))
let modulo2 : int -> (int monad) =
    fun n -> write "modulo deux !" >>= (fun () -> return (n mod 2))

let programme : (int monad) =
    (write "début du programme !")
    >>= (fun () -> carre 555)
    >>= (fun resCarre ->
            (modulo2 555)
```

```
            >>= (fun resModulo ->
                    (write "programme terminé !")
                    >>= (fun () -> return (resCarre + resModulo))
                )
        )
```

Toute gestion manuelle des logs a été remplacée par les fonctions `bind` (`>>=`) et `return` . Cependant cela impose l'utilisation de fonctions pour obtenir les valeurs dans les contextes, et le résultat est peu lisible.

4. La version immutable, avec monade *Writer*, et sucre syntaxique `perform`

   Nous utilisons un sucre syntaxique connu sous le nom de "do-notation", ici représenté syntaxiquement par le mot-clé `perform` . Chaque ligne dans `perform` (terminée par un `;`) est liée à la suivante par un `bind` implicite : `(perform begin a <-- ma ; mb end)` est équivalent à `(bind (fun a -> mb) ma)` .

```
let programme : (int monad) =
    perform begin
        () <-- write "début du programme !" ;
        resCarre <-- carre 555 ;
        resModulo <-- modulo2 555 ;
        () <-- write "programme terminé !" ;
        return (resCarre + resModulo)
    end
```

Pour conclure, les monades permettent de simuler des mécanismes pratiques de la programmation mutable, en conservant les propriétés de la programmation fonctionnelle. Avec du sucre syntaxique, l'apparence du programme final tend même vers celle d'un programme mutable.

Explications sur la fonction `bind`

La syntaxe "do-notation" exprime l'un des buts des monades : oublier l'effet et écrire le programme comme si les valeurs auxiliaires n'étaient pas là, en se concentrant sur l'algorithme principal. Ainsi, la fonction `bind` doit être considérée comme l'application standard, contenant implicitement des effets, et la fonction `return` comme la fonction identité. Nous en avons l'intuition en comparant les types de ces fonctions :

```
let id : 'a -> 'a = fun a -> a                                  (* identité standard *)

let return : 'a -> ('a monad) = ...                             (* identité monadique *)

let (<@) : ('a -> 'b) -> ('a -> 'b) = fun f a -> (f a)          (* application standard *)

let bind : ('a -> 'b monad) -> ('a monad -> 'b monad) = ...      (* application monadique *)
```

Code 1.b.3 – *Comparaison des fonctions* `bind` *et* `return` *avec* `app` *et* `id`

Pourquoi `bind` ne transforme pas plutôt une fonction de type `('a -> 'b)` ? Nous avons déjà pour cela la fonction `fmap` , qui peut être définie avec `bind` :

```
let fmap : ('a -> 'b) -> ('a monad -> 'b monad) =
    fun f ->
    (fun ma -> bind (lift f) ma)
```

Code 1.b.4 – *Définition de la fonction* `fmap`

De plus, `fmap` est moins expressive que `bind` . Le pouvoir de `bind` réside dans sa capacité à fusionner deux effets, celui de la valeur monadique et celui de la fonction à codomaine monadique. Avec `fmap` , la

fonction ne peut pas utiliser les effets de la monade, comme par exemple écrire des logs. L'utilisation de `fmap` est très linéaire :

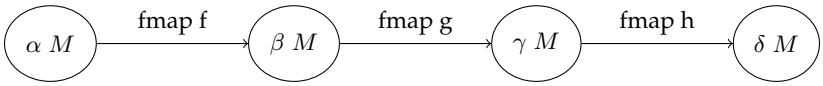


Figure 1.b.2 – *Schéma d'utilisation de la fonction* `fmap`

Au contraire, `bind` permet d'imbriquer plusieurs effets, et même de façon récursive :

```
let rec f : int -> (int monad) =
    fun n ->
        if n > 100
        then return n
        else bind (fun () -> f (n + 1)) (write "loop")
```

Code 1.b.5 – *Exemple de fonction monadique récursive*

Autrement dit, `bind` autorise son argument `f` à utiliser `bind`, ce qui n'est pas le cas de `fmap`.

Pourquoi ne pas directement écrire des fonctions de type `('a monad -> 'b monad)` sans passer par `bind` ? Si une fonction de ce type gère les effets de la même manière que `bind`, il s'agit de duplication de code, et nous perdons un des avantages principaux des monades. En revanche, il est parfois nécessaire de gérer les effets d'une façon différente de celle du `bind`, nous y reviendrons.

Les trois règles des monades

Une définition de monade doit respecter trois règles, nous allons les expliquer et les justifier. Reprenons la fonction d'application standard, elle permet d'écrire une fonction de composition, qui applique séquentiellement deux fonctions :

```
let (>>) : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c) =
    fun f g ->
    (fun a -> g (f a))
```

Code 1.b.6 – *Définition de la fonction de composition*

La fonction de composition `>>` crée une troisième fonction qui "englobe" les deux autres. La définition de l'application standard implique que la composition est associative :

```
(f >> g) >> h   ===   f >> (g >> h)
```

Code 1.b.7 – *Règle d'associativité de la composition*

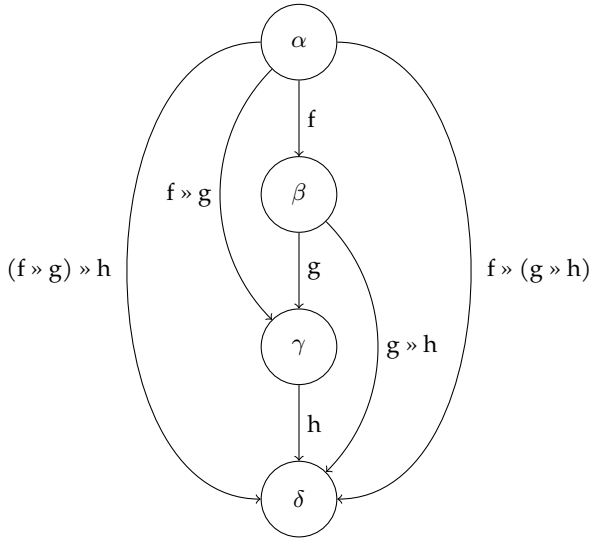


Figure 1.b.3 – *Schéma d'associativité de la composition*

La fonction `bind` est l'équivalent monadique de l'application, et à ce titre elle doit respecter les mêmes propriétés. Voici l'équivalent de la composition pour les monades :

```
let (>>>) : ('a -> 'b monad) -> ('b -> 'c monad) -> ('a -> 'c monad) =
    fun f g ->
    (fun a -> bind g (f a))
```

Code 1.b.8 – *Définition de la composition monadique*

```
(f >>> g) >>> h   ===   f >>> (g >>> h)    (* est-ce toujours vrai ? *)
```

Code 1.b.9 – *Règle d'associativité de la composition monadique*

Cette fois l'associativité reste à démontrer : la composition monadique utilise l'application monadique `bind`, qui ajoute des mécanismes de gestion des effets. Ces mécanismes doivent donc eux aussi présenter des propriétés "associatives". Le respect de l'associativité dépend donc de la définition de `bind` et `return`. L'absence d'associativité serait problématique, car il faudrait associer les fonctions selon la monade, et la monade en question n'est pas toujours connue, cela impacterait la modularité des monades, nous en reparlerons.

La composition standard a une "opérande neutralisante" à gauche et à droite, la fonction identité :

```
(id >> f)   ===   f   ===   (f >> id)
```

Code 1.b.10 – *Neutralité gauche et droite de la composition*

L'équivalent monadique est l'opérande neutralisante `return` :

```
(return >>> f)   ===   f   ===   (f >>> return)    (* est-ce toujours vrai ? *)
```

Code 1.b.11 – *Neutralité gauche et droite de la composition monadique*

La neutralité de `return` est importante car elle garantit que les élévations de fonctions ne vont pas ajouter des effets non neutres. Par exemple, il ne serait pas correct que la fonction `return` de la monade *Writer* ajoute un message. La neutralité de `return` assure de pouvoir combiner correctement des fonctions à effets avec des fonctions sans effets.

En général les trois règles des monades sont écrites sans utiliser la composition monadique [6] :

```
(ma >>= f) >>= g   ===   ma >>= (fun a -> (f a) >>= g)      (* associativité *)
                         (* a n'est pas libre dans f et g *)

(return a) >>= f   ===   (f a)                               (* neutralité gauche *)

(ma >>= return)    ===   ma                                  (* neutralité droite *)
```

Code 1.b.12 – *Les trois règles monadiques*

La monade *Writer* respecte ces règles : l'opération de concaténation des listes est associative, et l'opérande de la liste vide est neutre pour la concaténation.

Nous pouvons distinguer au moins deux sortes de monades.
La première est celle des "valeurs auxiliaires". Nous y plaçons *Writer* qui ajoute une variable auxiliaire de stockage, dont les règles de modifications respectent celles d'un monoïde, la monade *Reader* qui ajoute une variable constante (en lecture seule) implémentée par un paramètre, et la monade *State* qui ajoute une variable "mutable" sans restriction, implémentée par un paramètre d'entrée contenant la valeur actuelle de la variable, et pas une valeur auxiliaire de sortie représentant la nouvelle valeur.
La deuxième sorte est celle des "collections". Nous y plaçons *List* qui permet d'avoir plusieurs valeurs, *Option* qui permet d'avoir zéro ou une valeur.
Il existe beaucoup d'autres monades, très différentes, par exemple la monade *Continuation*, dont la valeur monadique attend une fonction de type `('a -> 'r)` exprimant la "suite du programme".

Exemple de définition de la monade *State* en OCaml

Voici une définition de la monade *State* en OCaml que nous utiliserons beaucoup. Cette monade ajoute une variable auxiliaire à toute valeur. Son type `'a monad` est un type fonctionnel, une expression de ce type reçoit en entrée la valeur actuelle de la variable auxiliaire, et rend en sortie une nouvelle valeur pour la variable auxiliaire :

6. Une preuve est donnée en annexe B.1.

```
module type TYPE = sig
    type t
end

module State (Var : TYPE) : sig
            type 'a monad = (Var.t -> 'a * Var.t)
            val return : 'a -> 'a monad
            val bind : ('a -> 'b monad) -> ('a monad -> 'b monad)
end = struct
    type 'a monad = Var.t -> ('a * Var.t)

            let return a = fun var -> (a, var)

            let bind f ma =
                    fun var0 ->
                    let (a, var1) = (ma var0) in
                    (f a var1)
end
```

Code 1.b.13 – *Définition de la monade* State

Le principe de la monade *State* est directement tiré de la gestion de la variable globale mutable en programmation fonctionnelle pure, rappelons-nous de l'exemple du générateur pseudo-aléatoire. Il s'agit de transporter la valeur immutable dans le programme, en l'ajoutant en entrée et en sortie de chaque fonction, ce qui explique le type monadique `(Var.t -> ('a * Var.t))`.

Cette "variable auxiliaire" n'est pas exactement une variable, elle n'a pas de nom ni d'espace mémoire spécifique. Chaque valeur monadique de type `(Var.t -> ('a * Var.t))` est libre de donner le nom qu'elle souhaite à son argument, ce qui constituera le "nom" de la "variable" auxiliaire à cet endroit.

Cette variable n'est pas non plus exactement mutable, ce n'est qu'une simulation. Le principe de la transparence référentielle est conservé, puisqu'aucune mutation n'est effectuée. Il faut considérer que c'est comme si l'on avait une valeur auxiliaire en permanence dans le contexte, qui serait automatiquement ajoutée aux fonctions et récupérée en sortie des fonctions.
Cela se rapproche de la liaison dynamique, où le nom libre dans la clôture est lié au même nom présent dans le contexte d'appel. Néanmoins, il y a deux différences.
La première est que la variable auxiliaire de *State* n'est pas nommée. La deuxième différence avec la liaison dynamique mutable est qu'il n'y a pas de réelle mutation, il est toujours possible d'appeler une fonction et de stocker son résultat dans une variable, sans autre modification du contexte local.

Ce que nous attendons des monades

Essayons de lister les objectifs que nous souhaitons atteindre en utilisant des monades.
Le principal objectif est de simuler des mécanismes mutables, tout en conservant la transparence référentielle. Cette simulation doit être centralisée dans `bind` et `return`. Les appels imbriqués à `bind` doivent être rendus implicites par l'utilisation de `perform`, afin de conserver une syntaxe raisonnable.

Il est peu probable que nous ayons besoin d'un seul mécanisme pour l'ensemble du programme. Il faut nécessairement pouvoir composer les monades afin de combiner les mécanismes. Cette composition doit se faire de la façon la plus automatique possible. Le nombre de combinaisons possibles est trop grand pour être géré entièrement manuellement.

Dans l'objectif d'écrire un programme de la façon la plus modulaire possible, nous souhaitons pouvoir écrire une fonction qui dépend d'un ensemble de monades données, et qui puisse être utilisée dans n'importe quel contexte de monades qui contient au moins cet ensemble. Par exemple, nous désirons pouvoir exprimer "n'importe quel ensemble de monades qui contient au moins la monade *State*". Cela peut se résoudre partiellement par l'utilisation de structures communes à plusieurs monades. Par exemple, le langage *Haskell* définit le groupe *MonadPlus* des monades dont les valeurs peuvent se combiner selon les

lois d'un monoïde. Ce groupe peut être nommé dans un type afin de cibler "toute monade qui est dans le groupe MonadPlus". La réalisation de cet objectif est encore expérimental, voir l'article de Tom Schrijvers et Bruno Oliveira [SO11].

De part le service élémentaire qu'elles assurent, certaines monades comme *Writer* et *State* ont un champ d'application très large. De plus, la syntaxe `perform` a pour objectif de les rendre pratiquement transparentes dans le code source du programme. Par conséquent, ces monades doivent laisser une empreinte en mémoire et en temps d'exécution la plus légère possible.

Combinaison de monade

Dans cette partie, nous développons un peu la question de la combinaison de monades.

Nous disposons de deux monades `X` et `Y`. Ce sont des monades quelconques, car le but est d'automatiser leur combinaison.

```
type 'a x = ...

let returnX : 'a -> 'a x = ...
let bindX : 'a x -> ('a -> 'b x) -> 'b x = ...

(* deux fonctions utiles, définies par returnX et bindX *)
let mapX : 'a x -> ('a -> 'b) -> 'b x = (fun aX f -> bindX aX (f >> returnX))
let flatX : 'a x x -> 'a x = (fun aX1X2 -> bindX aXX (fun aX -> aX))

type 'a y = ...
(* pareil que X, mais avec Y *)
```

Code 1.b.14 – *Définitions existantes des monades `X` et `Y`*

Nous souhaitons les combiner en une monade nommée `XY` :

```
type 'a xy = ???

 val returnXY : 'a -> 'a xy
 val bindXY : 'a xy -> ('a -> 'b xy) -> 'b xy

(* fonctions relatives aux monades combinées *)
 val liftFromX : 'a x -> 'a xy
 val liftFromY : 'a y -> 'a xy
```

Code 1.b.15 – *Signature des définitions de la monade combinée `XY`*

Les fonctions `liftFromX` et `liftFromY` sont importantes, elles permettent de composer des fonctions utilisant des mécanismes différents. Elles suivent le même principe que le `lift` initial des monades, elles ajoutent un effet neutre.

La monade combinée doit respecter des propriétés faisant intervenir les monades `X` et `Y`. Voici, de façon informelle, trois idées de règles[7] qu'une monade combinée `XY` devrait respecter (en plus des trois règles initiales des monades) :

7. Ces règles sont directement inspirées de celles données dans "Monad Transformers and Modular Interpreters" de Liang, Hudak et Jones [LHJ98].

```
(* équivalence de returnX/Y avec returnXY par liftFromX/Y *)
(returnX >> liftFromX)   ===   returnXY   ===   (returnY >> liftFromY)

(* équivalence entre bindX et bindXY par liftFromX *)
liftFromX (bindX aX f)   ===   bindXY (liftFromX aX) (f >> liftFromX)

(* équivalence entre bindY et bindXY par liftFromY *)
liftFromY (bindY aY f)   ===   bindXY (liftFromY aY) (f >> liftFromY)
```

Code 1.b.16 – *Règle de la monade combinée* `XY`

La première règle contraint `returnXY` à être neutre comme `returnX` suivi de `liftFromX`. Les fonctions `liftFromX` et `liftFromY` permettent de mettre en relation les définitions de la monade combinée avec celles des monades initiales.
La deuxième règle contraint `bindXY` à gérer les effets de la même façon que `bindX` lorsque les effets `Y` sont neutres. La troisième règle est l'équivalente pour `bindY` et les effets `X`.

Donnons un exemple avec une définition de `WriterOption`, la combinaison des monades *Writer* et *Option* [8] :

```
type 'a writer = 'a * (string list)

type 'a writerOption = 'a writer option   (* = Some of ('a * logs) | None *)

let returnWriterOption = returnWriter >> returnOption

let bindWriterOption aWriterOption f =
    match aWriterOption with
    | Some (a, logs1) -> match f a with
                         | Some (b, logs2) -> Some (b, logs1 ++ logs2)
                         | None -> None
    | None -> None

let liftFromWriter aWriter = returnOption aWriter
let liftFromOption aOption = mapOption aOption returnWriter
```

Code 1.b.17 – *Définition de la monade combinée* `WriterOption`

Imbrication automatique de monades

Nous développons l'idée d'imbriquer automatiquement deux monades, en s'inspirant de l'exemple `WriterOption` précédent. Prenons deux monades `X` et `Y` que nous combinons par imbrication en une monade `XY` :

```
type 'a xy = 'a x y

let returnXY = returnX >> returnY
let bindXY : 'a xy -> ('a -> 'b xy) -> 'b xy = ???

let liftFromX = returnY
let liftFromY aY = mapY aY returnX
```

Code 1.b.18 – *Définition incomplète de la monade combinée automatiquement* `XY`

Il nous reste à définir `bindXY`, c'est à dire à définir comment appliquer une fonction `f` de type `('a -> 'b xy)` à une valeur `m` de type `('a xy)`. Notons que :

8. La définition de la monade *Option* est donnée en annexe A.4, parmi les autres monades.

— nous ne pouvons pas appliquer `bindX`, car le codomaine de `f` est de type `('b x y)`
— nous ne pouvons pas appliquer directement `bindY`, car il faudrait que le domaine de `f` soit `('a x)`

Nous supposons une fonction `swapYX` de type `('a y x -> 'a x y)` capable de changer le sens d'imbrication des deux monades `X` et `Y`. Nous définissons ensuite `bindXY` en fonction de `swapYX` :

```
val swapYX : 'a y x -> 'a x y

let bindXY aXY f =
    bindY aXY (fun aX ->
        let bXYX = mapX aX f in
        let bXXY = swapYX bXYX in
        let bXY = mapY bXXY flatX in
        bXY)
```

Code 1.b.19 – *Définition de* `bind` *pour la monade combinée automatiquement* `XY`

Afin de réglementer `swapYX`, nous ajoutons que pour toute fonction `swapYX` ainsi utilisée, il doit exister une fonction `cancel` telle que :

```
(swapYX >> cancel)   ===   id      (* identité sur le type 'a y x *)
```

Code 1.b.20 – *Prérequis pour combiner automatiquement deux monades*

Faute de respecter cette propriété, la fonction `swapYX` n'est pas obligée de conserver toutes les informations du domaine `('a y x)`, et les règles de la monade `XY` risquent de ne pas être vérifiées (intuitivement, si `bind` perd des informations, les règles de neutralité sont cassées).

Donnons comme exemple la monade combinée `OptionWriter` :

```
type 'a writer = 'a * (string list)

type 'a optionWriter = ('a option) writer   (*   = (Val of 'a | Rien) * logs   *)

val swapWriterOption : 'a writer option -> 'a option writer

let swapWriterOption aWriterOption =
    match aWriterOption with
    | Some (a, logs) -> (Some a, logs)
    | None           -> (None,   [])

let cancel aOptionWriter =
    match aOptionWriter with
    | (Some a, logs) -> Some (a, logs)
    | (None, logs)   -> None
```

Code 1.b.21 – *Définitions pour combiner automatiquement* Option *avec* Writer

La composition `(swapWriterOption >> cancel)` donne bien l'identité, puisque les logs qui sont jetés dans le cas `(None, logs)` du `match` de `cancel` sont ceux qui ont été générés dans le cas `None` du `match` de `swapWriterOption`.

Nous remarquons que la construction opposée échoue, car il n'est pas possible d'écrire la fonction `swapOptionWriter` en respectant la règle d'identité avec `cancel`. Pourtant, nous avons déjà réussi à écrire

une monade combinée `WriterOption` qui était correcte, sans passer par une étape de `swap`. Redonnons la fonction `bind` de cette dernière :

```
let bindWriterOption aWriterOption f =
    bindOption aWriterOption (fun (a, logs1) ->  (* extraction de a *)
        mapOption (f a) (fun (b, logs2) ->
            (b, logs1 ++ logs2)
        ))
```

Code 1.b.22 – *Définition manuelle de bind de* Writer *avec* Option

Notre règle d'identité avec `cancel` était peut-être trop stricte, puisque dans cet exemple précis, l'action de `bindWriterOption` est de jeter `logs1` lorsque `(f a)` est égal à `None`, ce qui était justement ce que nous pouvions reprocher à `swapOptionWriter`.

Nous remarquons aussi que la définition de `bindWriterOption` ressemble à celle de `bindXY`, sauf que nous connaissons `X` (ici *Writer*), ce qui nous permet de savoir comment "extraire" la valeur `'a` et gérer les logs. De plus, l'utilisation de la monade *Option* dans `bindWriterOption` se limite aux fonctions `bindOption` et `mapOption`, ce qui est générique et ne dépend pas de la définition de la monade *Option*.
Nous concluons que s'il est difficile d'imbriquer automatiquement deux monades `X` et `Y` quelconques, il est plus souvent possible de le faire lorsque l'on fixe l'une des deux. Ces observations nous conduisent aux transformateurs de monades.

Transformateurs de monade

Un transformateur de monade est une "fonction" des monades vers les monades [LHJ98]. En reprenant la syntaxe des exemples précédents, un transformateur `XY` est défini en `X` et indéfini en `Y`.
Par exemple, il existe le transformateur de monade `WriterY`. Pour toute monade définie `M`, le transformateur `WriterY` transforme `M` en une monade `WriterY(M)`, qui est la combinaison de *Writer* et de `M`. Nous noterons parfois `WriterM` pour `WriterY(M)`.
Voici une définition en OCaml, avec des modules :

```
module type MONAD = sig
    type 'a monad
    val return : 'a -> 'a monad
    val bind : 'a monad -> ('a -> 'b monad) -> 'b monad
    val map : 'a monad -> ('a -> 'b) -> 'b monad
end

module WriterY (M : MONAD) = struct
    type 'a monad = 'a log M.monad

    let return a = returnWriter >> M.return

    let bind aWriterM f =
        M.bind aWriterM (fun (a, logs1) ->
            M.map (f a) (fun (b, logs2) ->
                (b, logs1 ++ logs2)
            ))

    let liftWriter = M.return
    let liftM aM = M.map aM returnWriter
end
```

Code 1.b.23 – *Définition du transformateur* `WriterY`

Il est important de noter que les transformateurs n'utilisent pas toujours une imbrication stricte des deux types. Par exemple, le transformateur `StateY` utilise le type suivant :

```
type 'a monad = var -> (('a * var) M.monad)
```

Code 1.b.24 – *Type du transformateur* `StateY`

En effet si nous utilisons le type `((var -> 'a * var) M.monad)`, la fonction `bind` ne sait pas quel `var` choisir pour obtenir `'a`, et ne peut donc pas appliquer `f`. Il faut placer `var` à l'extérieur de `M.monad` :

```
let bind ma f = fun var0 ->
    M.bind (ma var0) (fun (a, var1) -> f a var1)

let return a = returnState a >> M.return
let liftState aSt = aSt >> M.return
let liftM aM = fun var -> M.map aM (fun a -> (a, var))
```

Code 1.b.25 – *Définition du transformateur* `StateY`

Les transformateurs permettent d'éviter l'explosion du nombre de combinaisons manuelles à définir. Ils sont génériques : il suffit d'écrire le transformateur `WriterY` pour pouvoir obtenir toutes les combinaisons `WriterM`, par exemple `WriterWriterState`. Ils évitent la duplication de code de gestion d'un mécanisme. Il faut montrer que les règles de la monade sont respectées pour toute monade `M`, en procédant par réécriture de code. La preuve pour `WriterY` est donnée en annexes B.2.

Parfois il est difficile de trouver un transformateur qui soit très proche de la monade initiale, comme pour la monade `List`. C'est parce que deux mécanismes ne conservent pas nécessairement l'associativité lorsqu'ils sont combinés. L'associativité de la monade `List` repose sur l'équivalence suivante :

```
f : ('a -> 'b list)
g : ('b -> 'c list)

ma @> map f >> flat >> map g >> flat     ===     ma @> map (f >> map g) >> flat >> flat
```

Dans le membre de gauche, la fonction `f` est appliquée à chaque élément de `ma`, puis `flat` effectue la concaténation des sous-listes, et ainsi de suite avec `g`. Dans le membre de droit, à chaque élément de `ma`, est appliquée `f`, et directement `g` sur chaque élément. Voici un schéma d'exemple détaillant le contenu des listes intermédiaires :

membre gauche : $[\bigcirc, \bigcirc] \xrightarrow{map\ f} [[\oplus], [\oplus]] \xrightarrow{flat} [\oplus, \oplus] \xrightarrow{map\ g} [[\bullet], [\bullet]] \xrightarrow{flat} [\bullet, \bullet]$

membre droit : $[\bigcirc, \bigcirc] \xrightarrow{map\ (f \gg map\ g)} [[[\bullet]], [[\bullet]]] \xrightarrow{flat} [[\bullet], [\bullet]] \xrightarrow{flat} [\bullet, \bullet]$

Voici le schéma qui trace l'ordre d'exécution des transformations d'éléments :

membre gauche : $(\bigcirc \mapsto \oplus); (\bigcirc \mapsto \oplus); (\oplus \mapsto \bullet); (\oplus \mapsto \bullet)$

membre droit : $(\bigcirc \mapsto \oplus); (\oplus \mapsto \bullet); (\bigcirc \mapsto \oplus); (\oplus \mapsto \bullet)$

Ainsi les éléments ne sont pas traités dans le même ordre, mais il sont finalement assemblés dans le même ordre. Cela rend `List` associative, sauf si nous ajoutons une autre monade qui a un effet à chaque traitement d'éléments. Par exemple, si nous ajoutons une monade *Writer* et qu'on trace la même chose que dans notre schéma, l'associativité ne sera pas respectée puisque l'ordre des messages est différent.

La transformation n'est pas commutative, tout dépend des monades en question. Par exemple, `StateY(Writer)` est isomorphique à `WriterY(State)`, mais `OptionY(Writer)` n'est pas isomorphique à `WriterY(Option)`. Il semble que cela soit naturel : il n'y a pas toujours une combinaison unique de deux monades. Voici l'isomorphisme pour *Writer* et *State* :

```
val conv : 'a StateY(Writer) -> 'a WriterY(State)
let conv f = f >> (fun ((a, st), logs) -> ((a, logs), st))

val vnoc : 'a WriterY(State) -> 'a StateY(Writer)
let vnoc f = f >> (fun ((a, logs), st) -> ((a, st), logs))
```

Code 1.b.26 – *Isomorphisme entre `StateY(Writer)` et `WriterY(State)`*

Pour *Writer* et *Option*, cela paraît difficile étant donné que `WriterY(Option)` place les logs sous l'option, alors que `OptionY(Writer)` les place en dehors (donc il y a toujours des logs). Le transformateur `WriterY` ne permet pas de stocker d'informations en dehors de la monade `Y`. Une solution serait d'avoir une valeur spéciale de `'a` signifiant "nul", mais cela revient à ce pourquoi *Option* existe.

Le vrai défaut des transformateurs se trouve dans le manque de généricité de la "pile de monades". Lorsque nous écrivons une fonction qui écrit des logs, nous utilisons la monade *Writer*. Si nous voulons composer cette fonction avec une autre de codomaine `StateY(Writer)`, il nous faut utiliser la fonction `liftFromY` . Cela donne un fort encombrement de la syntaxe lorsque le nombre de monades à "lifter" augmente. En Haskell, ce problème est résolu avec des "typeclass" qui permettent d'appeler automatiquement autant de fois `lift` que nécessaire. Cependant, il reste des problèmes. Lorsque la même monade est utilisée plusieurs fois dans la pile, le compilateur utilise celle la plus haut placée dans la pile. De plus, l'injection est problématique. Si notre pile de monade est `XYZ`, et que nous avons une fonction de codomaine `XZ`, comment lui injecter `Y` de façon générique ?
Des travaux tentent de trouver des solutions à ces problèmes, par exemple l'article "Monads, Zippers and Views" de Schrijvers et Oliveira [SO11]. Leur but est d'abstraire la "pile de monades", et d'exprimer par exemple un codomaine tel que "une pile de monades qui contient *Writer*".
Dans ce document, nous ne chercherons pas à résoudre ces problèmes. Notre objectif sera d'être dans le cadre des transformateurs de monades, en acceptant leurs défauts actuels, car ils sont une solution raisonnable et assez répandue actuellement.

Chapitre 2

La monade `State` avec le respect d'un prédicat

2.a Respect d'un prédicat par un programme

Tout programme a un ensemble de règles à respecter. Par exemple, dans un programme d'attribution de salles à des séances de cours, le nombre de salles disponibles ne doit jamais être négatif. Au moins trois approches sont connues pour vérifier le respect des règles : la correction par construction, la vérification exhaustive et les tests manuels. Nous allons utiliser la correction par construction : en nous appuyant sur un ensemble restreint de preuves manuelles, et la méthode des types abstraits, nous garantirons que n'importe quel programme bien typé respecte le prédicat.

Nous exprimons la propriété sous la forme d'un prédicat sur les valeurs d'un type donné. Voici un exemple de prédicat sur les entiers :

$$\texttt{Pred} = \{n \mid n : \texttt{Int et } n \texttt{ modulo } 2 = 0\}$$

Ce prédicat est clairement faux pour certains programmes, par exemple `(fun x -> 3)` . Nous voulons certainement vérifier le prédicat `pair` sur certains entiers du programme, pas tous. Nous avons donc besoin d'un alias pour le type `Int`, afin de marquer les entiers qui doivent respecter le prédicat.

Un type abstrait est constitué :

- d'un type concret, par exemple `Int`
- d'un type abstrait, par exemple `Pair` , qui est un alias du type concret
- d'un constructeur, par exemple `pair : (Int -> Pair)` , qui convertit une valeur du type concret en type abstrait
- d'un destructeur, par exemple `unpair : (Pair -> Int)` , qui fait l'inverse
- d'une bibliothèque contenant des fonctions de manipulation du type abstrait, par exemple :

```
let zero = (pair 0)
let set = fun n:Int -> if n mod 2 = 0 then (some (pair n)) else (none of Pair)
let get = fun p:Pair -> (unpair p)
let succ = fun p:Pair -> (pair ((unpair p) + 2))
```

Généralement nous parlons de type "abstrait" parce que le constructeur n'est accessible que depuis les fonctions de la bibliothèque, et inaccessible dans le reste du programme. Cela donne à la bibliothèque le contrôle des valeurs de type `Pair` qui peuvent se créer dans le programme.

Notons que fonction `set` renvoie une erreur quand son argument n'est pas pair. Nous comprenons ici que nous ne pouvons pas garantir que le programme n'essaiera pas de créer un entier pair qui n'est pas pair : nous ne pouvons que garantir qu'il n'y arrivera pas (dans notre exemple il reçoit `none`).

Nous allons décrire les propriétés que doit suivre la bibliothèque afin d'obtenir les garanties sur le programme. Dans notre exemple, l'objectif est de montrer que le programme obtiendra bien un entier pair partout où il s'attend à en voir un (ce qui est visible par le type `Pair`).

2.b Vérification du prédicat par le type abstrait

L'intuition est que la bibliothèque ne doit construire que des valeurs `Pair` qui respectent le prédicat `Pred`. La bibliothèque assume ainsi que les valeurs `Pair` qu'elle reçoit en argument respectent aussi le prédicat. Cela lui permet d'éviter des vérifications systématiques sur les arguments de type `Pair` .

Nous utilisons un langage lambda calcul standard, avec `unit` , les booléens, les chaînes, les entiers, les bifurcations, les couples, les options, auquel il faut ajouter les opérations sur les entiers comme l'addition, les fonctions pour extraire une valeur d'une option, etc.

⟨*term*⟩ ::= ⟨*ident*⟩ | ⟨*value*⟩ | (⟨*term*⟩ ⟨*term*⟩) | (if ⟨*term*⟩ then ⟨*term*⟩ else ⟨*term*⟩) | (⟨*term*⟩, ⟨*term*⟩) | (fst ⟨*term*⟩)
| (snd ⟨*term*⟩) | (some ⟨*term*⟩)

⟨*value*⟩ ::= unit | true | false | ⟨*int*⟩ | ⟨*string*⟩ | ⟨*function*⟩ | (⟨*value*⟩, ⟨*value*⟩) | (some ⟨*value*⟩) | (none of ⟨*type*⟩)

⟨*function*⟩ ::= (λ ⟨*ident*⟩ :⟨*type*⟩ → ⟨*term*⟩)

⟨*type*⟩ ::= Unit | Bool | Int | String | (⟨*type*⟩ → ⟨*type*⟩) | (⟨*type*⟩, ⟨*type*⟩) | Option[⟨*type*⟩]

⟨*ident*⟩ ::= $token([a\text{-}z][a\text{-}zA\text{-}Z0\text{-}9]^*)$

⟨*int*⟩ ::= $token([0\text{-}9]^+)$

⟨*string*⟩ ::= $token(['']\neg['']^*[''])$

Définition 1 (Évaluation)**.**

$$\text{E-APP-1}\ \frac{t_1 \xrightarrow{\lambda a} t_1'}{(t_1\ t_2) \xrightarrow{\lambda a} (t_1'\ t_2)} \qquad \text{E-APP-2}\ \frac{t \xrightarrow{\lambda a} t'}{(f\ t) \xrightarrow{\lambda a} (f\ t')} \qquad \text{E-BETA}\ \frac{f = (\lambda x : A \to t) \qquad t' = t[x := v]}{(f\ v) \xrightarrow{\lambda a} t'}$$

$$\text{E-IF-1}\ \frac{t_1 \xrightarrow{\lambda a} t_1'}{(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) \xrightarrow{\lambda a} (\texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3)} \qquad \text{E-IF-T}\ \frac{\varnothing}{(\texttt{if true then}\ t_2\ \texttt{else}\ t_3) \xrightarrow{\lambda a} t_2} \qquad \text{E-IF-F}\ \frac{\varnothing}{(\texttt{if false then}\ t_2\ \texttt{else}\ t_3) \xrightarrow{\lambda a} t_3}$$

$$\text{E-TUPLE-1}\ \frac{t_1 \xrightarrow{\lambda a} t_1'}{(t_1, t_2) \xrightarrow{\lambda a} (t_1', t_2)} \qquad \text{E-TUPLE-2}\ \frac{t_2 \xrightarrow{\lambda a} t_2'}{(v_1, t_2) \xrightarrow{\lambda a} (v_1, t_2')} \qquad \text{E-FST-1}\ \frac{t \xrightarrow{\lambda a} t'}{(\texttt{fst}\ t) \xrightarrow{\lambda a} (\texttt{fst}\ t')} \qquad \text{E-FST}\ \frac{\varnothing}{(\texttt{fst}\ (v_1, v_2)) \xrightarrow{\lambda a} v_1}$$

$$\text{E-SND-1}\ \frac{t \xrightarrow{\lambda a} t'}{(\texttt{snd}\ t) \xrightarrow{\lambda a} (\texttt{snd}\ t')} \qquad \text{E-SND}\ \frac{\varnothing}{(\texttt{snd}\ (v_1, v_2)) \xrightarrow{\lambda a} v_2}$$

$$\text{E-SOME}\ \frac{t \xrightarrow{\lambda a} t'}{(\texttt{some}\ t) \xrightarrow{\lambda a} (\texttt{some}\ t')}$$

Définition 2 (Typage).

$$\text{T-IDENT}\ \frac{(x : A) \in \Sigma}{\Sigma \vdash x : A} \qquad \text{T-UNIT}\ \frac{\varnothing}{\Sigma \vdash \texttt{unit} : \texttt{Unit}} \qquad \text{T-TRUE}\ \frac{\varnothing}{\Sigma \vdash \texttt{true} : \texttt{Bool}} \qquad \text{T-FALSE}\ \frac{\varnothing}{\Sigma \vdash \texttt{false} : \texttt{Bool}} \qquad \text{T-INT}\ \frac{\varnothing}{\Sigma \vdash n : \texttt{Int}} \qquad \text{T-STRING}\ \frac{\varnothing}{\Sigma \vdash s : \texttt{String}}$$

$$\text{T-FUNC}\ \frac{\Sigma, (x : A) \vdash t : B}{\Sigma \vdash (\lambda x : A \to t) : (A \to B)} \qquad \text{T-APP}\ \frac{\Sigma \vdash t_1 : (A \to B) \qquad \Sigma \vdash t_2 : A}{\Sigma \vdash (t_1\ t_2) : B} \qquad \text{T-IF}\ \frac{\Sigma \vdash t_1 : \texttt{Bool} \qquad \Sigma \vdash t_2 : A \qquad \Sigma \vdash t_3 : A}{\Sigma \vdash (\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) : A}$$

$$\text{T-TUPLE}\ \frac{\Sigma \vdash t_1 : A \qquad \Sigma \vdash t_2 : B}{\Sigma \vdash (t_1, t_2) : (A, B)} \qquad \text{T-FST}\ \frac{\Sigma \vdash t : (A, B)}{\Sigma \vdash (\texttt{fst}\ t) : A} \qquad \text{T-SND}\ \frac{\Sigma \vdash t : (A, B)}{\Sigma \vdash (\texttt{snd}\ t) : B}$$

$$\text{T-SOME}\ \frac{\Sigma \vdash t : A}{\Sigma \vdash (\texttt{some}\ t) : \texttt{Option}[A]} \qquad \text{T-NONE}\ \frac{\varnothing}{\Sigma \vdash (\texttt{none of}\ A) : \texttt{Option}[A]}$$

Nous ajoutons le type abstrait pour l'exemple de `Pair`. Dans un véritable langage, il existerait un mécanisme d'ajout dynamique (c'est à dire exprimé dans le le langage lui-même).

⟨*term*⟩ ::= ... | **(pair** ⟨*term*⟩**)** | **(unpair** ⟨*term*⟩**)**

⟨*value*⟩ ::= ... | **(pair** ⟨*int*⟩**)**

⟨*type*⟩ ::= ... | **Pair**

Définition 3 (Évaluation pour `Pair`).

$$\text{E-PAIR}\ \frac{t \xrightarrow{\lambda a} t'}{(\texttt{pair}\ t) \xrightarrow{\lambda a} (\texttt{pair}\ t')} \qquad \text{E-UNPAIR-1}\ \frac{t \xrightarrow{\lambda a} t'}{(\texttt{unpair}\ t) \xrightarrow{\lambda a} (\texttt{unpair}\ t')} \qquad \text{E-UNPAIR}\ \frac{\varnothing}{(\texttt{unpair}\ (\texttt{pair}\ n)) \xrightarrow{\lambda a} n}$$

Définition 4 (Typage pour `Pair`).

$$\text{T-PAIR}\ \frac{\Sigma \vdash t : \texttt{Int}}{\Sigma \vdash (\texttt{pair}\ t) : \texttt{Pair}} \qquad \text{T-UNPAIR}\ \frac{\Sigma \vdash t : \texttt{Pair}}{\Sigma \vdash (\texttt{unpair}\ t) : \texttt{Int}}$$

Nous définissons la propriété `R`, que doit respecter toute fonction de la bibliothèque. Elle s'appuie sur le type de la fonction pour savoir quelle valeur doit respecter le prédicat. Elle complète les arguments attendus par les fonctions, puisqu'ils sont nécessaires pour vérifier le prédicat. Elle est définie par récurrence, ce qui lui permet de prendre en compte des types "complexes" comme `((Int -> Pair) -> Pair)`.

Définition 5 (prédicat `R` : respect du prédicat par un terme bien typé)**.** *Définition :*

$$\texttt{R}(\varnothing \vdash t : \texttt{Pair}) \wedge (t \xrightarrow{\lambda a} t') \Leftrightarrow \texttt{R}(t')$$
$$\texttt{R}(\varnothing \vdash (\texttt{pair}\ n) : \texttt{Pair}) \Leftrightarrow n \in \texttt{Pred}$$
$$\texttt{R}(\varnothing \vdash n : \texttt{Int}) \Leftrightarrow \texttt{vrai}$$
$$\texttt{R}(\varnothing \vdash f : (A \to B)) \Leftrightarrow \forall \texttt{R}(\varnothing \vdash v : A),\ \texttt{R}(\varnothing \vdash (f\ v) : B)$$
$$\texttt{R}(\Sigma, (x : A) \vdash t : B) \Leftrightarrow \forall \texttt{R}(\varnothing \vdash v : A),\ \texttt{R}(\Sigma \vdash (t\ v) : B)$$
$$\texttt{R}(\varnothing \vdash \texttt{unit} : \texttt{Unit}) \Leftrightarrow \texttt{vrai}$$
$$\texttt{R}(\varnothing \vdash b : \texttt{Bool}) \Leftrightarrow \texttt{vrai}$$
$$\texttt{R}(\varnothing \vdash s : \texttt{String}) \Leftrightarrow \texttt{vrai}$$
$$\texttt{R}(\varnothing \vdash (v_1, v_2) : (A, B)) \Leftrightarrow \texttt{R}(\varnothing \vdash v_1 : A) \wedge \texttt{R}(\varnothing \vdash v_2 : B)$$
$$\texttt{R}(\varnothing \vdash v : \texttt{Option}[A]) \Leftrightarrow \texttt{si}\ v = (\texttt{some}\ v')\ \texttt{alors}\ \texttt{R}(\varnothing \vdash v' : A)\ \texttt{sinon}\ \texttt{vrai}$$

Vérifions que notre bibliothèque d'exemple possède la propriété `R` :
- `zero` est correct puisque `0` est pair.
- `set` est correct puisque la seule possibilité pour renvoyer `some n` est que `n mod 2 = 0`, ce qui correspond à la définition de `Pred`.
- `get` est correct puisque la sortie est de type `Int`.
- `succ` est correct puisqu'on assume que le pair d'entrée vérifie le prédicat, et que l'on renvoie sa valeur ajoutée de `2`.

2.c Vérification du prédicat par un programme utilisant le type abstrait

La bibliothèque du type abstrait est utilisée par le programme, plus précisément le "reste" du programme, que nous nommerons "application". Nous pouvons représenter l'application par le terme :

$$x_{bib} : A_{bib} \vdash t_{app} : A_{app}$$

La bibliothèque par la valeur :

$$\varnothing \vdash v_{bib} : A_{bib}$$

Et enfin le programme par le terme issu de la substitution :

$$\varnothing \vdash t_{app}[x_{bib} \mapsto v_{bib}] : A_{app}$$

Nous ne pouvons pas nous contenter de vérifier `R` sur le programme, car nous voulons nous assurer que les valeurs de type `Pair`, y compris celles "intermédiaires", respectent le prédicat. Par exemple, $\texttt{R}((\lambda x : \texttt{Int} \to \texttt{pair}\ 2)\ (\texttt{pair}\ 3))$ est vrai. Demander qu'à chaque étape de son évaluation, le programme ne contiennent que des valeurs `Pair` qui respectent `Pred` n'est pas suffisant non plus. Cela autorise par exemple le terme $(\lambda x : (\texttt{Int} \to \texttt{Pair}) \to \texttt{pair}\ 2)\ (\lambda y : \texttt{Int} \to (\lambda z : \texttt{Int} \to \texttt{pair}\ z)\ 3)$. Nous souhaitons être "conservateurs", c'est à dire ne pas tenter de trouver quels termes ne seront pas évalués et peuvent être autorisés.
Rappelons que le constructeur `pair` fait partie du programme : il est interdit dans l'application, mais est incorporé à nouveau lors de l'ajout de la bibliothèque. Peut-être qu'utiliser un autre système d'exécution que la bêta réduction aurait empêché ce fait.
Pour être complet, il faut vérifier `R` sur le terme de l'application, et sur chacun de ses sous-termes. Ainsi ne se pose plus la question de savoir quels termes seront évalués ou non : tous sont vérifiés. Nous n'appliquons pas cette logique sur le programme complet : pour la partie bibliothèque, nous vérifions uniquement `R` sur

le terme "extérieur" (sinon, par exemple le terme $((\lambda x : \texttt{Int} -> \texttt{pair}\ x)\ 2)$ est refusé car la fonction seule ne respecte pas `R`). L'intérêt étant de montrer que la preuve manuelle de `R` sur la bibliothèque vaut pour le terme et les sous-termes de toute application.

2.d Prédicat sur les fonctions

Le type abstrait sur un type "atomique" comme `Int` permet uniquement de vérifier un prédicat sur les entiers. Nous pouvons imaginer des prédicats plus complexes, par exemple sur une fonction `(Int -> Int)` :

$$\{f \mid f \in (\texttt{Int} \to \texttt{Int}) \wedge \forall n : \texttt{Int},\ (f\ n) \geq n\}$$

Grâce à ces prédicats nous pouvons simuler les monades *Reader* et *Writer* avec la monade *State*. C'est à dire que nous pouvons définir une monade qui présente exactement la même interface.

$$\texttt{reader}\ (\texttt{f} : \texttt{Param} \to (\texttt{A} \times \texttt{Param})) = \begin{cases} \texttt{vrai} & \texttt{si} \quad \forall \texttt{p} : \texttt{Param},\ (\texttt{snd}\ (\texttt{f}\ \texttt{p})) = \texttt{p} \\ \texttt{faux} & \texttt{sinon} \end{cases}$$

$$\texttt{writer}\ (\texttt{f} : \texttt{Logs} \to (\texttt{A} \times \texttt{Logs})) = \begin{cases} \texttt{vrai} & \texttt{si} \quad \forall \texttt{logs1} : \texttt{Logs},\ \exists \texttt{logs2} : \texttt{Logs}, (\texttt{snd}\ (\texttt{f}\ \texttt{logs1})) = \texttt{logs1} \oplus \texttt{logs2} \\ \texttt{faux} & \texttt{sinon} \end{cases}$$

La monade *Reader* fait intervenir un argument supplémentaire, c'est à dire une valeur auxilliaire constante. Ainsi il suffit de ne pas fournir de fonction `set` dans l'interface pour simuler *Reader* avec *State* :

```
module Abs_State_Reader (Param : TYPE) : sig
    type 'a monad (* abstrait *)
    val return : 'a -> ('a monad)
    val bind : ('a -> 'b monad) -> ('a monad -> 'b monad)
    val get : (Param.t monad)
    val run : ('a monad) -> (Param.t -> 'a)
end = struct
    type 'a monad = Param.t -> ('a * Param.t)
    let return a = fun p -> (a, p)
    let bind f ma = fun p0 ->
        let (a, p1) = (ma p0) in
        (f a p1)
    let get = fun p -> (p, p)
    let run ma p = (fst (ma p))
end
```

Code 2.d.1 – *Définition et usage de la monade* Reader *implémentée avec une monade* State *abstraite*

La monade *Writer* permet à chaque valeur d'avoir une valeur auxiliaire associée, selon une structure monoïdale : il existe une valeur auxiliaire neutre, et une fonction de combinaison de valeurs auxiliaires. La fonction `bind` de *Writer* utilise cette fonction pour combiner les deux contextes, c'est à dire les deux valeurs auxiliaires. Pour simuler *Writer* avec *State*, nous fournissons une fonction `log` qui ajoute une valeur auxiliaire. La valeur auxiliaire initiale est la valeur neutre.

```
module Abs_State_Log (Log : MONOID) : sig
    type 'a monad (* abstrait *)
    val return : 'a -> ('a monad)
    val bind : ('a -> 'b monad) -> ('a monad -> 'b monad)
    val log : Log.t -> (unit monad)
    val get : Log.t monad
    val run : ('a monad) -> ('a * Log.t)
end = struct
    type 'a monad = Log.t -> ('a * Log.t)
    let return a = fun log -> (a, log)
    let bind f ma = fun log0 ->
                        let (a, log01) = ma log0 in
                        (f a log01)
    let log log2 = fun log1 -> ((), Log.(++) log1 log2)
    let get = fun log -> (log, log)
    let run ma = (ma Log.nil)
end

module S1 = Abs_State_Log (struct type t = log ... end)
open S1
let m1 : int monad = return 1
let f : int -> (int monad) = fun i -> bind (fun () -> return (i + 1)) (log "incrément")
let m2 : int monad = bind f m1
let mlog : log monad = get m2
let (i, log) : (int * log) = run m2
```

Code 2.d.2 – *Définition et usage de la monade* Log *implémentée avec une monade* State *abstraite*

Le prédicat sur fonction est nécessaire dès qu'on veut garantir le lien de construction entre l'entrée et la sortie des fonctions. Sans lui, rien n'empêche de fournir une sortie construite à la volée, qui n'a rien à voir avec l'entrée. Dans l'exemple des logs, une fonction de type `(Log -> ('a * Log))` ne doit pas pouvoir donner `Log.nil` en sortie. L'utilisation du type abstrait réduit les constructeurs du type `('a monad) := (Log -> ('a * Log))` à la bibliothèque du type abstrait, ce qui permet de forcer le lien entre l'entrée et la sortie.

Le type abstrait n'est pas l'unique solution. Il existe le typage linéaire [PZ17] qui garantit que l'argument sera utilisé exactement une seule fois dans le corps de la fonction. Nous pouvons aussi utiliser le polymorphisme, par exemple avec un type comme $\forall a.a \to a$ qui force la fonction à renvoyer l'argument si elle ne peut connaître le type de l'argument. Il est aussi possible de garantir le lien de construction en conservant explicitement les étapes de construction dans le type, dans notre exemple ce serait la séquence des transferts.

Comme exemple d'un type qui peut bénéficier du typage abstrait sur fonction, nous pouvons donner celui du générateur pseudo-aléatoire : la graine de sortie doit toujours être l'évolution de celle d'entrée.

$$\texttt{randgen}\ (\texttt{f} : \texttt{Graine} \to (\alpha \times \texttt{Graine})) = \begin{cases} \texttt{vrai} & \texttt{si} \quad \forall \texttt{g1} : \texttt{Graine},\ (\texttt{snd}(\texttt{f}\ \texttt{g1})) = \texttt{generator}^*(\texttt{g1}) \\ \texttt{faux} & \texttt{sinon} \end{cases}$$

Figure 2.d.1 – *Définition du prédicat du générateur pseudo-aléatoire*

```
module Abs_State_RandGen : sig
    type graine = int
    type 'a monad (* abstrait *)
    val return : 'a -> ('a monad)
    val bind : ('a -> 'b monad) -> ('a monad -> 'b monad)
    val get : (graine monad)
    val run : ('a monad) -> graine -> ('a * graine)
    val randgen : (int monad)
end = struct
    type graine = int
    type 'a monad = (gr -> ('a * gr))
    let return a = fun gr -> (a * gr)
    let bind f ma = fun gr0 -> let (a, gr1) = ma gr0 in (f a gr1)
    let get = fun gr -> (gr, gr)
    let run ma grinit = (ma grinit)
    let randgen = fun gr1 -> let (i, gr2) = generator gr1 in (i, gr2)
end
```

Code 2.d.3 – *Définition du type abstrait du générateur pseudo-aléatoire*

Chapitre 3

Optimisation de monades

Lorsque l'on programme un simulateur fonctionnel avec monades, un problème survient rapidement : l'exécution du programme s'arrête et une des deux erreurs *Out_Of_Memory Stack_Overflow* est affichée. Ces erreurs témoignent que le programme a eu besoin de plus d'espace mémoire qu'il ne lui en était autorisé.
Habituellement l'erreur *Stack_Overflow* est la conséquence d'une fonction récursive, à cause de laquelle trop d'appels à fonctions imbriqués ont été effectués. La solution est de modifier le code de la fonction afin d'activer l'optimisation d'appel terminal. L'erreur *Out_Of_Memory* survient lorsque trop de données doivent être stockées, par exemple une liste trop grande. Une des solutions est de se débarrasser des données dispensables, par exemple en les consommant le plus tôt possible et ainsi libérer l'espace mémoire associé.
Avec les monades le problème de *Stack_Overflow* vient toujours de la récursivité, mais il est plus discret car l'application standard est remplacée par l'application `bind` . Cette dernière n'est pas forcément terminale dans son argument fonctionnel. De plus, la valeur monadique peut être fonctionnelle, et la construction de cette valeur fonctionnelle peut engendrer des appels imbriqués non optimisés, qui risquent une *Out_Of_Memory* pour leur stockage, et une *Stack_Overflow* pour leur évaluation.
La solution pour les monades s'apparente à celle pour le problème standard. Il s'agit de forcer les appels terminaux à l'aide de fonctions de récursion bien écrites. Chaque monade doit avoir sa fonction spécialisée, et il n'est pas toujours possible d'en trouver une. Les fonction spécialisées peuvent aussi s'écrire pour un transformateur de monade.
Cependant le problème persiste lorsque le programmeur n'a pas le contrôle de la récursion. Nous utilisons alors une structure de "concret monadique", une valeur monadique dont la partie fonctionnelle a été complétée. Cette structure est également utile pour une utilisation *REPL* de la monade, "Read Eval Print Loop".

Voici le plan du chapitre :

- 3.a introduction aux erreurs *Stack_Overflow* et *Out_Of_Memory*
- 3.b utilisation des monades avec récursions
- 3.c cas des monades *State* et *Writer*
- 3.d solutions existantes imparfaites
- 3.e la bonne solution, existante mais peu répandue, non applicable en cas de récursion non contrôlée
- 3.f notre proposition de solution pour la récursion non contrôlée : les concrets monadiques
- 3.g utilisation des concrets monadiques pour *REPL*

3.a Introduction aux erreurs de taille mémoire

Nous introduisons les deux erreurs de mémoire : *Stack_Overflow* et *Out_Of_Memory*. Nous présentons le mécanisme d'appel terminal et son optimisation, en utilisant les mêmes schémas d'exécution utilisés en

introduction à la programmation fonctionnelle.

Les fonctions récursives à la place des boucles

En programmation mutable, les structures de récursion sont les boucles `for` et `while`, ainsi que les fonctions récursives. En programmation fonctionnelle, nous n'utilisons que les fonctions récursives, puisque nous ne pouvons pas modifier l'environnement d'une éventuelle structure de boucle. Voici un exemple de récursion où nous comptons le nombre d'éléments d'une liste, et le schéma d'exécution associé :

```
1 let rec taille_liste l =
2     if l = []
3     then 0
4     else let tr = taille_liste (List.tl l) in  (* appel récursif à taille_liste *)
5          (1 + tr)

6 let lc = ['a' ; 'b']
7 let tlc = (taille_liste lc)
```

Code 3.a.1 – *Code source de la fonction* `taille_liste` *sans récursion terminale*

Dans le schéma d'exécution Tableau 3.a.1, le nom `f` est un alias pour `taille_liste`. Nous rappelons que grâce au mot-clé `rec`, `f` est présent dans son propre contexte. Cela n'est pas présent dans les schémas car la syntaxe ne s'y prête pas, il faut donc le considérer comme implicite.

ligne	environnement courant	pile
init	{}	∅
1.	{ f=fun{} }	∅
2.	{ f=fun{} }	∅
3.	{ f=fun{} }	∅
4.	{ f=fun{} }	∅
5.	{ f=fun{} }	∅
6.	{ f=fun{} lc=['a';'b'] }	∅
début 7.	{ f=fun{} lc=['a';'b'] tlc=? }	∅
1.	{ l=['a';'b'] }	retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
début 4.	{ l=['a';'b'] tr=? }	retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
1.	{ l=['b'] }	retour 4. { f=fun{} lc=['a';'b'] tr=val_retour } retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
début 4.	{ l=['b'] tr=? }	retour 4. { f=fun{} lc=['a';'b'] tr=val_retour } retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
1.	{ l=[] }	retour 4. { f=fun{} lc=['b'] tr=val_retour } retour 4. { f=fun{} lc=['a';'b'] tr=val_retour } retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
3.	{ l=[] val_retour=0 }	retour 4. { f=fun{} lc=['b'] tr=val_retour } retour 4. { f=fun{} lc=['a';'b'] tr=val_retour } retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 4.	{ f=fun{} lc=['b'] tr=0 }	retour 4. { f=fun{} lc=['a';'b'] tr=val_retour } retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
5.	{ f=fun{} lc=['b'] tr=0 val_retour=1 }	retour 4. { f=fun{} lc=['a';'b'] tr=val_retour } retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 4.	{ f=fun{} lc=['a';'b'] tr=1 }	retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
5.	{ f=fun{} lc=['a';'b'] tr=1 val_retour=2 }	retour 7. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 7.	{ f=fun{} lc=['a';'b'] tlc=2 }	∅

Tableau 3.a.1 – *Schéma d'exécution de* `taille_liste` *sans récursion terminale*

Chaque appel récursif crée un élément dans la pile. Lorsque la fin de la liste est atteinte, `taille_liste` renvoie la valeur `0`, puis chaque appel est résolu jusqu'à donner la valeur finale `2` à `tlc`.

L'erreur *Stack_Overflow*

Pendant l'exécution du programme, la taille de la pile ne peut dépasser un maximum défini. Ce maximum peut être augmenté, mais il est borné par la capacité des composants physiques tels que la *RAM*. Si le programme tente de dépasser ce maximum, en allouant un élément supplémentaire sur la pile, il est arrêté et affiche une erreur nommée *Stack_Overflow*.

Le maximum de taille de pile borne le nombre d'appels de fonctions imbriqués. En effet chaque appel de fonction crée un élément sur la pile, qui est supprimé à la fin de l'appel.

Les deux concepts qui peuvent générer une erreur *Stack_Overflow*

En théorie, l'erreur *Stack_Overflow* peut arriver dans un programme qui ne contient pas de fonction récursive, comme l'exemple suivant :

```
let f1 x = x in
let f2 x = (f1 x) in
let f3 x = (f2 x) in
...
```

En pratique, vu le nombre d'éléments de pile nécessaires pour générer une erreur *Stack_Overflow*, un tel exemple est peu probable, et provoquera peut-être une erreur de compilation avant même une erreur d'exécution.

Les deux concepts qui peuvent générer une erreur *Stack_Overflow* sont :
- les fonctions récursives
- la création dynamique de fonctions imbriquées

Solution : optimisation d'appel terminal

Il existe une optimisation du compilateur qui permet d'éviter l'allocation d'un élément de pile pour un appel de fonction. Cela concerne les appels de fonctions dits "terminaux". Un appel terminal est la dernière action d'une fonction, et est un appel à une fonction (récursive ou non). Dans l'exemple de code suivant, l'appel à `h` est terminal dans la fonction `f`, mais pas l'appel à `g` :

```
let f x = h (g x)

let res = (f 10)
```

Voici la fonction `f'` qui est équivalente à la fonction `f` précédente mais est plus explicite sur son exécution :

```
let f' x =
    let y = (g x) in
    let z = (h y) in
    z

let res = (f' 10)
```

Lors de l'exécution de `f'`, l'appel à `(h y)` génère un élément de pile qui sauvegarde l'environnement courant de `f'`, ici la valeur associée à `x` et à `y`. Cet environnement est rétabli lorsque `(h y)` termine. Mais comme `(h y)` est la dernière action de `f'`, il n'y a plus d'action à effectuer. Il ne reste qu'à retourner la valeur `z` à l'appelant de `f'`. Nous avons donc sauvegardé et rétabli un environnement pour rien. L'optimisation d'appel terminal repère cette situation et prévient la génération d'élément de pile superflu. Le retour de `(h y)` sera directement le retour à l'appelant de `f'`.

Exemple d'optimisation d'appel terminal : calculer la taille d'une liste

Dans le premier exemple de la taille de liste, l'optimisation d'appel terminal n'est pas applicable car l'appel à `taille_liste` n'est pas terminal : la dernière action de la fonction `taille_liste` est le calcul de

`(1 + tr)` .

Il faut réécrire `taille_liste` de façon à ce que l'appel récursif à `taille_liste` soit terminal. Pour cela nous ajoutons un argument supplémentaire qui stocke le résultat courant (souvent nommé "accumulateur") :

```
1 let rec taille_liste l tr =
2     if l = []
3     then tr
4     else (taille_liste (List.tl l) (1 + tr))   (* appel terminal à taille_liste *)

5 let lc = ['a'; 'b']
6 let tlc = (taille_liste lc 0)
```

Code 3.a.2 – *Code source de* `taille_liste` *avec récursion terminale*

Avec cette nouvelle fonction récursive terminale, sans activer l'optimisation du compilateur, voici comment se déroule l'exécution :

ligne	environnement courant	pile
init	{}	∅
1.	{ f=fun{} }	∅
2.	{ f=fun{} }	∅
3.	{ f=fun{} }	∅
4.	{ f=fun{} }	∅
5.	{ f=fun{} lc=['a';'b'] }	∅
début 6.	{ f=fun{} lc=['a';'b'] tlc=? }	∅
1.	{ l=['a';b'] tr=0 }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
début 4.	{ l=['a';'b'] tr=0 val_retour=? }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
1.	{ l=['b'] tr=1 }	retour 4. { l=['a';'b'] tr=0 val_retour=val_retour } retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
début 4.	{ l=['b'] tr=1 val_retour=? }	retour 4. { l=['a';'b'] tr=0 val_retour=val_retour } retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
1.	{ l=[] tr=2 }	retour 4. { l=['b'] tr=1 val_retour=val_retour } retour 4. { l=['a';'b'] tr=0 val_retour=val_retour } retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
3.	{ l=[] tr=2 val_retour=2 }	retour 4. { l=['b'] tr=1 val_retour=val_retour } retour 4. { l=['a';'b'] tr=0 val_retour=val_retour } retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 4.	{ l=['b'] tr=1 val_retour=2 }	retour 4. { l=['a';'b'] tr=0 val_retour=val_retour } retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 4.	{ l=['a';'b'] tr=0 val_retour=2 }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 6.	{ f=fun{} lc=['a';'b'] tlc=2 }	∅

Tableau 3.a.2 – *Schéma d'exécution de* `taille_liste` *avec récursion terminale, sans l'optimisation*

Notons la présence de `val_retour = val_retour` qui montre l'inutilité de ces éléments de pile. Avec l'optimisation activée, l'exécution ne s'en encombre plus :

ligne	environnement courant	pile
init	{}	∅
1.	{ f=fun{} }	∅
2.	{ f=fun{} }	∅
3.	{ f=fun{} }	∅
4.	{ f=fun{} }	∅
5.	{ f=fun{} lc=['a';'b'] }	∅
début 6.	{ f=fun{} lc=['a';'b'] tlc=? }	∅
1.	{ l=['a';b'] tr=0 }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
début 4.	{ l=['a';'b'] tr=0 val_retour=? }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
1.	{ l=['b'] tr=1 }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour } *optimisation*
début 4.	{ l=['b'] tr=1 val_retour=? }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
1.	{ l=[] tr=2 }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour } *optimisation*
3.	{ l=[] tr=2 val_retour=2 }	retour 6. { f=fun{} lc=['a';'b'] tlc=val_retour }
fin 6.	{ f=fun{} lc=['a';'b'] tlc=2 }	∅

Tableau 3.a.3 – *Schéma d'exécution de* `taille_liste` *avec récursion terminale, avec l'optimisation*

Notons la disparition des éléments de pile de ligne `4` et le retour direct à la ligne `6`. Bien entendu, le calcul des arguments `(List.tl l)` et `(1 + tr)` doit se résoudre avant l'appel récursif, mais nous l'avons omis ici car il n'a pas d'impact sur la question *Stack_Overflow*. En effet ces appels sont créés puis consommés avant l'appel récursif.

Optimisation d'appel terminal et associativité

Si le code de `taille_liste` a pu être réécrit avec un appel terminal, c'est parce que l'action `(1 + tr)` est associative. C'est à dire que `(a + (b + _))` est égal à `((a + b) + _)` : sans même connaître la troisième opérande, nous pouvons associer les opérations de façon à disposer d'un groupe constitué d'opérandes connues. L'opération `(a + b)` est calculée et son résultat stocké dans l'argument supplémentaire de la fonction, l'accumulateur. Puisque nous avons calculé tout ce qui était à calculer avant l'appel récursif, il n'y a plus rien à faire au retour de cet appel, et cela active l'optimisation d'appel terminal. La modification de l'argument supplémentaire est manuelle, elle n'est pas effectuée automatiquement par le compilateur.

Notons que nous pouvons toujours accumuler les opérandes, sans les calculer. Lorsque la dernière opérande est atteinte, le calcul se fait. Il suffit de construire une structure de donnée "à trou", comme nous l'avons schématisé par `(a + (b + _))`, par exemple en utilisant le mécanisme de "passage par continuation". Attention néanmoins, car si la construction de la structure à trou se fait pendant l'exécution, elle nécessitera un espace mémoire pour chaque opérande à accumuler.

Fonction générique de récursion optimisée

Nous pouvons définir une fonction de récursion pour laquelle la récursion est optimisée. Cela permet de ne pas multiplier les fonctions récursives à vérifier.

```
val std_while : ('a -> bool) -> ('a -> 'a) -> ('a -> 'a)
let rec std_while test f =
    fun a ->
    if (test a)
    then (std_while test f (f a))
    else a
```

Code 3.a.3 – *Définition de la fonction de récursion* `std_while`

Voici l'exemple de `taille_liste` réécrit avec `std_while` :

```
let taille_liste li =
    snd (std_while (fun (li, t) -> li <> [])
                   (fun (li, t) -> (List.tl li, 1 + t))
                   (li, 0))
```

Voici également la fonction `std_for`, dont le nombre de répétitions est donné en argument :

```
val std_for : int -> ('a -> 'a) -> ('a -> 'a)

let std_for n f a = fst (std_while (fun (a, n) -> n > 0)
                                   (fun (a, n) -> (f a, n - 1))
                                   (a, n))
```

Code 3.a.4 – *Définition de la fonction de récursion* `std_for`

L'erreur *Out_Of_Memory*

L'erreur *Out_Of_Memory* est générée lorsque le programme tente d'allouer un espace de mémoire supplémentaire alors qu'il a déjà atteint la limite maximale (bornée par les capacités matérielles de l'ordinateur). Dans nos schémas, cela concerne l'environnement courant, ainsi que les environnements contenus dans la pile.[1] Un exemple de situation qui peut générer une *Out_Of_Memory* est la création d'une liste avec un très grand nombre d'éléments. Un autre exemple est la création dynamique d'un nombre trop élevé de fonctions, possible en programmation fonctionnelle :

```
1 let (>>) f g =
2     (fun x -> g (f x))

3 let f x =
4     (x + 1)

5 let rec loop acc =
6     let acc = (acc >> f) in
7     (loop (acc >> f))

8 let fff = loop f
```

Code 3.a.5 – *Exemple de création dynamique de compositions de fonctions*

1. Les environnements dans la pile appartiennent à un concept dû à notre schématisation simplifiée de l'exécution. En réalité un élément de pile ne contient pas de liste, par exemple. Ainsi l'erreur *Stack_Overflow* tient plus du nombre d'éléments dans la pile que de leur taille.

ligne	environnement courant	pile
init	{}	∅
1.	{ (»)=fun{} }	∅
2.	{ (»)=fun{} }	∅
3.	{ (»)=fun{} f=fun{} }	∅
4.	{ (»)=fun{} f=fun{} }	∅
5.	{ (»)=fun{} f=fun{} loop=fun{} }	∅
6.	{ (»)=fun{} f=fun{} loop=fun{ f=fun{} } }	∅
7.	{ (»)=fun{} f=fun{} loop=fun{ f=fun{} } }	∅
début 8.	{ (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=? }	∅
5.	{ acc=f }	retour 8. { (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=val_retour }
6.	{ acc=(fun x -> f (f x)) }	retour 8. { (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=val_retour }
début 7.	{ acc=(fun x -> f (f x)) val_retour=? }	retour 8. { (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=val_retour }
5.	{ acc=(fun x -> f (f x)) }	retour 8. { (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=val_retour }
6.	{ acc=(fun x -> f ((fun x -> f (f x)) x)) }	retour 8. { (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=val_retour }
…	acc *de plus en plus grand*	retour 8. { (»)=fun{} f=fun{} loop=fun{ f=fun{} } fff=val_retour }

Tableau 3.a.4 – *Résultat de test du Code 3.a.5*

Tests de calcul de taille de liste

Voici pour appuyer les propos précédents des tests de programmes de taille de liste. Quelques précisions :

- les tests comprennent la construction de la liste
- l'erreur *Stack_Overflow* est abrégée en `SO` et *Out_Of_Memory* en `OOM`
- le nombre `n` affiché en tête de tableau est le nombre d'appels de fonction effectués pendant le test. Ici ce nombre est égal au nombre d'éléments dans la liste. Nous utilisons la convention `K` pour mille, `M` pour million et `G` pour milliard
- les tailles pour l'environnement et la pile sont toujours en bytes (égaux à des octets, 8 bits) et en base 10, jamais en base 2
- la taille de l'environnement est fixée à 3 gigabytes quand nous faisons varier la taille de la pile
- la taille de la pile est fixée à 1 gigabyte quand nous faisons varier la taille de l'environnement
- l'optimisation d'appel terminal est activée
  Test de calcul de taille de pile, sans appel terminal :

taille de pile	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.03s	0.48s	4.31s (SO)	5.03s (OOM)	5.00s (OOM)
5G	0.00s	0.03s	0.47s	4.31s (SO)	5.00s (OOM)	5.00s (OOM)
1G	0.00s	0.03s	0.48s	4.33s (SO)	5.00s (OOM)	5.04s (OOM)
100M	0.00s	0.04s	0.41s (SO)	4.41s (SO)	5.23s (OOM)	5.25s (OOM)
10M	0.00s	0.03s (SO)	0.40s (SO)	4.32s (SO)	4.97s (OOM)	5.02s (OOM)

L'erreur `OOM` est dûe à la taille trop grande de la liste à analyser. Dans le reste du tableau nous observons l'erreur `SO` qui apparaît de plus en plus tôt lorsque nous réduisons le maximum de taille de pile. Le tableau suivant, avec l'appel terminal, montre que de tels besoins de pile sont dispensables.

Test de calcul de taille de pile, avec appel terminal :

taille de pile	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.03s	0.42s	4.41s	4.99s (OOM)	5.03s (OOM)
5G	0.00s	0.03s	0.41s	4.41s	5.00s (OOM)	5.00s (OOM)
1G	0.00s	0.03s	0.41s	4.43s	5.01s (OOM)	5.24s (OOM)
100M	0.00s	0.03s	0.43s	4.61s	5.24s (OOM)	5.21s (OOM)
10M	0.00s	0.03s	0.43s	4.61s	5.21s (OOM)	5.20s (OOM)

Tableau 3.a.5 – *Résultats d'exécution de taille de pile avec appel terminal et optimisation activée*

3.b Monades et récursions

Nous étudions maintenant l'utilisation des monades dans un programme et les relations avec la récursion. Pour cela nous allons nommer une série de concepts et de propriétés qui nous serviront dans la suite du chapitre.

Récursion gauche et droite

Le programme est une séquence d'applications monadiques `bind`. Nous distinguons des sous-séquences dites "imbriquées à gauche" ou "à droite". Voici un schéma montrant une imbrication à gauche et une à droite :

```
((((ma >>= f) >>= g) >>= h) >>= k)     (* à gauche *)

(ma >>= (fun a ->
    (f a) >>= (fun b ->                (* à droite *)
          (g b) >>= (fun c ->
                (h c) >>= k ))))
```

Code 3.b.1 – *Imbrication à gauche et à droite avec* `bind`

Par la règle d'associativité des monades, les deux imbrications sont équivalentes. Cependant, notons que l'imbrication à droite permet l'accès aux valeurs intermédiaires `a`, `b` et `c`. Cela permet par exemple d'écrire le code suivant :

```
ma >>= (fun a ->
    (f a) >>= (fun b ->
        return (a, b) ))
```

Sur le plan de la consommation mémoire, l'équivalence n'est pas garantie car l'imbrication à gauche peut se faire étape par étape, comme dans le code suivant :

```
let mb = (ma >>= f) in
let mc = (mb >>= g) in
(mc >>= h)
```

Cela prévient les erreurs *Stack_Overflow* pour l'application de `bind`. En revanche en application à droite nous ne pouvons pas découper l'argument fonctionnel du `bind`. En conséquence, si le `bind` n'est pas terminal dans son argument fonctionnel, nous risquons une erreur *Stack_Overflow*.

Points de vue interne et externe

Nous distinguons deux points de vue lors de l'utilisation d'une monade.

Le point de vue "interne" est celui qui s'inscrit dans les règles de la monade. C'est une valeur lue dans un contexte monadique. C'est une fonction `f` de type `('a -> 'b monad)` appliquée à `bind`. Par opposition, le point de vue "externe" manipule une valeur monadique "de l'extérieur". C'est une valeur extraite d'une valeur monadique sans passer par `bind`. C'est une fonction de type `('a monad -> 'b monad)`. Donnons quelques exemples.

Avec la monade *List*, lire la valeur avec `List.head` est une utilisation externe de `ma` :

```
val ma : 'a list_monad

let first_a = (List.head ma)
```

En revanche, l'argument `a` dans le code suivant est interne, puisqu'il provient de l'application de `bind` :

```
val ma : 'a list_monad

let maa = ma >>= (fun a -> [ a ; a + 1 ])
```

La fonction suivante est externe, puisqu'elle manipule directement la liste des messages, au lieu de laisser cette tâche à `bind` :

```
val incrementer : (int writer_monad) -> (int writer_monad)

let incrementer (n, msgs) = (n + 1, "Incrément" :: (List.tail msgs))
```

Les deux points de vue sont utiles dans le programme. Le point de vue interne permet d'utiliser les concepts de la monade dans leur fonctionnement standard, par exemple écrire des logs, lire la valeur auxiliaire, etc. Il permet de déléguer la gestion des concepts `bind`. Le point de vue externe permet de sortir du contexte. Cela peut consister en une lecture, par exemple pour la monade *Writer* il s'agit de lire tous les messages. Cela peut aussi être nécessaire pour compenser un mécanisme absent du `bind`, par exemple vider la liste des messages. Le point de vue externe peut être indispensable : par exemple pour la monade *State* c'est la seule façon de lancer le calcul.

Fonctions génériques de récursion `m_while` et `m_for`

Nous allons étudier une fonction générique de récursion monadique. Le type de la fonction à répéter est `('a -> 'a monad)`. Le domaine n'est pas monadique, sinon nous utiliserions simplement `std_while`. La fonction est appliquée avec `bind`. Les deux questions sont : dans quel sens imbriquer, et quel est le type de la condition d'arrêt ?

Commençons par une condition d'arrêt externe, c'est à dire de type `('a monad -> bool)`. Le sens d'imbrication est alors à gauche, c'est à dire que nous sortons de la monade entre chaque étape de récursion :

```
val m_while_ext : ('a monad -> bool) -> ('a -> 'a monad) -> ('a -> 'a monad)

let m_while_ext test f a = std_while test (bind f) (return a)
```

Code 3.b.2 – *Définition basique de* `m_while_ext`

Symmétriquement, avec une condition d'arrêt interne, le sens d'imbrication est à droite :

```
val m_while_int : ('a -> bool) -> ('a -> 'a monad) -> ('a -> 'a monad)

let rec m_while_int test f a =
    if (test a)
    then (f a) >>= (m_while_int test f)
    else (return a)
```

Code 3.b.3 – *Définition basique de* `m_while_int`

Impossible d'imbriquer à gauche, car nous ne pouvons sortir `'a` de la monade de façon générique, donc nous ne pouvons sortir `bool` de la monade de façon générique.

Les deux fonctions `m_while_ext` et `m_while_int` n'ont pas la même expressivité. Le point de vue extérieur donne la valeur complète monadique. Dans le cas de la monade `List`, cela donne directement l'ensemble des valeurs actuelles de la liste. Traduire un prédicat sur l'ensemble de la liste en un prédicat sur un seul

élément demande du travail supplémentaire : il faut ajouter à chaque élément de la liste une information supplémentaire qui permet d'exprimer l'arrêt, et organiser le tout afin de respecter le prédicat sur la liste complète. Si cela peut être trivial dans certains cas, c'est un travail non négligeable dans d'autres, au point que nous ne pouvons pas conclure que les deux points de vue sont équivalents.
Dans l'autre sens, nous pouvons donner l'exemple de la monade *State*. Un prédicat sur la valeur `'a` n'est pas trivialement traduisible en un prédicat sur la valeur `('a state)`, qui contient une inconnue dont dépend `'a`. Il est vrai que dans l'implémentation de `m_while_int/ext`, le premier "effet" monadique est celui de `return`, qui est neutre sur `'a`. Cependant, la fonction répétée peut utiliser la valeur auxiliaire, et c'est sur ce point qu'il faudra être capable d'ajuster. Le plus douteux sera le test entre chaque répétition, qui devra fournir une valeur auxiliaire de départ.
En résumé, passer d'une fonction à l'autre nécessite à la fois des calculs de traduction, et à la fois des mécanismes de contournement des effets de la monade, ce qui nous permet de conclure que les deux fonctions n'ont pas la même expressivité. Voici deux exemples de fonctions, triviales mais suffisantes pour illustrer :

```
(* monade List *)
m_while_ext (fun la -> List.size la < 100)
            (fun a -> if a mod 2 then [ a ; a ] else [ a ; a ; a ])

(* monade State *)
m_while_int (fun a -> a < 100) (fun a aux -> (aux, aux + 1))
```

Nous pouvons aussi utiliser un `m_while` interne plus complet avec une condition d'arrêt de type `('a -> bool monad)`.

Lorsque la condition n'a à voir ni avec `'a` ni avec `('a monad)`, l'imbrication peut se faire à gauche ou à droite, par exemple la fonction `m_for` :

```
val mfor : int -> ('a -> 'a monad) -> ('a -> 'a monad)

let mfor_l n f a = std_for n (bind f) (return a)

let rec mfor_r n f a =
    if (n > 0)
    then (f a) >>= (mfor_r (n -1) f)
    else (return a)
```

Notons qu'il existe un type de récursion appelé "récursion de valeur monadique". Celui-ci est différent car il ne cumule pas les effets monadiques, contrairement aux fonctions que nous venons de voir. Quelques explications sur cette récursion sont données en annexe D.1.

Non-contrôle de la récursion

Les fonctions précédentes supposent le contrôle de la récursion, c'est à dire du mot-clé `rec`, ou bien la disposition d'une fonction telle que `std_fix`. Il existe des cas où le programmeur ne contrôle pas la récursion, c'est à dire qu'il doit fournir une fonction, par exemple de type `('a -> 'a)`, qui sera répétée pour lui par un code déjà écrit. Prenons par exemple le parcours de structures de données, par la fonction `fold : ('elt -> 'acc -> acc) -> ('elt 'struct) -> ('acc -> 'acc)`. La fonction est généralement déjà définie dans la bibliothèque de la structure, ainsi il n'est pas satisfaisant de la réécrire manuellement avec `rec`. Il faut donc accepter d'utiliser `fold` et de fournir une fonction de type `('elt -> 'acc -> 'acc)`, qui sera répétée à chaque élément de la structure. Quand l'accumulateur est monadique, cela donne une imbrication à gauche, puisque nous sortons de la monade à chaque étape. L'absence de contrôle pose donc un problème pour les monades qui ne sont pas efficaces en imbrication à gauche, comme *State*.

3.c Cas des monades *State* et *Writer*

Étudions la consommation des monades *State* et *Writer* avec plusieurs types de récursion (interne, externe).

La monade *State* doit s'imbriquer à droite

```
let bind f ma =
    fun aux0 ->
        let (a, aux1) = (ma aux0) in
        (f a aux1)
```

Code 3.c.1 – *Rappel de* `bind` *de* State

La principale caractéristique du `bind` de *State* est qu'il crée une nouvelle fonction d'argument `aux0` . Imbriquer à gauche multiplie donc la création de fonctions, ce qui prend de l'espace mémoire. En témoignent le code et le tableau de tests suivants :

```
let m_for_l n f a =
    let rec loop i ma =
        if i < n
        then loop (i + 1) (bind f ma)
        else ma
    in
    loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = fun i -> (a, i + 1)

let bigFunction = m_for_l n f 'a'
```

Code 3.c.2 – *Test d'imbrication à gauche de* State, *sans évaluation de la composition*

taille de l'environnement	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.05s	0.63s	6.95s	16.55s (OOM)	16.46s (OOM)
5G	0.00s	0.05s	0.61s	6.79s	8.05s (OOM)	8.13s (OOM)
1G	0.00s	0.05s	0.62s	1.51s (OOM)	1.50s (OOM)	1.49s (OOM)
100M	0.00s	0.05s	0.12s (OOM)	0.12s (OOM)	0.12s (OOM)	0.12s (OOM)

Tableau 3.c.1 – *Résultat de test d'imbrication à gauche de* State, *sans évaluation de la composition*

De plus, ces fonctions sont composées à gauche, ce qui signifie que pour évaluer la composition avec un argument, il faut tout d'abord remonter toute la composition jusqu'à la première fonction, pour lui donner l'argument. Cette remontée peut provoquer une erreur *Stack_Overflow*, comme en témoignent le code et le tableau de tests suivants :

```
...
let bigFunction = m_for_l n f 'a'
let res = (bigFunction 0)
```

Code 3.c.3 – *Test d'imbrication à gauche de* State, *avec évaluation de la composition*

taille de pile	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.10s	4.22s	4.70s (OOM)	4.68s (OOM)	4.69s (OOM)
5G	0.00s	0.09s	4.21s	4.72s (OOM)	4.69s (OOM)	4.71s (OOM)
1G	0.00s	0.09s	4.22s	4.79s (OOM)	4.83s (OOM)	4.87s (OOM)
100M	0.00s	0.10s	0.65s (SO)	4.88s (OOM)	4.87s (OOM)	4.82s (OOM)
10M	0.00s	0.05s (SO)	0.64s (SO)	4.83s (OOM)	4.84s (OOM)	4.84s (OOM)

Tableau 3.c.2 – *Résultat de test d'imbrication à gauche de* State, *avec évaluation de la composition*

Dans ce dernier tableau, les erreurs `OOM` arrivent plus tôt car quand nous varions la borne de la taille de la pile, celle de l'environnement est fixée à `3G`.

En imbrication à droite, la consommation est constante. La récursion est bloquée par l'attente de `aux0` , donc la construction des fonctions est elle aussi bloquée. Lorsque `aux0` est reçu, l'appel à `f` est terminal, donc il n'y a pas de création d'élément de pile. En témoignent le code et le tableau de tests suivants :

```
let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a = fun i -> (a, i + 1)
let bigFunction = m_for_r n f 'a'
let res = bigFunction 0
```

Code 3.c.4 – *Test d'imbrication à droite de* State, *avec évaluation de la composition*

taille de l'environnement	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.00s	0.08s	0.87s	4.33s	8.68s
5G	0.00s	0.00s	0.08s	0.87s	4.30s	8.61s
1G	0.00s	0.00s	0.08s	0.85s	4.29s	8.59s
100M	0.00s	0.00s	0.08s	0.85s	4.28s	8.60s

Tableau 3.c.3 – *Test d'imbrication à droite de* State, *variation de borne d'environnement*

taille de pile	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.00s	0.08s	0.89s	4.46s	8.89s
5G	0.00s	0.00s	0.08s	0.88s	4.44s	8.87s
1G	0.00s	0.00s	0.08s	0.88s	4.45s	8.90s
100M	0.00s	0.00s	0.08s	0.88s	4.46s	8.88s
10M	0.00s	0.00s	0.08s	0.88s	4.46s	8.94s

Tableau 3.c.4 – *Test d'imbrication à droite de* State, *variation de borne de pile*

Pour être économique, la monade *State* doit donc s'imbriquer à droite, c'est à dire utiliser une récursion interne.

La monade *Writer* doit s'imbriquer à gauche

```
let bind f ma =
    let (a, msgsA) = ma in
    let (b, msgsB) = (f a) in
    (b, Log.combine msgsA msgsB)
```

Code 3.c.5 – *Rappel du* `bind` *de* Writer

La principale caractéristique du `bind` de *Writer* est qu'il cumule les messages des deux arguments, il n'est pas terminal avec son argument fonctionnel. Ainsi, en imbrication à droite, cela risque une erreur *Stack_Overflow*, car l'argument fonctionnel est récursif. En témoignent le code et le tableau de tests suivants :

```
let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a = (a, 1)
let res = m_for_r n f 'a'
```

Code 3.c.6 – *Test d'imbrication à droite de* Writer

taille de l'environnement	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.19s	13.46s	6m22s (SO)	6m20s (SO)	6m22s (SO)
5G	0.00s	0.19s	13.21s	6m20s (SO)	6m18s (SO)	6m17s (SO)
1G	0.00s	0.19s	13.25s	50.38s (SO)	51.39s (SO)	50.09s (SO)
100M	0.00s	0.19s	0.50s (OOM)	0.49s (OOM)	0.49s (OOM)	0.49s (OOM)

Tableau 3.c.5 – *Résultat de test d'imbrication à droite de* Writer, *variation de borne d'environnement*

taille de pile	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.19s	13.22s	6m55s (OOM)	6m54s (OOM)	6m55s (OOM)
5G	0.00s	0.19s	13.29s	6m59s (OOM)	7m1s (OOM)	6m57s (OOM)
1G	0.00s	0.19s	13.34s	6m22s (SO)	6m19s (SO)	6m21s (SO)
100M	0.00s	0.19s	4.16s (SO)	4.15s (SO)	4.16s (SO)	4.15s (SO)
10M	0.00s	0.07s (SO)	0.07s (SO)	0.07s (SO)	0.07s (SO)	0.07s (SO)

Tableau 3.c.6 – *Résultat de test d'imbrication à droite de* Writer, *variation de borne de pile*

En imbrication à gauche, l'absence d'appel terminal ne pose pas de problème, étant donné que la fonction répétée est calculée à chaque étape. En témoignent le code et le tableau de tests suivants :

```
let m_for_l n f a =
    let rec loop i ma =
        if i < n
        then loop (i + 1) (bind f ma)
        else ma
    in
    loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = (a, 1)
let res = m_for_l n f 'a'
```

Code 3.c.7 – *Test d'imbrication à gauche de* Writer

taille de pile	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.00s	0.04s	0.40s	2.04s	4.08s
5G	0.00s	0.00s	0.04s	0.40s	2.04s	4.08s
1G	0.00s	0.00s	0.04s	0.40s	2.04s	4.07s
100M	0.00s	0.00s	0.04s	0.40s	2.03s	4.07s
10M	0.00s	0.00s	0.04s	0.40s	2.04s	4.08s

Tableau 3.c.7 – *Résultat de test d'imbrication à gauche de* Writer

Le tableau est le même pour les tests d'environnements.

Pour être économique, la monade *Writer* doit donc s'imbriquer à gauche, c'est à dire utiliser une récursion externe.

Résumé du problème

Nous venons de voir que selon la monade utilisée, pour faire de la récursion économique il faut imbriquer à gauche ou à droite. Autrement dit il n'existe pas de méthode générique, économique pour n'importe quelle monade. Cependant nous avons vu que les monades à type fonctionnel (*State*, *Reader*) tendent à avoir des soucis en imbrication à gauche, à cause du cumul des suspensions et de la pile d'appel. Aussi, les monades qui doivent combiner les deux arguments (*Writer*) tendent à avoir des soucis en imbrication à droite, à cause de l'appel non terminal effectué avant la fin de la combinaison.

3.d Solutions existantes

Dans cette partie nous étudions des solutions qui sont proposées et utilisées dans plusieurs langages tels que *Haskell*, *Scala*, qui utilisent aussi les monades. Nous verrons que les meilleures d'entre elles répondent au problème de *Stack_Overflow*, mais pas au problème *Out_Of_Memory*.

Le passage par continuation pour imbriquer à droite de façon externe

En utilisant le "passage par continuation", nous obtenons une structure "à trou" qui permet de composer à droite "depuis l'extérieur". Voici la version monadique de ce mécanisme :

```
type ('a,'r) monad_cont = (('a -> 'r) -> 'r)

val return_cont : 'a -> ('a -> 'r) -> 'r
let return_cont a suiteDeA = (suiteDeA a)

val bind_cont : ('a -> ('b -> 'r) -> 'r) -> (('a -> 'r) -> 'r) -> (('b -> 'r) -> 'r)

let bind_cont f ca =
    fun suiteDeB ->
        ca (fun a -> f a suiteDeB)
```

Code 3.d.1 – *Définition de la monade* Continuation *en* OCaml

Nous donnons un exemple dans lequel nous nous appliquons à bien placer la continuation (`suite`) dans l'argument fonctionnel de `bind_state` , afin de générer une imbrication à droite :

```
let f a s = (a, s + 1)

let csta = m_for_l
              (1000 * 1000)
              (fun sta suite -> bind_state (fun a -> suite (f a)) sta)
              (return_cont (return_state 'a'))

let sta = csta (fun sta -> sta)
let (a, s) = (sta 0)
```

Code 3.d.2 – *Test d'utilisation de* Continuation *afin d'imbriquer* State *à gauche de façon efficace*

C'est peut-être plus clair avec le schéma d'exécution suivant, dans lequel nous nous autorisons à évaluer à l'intérieur des fonctions, à condition que ça ne corrige pas d'éventuelle *Stack_Overflow*, et que ça ne change pas l'ordre d'évaluation :

```
bind_ct (fun sta sui -> bind_st (fun a -> sui (f a)) sta) csta
-------------------------------------------------------------------------------
fun sui -> csta (fun sta -> (fun sta sui -> bind_st (fun a -> sui (f a)) sta) sta sui)
-------------------------------------------------------------------------------
fun sui -> csta (bind_st (fun a -> sui (f a)))


(* si nous répétons avec le résultat précédent *)

fun sui -> (fun sui -> csta (bind_st (fun a -> sui (f a)))) (bind_st (fun a -> sui (f a)))
-------------------------------------------------------------------------------------------
fun sui -> csta (bind_st (fun a -> (bind_st (fun a -> sui (f a))) (f a)))
```

Code 3.d.3 – *Schéma d'évaluation du Code 3.d.2*

Cette technique permet d'éviter l'erreur *Stack_Overflow* lors de l'évaluation de la valeur monadique *State*. En revanche, cela requiert de stocker la composition en mémoire, ce qui peut risquer l'erreur *Out_Of_Memory*.

taille de l'environnement	**n = 100K**	**n = 1M**	**n = 10M**	**n = 100M**	**n = 500M**	**n = 1G**
10G	0.00s	0.05s	0.74s	8.09s	17.03s (OOM)	17.51s (OOM)
5G	0.00s	0.05s	0.75s	8.21s	8.49s (OOM)	8.47s (OOM)
1G	0.00s	0.05s	0.74s	1.46s (OOM)	1.46s (OOM)	1.47s (OOM)
100M	0.00s	0.06s	0.12s (OOM)	0.12s (OOM)	0.11s (OOM)	0.12s (OOM)

Tableau 3.d.1 – *Résultat de test du Code 3.d.2*

Quitte à avoir une solution spécialisée à *State*, il existe une meilleure solution. Cependant le passage par continuation reste utile en cas de non contrôle de la récursion.

Trampoline pour pallier l'absence d'optimisation du langage

Le trampoline est une construction permettant de pallier l'absence d'optimisation d'appel terminal dans le langage. Par exemple en *Java*, l'optimisation ne fonctionne que sur les appels terminaux récursifs, pas sur les appels terminaux mutuellement récursifs par exemple. Le principe est de stocker l'appel terminal dans une structure de donnée. L'évaluation se fait alors à l'aide d'une boucle (ou tout mécanisme optimisé) qui résoud la structure. Au lieu d'effectuer l'appel terminal, le principe est de le stocker dans une structure temporaire `Thunk` dont le but est de suspendre l'exécution d'un terme momentanément.

```
type 'a trampoline = Value of 'a | Thunk of (unit -> 'a trampoline)

val eval : 'a trampoline -> 'a

let eval ta =
    let rta = ref ta in
    let roa = ref None in
    let continuer = ref true in
    while ! continuer do
        match ! ta with
        | Value a -> (roa := Some a) ;
                     (continuer := false)
        | Thunk f -> (rta := f ())
    done ;
    let (Some a) = ! roa in
    a

let rec fact n res =
    if n <= 1
    then Value res
    else Thunk (fun () ->
                let res2 = (n * res) in
                let n2   = (n - 1) in
                fact n2 res2)
```

Code 3.d.4 – *Définition d'un mécanisme de trampoline en* OCaml

Le trampoline tel quel ne nous aide pas à résoudre nos problèmes de mémoire, puisqu'il ne peut pas faire plus que simuler l'optimisation d'appel terminal. Mais il introduit le concept de la composition d'une fonction dans une structure de données manipulable. Le concept peut être approfondi[2] et mène à la monade *Freer*.

2. Dans "Stackless Scala with Free monads" [Bja12], Bjarnason montre une structure de trampoline capable de stocker des `bind` en attente, et de les réimbriquer dynamiquement à droite au moment de l'évaluation.

La monade *Freer* pour séparer construction et évaluation

La monade *Freer* [KI15] s'appuie sur la séparation entre construction et évaluation. C'est comme *State*, mais le stockage des fonctions se fait dans une structure de données manipulable. La fonction `bind` se contente d'accumuler des "actions" en les imbriquant à droite, aucun calcul n'est réalisé. Toute "l'intelligence" est placée dans une fonction `run`, qui est écrite de façon récursive terminale (avec un trampoline si besoin). Voici une définition de *Freer* avec un exemple d'actions *State* :

```
module MFreer (Action : TYPE) = struct
    type _ monad =
    | Return : 'a -> 'a monad
    | Action : 'a Action.t * ('a -> 'b monad) -> 'b monad

    let return a = Return a

    let rec bind g mb =
        match mb with
        | Return b -> (g b)
        | Action (actA, f) -> Action (actA, fun a -> bind (f a) g)
end

module Action_State_Int : TYPE = struct
    type _ t =
    | Get : int t
    | Set : int -> unit t
end

module M = MFreer (Action_State_Int)

val run : 'a M.monad -> int -> ('a * int)

let rec run ma st =
    match ma with
    | Return a -> (a, st)
    | Action (actA, f) ->
          match actA with
          | Get -> run (f st) st
          | Set st' -> run (f ()) st'
```

Code 3.d.5 – *Définition de la monade* Freer *en* OCaml

Cela permet aussi plus de souplesse pour les actions de la monade. En effet, celles-ci n'ont aucune influence dans la vérification des règles du `bind`, et la fonction `run` ne demande aucune règle. De plus, peu importe l'association des `bind`, les actions sont toujours imbriquées à droite, dans le même ordre. Par exemple, dans la monade *Writer* les logs ne sont plus nécessairement un monoïde, puisqu'il n'y a plus qu'une seule façon d'associer les opérations.

L'avantage de *Freer Monad* est qu'elle respecte strictement les règles d'une monade, et qu'elle permet une généricité sur les actions. Son désavantage est son coût temporel quadratique en `bind`. Ce problème est réglé dans "Reflection Without Remorse : Revealing a Hidden Sequence to Speed Up Monadic Reflection" [PK14] de Kiselyov et Van der Ploeg. Cette dernière proposition connaît toujours le problème d'accumulation des opérations et donc de risque d'erreur *Out_Of_Memory*. Ensuite, il s'agit encore d'ajouter une couche d'interprétation. En général, cette couche fait l'objet d'un compilateur, objet complexe car poussant au maximum l'optimisation, et la fonction `run` semble simpliste en comparaison. Aussi, le compilateur de base est prévu pour optimiser un code source écrit dans le langage source. Il ne comprend pas les "extensions" dynamiques ajoutées au langage avec une monade, et nous pouvons penser qu'il ne les optimise pas aussi bien.

La monade `Freer` a une portée plus grande que le problème que nous abordons ici, puisqu'elle existe aussi dans le but d'exprimer des effets de façon expressive et incrémentale. Par sa séparation entre construction

et évaluation, elle résoud le problème de *Stack_Overflow*, mais pas le problème *Out_Of_Memory*, et elle est complexe à optimiser.

Appel par besoin et *Haskell*

Nous étudions si l'appel par besoin, qui suspend l'évaluation des termes jusqu'à nécessité, et retient les valeurs calculées, permet de répondre au problème.

Le premier point est que l'appel par besoin (et par nom) peut générer des *Stack_Overflow* à cause de la suspension activée par défaut. Ainsi dans un `std_for` les applications de la fonction sont accumulées car le résultat n'est donné qu'à la fin du `for`, donc n'est évalué qu'à la fin :

```
std_for :: Int -> (a -> a) -> (a -> a)

std_for n f a =
    if (n > 0)
    then (std_for (n - 1) f (f a))  -- (f a) n'est pas évalué et reste en suspension
    else a
```

Code 3.d.6 – *Définition de* `std_for` *en* Haskell

Une erreur *Stack_Overflow* est possible à cause de l'empilement d'applications :

```
std_for 3 (\x -> 1 + x) 0
---------------------------------------
std_for 2 (\x -> 1 + x) ((\x -> 1 + x) 0)
-----------------------------------------------------------
std_for 1 (\x -> 1 + x) ((\x -> 1 + x) ((\x -> 1 + x) 0))
-------------------------------------------------------------------------
std_for 0 (\x -> 1 + x) ((\x -> 1 + x) ((\x -> 1 + x) ((\x -> 1 + x) 0)))
-------------------------------------------------------------------------
((\x -> 1 + x) ((\x -> 1 + x) ((\x -> 1 + x) 0)))
-------------------------------------------------
1 + ((\x -> 1 + x) ((\x -> 1 + x) 0))
    --------------------------------- stack + 1
    1 + ((\x -> 1 + x) 0)
        ----------------- stack + 2
```

Code 3.d.7 – *Schéma d'exécution de* `std_for` *en* Haskell

Pour forcer l'évaluation à chaque étape nous ajoutons `seq a` au début du code du `then`, ce qui signifie que si l'appel récursif `std_for` a besoin d'être évalué, alors il faut évaluer `a`. Lorsque `a` est une structure composée, le programmeur doit prendre garde à bien évaluer les suspensions sur les sous-structures, sinon elles s'accumulent, comme dans le code suivant :

```
std_for (1000 * 1000) (\(x, y) -> (1 + x, 1 + y)) 0
```

Notons que la suspension par défaut tend à libérer certains liens entre les termes. C'est à dire que pour calculer une valeur, il n'est pas nécessaire de calculer la structure complète auquelle elle appartient (par exemple un couple). L'ordre d'évaluation est moins contraint par la syntaxe. Par exemple pour la monade *Writer* il est possible d'imbriquer à droite et d'obtenir la valeur principale sans *Stack_Overflow*. Aussi, si les logs sont stockés dans une structure telle que `List`, qui permet la suspension et le calcul de valeur partielle, les résultats peuvent être récupérés sans *Stack_Overflow* :

```
import Control.Monad.Writer.Lazy  -- si nous prenons Strict cela donne un Stack Overflow

type ListInt = [Int]

m_while_r :: (a -> Bool) -> (a -> Monad a) -> (a -> Monad a)
m_while_r test f a =
    if (test a)
    then (f a) >>= (m_while_r test f)
    else (return a)

test :: Int -> Bool
test n = (n > 0)

f :: Int -> Writer ListInt Int
f n = writer (n - 1, [n])      -- writer est un "wrapper" neutre

ma = (m_while_r test f (10 * 1000 * 1000))

nl = (runWriter ma)            -- runWriter est un "un-wrapper" neutre

main = do
  putStrLn (show (fst nl)) ;
  putStrLn (show (take 10 (snd nl)))

-- output :
-- 0
-- [1000000000,999999999,999999998,999999997,999999996,
--  999999995,999999994,999999993,999999992,999999991]
```

Code 3.d.8 – *Récupération des premiers logs sans évaluation complète*

Dans le schéma d'exécution suivant, nous constatons comment fonctionne le délestage des logs qui permet d'obtenir la valeur principale finale sans problème de mémoire. Ceci dépend de la version `Lazy` de la monade *Writer*, qui permet de garder sous forme de *thunk* l'appel récursif à `while`. Dans la version `Strict`, l'appel doit être évalué jusqu'à la paire, afin d'obtenir `b` (ce dernier peut être un *thunk*). Puisque l'appel récursif doit être évalué, cela risque un *Stack_Overflow*. La version `Lazy` place l'appel récursif sous `fst` donc celui-ci n'est pas évalué immédiatement, ce qui permet de résoudre le `fst` extérieur :

```
fst (while 10)
--------------------------
fst ((f 10) >== (while 9))
-----------------------------------------------------------------------------
fst (let ~(a,logA) = (f 10) in let ~(b,logB) = (while 9 a) in (b, logA + logB))
-----------------------------------------------------------------------------
fst (fst (while 9 (fst (f 10))), (snd (f 10)) + (snd (while 9 (fst (f 10)))))
-------------------------------------------------------------------------
fst (while 9 (fst (f 10)))     -- le calcul des logs est délesté
--------------------------
fst ((f 9) >>= (while 8))      -- (fst (f 10)) est réduit par le test
```

Code 3.d.9 – *Schéma d'exécution où le calcul des logs est délesté*

Nous pouvons donc utiliser la structure `List` et la convertir à la fin dans le type de `Log` souhaité. Par exemple, le type `Int` risque une erreur *Stack_Overflow*, alors nous utilisons `ListInt` et effectuons la somme à la fin.

En ce qui concerne le problème d'imbrication à gauche de *State*, l'appel par besoin n'est pas une solution en lui-même, mais il n'empêche pas d'implémenter les solutions existantes que nous avons vues.

En conclusion que ce soit en *Haskell* ou en *Ocaml*, des solutions sont nécessaires car il n'y a pas d'efficacité systématique des récursions avec monades, et les fonctions telles que `iterateUntilM` du paquet "Monadic Loops", ne garantissent pas l'absence d'erreurs de mémoire :

```
iterateUntilM :: (Monad m) => (a -> Bool) -> (a -> m a) -> a -> m a

iterateUntilM test f a
    | test a    = (f a) >>= (iterateUntilM test f)
    | otherwise = (return a)
```

3.e Fonctions de récursion spécialisées par monade

Nous étudions à présent une solution qui est à la fois efficace, générique, et simple. Il s'agit de définir une fonction `m_while` pour chaque monade, et de l'implémenter de façon optimisée en mémoire, lorsque c'est possible. La même interface est présentée pour toutes les monades, ainsi le programmeur sait qu'en utilisant `m_while` , il obtiendra une fonction optimisée, si elle existe. De plus, la fonction `m_while` peut aussi être définie dans un transformateur de monades, ce qui donne une réelle qualité pratique à la solution.

La bonne solution est appliquée dans le langage *PureScript*, voir le document "Stack Safety for Free"[Fre15] de Phil Freeman, le créateur de PureScript. Ce langage est fortement inspiré par Haskell, mais applique l'appel par valeur, et compile vers Javascript. Dans le paquet `tailrec` [3], nous trouvons la fonction suivante, et son implémentation pour chaque monade qui permet une récursion optimisée :

3. Documentation du paquet "tailrec" de *PureScript* sur le site officiel *pursuit* : `https://pursuit.purescript.org/packages/purescript-tailrec/4.0.0/docs/Control.Monad.Rec.Class#t:MonadRec`, visitée le 15 Juillet 2019

```
tailRecM :: forall a b. (a -> monad (Step a b)) -> (a -> monad b)

data Step a b = Loop a | Done b
```

Nous recommandons la solution de *PureScript* dans les librairies de monade. Nous présentons maintenant cette solution adaptée en OCaml (le changement de type est anodin et garde simplement la cohérence avec le début du chapitre) :

```
val m_while :: ('a -> bool) -> ('a -> 'a monad) -> ('a -> 'a monad)
```

Code 3.e.1 – *Signature de* `m_while`

Monade *State*

Le principe de la monade *State* est d'exprimer une valeur auxilliaire, non encore connue, dont la valeur principale dépend. Le calcul de la valeur principale peut changer la valeur secondaire. Une valeur monadique de *State* permet donc plusieurs "exécutions", avec des valeurs auxiliaires initiales différentes.
Dans une récursion `m_while` où une fonction est répétée, il n'y a pas lieu de conserver cette expressivité, car les valeurs auxiliaires sont liées entre elles par `bind` , et seule la valeur initiale, celle donnée à la première occurence de la fonction répétée, est inconnue. En conclusion, nous pouvons simplifier la structure à l'intérieur de la récursion :

```
(* monade State *)
let m_while test f a aux0 =
    std_while (fun (a, aux) -> test a)
              (fun (a, aux) -> f a aux)
              (a, aux0)
```

Code 3.e.2 – *Définition de* `m_while` *pour* State

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.04s	0.46s	2.29s	4.60s
5G	0.00s	0.00s	0.04s	0.46s	2.29s	4.65s
1G	0.00s	0.00s	0.04s	0.46s	2.31s	4.60s
100M	0.00s	0.00s	0.04s	0.46s	2.32s	4.74s
10M	0.00s	0.00s	0.04s	0.46s	2.32s	4.66s

Tableau 3.e.1 – *Résultats de* State *avec* `m_while`

Voici la version pour le transformateur de monade *State* :

```
(* transformateur de monade State *)
type 'a monad = (aux -> ('a * aux) M.monad)

let m_while test f a aux0 =
    M.m_while (fun (a, aux) -> test a)
              (fun (a, aux) -> f a aux)
              (a, aux0)
```

Code 3.e.3 – *Définition de* `m_while` *pour le transformateur* State

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.06s	0.62s	3.12s	6.26s
5G	0.00s	0.00s	0.06s	0.62s	3.12s	6.28s
1G	0.00s	0.00s	0.06s	0.60s	3.12s	6.28s
100M	0.00s	0.00s	0.06s	0.63s	3.14s	6.23s

Tableau 3.e.2 – *Résultats de* StateWriter *avec* `m_while`

Monade *Writer*

La version efficace de *Writer* est simplement son imbrication à gauche. Cependant la solution conserve son intérêt, puisqu'il s'agit de faire le choix d'une implémentation efficace, et de la placer derrière l'interface générique `m_while` .

```
(* monade Writer *)
let m_while test f a =
    std_while (fun (a, msgs) -> test a)
              (fun (a, msgs) -> let (a2, msgs2) = (f a) in (a2, Msg.add msgs msgs2))
              (a, Msg.nil)
```

Code 3.e.4 – *Définition de* `m_while` *pour* Writer

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.03s	0.38s	1.89s	3.80s
5G	0.00s	0.00s	0.03s	0.38s	1.90s	3.79s
1G	0.00s	0.00s	0.03s	0.38s	1.90s	3.81s
100M	0.00s	0.00s	0.03s	0.38s	1.90s	3.79s

Tableau 3.e.3 – *Résultats de* Writer *avec* `m_while`

Voici la version pour le transformateur :

```
(* transformateur de monade Writer *)
type 'a monad = ('a * Msg.t) M.monad

let m_while test f a =
    M.m_while (fun (a, msgs) -> test a)
              (fun (a, msgs) -> M.map (fun (a2, msgs2) -> (a2, Msg.add msgs msgs2)) (f a))
              (a, Msg.nil)
```

Code 3.e.5 – *Définition de* `m_while` *pour le transformateur* Writer

3.f Les concrets monadiques pour la récursion non contrôlée

Dans la fonction `m_while` , le problème de mémoire est résolu car les états auxiliaires internes sont liés. Nous "sortons" de la monade de façon contrôlée, car confinée à l'intérieur de `m_while` . L'attente de la valeur auxiliaire initiale bloque l'évaluation, mais aussi la construction des fonctions à appliquer, ce qui prévient le problème de stockage.

En récursion non contrôlée, la fonction `m_while` n'est pas utilisable. Nous ne pouvons pas bloquer l'évaluation de la répétition, nous sommes donc forcés de stocker les suspensions en attente de la valeur auxiliaire initiale.

Les solutions de passage par continuation et de monade `Freer` protègent du *Stack_Overflow* mais pas de *Out_Of_Memory*.

La solution que nous proposons consiste à sortir de la monade comme le fait `m_while` . La sortie est encadrée par un type dédié, et par une fonction servant de "bloc" de programmation pour cette sortie.
Le principe est d'attendre la valeur auxiliaire initiale, puis de la placer dans un couple, à côté de la valeur principale. Ce couple peut ensuite être utilisé de façon optimisée dans une récursion. Un des enjeux de la solution est la protection de ce couple des mauvaises utilisations.
La structure se nomme "concret monadique", en opposition au type monadique *State*, qui contient une abstraction.

Définition de base, exemple avec *State*

Voici les trois définitions de base de la structure : le type paramétré, la fonction `with_mconc` qui permet d'introduire l'usage du concret au sein du programme monadique, et la fonction `apply` qui permet d'utiliser un concret monadique.

```
type 'a mconc

val with_mconc : ('a mconc -> 'b mconc) -> ('a -> 'b monad)

val apply : ('a -> 'b monad) -> ('a mconc -> 'b mconc)
```

Code 3.f.1 – *Signature d'une bibliothèque de concret monadique*

Le type `('a mconc)` contient toutes les informations, dans le cas de *State* il s'agit des valeurs principale et auxiliaire :

```
(* monade State *)
type 'a mconc = ('a * aux)
```

Code 3.f.2 – *Type de concret monadique pour* State

La fonction `with_mconc` bloque l'évaluation grâce au co-domaine monadique `('b monad)` . Son argument fonction sur les concrets reste sous forme de code en suspension, c'est à dire que les récursions ne sont pas déroulées, ce qui ne risque pas d'erreur mémoire. Voici l'implémentation avec *State* :

```
(* monade State *)
let with_mconc fconc a aux = fconc (a, aux)
```

Code 3.f.3 – *Définition de* `with_mconc` *pour* State

La fonction `apply` applique une fonction monadique à un concret monadique. Elle retourne un concret monadique, puisque les informations sont complètes. La fonction monadique est immédiatement évaluée et non mise en suspension comme dans le `bind` de *State* :

```
(* monade State *)
let apply f (a, aux) = (f a aux)
```

Code 3.f.4 – *Définition de* `apply` *pour* State

Le choix `('a -> 'b monad)` pour le type de `f` est important. Volontairement nous ne proposons pas le type `('a -> 'b mconc)` . Il n'est pas possible, car cela implique de fusionner deux concrets monadiques,

soit deux choix de valeurs auxiliaires initiales pour *State*, ce qui n'a pas de sens, et ne garantit pas l'efficacité. C'est pour cette raison que la structure d'un concret monadique n'est pas celle d'une monade.
Le but d' `apply` est de pouvoir appeler les fonctions du programme et de les utiliser sur un concret monadique.

Voici un exemple complet avec un parcours de liste :

```
(* monade State *)
type aux = int

val g : int -> 'a -> ('a monad)

let g n a aux = if n <= aux then (a, aux - n) else (a, aux)

val f : int list -> 'a -> ('a monad)

let f ln = with_conc (fun mca ->
    List.fold_left (fun elt_n mca -> apply (g elt_n) mca) mca ln)
```

Code 3.f.5 – *Exemple d'utilisation de concret monadique* State *avec un parcours de liste*

Nous observons que `with_conc` définit un environnement dans lequel les arguments initiaux de la valeur monadique ont été donnés. Cet environnement est borné afin qu'il puisse avoir une valeur monadique, et s'inscrire dans le programme complet. Nous ne souhaitons pas utiliser les concrets monadiques dans l'ensemble du programme : nous en avons besoin lorsqu'il y a un risque mémoire. Le but de la monade est d'abstraire l'argument, et le concret monadique le fait apparaître momentanément en l'associant à la valeur principale (`mca` dans notre exemple), ce qui permet l'efficacité aux points critiques du programme. En effet le concret monadique, lors d'un `apply` , ne crée pas de suspension, ce qui garantit une empreinte constante sur la mémoire (ajoutée à la mémoire nécessaire pour stocker la valeur elle-même, indépendamment de la monade).

Voici un essai avec une répétition (le parcours d'une grande liste influencerait les résultats du test) :

```
let apply f (a, st) = (f a st)
let with_mconc fconc a st = fconc (a, st)

let std_for n f a =
  let rec loop i a =
    if i < n
    then loop (i + 1) (f a)
    else a
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a aux = (a, aux + 1)
let res = with_mconc (std_for n (apply f)) 'a' 0
```

Code 3.f.6 – *Répétition de* `apply` *de concret* State

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.03s	0.34s	1.75s	3.50s
5G	0.00s	0.00s	0.03s	0.34s	1.76s	3.50s
1G	0.00s	0.00s	0.03s	0.34s	1.75s	3.50s
100M	0.00s	0.00s	0.03s	0.34s	1.75s	3.50s

Tableau 3.f.1 – *Résultats de répétition de* `apply` *de concret* State

Le tableau de résultats 3.f.1 permet de constater l'absence d'erreur *Out_Of_Memory*.

Garantie des règles de la monade pour *Reader*

Voici la définition pour *Reader* :

```
(* monade Reader *)
type 'a mconc = ('a * aux)

let with_mconc fconc a aux = (fst (fconc (a, aux))

let apply f (a, aux) = (f a aux, aux)
```

Code 3.f.7 – *Définition de concret monadique pour* Reader

La différence avec *State* se situe dans le respect des règles de la monade. *State* n'a pas beaucoup de règles puisqu'il n'y a aucune restriction sur la valeur auxiliaire. Pour *Reader*, l'intuition est que le paramètre est immutable. Dans les faits, avec un point de vue externe nous pouvons écrire un code qui ressemble à un changement de paramètre :

```
(* monade Reader *)
val paramAutre : param
val f : 'a -> ('a monad)
val f2 : 'a -> ('a monad)

let f2 a paramInitial = f a paramAutre
```

Code 3.f.8 – *Exemple de changement interne de paramètre* Reader

Notons que le changement est interne à `f2`. Dans le code `(ma >>= f2 >>= g)`, le paramètre obtenu par `g` n'est pas `paramAutre`, mais bien celui donné à `ma` initialement.

Le problème du concret monadique est qu'il offre un point de vue externe, ce qui permet d'écrire le code suivant :

```
val paramAutre : param
val f : 'a -> ('a monad)
val f2 : 'a -> ('a monad)

let f2 = with_mconc (fun mca ->
    let (a, paramInitial) = mca in
    let mca = (a, paramAutre) in
    apply f mca)
```

Code 3.f.9 – *Exemple de changement de paramètre de concret monadique* Reader

C'est beaucoup plus facile, donc tentant et discret, de modifier le paramètre de *Reader* avec le concret monadique. Cela ne respecte pas l'intention initiale, qui est de fixer le paramètre initial et d'appliquer des fonctions monadiques. La comparaison des deux codes ne met pas vraiment en valeur cet argument, alors il faut imaginer que le premier code est intégré à un programme monadique, où adopter un point de vue externe ne s'intègre pas si facilement syntaxiquement, puisque celà interfère avec la syntaxe `perform`.

Nous rendons le type du concret monadique abstrait afin de protéger la valeur auxiliaire. Pour modifier celle-ci, il faut créer un autre concret monadique.

Garantie des règles de la monade *State* abstraite

Ce qui a été dit dans la partie précédente est valable pour la monade *Reader* standard. Cela ne vaut pas pour la monade *Reader* simulée par une monade *State* abstraite, que nous avons vue au chapitre 2. Rappelons qu'il s'agit simplement d'une monade *State* dont le type est abstrait, et qui ne propose pas de fonction `set` permettant de modifier la valeur auxiliaire. La différence principale avec la monade *Reader* standard est que le paramètre est transporté comme valeur auxiliaire, puisque c'est *State*. L'ajout du mécanisme de concret monadique perce une brèche dans la garantie d'immutabilité de cette valeur auxiliaire :

```
(* monade State Abstraite Reader *)
val h : 'a -> ('a monad)

let mautre = with_mconc (fun mcautre ->

    let g = with_mconc (fun mca -> mcautre) in

    let ma2 = ma >>= g >>= h in

    ...
    mcautre
) a

let res = (run mautre paramAutre)
```

Code 3.f.10 – *Exemple de modification globale de paramètre de concret monadique* State *abstraite* Reader

Dans ce code, `h` est séquencé monadiquement avec `g`. Selon les règles de *Reader*, les deux fonctions doivent recevoir le même paramètre en entrée. C'est faux ici, car `g` parvient à modifier le paramètre non seulement localement, mais pour toutes les fonctions suivantes puisque c'est la monade *State*. C'est seulement possible grâce aux concrets monadiques qui donnent trop d'expressivité avec une fonction `('a mconc -> 'a mconc)`.

Ce contre-exemple fait échouer la preuve de la propriété `R` (voir chapitre 2) pour la fonction `with_mconc`.

Le problème se pose pour bien des versions de la monade *State* abstraite, par exemple la monade de pseudo-génération aléatoire :

```
(* monade State Abstraite RandGen *)
val h : int -> ('a monad)

let mautre = with_mconc (fun mcautre ->

    let g = with_mconc (fun mca -> mcautre) in

    let ma2 = ma >>= g >>= randgen >>= h in

    ...
    mcautre
) a

let res = (run mautre graineAutre)
```

Code 3.f.11 – *Exemple de modification illégale de graine de concret monadique* State *abstraite* RandGen

Dans cet exemple, la fonction `g` modifie la graine, ce qui affecte le nombre généré par `randgen` et utilisé par la fonction `h`. Il n'y a donc plus aucun respect du prédicat associé à cette monade.

La solution proposée est d'ajouter un paramètre de type polymorphique à `mconc`, afin d'empêcher de donner en sortie de `with_mconc` un concret venant d'un différent `with_mconc` :

```
type ('a,'id) mconc = ('a * aux)

type ('a,'b) fmconc = { f : 'id. ('a,'id) mconc -> ('b,'id) mconc }

val with_mconc : ('a,'b) fmconc -> ('a -> 'b monad)

val apply : ('a -> 'b monad) -> (('a,'id) mconc -> ('b,'id) mconc)
```

Code 3.f.12 – *Signature de bibliothèque concret monadique polymorphique*

Grâce à cette structure mise à jour, l'exemple échoue à la compilation car `mcautre` n'a pas la même variable de type que `mca` :

```
(* monade State Abstraite RandGen *)
val h : int -> ('a monad)

let mautre = with_mconc { f = fun mcautre ->

    let g = with_mconc { f = fun mca -> mcautre } in
                                        (* erreur de type *)
```

Code 3.f.13 – *Exemple de protection offerte par le concret monadique polymorphique*

En conclusion, grâce au type paramètre polymorphique, il est possible d'enforcer les garanties offertes sur les monade *State* du chapitre 2 sur les concrets monadiques. Un type fonctionnel est un moyen de garantir une propriété, et sa disparition, comme dans le type du concret, nécessite un moyen supplémentaire qui est le type paramètre.

Difficulté d'application de l'optimisation "mutable" de *State* abstraite

Il existe une optimisation de la monade *State* abstraite qui utilise la mutabilité. [4] Il s'agit d'utiliser une valeur auxiliaire mutable au lieu d'une immutable. L'utilité se trouve par exemple dans le cas de *Writer*, où nous pouvons ajouter le nouveau message en mutant une valeur existante, ce qui peut être plus rapide, selon la structure de données utilisée.

Cela nécessite un encadrement afin de conserver la propriété de transparence référentielle. Précisons le contexte : nous ne voulons pas ici une mutabilité complète, sans quoi nous ne prendrions pas de précautions. Nous voulons conserver toute l'apparence de l'immutabilité, mais profiter des performances de la mutabilité sur certaines opérations.

Pour commencer, nous ne pouvons pas donner la valeur auxiliaire mutable à la valeur monadique, comme d'habitude. Cela suppose une création mutable en dehors de la monade, et nous n'en voulons pas. La monade possède la création de la valeur auxiliaire initiale, qui pourra éventuellement être paramétrée par un argument (immutable). Par exemple, rappelons que la monade *State* abstraite pour *Writer* crée elle-même les logs vides de départ dans sa fonction `run`. Il s'agit ici du même principe, sauf que la création sera une valeur mutable initiale.

Ensuite, nous ne pouvons plus donner un accès direct à la valeur auxiliaire, à travers `get_state`. Puisqu'elle est mutable, ce serait encore une fois casser la transparence référentielle. Le mieux que nous puissions faire est donner des fonctions de lecture, dont la valeur de retour sera immutable. Pour les mêmes raisons, en

4. Voir les explications de Philip Wadler à la section 4.1 de "Monads for functional programming" [Wad95].

sortie de l'application de la monade (la fonction `run`), nous ne donnons plus la valeur auxiliaire mutable directe.

Voici pour exemple la structure des comptes bancaires, exprimée dans une structure `Hashtbl` mutable de la bibliothèque d'*OCaml*. L'avantage obtenu est qu'une seule version de la structure est nécessaire.

```
module MutAccs = struct
    type t = (string, int) Hashtbl.t
    let create () = Hashtbl.create 1000
    let get id accs = try Hashtbl.find id accs with Not_found -> 0
    let transfert (src, amount, dst) accs =
        if get src >= amount || src "banque"
        then let () = Hashtbl.replace accs src (get src - amount) in
             let () = Hashtbl.replace accs dst (get dst + amount) in
             ()
        else ()
    let to_immut accs =
        List.rev (
            Hashtbl.fold
                (fun id amount list -> (id, amount) :: list)
                accs [] )
end
```

Code 3.f.14 – *Définition de comptes bancaires mutables en* OCaml

Intégré à la monade *State* abstraite, cette structure est cachée derrière le type `monad`. Les fonctions de modification et de lecture y donnent accès par leur codomaine monadique.

```
module Abs_State_Sfc_Mut : sig
    type 'a monad (* abstrait *)
    val return : a -> ('a monad)
    val bind : ('a -> 'b monad) -> ('a monad -> 'b monad)
    val get_amount : string -> (int monad)
    val transfert : (string * int * string) -> (unit monad)
    val run : ('a monad) -> ('a * ((string * int) list))
end = struct
    type 'a monad = (accs -> 'a * accs)

    let return a accs = (a, accs)
    let bind f ma accs0 =
        let (a, accs1) = (ma accs0) in
        (f a accs1)

    let get_amount id accs =
        let () = Accs.get id accs in
        accs

    let transfert details accs =
        let () = Accs.transfert details accs in
        accs

    let run ma =
        let accs0 = Accs.create () in
        let (a, accs1) = (ma accs0) in
        let immut_accs1 = Accs.to_immut accs1 in
        (a, immut_accs1)
end
```

Code 3.f.15 – *Définition de monade* State *abstraite de comptes bancaires mutables*

Le problème est que la propriété "single thread", qui permet de garantir l'immutabilité d'une variable mu-

table en imposant une seule référence à la fois, n'est possible que par le type monadique fonctionnel. Nous disposons d'une valeur monadique, nous lui appliquons un `bind` : dans le cas d'un type fonctionnel, rien ne s'est encore passé, le calcul ne se fait qu'au `run` . Dans le cas d'un type non fonctionnel, le calcul s'effectue dès le `bind` , ce qui comprend la mutation. Or le type d'un concret est justement non fonctionnel. Nous en concluons que la solution du concret monadique n'est pas naturellement compatible avec l'optimisation de *State* mutable.

Une alternative éventuelle est imaginable pour les types auxiliaires qui sont "incrémentaux". Les comptes bancaires ne le sont pas, car le montant sur un compte peut augmenter et diminuer. Les listes de messages sont incrémentales, parce qu'au fur et à mesure de l'évaluation, la liste ne peut qu'augmenter ou rester la même. Ainsi, il serait possible de fournir au concret une interface immutable à la structure mutable, qui serait bornée à un certain point dans la structure. Par exemple pour la liste de message, la structure enregistrerait la position maximale dans la liste. Même si des messages supplémentaires apparaissaient dans la structure mutable suite à un calcul, la borne ne les prendrait pas en compte. Pour les types non incrémentaux comme les comptes bancaires, nous pouvons aussi penser à une structure de "diff", ou convertir en immutable le temps du concret, dans les deux cas cela a un coût.

3.g Les concrets monadiques pour l'utilisation REPL

Pour un certain type d'utilisation, la monade *State* n'est pas du tout efficace. Il s'agit de l'utilisation *REPL*, *Read Eval Print Loop*. Trois étapes sont répétées : lecture d'information (ou de commande, d'ordre), mise à jour de l'état, et affichage du nouveau résultat (ou de la différence avec l'ancien). L'arrêt est généralement décidé par une certaine commande, ou par un échec d'application d'une commande.

Appliquons le *REPL* avec *State*. La lecture d'information ne change pas, nous ne pouvons pas la considérer comme l'état initial, car l'information est nouvelle à chaque étape. Les deux étapes qui nous intéressent le plus sont *Eval* et *Print*. Il y a deux façons de les appliquer.

La première façon cherche à rester dans le cadre de la monade. Nous disposons d'un état monadique courant `ma` , et nous avons une commande monadique à effectuer de type `('a -> 'a monad)` . Nous utilisons `bind` pour l'appliquer à `ma` , puis nous lançons l'évaluation de `(ma aux0)` afin d'obtenir les valeurs principale et auxiliaire finales, nous les affichons.
Le grand défaut de cette première méthode est sa complexité d'évaluation en temps quadratique en rapport au nombre de commandes reçues. En effet, `ma` est contient une collection de commandes suspendues, et son évaluation `(ma aux0)` demande de recalculer ces commandes.
L'avantage de cette méthode est qu'elle reste dans le cadre de l'application monadique `bind` . Ainsi elle fonctionne encore si nous transformons *State* avec d'autres monades.

La seconde façon est pragmatique et ne cherche pas à rester dans la monade. Une fois la première commande évaluée, elle crée une nouvelle valeur `ma` à partir du résultat. Cette nouvelle valeur sera utilisée à la prochaine étape.
L'avantage de cette seconde méthode est son efficacité, mais son défaut est qu'elle nécessite un "branchement manuel" entre les résultats de l'étape et la création du nouvel `ma` .

Nous voyons ici que les concrets monadiques effectuent ces "branchements manuels", de façon encadrée, grâce à la fonction `mconc` . Cela nous permet de cumuler les avantages des deux méthodes, et d'éviter leurs défauts.

Il faut ajouter à la structure des concrets le type et la fonction `extract` , car nous avons rendu le type `('a mconc)` abstrait :

```
type 'a mconc  (* abstrait *)

type 'a extract  (* public *)

val extract : ('a mconc) -> ('a extract)
```

Code 3.g.1 – *Signature de supplément de bibliothèque de concret monadique : extraction*

Cela fonctionne aussi avec le paramètre de type polymorphique.

Voici un exemple de *REPL* avec concret monadique :

```
(* monade State *)
type cmd
type state
type aux

val get_cmd : unit -> cmd
val exec_cmd : cmd -> state -> (state monad)
val print : aux -> state -> unit

val repl : state -> (state monad)

let repl = with_mconc { f = fun mcstate ->
    let rec loop mcstate =
        let cmd = get_cmd () in
        if cmd = "Stop"
        then mcstate
        else let mcstate = apply (exec_cmd cmd) mcstate in
             let (state, aux) = (extract mcstate) in
             let () = (print aux state) in
             (loop mcstate)
    in
    (loop mcstate) }
```

Code 3.g.2 – *Exemple de concret monadique avec* REPL

3.h Plus de détails sur les contextes d'utilisation

Lorsque nous parlons de non-contrôle de récursion, nous prenons par exemple le parcours d'une liste. Or, il existe une structure "d'itérateur" qui permet de parcourir une structure de données en gardant le contrôle. Si nous disposons d'un itérateur, alors nous pouvons nous passer d'un concret monadique et utiliser `m_while` :

```
type 'elt iter

val has_next : ('elt iter) -> bool
val next : ('elt iter) -> ('elt * ('elt iter))

val f : 'elt -> 'a -> ('a monad)

let res =
    map fst (m_while (fun (a,iter) -> has_next iter)
                     (fun (a,iter) ->
                          let (elt,iter) = (next iter) in
                          let ma = (f elt a) in
                          (map (fun a -> (a,iter)) ma)))
```

Code 3.h.1 – *Parcours de structure avec itérateur et* `m_while`

Pour le contexte *REPL*, il est aussi possible de se passer des concrets monadiques si la lecture interne donne autant d'information que la lecture externe. C'est le cas de la monade *State*, et ce n'est pas le cas de la monade *Writer*, sauf si nous utilisons cette dernière sous forme de *State* abstraite. En tout cas, si nous nous contentons d'une lecture interne, nous pouvons écrire un *REPL* avec `m_while3` ainsi (`m_while3` est une variante de `m_while` dans laquelle le test est intégré à l'étape) :

```
(* monade State *)
type cmd
type state
type aux

val get_cmd : unit -> cmd
val exec_cmd : cmd -> state -> (state monad)
val print : aux -> state -> unit
val m_while3 : ('a -> ('a * bool) monad) -> ('a -> 'a monad)

let repl = m_while3 (fun state ->
    let cmd = get_cmd () in
    if cmd = "Stop"
    then return (state, false)
    else perform begin
             state <-- (exec_cmd cmd state) ;
             aux <-- get_aux ;  (* lire la valeur auxiliaire *)
             let () = (print aux state) in
             (return (state, true))
         end)
```

Code 3.h.2 – REPL *avec lecture interne et* `m_while`

Les concrets monadiques peuvent rappeler les *générateurs* ou les *iteratee*.[5] Mais les générateurs ne remplissent pas le même rôle : ils fournissent un mécanisme de découpe du calcul avec une valeur fournie à chaque pause. Cela ne résoud pas directement notre problème : en sortant de la monade, nous accumulons les suspensions, et en restant dans la monade nous avons une information partielle. Les générateurs peuvent être une façon de présenter un calcul, en ajoutant un mécanisme d'étapes. C'est compatible avec les concrets monadiques, par exemple dans le rôle de couche supérieure. La réflexion est la même pour les *iteratee* et l'étape *Read* d'un *REPL* : cela donne un mécanisme supplémentaire, cela ne résoud pas nos problèmes.

3.i Conclusion

Lorsque nous utilisons une monade comme *State*, dont le type monadique est fonctionnel, nous avons une information abstraite. La monade peut être "exécutée" en fournissant une valeur concrète, donc la monade représente autant d'exécutions que de valeurs du domaine.
Lorsque la suite du programme est placée à droite du `bind` , elle est mise en pause par l'attente de l'argument auxiliaire. Du point de vue de la consommation mémoire, c'est donc la façon la plus économique de programmer avec la monade *State*.
Cependant ce n'est pas toujours possible, car parfois nous devons "sortir" de la monade : soit parce que nous ne maîtrisons pas une récursion, soit parce que nous devons effectuer une lecture externe de la valeur monadique. Dans le cas d'une récursion, cela arrive par exemple quand nous voulons parcourir une structure de données qui ne fournit pas de mécanisme d'itération. Le contrôle de la récursion appartient à la bibliothèque de la structure et nous devons sortir de la monade. Dans le cas de la lecture externe, cela arrive par exemple lorsque nous suivons un schéma *REPL*, et qu'après chaque `bind` nous devons afficher la valeur monadique. Si nous ne pouvons pas obtenir toutes les informations en en restant à l'intérieur de

5. Pour une introduction, voir *Coroutine Pipelines* de Mario Blaževic [Bla11].

la monade, nous devons en sortir.
Ces deux situations nous imposent donc d'imbriquer *State* à gauche, ainsi contrairement à l'imbrication droite l'exécution n'est pas mise en pause : nous accumulons les suspensions d'actions en attente d'une valeur auxiliaire initiale concrète.
Ceci peut donner une erreur *Stack_Overflow*, qui se règle par les multiples solutions existantes que nous avons présentées. Cependant, nous souhaitons insister sur le fait que l'accumulation de suspensions peut donner dans le pire des cas une erreur *Out_Of_Memory*, et dans le meilleur une consommation superflue d'espace mémoire.
Pour le résoudre, il ne faut pas accumuler les opérations, mais nous avons vu qu'il existe des cas où nous ne pouvons pas imbriquer à droite. Nous ne pouvons pas rendre le type monadique non fonctionnel : ça ne définit plus une monade qui rend implicites une entrée et une sortie auxiliaires. Pour ne pas accumuler les opérations, il faut placer des "pauses", c'est à dire une fonction qui permet d'attendre la valeur auxiliaire concrète, et de bloquer le code qui génère des opérations.
C'est ce que fait la fonction `with_mconc`, grâce à son type de retour monadique, qui permet d'attendre les arguments auxiliaires, s'ils existent, selon la monade en question. Elle met à disposition une valeur appelée concret monadique, dont le type n'est idéalement pas fonctionnel (il y aura des monades pour lesquels ce sera difficile, comme *Continuation*). Toute opération appliquée sur le concret monadique est donc calculée et non mise en suspension. La valeur auxiliaire accompagne directement la valeur principale, elle n'est pas simulée par une fonction. C'est un point de vue différent, et ce n'est pas souhaitable de l'appliquer à l'ensemble du programme. La monade *State* permet de simuler un argument supplémentaire à une fonction, en rendant le codomaine monadique. Le concret monadique ne le permet pas, il est forcé de rendre les deux domaine et codomaine monadiques. Cela change beaucoup la façon de programmer puisque cela donne un point de vue externe par défaut.
La fonction `with_mconc` intègre directement les concrets dans le programme, c'est à dire dans la séquence de `bind`. Il évite d'ajouter une structure de *run*, comme c'est le cas des solutions existantes, qui poussent à réécrire un langage et un compilateur dans le programme-même. Les concrets monadiques peuvent être vus comme un mécanisme de *run* puisqu'ils peuvent appliquer des opérations et lire les résultats. Nous les utilisons donc soit comme mécanismes de *run*, par exemple pour un *REPL*, soit comme mécanisme de prévention d'erreur *Out_Of_Memory*en empêchant l'accumulation d'opérations suspendues.

Chapitre 4

Réalisation : un simulateur d'économie en `OCaml` avec monades

Dans ce chapitre, nous présentons notre étude du simulateur d'économies *Jamel* [Sep16], et expliquons pourquoi les monades s'y prêtent difficilement.
Nous développons alors les caractéristiques d'un simulateur d'économies écrit complètement en programmation fonctionnelle. Ce simulateur est disponible dans un dépôt git [1]. Il permet d'éprouver les solutions proposées dans les deux chapitres précédents. Il ne possède pas beaucoup de fonctionnalités économiques mais a déjà une complexité suffisante : le temps d'exécution est d'au moins 8 secondes. Il utilise une fonction non triviale de distribution, qui met en relation des entreprises vendeuses de biens avec des ménages acheteurs. Ce genre de fonction permet de tester les solutions proposées, à la fois du point de vue des performances et du point de vue de la qualité syntaxique.
Le simulateur fonctionnel a été écrit avec une bibliothèque de monades qui implémente les fonctions de répétitions et les concrets monadiques présentées. Elle est indépendante du simulateur et a son propre dépôt git [2].

4.a Étude du simulateur d'économies *Jamel*

Nous avons étudié le simulateur *Jamel* [3] écrit par Pascal Seppecher [Sep14]. C'est un modèle d'agents générant des ménages et des firmes hétérogènes, ainsi qu'une banque unique. Il respecte la cohérence des stocks et des flux : l'argent est créé et détruit par la banque. Le but du programme est que le nombre d'agents et d'interactions soit assez grand et complexe pour générer des dynamiques macro-économiques intéressantes.
La banque prête de l'argent, reçoit de l'intérêt des prêts, et verse des dividendes à son propriétaire (un des ménages).
Les entreprises embauchent des salariés (des ménages), paient les salaires, vendent des biens, empruntent à la banque, paient les intérêts, remboursent les prêts et versent des dividendes aux propriétaires.
Les ménages reçoivent des salaires, travaillent pour les entreprises, et leur achètent des biens.

La simulation se déroule comme une séquence périodique des étapes suivantes :
- la banque verse des dividendes
- les entreprises versent des dividendes
- les entreprises planifient leur production (embauche, prix de vente…)

1. Le simulateur est disponible à l'adresse `https://github.com/antoinekagit/sim-func`
2. La bibliothèque de monades est disponible à l'adresse `https://github.com/antoinekagit/concrete-monads`
3. voir le projet *Jamel* sur Github à l'url `https://github.com/pseppecher/jamel`

— les entreprises embauchent sur le marché du travail
— les entreprises empruntent à la banque
— les entreprises paient les salaires
— les entreprises produisent des biens
— les ménages achètent (et consomment immédiatement) des biens
— les entreprises remboursent les prêts et paient les intérêts

Les ménages possèdent un compte bancaire, où sont stockés leur salaire et l'argent restant après l'achat de biens. Les entreprises possèdent un compte bancaire, où est stocké l'argent de leurs prêts et de leurs ventes, et qui sert à payer les salaires.

Nous allons décrire sommairement le système des comptes, qui ne se résume pas à un tableau de montants. Chaque compte d'entreprise et de ménage a un champ "dette" et un champ "montant". Quant à la banque elle possède trois champs : "passif", "actif" et "capital". Lorsqu'une entreprise emprunte, le montant de son compte, la dette de son compte, l'actif et le passif de la banque augmentent du montant. Lorsqu'elle rembourse, ces champs sont réduits du même montant. Le capital de la banque augmente lors du paiment d'intérêts.
Les comptes bancaires dans *Jamel* doivent respecter la propriété de *cohérence des stocks et des flux*, qui se traduit du point de vue des comptes bancaires ainsi : la somme des montants de tous les comptes doit être égale à zéro, et seule la banque peut être en négatif (c'est elle qui crée l'argent de la simulation).

L'application d'une monade n'est pas possible directement sur *Jamel* sans une grande réécriture : il est écrit en langage *Java*, qui ne met pas à disposition de facilité pour les monades. De plus il fait un grand usage de mutabilité, ce qui entrave les monades qui sont prévues dans un cadre où le calcul est représenté par des applications imbriquées. La méthode des monades est donc à utiliser dans le cas de simulateurs d'économies exprimées par un langage fonctionnel.

Nous voyons maintenant comment effectuer le traitement des comptes bancaires dans une monade *State*, et garantir la propriété de cohérence des stocks et des flux pour toute valeur monadique [BKP15] :

```
val init : accounts
val get : string -> accounts -> int
val transfert : (string * int * string) -> (accounts -> accounts option)

let init = []
let get id accounts = List.assoc id accounts
let transfert (src, amount, dst) accounts =
    if get src >= amount || src = "banque"
    then Some (
        List.map
            (fun (id, balance) ->
                let balance = if id = src then balance - amount else balance in
                let balance = if id = dst then balance + amount else balance in
                (id, balance))
            accounts
    else None
```

Code 4.a.1 – *Bibliothèque (sans type abstrait) des comptes bancaires*

Ici, nous pouvons appliquer l'abstraction sur le type `accounts`. Il n'y a pas lieu de placer l'abstraction sur le type fonctionnel `accounts -> accounts` : toute fonction de ce type est "correcte", étant donné que le "graphe des comptes bancaires corrects est connexe". Depuis tous comptes bancaires corrects, nous pouvons rendre tout l'argent à la banque, puis atteindre n'importe quels comptes corrects en prêtant l'argent de la banque comme il faut. Par exemple :

```
{ Banque = -100 ; Damien = 50 ; François = 50 }
{ Banque = -50  ; Damien = 0  ; François = 50 }  # Damien   --50--> Banque
{ Banque = 0    ; Damien = 0  ; François = 0  }  # François --50--> Banque
{ Banque = -13  ; Damien = 0  ; François = 13 }  # Banque   --13--> François
```

Cependant, nous pouvons trouver une différence si nous appliquons l'abstraction sur le type fonctionnel `accounts -> accounts` . Cela oblige toute valeur abstraite de ce type à être une composition de fonctions de transfert. Ainsi, il devient impossible d'écrire une fonction qui rend tout l'argent à la banque en une seule fois : `fun x:accounts -> init` . Cela forme une sorte de sécurité qui peut être souhaitable.

Nous passons à présent à la description du simulateur d'économies complètement fonctionnel.

4.b Monades utilisées dans le simulateur fonctionnel

Les attributs des agents sont représentés dans un type enregistrement. Les méthodes de l'agent sont définies dans le module qui définit le type de l'agent. Les modules suffisent car il n'y a pas besoin de mécanisme de génériques et d'héritage avancés.

La mutation est un point central. Le programme fonctionnel ne permet pas les clôtures mutables, alors nous utilisons les monades *Reader*, *State* et *Writer*.
La monade *Reader* permet de diffuser les paramètres initiaux dans le programme, par exemple le nombre d'entreprises au départ. Il est vrai que nous pourrions utiliser un module pour stocker statiquement ces paramètres, mais la monade *Reader* fournit plus de souplesse, par exemple elle permet de contrôler de façon programmable les paramètres, ce qui est plus difficile avec un module statique.
La monade *State* permet de simuler tous les états mutables, en particulier les générateurs d'identifiants des agents, les générateurs pseudo-aléatoires. Nous y plaçons les états qui ont un fort caractère auxiliaire, comme le générateur d'identifiant d'agent (il s'agit simplement d'un entier représentant le prochain identifiant non encore utilisé). Nous y plaçons également les états qui sont utilisés très régulièrement dans l'ensemble du programme, et qui seraient encombrants à placer en valeurs principales, par exemple la graine du générateur pseudo-aléatoire.
La monade *Writer* permet d'enregistrer tous les évènements de la simulation. L'état entre les périodes de la simulation ne suffit pas toujours, car certains évènements ne sont pas calculables à partir de l'état, et doivent être explicitement enregistrés.

Dans la simulation d'économies, les ménages et les entreprises ont chacun un compte bancaire. Dès qu'ils réalisent des opérations d'achat et de vente, il y a une répercution sur les comptes.
Cette opération arrive trop souvent dans la simulation pour que nous plaçions les comptes en valeur principale. Cela nous forcerait à les donner explicitement en entrée et en sortie de la plupart des fonctions de la simulation. D'ailleurs, dans *Jamel* les transferts sont implémentés comme des effets de bord sur des variables mutables.
De plus, les comptes bancaires de *Jamel* doivent respecter la propriété de cohérence des stocks et des flux, et nous pouvons exprimer cette propriété grâce à la monade *State* abstraite.
Pour toutes ces raisons les comptes bancaires font partie des valeurs auxiliaires de la simulation, donc du type de l'état de la monade *State*.

4.c Bibliothèque de monades

Nous avons écrit une bibliothèque de monades contenant *Writer*, *Reader*, *State*. Elle comprend les transformateurs de monade associés. Nous avons ajouté les fonctions de récursions spécialisées `m_while` venant de *PureScript*, sous cette interface :

```
val m_for : int -> ('a -> 'a monad) -> ('a -> 'a monad)
val m_while : ('a -> bool) -> ('a -> 'a monad) -> ('a -> 'a monad)
val m_while2 : ('a -> bool monad) -> ('a -> 'a monad) -> ('a -> 'a monad)
val m_while3 : ('a -> ('a * bool) monad) -> ('a -> 'a monad)
val m_while4 : ('a -> 'a option monad) -> ('a -> 'a monad)
```

Code 4.c.1 – *Signature des fonctions de récursion monadique*

Les fonctions `m_for` , `m_while2` et `m_while4` sont générées automatiquement grâce au système de modules, à partir des définitions de `m_while` et `m_while3` .

Nous avons également ajouté les concrets pour la récursion non contrôlée et l'usage *REPL*, sous cette interface :

```
type ('a,'id) mconc      (* always abstract *)
type ('a,'b) f_mconc = { f : 'id. ('a,'id) mconc -> ('b,'id) mconc }
type 'a extract   (* never abstract  *)

val apply : ('a,'id) mconc -> ('a -> 'b monad) -> ('b,'id) mconc
val with_mconc : 'a -> ('a,'b) f_mconc -> ('b monad)
val cmap : ('a -> 'b) -> ('a,'id) mconc -> ('b,'id) mconc
val cset : 'b -> ('a,'id) mconc -> ('b,'id) mconc
```

Code 4.c.2 – *Signature des concrets monadiques*

Nous avons aussi ajouté les monades *State* abstraites pour *Writer*, les comptes bancaires, et les types simples. Le but de cette dernière est simplement de protéger le type fonctionnel de la monade *State*, afin de maintenir une bonne pratique d'utilisation de dépendre le moins possible de la monade utilisée.
Notons que pour les monades abstraites, le transformateur a dû être intégré au module de base, afin qu'il aie connaissance du type, et que nous puissions écrire la fonction `lift_ext` .

Le paquet *omonad* a été utilisé afin d'obtenir la syntaxe `perform` (syntaxe *do* en *Haskell*).

Étant donné la potentielle omniprésence des monades dans un programme (par exemple dans un simulateur), l'implémentation des fonctions de la bibliothèque est très sensible et fait facilement varier le temps d'exécution.

4.d Difficulté de `lift`

Malgré les transformateurs, la combinaison de monades reste difficile. Lors de la programmation il est souhaitable de donner un type précis à une fonction, c'est à dire déclarer uniquement les monades utilisées au sein de celle-ci. Ceci oblige ensuite à "lifter" la fonction lorsqu'elle est composée avec une autre fonction utilisant plus de monades. Dans le meilleur des cas nous avons un de ces schémas :

```
M2_Y (M1)
M3_Y (M2_Y (M1))
(* nous utilisons lift_int pour ajouter M3 *)

M1
M3_Y (M2_Y (M1))
(* nous utilisons deux fois lift_int *)
```

En *Haskell* la définition par *TypeClass* de `lift` permet d'appeler une seule fois la fonction et elle sera répétée automatiquement jusqu'à `M1`.

Dans les autres cas, nous trouvons :

— il manque une monade au milieu, par exemple nous voulons passer de `M3_Y(M1)` à `M3_Y(M2_Y(M1))`, ce qui ne peut pas se faire avec seulement `lift_int` et `lift_ext`

— les monades sont commutatives mais dans le désordre, par exemple nous voulons passer de `M1_Y(M2_Y(M3))` à `M3_Y(M2_Y(M1))`, ce qui n'est pas compliqué mais fastidieux
— il y a plusieurs fois la même monade, par exemple `M2_Y(M2_Y(M1))`, ce qui porte à confusion, et complique une automatisation du mécanisme

Dans ces cas, trouver le bon enchaînement de `lift` et autres fonctions à appliquer demande plus de réflexion. Ceci augmente la dépendance de la fonction aux monades choisies. Dans le cas où il manque une monade au milieu, il n'existe pas à notre connaissance de fonction standard permettant de l'ajouter.

Il ne semble pas pertinent d'abandonner les transformateurs de monades. Ceux-ci permettent une définition modulaire, donc facilement compréhensible et interfaçable d'une monade.

La solution pragmatique que nous avons choisie est d'effectuer toutes les transformations au début du programme, afin de créer "la" monade du programme, qui sera utilisée dans toutes les fonctions. L'avantage est qu'il y a seulement besoin de `lift` au début du programme, pour élever les fonctions telles que `get_state`. Le défaut est la perte de précision du type des fonctions, et l'éventuelle perte de performance dûe à l'utilisation de toutes les monades. Voici la construction de la monade tirée du code du simulateur :

```
module Env = StateAbs(SimEnv)
module Log = StateAbs_Writer(SimLog)
module Sfc = StateAbs_Accs
module ParT = ReaderT(SimParam)

module EnvLog = Log.T(Env)
module EnvLogSfc = Sfc.T(EnvLog)

module EnvLogSfcPar = struct
    include ParT(EnvLogSfc)
    let get_env = lift_ext (EnvLogSfc.lift_ext (EnvLog.lift_ext Env.get_state))
    let set_env = lift_ext (EnvLogSfc.lift_ext (EnvLog.lift_ext Env.set_state))
    let log = lift_ext (EnvLogSfc.lift_ext EnvLog.write)
    let get_accs = lift_ext (EnvLogSfc.get_accs)
    let get_amount = lift_ext (EnvLogSfc.get_amount)
    let exc_make_transfert = lift_ext (EnvLogSfc.exc_make_transfert)
    let run ma param accs env =
        Env.run (EnvLog.run (EnvLogSfc.run (ma param) accs)) env
end

open EnvLogSfcPar
```

Code 4.d.1 – *Définition de la "grande" monade pour le simulateur*

Pour optimiser les performances, nous pourrions combiner les états des paramètres, de l'environnement, des logs et des comptes bancaires en un seul quadruplet. Cela éviterait le type monadique quatre fois fonctionnel. Nous pensons à la monade *Freer* qui explore "l'extensibilité" des structures de données. L'extensibilité est définie comme une monade à laquelle nous pouvons ajouter un concept sans recompiler le code initial. [4] Les transformateurs entrent plus ou moins dans cette définition : le code de la monade initiale n'a pas besoin d'être recompilé, il faut ajouter des `lift` donc recompiler le programme qui utilise la monade.

Notons que dans le même esprit que la monade unique au programme, il est confortable de définir en haut du programme les parcours de structures avec concrets monadiques, par exemple :

4. Voir "Freer Monads, More Extensible Effects"[KI15] d'Oleg Kiselyov et Ishii Hiromi.

```
(* monade EnvLogSfcPar *)
val liste_fold_mconc :
    ('elt list) -> 'acc ->
    ('elt -> 'acc -> 'acc monad) ->
    ('acc monad)

let liste_fold_left_mconc lelt acc f =
    with_mconc acc { f = fun mcacc ->
        List.fold_left (fun mcacc elt -> apply mcacc (f elt)) lelt mcacc
    }
```

Code 4.d.2 – *Fonctions de parcours de liste avec concret monadique*

4.e Algorithmes à structure de données mutable

Le simulateur contient des algorithmes appelés très régulièrement, comme l'algorithme de mélange pseudo-aléatoire de liste, qui intervient par exemple pour sélectionner un sous-ensemble aléatoire des ménages. Mais les listes en *OCaml* sont immutables, et sont moins efficaces pour réaliser cette opération de mélange que les tableaux mutables. Utiliser les tableaux dans l'algorithme permet d'améliorer les performances.
Les tableaux mutables ne sont pas compatibles avec la transparence référentielle. Toute mutabilité est en général déconseillée avec les monades puisque cela peut invalider les règles monadiques.
La solution est de confiner la mutabilité à l'intérieur de l'algorithme, et de vérifier ce dernier avec une attention spéciale. Nous recopions la liste immutable en tableau mutable au début de l'algorithme, et en sens inverse à la fin.

```
let shuffle_array arr : ('a array monad) =
    let n = (Array.length arr) - 1 in
    let m_arr_i = m_for n
        (fun (arr, i) ->
             perform begin
               k <-- rand_int (i + 1) ;
               let tmp = arr.(i) in
               let () = arr.(i) <- arr.(k) in
               let () = arr.(k) <- tmp in
               return (arr, i + 1)
             end)
        (arr, 0)
    in
    (map fst m_arr_i)

let shuffle_list la : ('a list monad) =
  let arr = (Array.of_list la) in
  let m_arr = (shuffle_array arr) in
  (map Array.to_list m_arr)
```

Code 4.e.1 – *Exemple d'algorithme utilisant structure mutable et monades*

Le code ci-dessus fonctionne avec les monades *State*, *Writer*, *Reader*. En revanche, si par exemple nous ajoutons la monade *List*, il ne faut pas confondre avec la liste `la` que nous mélangeons, nous parlons ici d'ajouter la monade *List* aux monades représentées par le type paramétré `monad`. Cette monade *List* a la capacité de rendre plusieurs valeurs au lieu d'une seule, et son `bind` répète la fonction pour chaque valeur de la liste. Imaginons à présent que pour une raison obscure, la fonction `rand_int` de l'exemple crée deux valeurs de sortie (alors que bien sûr la véritable `rand_int` ne renvoie qu'une seule graine). La conséquence est la duplication des actions d'affectation du tableau qui suivent, par exemple `(arr.(i) <- arr.(k))`, avec les deux `k` différents donnés par `rand_int`. À priori ça ne va pas faire grand mal à l'algorithme de mélange, mais cela donne une idée des situations qui peuvent arriver et être éventuellement dommageables. Il faut

faire attention lorsque l'on associe mutabilité et monades.
Rappelons que nous avons vu en 3.f.4 une méthode pour utiliser une structure mutable dans la monade *State*. Ce serait approprié ici et supprimerait les risques de la monade *List*, en créant plusieurs tableaux mutables.

4.f Présence des concrets monadiques

Nous notons que la plupart des récursions présentes dans le simulateur utilisent les concrets monadiques au lieu des fonctions de répétition. Cela s'explique par le fait qu'il s'agit généralement de parcourir une structure de données, en appliquant une fonction de transformation (`map`) ou en effectuant un calcul accumulatif (`fold`).

Voici un exemple tiré du simulateur. Il commence par une fonction qui définit le parcours des listes à l'aide d'un concret, ce qui évite de répliquer ce code dans la simulation. Ensuite vient la fonction `travail_machine_firm`, dont le rôle est de mettre à jour l'état de production des machines de la simulation. L'état de production est un entier : il augmente de 1 par période, et lorsque qu'il atteint un pallier, il revient à zéro et un "bien" est enregistré.

```
let liste_fold_left_mconc l b f : 'b monad =
        with_mconc b { f = fun mcb ->
                Liste.fold_left l mcb (fun elt -> apply' (f elt)) }

let travail_machine_firm firm : firm monad = perform begin
        param <-- get_param () ;
        (machines, biens) <--
                liste_fold_left_mconc firm.machines ([], 0)
                        (fun machine (machines, biens) -> perform begin
                                (machine2, fini) <-- travail_machine_1 machine ;
                                if fini
                                then return (machine2 :: machines, biens +
                                ↪  param.nbBiensParMachine)
                                else return (machine2 :: machines, biens)
                         end) ;
        log (Fifo.one (TravailMachines (firm.id, Liste.size machines, biens))) ;
        return { firm with machines ; biens = firm.biens + biens }
end
```

Il faut à la fois mettre à jour la liste des machines et retenir le nombre de biens créés. Nous utilisons donc une fonction de parcours de liste `fold_left`. Bien que dans notre cas le nombre de machines est petit, par sécurité nous utilisons un concret monadique afin d'éviter l'accumulation de fonctions. C'est aussi un peu plus rapide. La fonction `liste_fold_left_mconc` crée un concret avant le parcours : `mcb`. Ensuite, elle appelle la fonction `Liste.fold_left` (version alternative de la standard qui utilise l'appel terminal) et à chaque étape utilise `apply` sur le concret `mcb`.

La fonction `get_param` appartient à la monade *Reader* et permet ici de récupérer le pallier de l'état de production des machines. La fonction `log` appartient à la monade *Writer* et permet d'enregistrer un évènement.

4.g Fonctionnel et monades versus mutabilité globale

Est-ce que le contexte de la programmation de simulateur bénéficie de la programmation fonctionnelle et des monades ? Est-ce qu'il aurait été meilleur d'utiliser des variables globales mutables, par exemple pour exprimer la graine du générateur pseudo-aléatoire ?

Si nous considérons que chaque simulation aura son processus, c'est à dire que le programme est compilé, et que chaque exécution du simulateur consiste à créer un processus avec l'exécutable, alors la mutabilité n'est pas un problème. Chaque processus aura son propre environnement mutable privé. Le problème de la mutabilité ne se situe pas dans cet usage.

Considérons que nous souhaitions améliorer le simulateur, par exemple en ajoutant un autre pays avec des comptes bancaires séparés. Nous pouvons adapter les variables globales des comptes bancaires pour gérer deux pays différents. La mutabilité ne pose pas de problèmes, l'enjeu ici est plutôt la capacité d'évolution des structures de données et leurs interfaces dans le programme.

Maintenant considérons que nous voulons intégrer notre simulateur au sein d'un autre programme, dont l'objectif est plus grand. Nous ne voulons pas lancer l'exécutable du simulateur dans un sous-processus, nous voulons ajouter le code au projet, comme une bibliothèque. Ici nous pouvons imaginer une difficulté, par exemple si nous souhaitons avoir plusieurs simulations simultanées. Il faut alors effectuer des sortes de commutations de contexte pour changer les variables globales selon la simulation. Ici c'est un avantage pour la programmation fonctionnelle puisque le simulateur est une fonction pure qui prend en entrée le contexte de la simulation. Notons que la programmation objet mutable est une méthode pour lier les fonctions d'interface directements aux contextes.

L'avantage d'une fonction qui ne dépend pas d'un contexte mutable est qu'elle est plus facilement réutilisable, par exemple dans un contexte un peu différent de l'initial. Il est évident que l'interface simple "entrée - sortie" est plus facile à manipuler que la même interface avec une clôture contenant des variables mutables. Ceci dit, la supression des variables globales mutables encombre largement les entrées et les sorties, au point que nous sommes obligés de recourir à des mécanismes automatiques (monades). Ces derniers ne sont pas triviaux, ils posent des problèmes de mémoire, de performance et de composition. Du point de vue syntaxique, dans un simulateur de taille petite ou moyenne, réalisé par un petit nombre de personnes, et non destiné à être intégré au sein d'un autre programme de façon complexe, les variables globales mutables semblent être une solution raisonnable.

Deuxième partie

Simulation par acteurs

Chapitre 5

Modèle acteur, Akka et typage des messages

Les architectures informatiques de calcul sont de plus en plus parallèles. L'industrie des microprocesseurs a fait le choix d'orienter l'augmentation régulière du nombre de transistors vers la multiplication des cœurs de calcul au sein des processeurs, ainsi que la production de microprocesseurs peu coûteux accompagnant le développement de la téléphonie mobile et de l'internet des objets. Dans le même temps, l'augmentation des performances de calcul par cœur a cédé la place à une plus grande distribution de ces ressources dans les matériels informatiques. La bonne exploitation des ressources de calcul demande donc d'adopter des méthodes de programmation adaptées. De nombreux modèles de programmation concurrentes ont été étudiés depuis les années 60, mais les méthodes actuelles de programmation des simulations multi-agents, basées essentiellement sur le paradigme de programmation objet, relèvent essentiellement d'un modèle de concurrence basé sur les états mutables partagés et les fils de calcul (threads), avec lequel il est difficile de raisonner.

Le modèle d'acteur est un modèle de calcul concurrent théorique introduit dans les années 70 [HBS73] dans lequel des unités de programmes appelées acteurs communiquent entre-elles uniquement de manière asynchrone par envoi et réception de messages. Le modèle d'acteur a connu un succès tardif avec son implémentation dans le langage *Erlang* [Arm03].

Ce modèle peut être vu d'une part comme comme un renforcement de l'encapsulation du paradigme objets car si les acteurs possèdent des états mutables, ceux-ci ne sont jamais partagés et d'autre part comme une abstraction de la notion de thread car chaque acteur fait au plus une chose à la fois en étant le moins possible en attente d'autres acteurs.

Un programme consiste alors en un système d'acteurs, communiquant entre eux et se partageant un ensemble de ressources de calcul (les multiples cœurs de plusieurs processeurs répartis sur un réseau informatique). Un système d'acteurs peut ainsi consister en des millions d'acteurs, réparties sur différents matériels. Un point important assuré par les implémentations est que la programmation des acteurs ne dépend pas de leur localisation sur le matériel, la communication est possible de la même façon quelle que soit la localisation réelle des acteurs. Un autre élément essentiel que nous n'aborderons pas ici est la supervision des systèmes d'acteurs, essentiel pour la résilience, qui s'exerce de façon décentralisée entre acteurs via le système de remise de messages.

Les données contenues dans les messages sont immutables et aucune référence vers un état local ne peut être partagée entre acteurs.

Dans cette partie nous envisageons l'utilisation des systèmes d'acteurs pour réaliser des simulateurs basés agents, de part les similitudes entre agents et acteurs.

Nous présentons *Akka*, une bibliothèque implémentant ce modèle, et nous voyons que dans sa version standard elle n'applique pas de vérification sur les types des messages transmis, ce qui peut mener le programme vers un état non prévu. Nous exprimons alors les acteurs dans un langage simplifié, auquel nous montrons comment ajouter du typage simple (à la manière du lambda calcul simplement typé). En fin de chapitre, nous présentons quelques exemples illustrant pourquoi ce typage doit être augmenté pour retrouver l'expressivité initiale.

5.a Un modèle de concurrence : acteurs Akka

Nous allons introduire plus précisément le modèle acteurs et la bibliothèque *Akka* qui l'implémente. Ce modèle a pour but de décrire des programmes dits "concurrents". Nous commençons par donner une brève introduction aux programmes concurrents, ce qui nous permet de cerner le contexte et ses problématiques, et de préciser le vocabulaire.

Les programmes concurrents

Un programme est une suite d'instructions à destination du processeur de l'ordinateur. Le processeur (ou plus précisément un cœur de calcul sur un processeur multi-cœurs) est le mécanisme chargé d'exécuter un programme. Il crée une instanciation du programme que nous appelons "processus". Le processus contient la suite d'instructions et un marqueur de lecture. Le processeur exécute les instructions une par une, en respectant l'ordre.

Un processus a un effet s'il modifie son environnement. Dans l'ordinateur l'environnement d'un processus est constitué de la mémoire de travail, de la mémoire de stockage, des périphériques (écran, clavier), etc. Les suites d'instructions d'un programme consistent donc en grande partie à manipuler des variables : des petites zones de la mémoire de travail.

Le programme est conservé dans la mémoire de stockage, qui est grande, lente et permanente. Un processus est stocké dans la mémoire de travail, qui est petite, rapide et ephémère. La taille d'un processus est principalement liée au nombre de variables que celui-ci utilise.

La mémoire de travail est de taille suffisante pour accueillir plusieurs processus. Le processeur donne alors l'illusion de simultanéité des exécutions. Pour ce faire, il exécute quelques instructions de chaque processus, tour à tour.

Nous avons décrit un contexte suffisant pour définir les programmes concurrents. Un programme concurrent suit la définition d'un programme que nous avons donnée, à la différence que son exécution donne lieu à la création de plusieurs processus au lieu d'un seul. Ces processus vont collaborer ou entrer en compétition afin de remplir le but du programme, d'où l'expression "concurrent" pour "courir ensemble".

Pourquoi vouloir plusieurs processus pour un seul programme ? Cela permet d'améliorer le temps d'exécution de certains programmes. Notamment lorsqu'un processus est un attente d'une autre ressource physique de l'ordinateur que le processeur (accès à la mémoire de stockage, au réseau). Ce peut aussi être une contrainte inhérente au but du programme, par exemple une messagerie instantanée dans laquelle les communiquants sont nécessairement en concurrence. Nous allons décrire trois schémas de programme concurrents.

Dans le schéma "concurrent-léger", l'exécution du programme donne lieu à la création d'un processus (appelé "processus initial"). Le processus initial crée des sous-processus légers, appelés "threads". Les threads ont chacun leur propre suite d'instructions, mais tous travaillent dans la même zone mémoire que le processus principal. La principale caractéristique de ce schéma est le partage de mémoire.
La première problématique est de synchroniser les threads. Par exemple, un thread d'affichage attend que le thread de calcul ait terminé avant de mettre à jour l'interface. Pour se synchroniser, deux threads peuvent utiliser un "signal" (sorte de message rudimentaire) ou déposer leur message directement dans la mémoire (qui est partagée).
La deuxième problématique est d'assurer la cohérence de la mémoire. Elle n'est pas garantie puisque la

même mémoire est lue et modifiée par tous les threads. Par exemple, deux threads achètent par erreur le même billet d'avion.

Dans le schéma "concurrent-lourd", le processus principal crée des sous-processus lourds, qu'on appelle simplement sous-processus. Chacun travaille sur sa propre zone de mémoire privée. Chaque sous-processus s'assure de la cohérence de sa mémoire privée qu'il est seul à utiliser. Il n'y a qu'un lien de filiation entre le processus principal et ses sous-processus.
Ce schéma protège les zones mémoires entre les sous-processus. Il implique une communication plus élaborée, car les signaux ne suffisent pas pour échanger toutes les données, par exemple un tableau de noms de clients. L'ordinateur fournit des mécanismes de communication ("pipe", fichier, bases de données, "IPC sockets", etc).
La problématique de synchronisation reste présente. La problématique d'un sous-processus est cohérente, mais la séparation des mémoires fait apparaître l'absence de cohérence des mémoires entre elles. Autrement dit, au lieu d'avoir un entremêlement de deux informations contradictoires au même endroit, nous avons deux informations contradictoires dans deux endroits séparés, quand le programme implique que l'ensemble soit cohérent.

Dans le schéma qu'on nomme "concurrent-distribué", il y a plusieurs ordinateurs, donc plusieurs processeurs, plusieurs mémoires et plusieurs processus. Ce schéma ressemble au schéma "concurrent-lourd" , mais les communications entre processus sont plus difficiles, car il faut utiliser le réseau.
Les communications par réseau impliquent un travail supplémentaire lors de l'écriture du programme, et sont plus lentes que les communications entre sous-processus.
L'utilisation de plusieurs ordinateurs est un gain temporel. Ce gain est contre-balancé par le coût des communications.
En plus des problématiques de synchronisation et de cohérence globale, ce schéma ajoute la problématique de gestion des échecs. Avec un grand nombre d'ordinateurs le risque de panne ou de perte de communication réseau est non négligeable, et il faut le prévoir dans le programme.

Les trois schémas sont cumulables : un programme peut être constitué de plusieurs processus répartis sur plusieurs ordinateurs. Chacun de ces processus crée des sous-processus, et chaque sous-processus crée des threads.

Le modèle Acteurs

Les programmes concurrents sont réputés difficiles à écrire, à cause des problématiques évoquées : synchronisation, cohérence de la mémoire, problèmes réseaux, etc. Le modèle "Acteurs"[1] est un langage de description de programmes concurrents. Il abstrait les mécanismes de sous-processus dans le concept d'acteur.

Un acteur a une mémoire privée dont il est seul manipulateur. Comme pour les sous-processus lourds, il y a donc une cohérence locale des données. La cohérence globale est laissée au programmeur. C'est un des principes du modèle "Acteur" : certaines problématiques ne sont pas résolubles au niveau du cadre de programmation et doivent être résolus par le programme lui-même.

Un système de messagerie permet de faire communiquer deux acteurs. Chaque acteur est doté d'une adresse qui est unique. Un acteur peut envoyer un message à un autre acteur s'il connaît l'adresse de ce dernier. L'envoi de message est non-bloquant, c'est à dire que l'émetteur n'attend pas que le destinataire confirme sa disponiblité à recevoir le message. L'émetteur envoie le message et continue son exécution. En un sens cela résoud le problème de synchronisation, en interdisant la synchronisation. L'idée est qu'il ne faut pas qu'un sous-processus se bloque parce qu'il attend une condition. Il doit continuer à être disponible à l'interaction avec les autres sous-processus.

Un acteur est un quadruplet contenant une adresse unique, une mémoire privée, une boîte aux lettres et une fonction de réaction. La boîte aux lettres stocke dans une file les messages reçus par l'acteur. L'exécution de l'acteur consiste à attendre un message dans la boîte, et à le traiter en utilisant la fonction de réaction. Un

1. Inventé en 1973, voir "A Universal modular Actor formalism for artificial intelligence" [HBS73]

seul message est traité à la fois, ce qui garantit la cohérence locale de la mémoire. La fonction de réaction a plusieurs possibilités : lire et écrire dans la mémoire, terminer l'acteur, envoyer des messages, créer de nouveaux acteurs.

La bibliothèque Akka

La bibliothèque Akka est une implémentation du modèle Acteurs en langage "Scala". Scala est un langage objet et fonctionnel qui s'exécute sur une "JavaMachine". Il possède un sytème de types avancé, qui va nous intéresser dans le projet d'étudier le typage d'acteurs.

Akka suit la description du modèle Acteurs qu'on a donnée. Il y a des fonctionnalités supplémentaires comme la supervision d'acteurs qui ne nous concernent pas ici. Nous notons que l'envoi de message dans Akka est "at-most-once" c'est à dire qu'un message sera transmis entre zéro et une fois. Cela est un reflet supplémentaire de l'impossibilité pour la bibliothèque de garantir une transmission fiable à 100%. La bibliothèque préfère obliger le programmeur à gérer les cas d'erreur au niveau du programme.

Pour créer un acteur en Akka nous définissons une classe constructeur d'acteur. Tous les acteurs créés à partir de cette classe partageront la même fonction de réaction. Voici un exemple de classe d'acteur :

```
class Fireman (chief_param:ActorRef) extends akka.Actor {
    // mutable memory
    var chief :ActorRef = chief_param
    var state :String = "waiting for fire"

    // fonction de réaction
    def receive = {
        case FireAlert(where) =>
            if (where == "the house") state = "looking elsewhere"
            else state = "going for fire"
        case Report() =>
            chief ! Report()
        case HireCousin() =>
            var cousin :ActorRef = context.newActor(Fireman, chief)
            cousin ! Report()
    }
}

// classes des messages
case class FireAlert (where:String)
case class Report ()
case class HireCousin ()
```

Le nom de la classe est `Fireman`. `chief_param` est un paramètre du constructeur de la classe. La mémoire privée est constituée des deux variables `chief` et `state`. La fonction de réaction se nomme `receive` et est définie pour trois classes de messages : `FireAlert`, `Report` et `HireCousin`. Ces trois classes sont définies à la fin du code. Ce sont des `case class`, une spécificité de Scala qui permet de les construire et les "matcher" sans utiliser de symbole `new`, à la façon des langages fonctionnels. Dans la réaction à `FireAlert`, il y a une modification de la mémoire. Dans la réaction à `HireCousin`, il y a une création de nouvel acteur, à partir de la classe `Fireman`. La variable `context` fait appel à la bibliohèque `Akka`. Dans la réaction à `Report`, il y a un envoi d'un message de classe `Report` à l'adresse contenue dans la variable `chief`.

Bien qu'il ne soit pas écrit explicitement dans la classe, le type de la fonction de réaction `receive` est `(Any -> Unit)`, ce qui dénote une fonction du type `Any` vers le type `Unit`. Le type `Any` est le "sur-type" de tous les types, autrement dit n'importe quel type peut être utilisé à la place de `Any`. Donc, notre fonction `receive` peut être appliquée sur un message de n'importe quel type au moment de la compilation. En revanche, à l'exécution les trois `case` de la fonction seront successivement testés, et si aucun n'est vérifié le message sera jeté. Cette absence de vérification au moment de la compilation nous mène à notre objectif.

Vérification des communications

La bibliothèque Akka ne vérifie pas la correction des communications à la compilation. Une communication est dite incorrecte lorsque la fonction de réaction du destinataire refusera toujours le type du message transmis. Puisqu'Akka utilise le type `Any` pour la fonction, la compilation valide toujours la communication, alors que l'exécution va potentiellement la faire échouer. Notre objectif est de spécifier un système de types qui peut vérifier à la compilation la correction des communications.
Nous allons prendre un peu de recul par rapport à Akka et utiliser une syntaxe un peu plus abstraite, un lambda-calcul avec quelques constructions supplémentaires.

5.b Grammaire d'un langage acteurs simplifié

Le langage est organisé en deux parties : les termes, qui sont des expressions algorithmiques pouvant s'évaluer (conditions, déclarations et affectations d'identifiants…), et les systèmes, qui sont des ensembles de termes en concurrence.

Les termes sont utilisés pour exprimer les définitions d'acteurs, les acteurs en cours, et les messages.

De façon générale dans ce chapitre, le symbole λa sera associé aux concepts concernant les termes, et la lettre s sera pour les systèmes.

Grammaire des termes

Nous utilisons le lambda calcul simplement typé, avec évaluation par valeur à petits pas [2]. Nous l'étendons avec des termes supplémentaires relatifs aux acteurs : `spawn` et `sendmsg`. L'action `self` (qui permet d'obtenir l'adresse de l'acteur courant) sera intégrée comme un argument supplémentaire de la fonction de réception de l'acteur. L'action `wait-msg`, qui bloque l'exécution de l'acteur jusqu'à réception d'un message, n'est pas rendue disponible dans les termes. La raison est que nous utilisons un langage acteurs "par définitions", ainsi cet évènement est supposé se déclencher une fois avant chaque exécution de la fonction de réception, et c'est donc le système qui s'en charge.

Dans la grammaire Figure 5.b.1, nous donnons le lambda calcul, avec en gras la syntaxe supplémentaire pour les acteurs.

⟨*term*⟩ ::= ⟨*ident*⟩
| ⟨*value*⟩
| (⟨*term*⟩ ⟨*term*⟩)
| (if ⟨*term*⟩ then ⟨*term*⟩ else ⟨*term*⟩)
| (some ⟨*term*⟩)
| ⟨**actor-action**⟩

⟨*value*⟩ ::= unit | true | false
| ⟨*int*⟩ | ⟨*string*⟩ | ⟨**address**⟩
| ⟨*function*⟩
| (some ⟨*value*⟩) | (none of ⟨*type*⟩)

⟨*function*⟩ ::= (λ ⟨*ident*⟩ :⟨*type*⟩ $\rightarrow$ ⟨*term*⟩)

⟨**actor-action**⟩ ::= (**send-msg** ⟨*term*⟩ **to** ⟨*term*⟩)
| (**spawn** ⟨*string*⟩ **with** ⟨*term*⟩)

Figure 5.b.1 – *Termes* λa

2. voir le livre *Types and Programming languages* de *Benjamin C. Pierce*[Pie02]

Les symboles `<ident>`, `<int>`, `<string>` et `<address>` sont tous des jetons (tokens) dont la définition est respectivement : une chaîne de caractères commençant par une lettre minuscule et sans symboles spéciaux, une suite de chiffres, une chaîne de caractères entourée de guillemets, et pour les adresses nous ne donnons pas de définition précise (ça peut être par exemple une suite de chiffres commençant par un arobase). Notons qu'on autorise le sucre syntaxique `let-in` :

`let <ident>:<type> = <term`$_1$`> in <term`$_2$`>`
≡
`((`λ`<ident>:<type>`$\rightarrow$ `<term`$_2$`>) <term`$_1$`>)`

Ce sucre syntaxique ne sera pas traité dans les règles d'évaluation et de typage, il faut considérer qu'il est implicitement écrit dans sa forme basique (application et lambda).

Nous donnons en Figure 5.b.2 les types pour les termes.

$\langle type \rangle$ ::= Unit | Bool | Int | String | **Address**
| ($\langle type \rangle \rightarrow \langle type \rangle$)
| Option[$\langle type \rangle$]

Figure 5.b.2 – *Types* λa

Grammaire des systèmes

Il y a quelques variantes possibles de définition d'un système d'acteur. Le point commun est le traitement d'un message reçu à la fois, et l'envoi de message non bloquant. Le point divergent est la façon de définir le comportement de l'acteur. Dans certains systèmes, appelés "acteurs processus" [3], il y a une action `wait-msg` qui bloque l'exécution jusqu'à réception d'un message. Un acteur processus est défini par un terme (souvent) récursif qui contient des actions `wait-msg`. Ici nous allons plutôt utiliser les acteurs dits "classiques", qui utilisent comme définition une fonction décrivant les actions à effectuer en réponse à un message. Cette fonction ne contient pas d'action `wait-msg`. Le type de notre fonction est (`Address` $\rightarrow A \rightarrow M \rightarrow$ `Option`$[A]$). Le premier argument est l'adresse de l'acteur courant : c'est le mécanisme qui joue le rôle de l'action `self`. Le deuxième argument est la mémoire de l'acteur. Nous remarquons que ce même type est redonné en sortie, à l'intérieur d'une `Option`. Le système va répéter la fonction en branchant la sortie sur l'entrée, jusqu'à ce que la sortie soit nulle. Enfin, le troisième argument est le message. Lorsqu'un acteur est créé, nous connaissons l'adresse et la mémoire initiale. Le message est inconnu : l'acteur est alors en attente de le recevoir.

Un système d'acteurs est composé d'un ensemble de définitions (similaires à des classes en programmation orientée objet (*POO*)), d'acteurs en cours d'exécution, d'acteurs en attente de message, et de messages en attente de réception.

3. voir les définitions dans la thèse de *Simon Fowler*[Fow19]

⟨*system*⟩ ::= ⟨*definitions*⟩ ⟨*address*⟩ ⟨*processus*⟩

⟨*definitions*⟩ ::= ∅
| ⟨*definitions*⟩ || ⟨*actor-def*⟩

⟨*actor-def*⟩ ::= (actdef ⟨*string*⟩ ⟨*type*⟩ ⟨*term*⟩)

⟨*processus*⟩ ::= ∅
| ⟨*processus*⟩ || ⟨*actor-waiting*⟩
| ⟨*processus*⟩ || ⟨*actor-in-eval*⟩
| ⟨*processus*⟩ || ⟨*msg*⟩

⟨*actor-waiting*⟩ ::= (actwait ⟨*string*⟩ ⟨*address*⟩ ⟨*term*⟩)

⟨*actor-in-eval*⟩ ::= (acteval ⟨*string*⟩ ⟨*address*⟩ ⟨*term*⟩)

⟨*msg*⟩ ::= (msg ⟨*value*⟩ to ⟨*address*⟩)

Figure 5.b.3 – `grammaire`$_s$

Un système contient une adresse : à chaque création d'acteur, elle est attribuée au nouvel acteur, puis mise à jour à une adresse qui n'existe pas déjà dans le système, de façon à garantir l'unicité des adresses.
Nous peouvons considérer `<definitions>` comme un ensemble, avec une condition supplémentaire : chaque nom de définition d'acteur s doit être unique. Une définition `<actor-def>` est composée d'un `<string>` pour le nom de la definition, d'un `<type>` pour le type de la mémoire que posséderont les acteurs issus de cette définition, et d'un `<term>` qui est une fonction de réception telle que nous l'avons décrit précédemment.
Un point important à noter est que les définitions d'acteurs sont mutuellement récursives de façon indirecte par les noms s. En effet, le terme d'acteur t d'une définition peut contenir une action d'acteur `spawn`. Cette action a pour but de créer un nouvel acteur en utilisant la définition correspondant à un nom s fourni en argument. Nous pouvons par exemple imaginer une définition d'acteur de nom "Fork" contenant une instruction `spawn` avec le nom "Fork" (voir Figure 5.b.4).

```
(actdef ``Fork'' Unit (λself:Address→ (λmem:Unit→
    (λmsg:String→
        let addrOfNewActor:Address = (spawn ``Fork'' with unit) in
        let unit:Unit = (send-msg ``hello world'' to addrOfNewActor) in
        (some mem)
    )
)))
```

Figure 5.b.4 – *Exemple de définition d'acteur qui se réplique à l'infini*

Les `<processus>` sont à considérer comme un multi-ensemble, bien que la grammaire les présente comme construits incrémentalement de gauche à droite.

Nous donnons en Figure 5.b.5 un schéma du cycle de vie d'un acteur. Un acteur peut se trouver en deux modes : attente (de message), et évaluation (lorsqu'il a reçu un message et effectue son action de réponse). L'état `mort` est définitif : l'acteur ne fera plus rien (et peut être nettoyé de la mémoire dans une implémentation réelle).

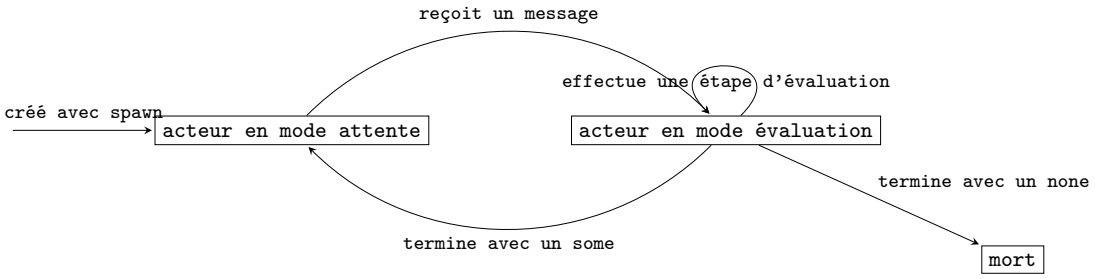


Figure 5.b.5 – *Automate schématisant le cycle de vie d'un acteur*

Un acteur en mode attente est exprimé par `<actor-waiting>`, qui contient un `<string>` pour le nom de la définition d'où vient l'acteur, une `<address>` qui est celle de l'acteur lui-même, et un `<term>` qui doit être l'application de la fonction de réception à l'adresse de l'acteur et une mémoire initiale.
Un acteur en mode évaluation est exprimé par `<actor-in-eval>`, et est similaire à un acteur en mode attente pour le nom et l'adresse, et son terme a été appliqué à un message, et a potentiellement déjà effectué des étapes d'évaluation.

5.c Évaluation par valeur à petits pas

Nous utilisons l'évaluation à petits pas. Cela permet de rappeler que les acteurs s'évaluent en concurrence, et cela est mieux adapté aux systèmes qui peuvent ne pas terminer. Le mode "par valeur" convient dans ce langage contenant des mécanismes mutables.

L'évaluation des termes est standard pour sa partie lambda calcul. Pour les actions d'acteurs, elle fait intervenir le contexte des autres processus et des définitions du système. L'évaluation système consiste à lancer des évaluations de termes d'acteurs, de relancer des acteurs, et de transmettre les messages.

Évaluation de terme $\xrightarrow{\lambda_a}$

De façon simplifiée, la relation $\xrightarrow{\lambda_a}$ est définie sur `<term>`, car il s'agit d'évaluer un terme. En réalité, elle est définie entre (`<definitions> <address> - <term>`) et (`<address> <processus> - <term>`). Un terme est évalué dans le contexte d'un système. Nous utiliserons $\xrightarrow{\lambda_a}$ pour évaluer les termes à l'intérieur des `<actor-in-eval>`.
Nous avons besoin des `<definitions>` pour évaluer l'action `spawn`, qui génère un nouvel acteur en utilisant la fonction de réception trouvée dans les définitions.
Nous avons besoin des `<processus>` car l'évaluation des actions `spawn` et `send-msg` génère respectivement un nouvel acteur et un nouveau message, et qu'il faut placer ceux-ci quelque part, et ce ne peut être dans `<term>`.
Nous avons besoin de `<address>` pour générer une nouvelle adresse pour un nouvel acteur lors de `spawn`.
Nous notons que dans un système d'acteur non classique, où une action `wait-msg` est disponible, nous aurions besoin aussi des processus à gauche de $\xrightarrow{\lambda_a}$, pour consommer un message lors de la réception. Ici ce n'est pas le cas puisque l'action n'est pas disponible et est gérée au niveau du système.

Dans la définition suivante, Δ est une méta-variable pour `<definitions>`, α est une méta-variable pour `<address>`, Π est une méta-variable pour `<processus>`, t est une méta-variable pour `<term>`, f est une méta-variable pour `<function>`, x est une méta-variable pour `<ident>`, v est une méta-variable pour `<value>`.

Définition 6 (Évaluation de terme $\xrightarrow{\lambda_a}$)**.**

E-APP-1
$$\frac{\Delta\alpha - t_1 \xrightarrow{\lambda_a} \alpha'\Pi - t_1'}{\Delta\alpha - (t_1\ t_2) \xrightarrow{\lambda_a} \alpha'\Pi - (t_1'\ t_2)}$$

E-APP-2
$$\frac{\Delta\alpha - t \xrightarrow{\lambda_a} \alpha'\Pi - t'}{\Delta\alpha - (f\ t) \xrightarrow{\lambda_a} \alpha'\Pi - (f\ t')}$$

E-BETA
$$\frac{f = (\lambda x : A \to t) \qquad t' = t[x := v]}{\Delta\alpha - (f\ v) \xrightarrow{\lambda_a} \alpha\varnothing - t'}$$

E-IF-1

$$\frac{\Delta\alpha - t_1 \xrightarrow{\lambda_a} \alpha'\Pi - t_1'}{\Delta\alpha - (\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) \xrightarrow{\lambda_a} \alpha'\Pi - (\texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3)}$$

E-IF-T

$$\frac{\varnothing}{\Delta\alpha - (\texttt{if true then}\ t_2\ \texttt{else}\ t_3) \xrightarrow{\lambda_a} \alpha\varnothing - t_2}$$

E-IF-F

$$\frac{\varnothing}{\Delta\alpha - (\texttt{if false then}\ t_2\ \texttt{else}\ t_3) \xrightarrow{\lambda_a} \alpha\varnothing - t_3}$$

E-SOME

$$\frac{\Delta\alpha - t \xrightarrow{\lambda_a} \alpha'\Pi - t'}{\Delta\alpha - (\texttt{some}\ t) \xrightarrow{\lambda_a} \alpha'\Pi - (\texttt{some}\ t')}$$

E-SEND-1

$$\frac{\Delta\alpha - t_1 \xrightarrow{\lambda_a} \alpha'\Pi - t_1'}{\Delta\alpha - (\texttt{send-msg}\ t_1\ \texttt{to}\ t_2) \xrightarrow{\lambda_a} \alpha'\Pi - (\texttt{send-msg}\ t_1'\ \texttt{to}\ t_2)}$$

E-SEND-2

$$\frac{\Delta\alpha - t_2 \xrightarrow{\lambda_a} \alpha'\Pi - t_2'}{\Delta\alpha - (\texttt{send-msg}\ v_1\ \texttt{to}\ t_2) \xrightarrow{\lambda_a} \alpha'\Pi - (\texttt{send-msg}\ v_1\ \texttt{to}\ t_2')}$$

E-SEND

$$\frac{\varnothing}{\Delta\alpha_1 - (\texttt{send-msg}\ v\ \texttt{to}\ \alpha_2) \xrightarrow{\lambda_a} \alpha_1\,(\texttt{msg}\ v\ \texttt{to}\ \alpha_2) - \texttt{unit}}$$

E-SPAWN-1

$$\frac{\Delta\alpha - t \xrightarrow{\lambda_a} \alpha'\Pi - t'}{\Delta\alpha - (\texttt{spawn}\ s\ \texttt{with}\ t) \xrightarrow{\lambda_a} \alpha'\Pi - (\texttt{spawn}\ s\ \texttt{with}\ t')}$$

E-SPAWN

$$\frac{(\texttt{actdef}\ s\ A\ t) \in \Delta \qquad \alpha' = next(\alpha)}{\Delta\alpha - (\texttt{spawn}\ s\ \texttt{with}\ v) \xrightarrow{\lambda_a} \alpha'\,(\texttt{actwait}\ s\ \alpha\ (t\ \alpha\ v)) - \alpha}$$

Dans la règle E-SPAWN, le t est la "fonction de réception", elle reçoit l'adresse de l'acteur créé, car elle doit connaître l'adresse de l'acteur courant (action "self"). Elle reçoit aussi la valeur attribuée comme mémoire initiale. Le terme d'acteur est remplacé par l'adresse de l'acteur créé, afin que celle-ci soit connue et puisse être diffusée à d'autres acteurs.

Évaluation de système $\xrightarrow{s}$

Il y a trois évaluations possibles dans un système : l'évaluation du terme d'un acteur en cours, la relance d'un acteur qui a terminé son traitement, et la transmission d'un message envoyé à un destinataire. Il reste deux évaluations qui nettoient les acteurs terminés et les messages sans destinataire.

La relation $\xrightarrow{s}$ est définie sur `<system>`. Nous notons que les règles utilisent l'acteur ou le message complètement à droite des processus, et nous rappelons que les processus sont à considérer comme un multi-ensemble, dont l'ordre n'a pas d'importance.

Définition 7 (Évaluation système $\xrightarrow{s}$)**.**

ES-EVAL

$$\frac{\Delta\alpha_1 - t \xrightarrow{\lambda_a} \alpha_1'\Pi' - t'}{\Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ \alpha_2\ t) \xrightarrow{s} \Delta\alpha_1'\Pi \parallel \Pi' \parallel (\texttt{acteval}\ s\ \alpha_2\ t')}$$

ES-RESPAWN

$$\frac{(\texttt{actdef}\ s\ A\ t) \in \Delta}{\Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ \alpha_2\ (\texttt{some}\ v)) \xrightarrow{s} \Delta\alpha_1\Pi \parallel (\texttt{actwait}\ s\ \alpha_2\ (t\ \alpha_2\ v))}$$

ES-MSG

$$\frac{\varnothing}{\Delta\alpha_1\Pi \parallel (\texttt{actwait}\ s\ \alpha_2\ t) \parallel (\texttt{msg}\ v\ \texttt{to}\ \alpha_2) \xrightarrow{\texttt{s}} \Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ \alpha_2\ (t\ v))}$$

ES-CLEAN-ACTOR

$$\frac{\varnothing}{\Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ \alpha_2\ (\texttt{none of}\ A)) \xrightarrow{\texttt{s}} \Delta\alpha_1\Pi}$$

ES-CLEAN-MSG

$$\frac{\nexists\, (\texttt{actwait}\ s\ \alpha_2\ t), (\texttt{acteval}\ s\ \alpha_2\ t) \in \Pi}{\Delta\alpha_1\Pi \parallel (\texttt{msg}\ v\ \texttt{to}\ \alpha_2) \xrightarrow{\texttt{s}} \Delta\alpha_1\Pi}$$

La règle ES-RESPAWN s'applique uniquement aux acteurs qui contiennent une valeur `some`. Les acteurs contenant une valeur `none` sont des acteurs "terminés" qui sont supprimés par ES-CLEAN-ACTOR.

5.d Forme normale et valeur

Un terme (ou système) en forme normale $\texttt{FN}_{\lambda\texttt{a}}$ (ou $\texttt{FN}_{\texttt{s}}$) ne peut pas s'évaluer. Aucune règle d'évaluation ne s'applique à lui. Toutes les valeurs sont des formes normales. Les formes normales qui ne sont pas des valeurs sont considérées comme des erreurs. Ces termes erronés ne sont pas souhaitables, nous voulons ne pas les voir apparaître, ce que nous garantirons dans la suite avec le typage.

Forme normale de terme $\texttt{FN}_{\lambda\texttt{a}}$

Le prédicat $\texttt{FN}_{\lambda\texttt{a}}$ est défini sur (`<definitions> <address> - <term>`). Nous avons besoin des définitions et de l'adresse puisqu'elles font partie des conditions de certaines règles, par exemple E-SPAWN. Notons que le domaine est exactement celui de gauche de $\xrightarrow{\lambda\texttt{a}}$.

Définition 8 (Prédicat $\texttt{FN}_{\lambda\texttt{a}}$)**.** *Si et seulement s'il n'existe pas de règle d'évaluation $\xrightarrow{\lambda\texttt{a}}$ applicable, le terme est en forme normale* (*le terme avec les définitions et l'adresse*)*.*

$$\texttt{FN}_{\lambda\texttt{a}}(\Delta\alpha - t) = \texttt{il n'existe pas}\ \Pi, \alpha', t'\ \texttt{tel que}\ (\Delta\alpha - t) \xrightarrow{\lambda\texttt{a}} (\Pi\alpha' - t')$$

Autrement formulé, le terme est en forme normale lorsqu'au moins une des conditions de chaque règle de $\xrightarrow{\lambda\texttt{a}}$ est invalidée.

Les valeurs de terme sont déjà données dans la grammaire λa par `<value>`.

Proposition 1 (Tout terme qui est une valeur est en forme normale)**.**

Démonstration. Par récurrence structurelle sur `<value>` de la grammaire λa, avec n'importe quelles définitions Δ et addresse α.

- cas `unit` : il n'existe pas de règle dans laquelle $t = \texttt{unit}$. Même conclusion pour `true`, `false`, `<int>`, `<string>`, `<address>`, `<function>`, et `(none of <type>)`.
- cas (`some` v) : la règle E-SOME nécessite d'évaluer v, or par hypothèse de récurrence nous avons que v, qui est une valeur, ne peut pas s'évaluer. En conclusion, (`some` v) ne peut pas non plus s'évaluer et c'est bien une forme normale.

□

Les formes erronées (normales et non valeur) des termes correspondent à des constructions non interprétables, par exemple la condition d'un `if` qui ne serait pas un booléen, ou bien l'application d'un argument à un entier au lieu d'une fonction.
Notons qu'un terme peut perdre sa part de non-sens durant l'évaluation, par exemple `(if true then 3 else (3 3))`.

Forme normale de système $\texttt{FN}_\texttt{s}$ et valeur de système $\texttt{Value}_\texttt{s}$

Le prédicat $\texttt{FN}_\texttt{s}$ est défini sur `<system>`.

Définition 9 (prédicat $\texttt{FN}_\texttt{s}$)**.** *Si et seulement s'il n'existe pas de règle d'évaluation $\xrightarrow{\texttt{s}}$ applicable, le système est en forme normale.*

$$\texttt{FN}_\texttt{s}(\Delta\alpha\Pi) = \texttt{il n'existe pas } \alpha', \Pi' \texttt{ tel que } (\Delta\alpha\Pi) \xrightarrow{\texttt{s}} (\Delta\alpha'\Pi')$$

Définition 10 (prédicat $\texttt{Value}_\texttt{s}$)**.** *Un système est une valeur (système) s'il ne contient pas d'acteurs en cours d'évaluation et de message.*

$$\begin{aligned}&\texttt{Value}_\texttt{s}(\Delta\alpha\Pi) = \\ &\quad \nexists\, (\texttt{acteval } s_1\ \alpha_1\ t_1) \in \Pi \\ &\quad \nexists\, (\texttt{msg } v_2 \texttt{ to } \alpha_2) \in \Pi\end{aligned}$$

Proposition 2 (Tout système valeur est en forme normale $\texttt{FN}_\texttt{s}$)**.**

Démonstration. Par cas sur les règles d'évaluation $\xrightarrow{\texttt{s}}$:
- cas ES-EVAL : Impossible car il n'y a pas d'acteur en évaluation. Même conclusion pour ES-RESPAWN et ES-CLEAN-ACTOR.
- cas ES-MSG : Impossible car il n'y a pas de message. Même conclusion pour ES-CLEAN-MSG.

□

Les formes erronées des systèmes contiennent des acteurs en évaluation erronés. Les messages qui n'ont pas de destinataire sont simplement nettoyés, ce n'est pas considéré comme une erreur car il est légitime qu'un message soit à destination d'un acteur qui a déjà terminé. Il y a aussi l'action ES-RESPAWN qui peut échouer si la définition n'existe pas, mais nous considèrons que les acteurs du système ont été créés à l'aide d'une définition.

5.e Typage, progression, conservation

Le typage garantit qu'un terme (un système) soit est une valeur, soit peut s'évaluer en conservant cette même propriété. Autrement dit un terme (système) bien typé n'est pas et ne peut pas devenir erroné.

Pour les termes, nous utilisons le typage simple, en ajoutant les cas pour les deux actions d'acteurs. Pour le système, il s'agit de vérifier que tous les éléments du processus sont indépendamment bien typés en respect des définitions.

Ensuite, nous définissons les propriétés de progression et de conservation du typage. Pour les acteurs, cela garantit la "terminaison" puisqu'ils ne sont pas récursifs, mais nous remarquons qu'ils peuvent être relancés sans fin par le système. Les deux propriétés nous donnent qu'un terme (ou un système) bien typé soit est une valeur, soit peut s'évaluer en conservant le type.

Typage de terme (:)

Le typage pour les termes lambda est classique. Pour l'action `spawn`, il s'agit de vérifier que la mémoire donnée en argument est compatible avec le type mémoire déclaré dans l'`actdef` correspondant. Pour l'action `send-msg`, pour le moment nous ne vérifions pas la compatibilité du type du message avec le type de la fonction de réception. Nous ne pouvons pas le faire directement, puisque nous n'avons pas d'information triviale sur l'adresse donnée à l'action `send-msg`. Donc selon le typage actuel, n'importe quelle valeur typée peut être envoyée à n'importe quelle adresse.

La relation de typage (:) se fait entre (`<definitions>` − `<environnement>` ⊢ `<term>`) et `<type>`.
L'ensemble `<environnement>` est standard : il associe à des identifiants des types. Il est défini au dessus de

`<term>` et contient les informations de type sur les variables libres de ce dernier.
Le typage de terme servira pour ceux contenus dans les `<actor-def>`, les `<actor-waiting>`, les `<actor-in-eval>`, et les `<msg>`.

Nous définissons un typage qui ne nécessite pas d'inférence de type, ce qui nous évite par exemple de devoir utiliser un mécanisme de résolution de contraintes. L'arbre de dérivation de typage suffit, et les feuilles utilisent une information locale.

Dans la définition suivante, x est une méta-variable pour `<ident>`, A et B sont des méta-variables pour `<type>`, Σ est une méta-variable pour `<environnement>`, Δ est une méta-variable pour `<definitions>`, $\varnothing$ signifie l'absence de conditions à une règle, n est une méta-variable pour `<int>`, s est une méta-variable pour `<string>`, α est une méta-variable pour `<address>`, t est une méta-variable pour `<term>`.

Définition 11 (typage de terme (:))**.**

$$\text{T-IDENT}\ \frac{(x : A) \in \Sigma}{\Delta - \Sigma \vdash x : A} \qquad \text{T-UNIT}\ \frac{\varnothing}{\Delta - \Sigma \vdash \texttt{unit} : \texttt{Unit}} \qquad \text{T-TRUE}\ \frac{\varnothing}{\Delta - \Sigma \vdash \texttt{true} : \texttt{Bool}} \qquad \text{T-FALSE}\ \frac{\varnothing}{\Delta - \Sigma \vdash \texttt{false} : \texttt{Bool}}$$

$$\text{T-INT}\ \frac{\varnothing}{\Delta - \Sigma \vdash n : \texttt{Int}} \qquad \text{T-STRING}\ \frac{\varnothing}{\Delta - \Sigma \vdash s : \texttt{String}} \qquad \text{T-ADDRESS}\ \frac{\varnothing}{\Delta - \Sigma \vdash \alpha : \texttt{Address}}$$

$$\text{T-FUNC}\ \frac{\Delta - \Sigma, (x : A) \vdash t : B}{\Delta - \Sigma \vdash (\lambda x : A \to t) : (A \to B)} \qquad \text{T-APP}\ \frac{\Delta - \Sigma \vdash t_1 : (A \to B) \qquad \Delta - \Sigma \vdash t_2 : A}{\Delta - \Sigma \vdash (t_1\ t_2) : B}$$

$$\text{T-IF}\ \frac{\Delta - \Sigma \vdash t_1 : \texttt{Bool} \qquad \Delta - \Sigma \vdash t_2 : A \qquad \Delta - \Sigma \vdash t_3 : A}{\Delta - \Sigma \vdash (\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) : A}$$

$$\text{T-SOME}\ \frac{\Delta - \Sigma \vdash t : A}{\Delta - \Sigma \vdash (\texttt{some}\ t) : \texttt{Option}[A]} \qquad \text{T-NONE}\ \frac{\varnothing}{\Delta - \Sigma \vdash (\texttt{none of}\ A) : \texttt{Option}[A]}$$

$$\text{T-SEND}\ \frac{\Delta - \Sigma \vdash t_1 : M \qquad \Delta - \Sigma \vdash t_2 : \texttt{Address}}{\Delta - \Sigma \vdash (\texttt{send-msg}\ t_1\ \texttt{to}\ t_2) : \texttt{Unit}} \qquad \text{T-SPAWN}\ \frac{\Delta - \Sigma \vdash t_1 : A \qquad (\texttt{actdef}\ s\ A\ t_2) \in \Delta}{\Delta - \Sigma \vdash (\texttt{spawn}\ s\ \texttt{with}\ t_1) : \texttt{Address}}$$

Notons que le type présent dans (`none of` A) est ce qui permet à la règle T-NONE de connaître le type localement.

Nous définissons une proposition d'inversion qui permet d'obtenir des informations sur une valeur grâce à son type.

Dans la définitions suivante, Δ est une méta-variable pour `<definitions>`, Σ est une méta-variable pour `<environnement>`, v est une méta-variable pour `<value>`, A et B sont des méta-variables pour `<type>`.

Proposition 3 (`inversion-type`$_{\lambda a}$)**.**

$$\begin{aligned}
&\forall \Delta, \Sigma, v, A, B, \\
&\quad \texttt{si}\ (\Delta - \Sigma \vdash v : (A \to B))\ \texttt{alors}\ v\ \texttt{est une fonction} && \text{(i)} \\
&\quad \texttt{si}\ (\Delta - \Sigma \vdash v : \texttt{Bool})\ \texttt{alors}\ v = \texttt{true ou false} && \text{(ii)} \\
&\quad \texttt{si}\ (\Delta - \Sigma \vdash v : \texttt{Address})\ \texttt{alors}\ v\ \texttt{est une adresse} && \text{(iii)}
\end{aligned}$$

Démonstration. Par définition de la relation de typage,

— cas (i) : la seule règle donnant un type de la forme $(A \to B)$ est celle qui type une fonction. La règle pour l'application ne compte pas car une application n'est pas une valeur, il en va de même pour `if then else`.
— cas (ii) : il n'y a que deux règles qui donnent le type `Bool` à une valeur, et elles le font respectivement pour les valeurs `true` et `false`.
— cas (iii) : il n'y a qu'une règle qui donne le type `Address` à une valeur, et elle le donne à des termes `<addr>`.

□

La proposition 4 de progression typée dit que si un terme est bien typé avec un environnement vide, soit c'est une valeur, soit il n'est pas en forme normale. Cela exclut un terme bien typé d'être en forme normale sans être une valeur.
Nous imposons un environnement vide afin de garantir la possibilité d'évaluation. Nous rappelons qu'on ne considère pas un identifiant comme une valeur. La restriction sur l'environnement exclut des programmes qui pourraient effectuer des étapes d'évaluation, mais nous considérons que capturer les programmes sans identifiant "libre" est suffisant.

Proposition 4 (`progression-type`$_{\lambda a}$)**.** *Tout terme bien typé soit est une valeur, soit peut s'évaluer.*

$$\begin{array}{l} \texttt{pour tout } \Delta, \alpha, t, A, \\ \quad \texttt{si } (\Delta - \varnothing \vdash t : A) \texttt{ alors :} \\ \quad\quad \texttt{soit } (t \texttt{ est une valeur}) \qquad \text{(i)} \\ \quad\quad \texttt{soit } \neg \texttt{FN}_{\lambda a}(\Delta\alpha - t) \qquad \text{(ii)} \end{array}$$

Démonstration. Par récurrence sur la définition de la relation de typage $(:)$, avec $\Sigma = \varnothing$.
— cas T-IDENT $(\Delta - \varnothing \vdash x : A)$: ce cas est impossible car $(x : A) \notin \varnothing$.
— cas T-UNIT $(\Delta - \varnothing \vdash \texttt{unit} : \texttt{Unit})$: `unit` est une valeur. Même conclusion pour les cas T-TRUE, T-FALSE, T-INT, T-STRING, T-ADDRESS.
— cas T-FUNC $(\Delta - \varnothing \vdash (\lambda x : A \to t) : (A \to B))$: une fonction est une valeur.
— cas T-APP $(\Delta - \varnothing \vdash (t_1\ t_2) : B)$:
Par l'hypothèse de récurrence sur $(\Delta\, \varnothing \vdash t_1 : (A \to B))$:
  — cas t_1 est une valeur :
  Par la proposition `inversion-type`$_{\lambda a}$, nous savons qu'une valeur typée $(A \to B)$ est une fonction.
  Ensuite, par l'hypothèse de récurrence sur $(\Delta\, \varnothing \vdash t_2 : A)$:
    — cas t_2 est une valeur : Nous pouvons appliquer la règle E-BETA, ce qui montre le cas (ii).
    — cas t_2 n'est pas une forme normale : Nous pouvons appliquer la règle E-APP-2, ce qui montre le cas (ii).
  — cas t_1 n'est pas une forme normale : Nous pouvons appliquer la règle E-APP-1, ce qui montre le cas (ii).
— cas T-IF $(\Delta - \varnothing \vdash (\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3) : A)$:
Un `if then else` n'est jamais une valeur donc pour respecter la propriété il doit forcément être évaluable. Par la relation de typage, nous savons que t_1 est bien typé par `Bool`. Par l'hypothèse de récurrence, nous savons alors que t_1 soit est une valeur, soit peut s'évaluer. Si t_1 est une valeur, nous avons par la proposition d'inversion qu'il vaut soit `true` soit `false`. Nous pouvons alors appliquer E-IF-T ou E-IF-F. Si t_1 n'est pas une valeur, selon l'hypothèse de récurrence il doit pouvoir s'évaluer. Nous pouvons alors effectuer E-IF-1.
— cas T-SOME $(\Delta - \varnothing \vdash (\texttt{some } t_1) : \texttt{Option}[A])$:
Par l'hypothèse de récurrence, soit t_1 est une valeur, et dans ce cas $(\texttt{some } t_1)$ est une valeur, soit t_1 peut s'évaluer et nous pouvons effectuer E-SOME.
— cas T-NONE $(\Delta - \varnothing \vdash (\texttt{none of } A) : \texttt{Option}[A])$: c'est une valeur.

— cas T-SEND ($\Delta - \varnothing \vdash (\texttt{send-msg}\ t_1\ \texttt{to}\ t_2) : \texttt{Unit}$) :
Par la relation de typage, t_1 et t_2 sont bien typés, respectivement par A et `Address`. Par l'hypothèse de récurrence, soit t_1 est une valeur, et nous passons au cas de t_2, soit t_1 peut s'évaluer, et dans ce cas nous effectuons E-SEND-1. Ensuite, soit t_2 est aussi une valeur, et par la proposition d'inversion t_2 est une adresse. Alors, nous pouvons effectuer E-SEND. Si t_2 n'est pas une valeur, il peut s'évaluer et nous pouvons effectuer E-SEND-2.
— cas T-SPAWN ($\Delta - \varnothing \vdash (\texttt{spawn}\ s\ \texttt{with}\ t_1) : \texttt{Address}$) :
Par la relation de typage et l'hypothèse de récurrence, soit t_1 est une valeur, et dans ce cas nous pouvons déclencher la création d'acteur avec E-SPAWN. Nous notons que la relation de typage a aussi vérifié que la définition d'acteur existe, et nous supposons qu'il existe toujours une nouvelle adresse disponible. Si t_1 n'est pas une valeur, alors il peut s'évaluer, et nous pouvons effectuer E-SPAWN-1.

□

Nous cherchons à présent à montrer une propriété de conservation du type pendant l'évaluation. Cela avec la propriété de progression nous garantira qu'un terme bien typé soit finit par s'évaluer en une valeur, soit s'évalue sans fin.
Pour commencer nous montrons une première propriété seulement sur les bêta réduction, pour modulariser la preuve de la propriété.

Dans la proposition et la démonstration suivantes, Δ est une méta-variable pour `<definitions>`, Σ est une méta-variable pour `<environnement>`, x et y sont des méta-variables pour `<ident>`, A, B, C et D sont des méta-variables pour `<type>`, t est une méta-variable pour `<term>`, v est une méta-variable pour `<value>`.

Proposition 5 ($\texttt{conservation-type-beta-reduction}_{\lambda a}$)**.** *Un terme bien typé conserve son type quand nous lui appliquons une bêta réduction.*

$$\begin{array}{l}\texttt{pour tout}\ \Delta, \Sigma, x, A, t, B, v,\\ \quad \texttt{si}\ (\Delta - \Sigma, (x : A) \vdash t : B)\ \texttt{et}\ (\Delta - \Sigma \vdash v : A),\\ \quad\quad \texttt{alors}\ (\Delta - \Sigma \vdash t[x := v] : B)\end{array}$$

Démonstration. Par récurrence sur la définition de la relation de typage (:) pour $(\Delta - \Sigma, (x : A) \vdash t : B)$.
— cas T-IDENT ($\Delta - \Sigma, (x : A) \vdash y : B$) :
Si $x = y$, alors $A = B$. Remplacer x par v implique de montrer $(\Delta\ \Sigma \vdash v : A)$, ce qui est donné en hypothèse.
Si $x \neq y$, alors la bêta réduction ne change pas le terme, et nous devons montrer $(\Delta - \Sigma \vdash y : B)$. Par l'hypothèse et par le typage nous savons que $(y : B) \in \Sigma$.
— cas T-UNIT ($\Delta - \Sigma, (x : A) \vdash \texttt{unit} : \texttt{Unit}$) : La bêta réduction ne change pas le terme, et le terme n'est pas dépendant du contexte $(x : A)$, donc nous prouvons $(\Delta - \Sigma \vdash \texttt{unit} : \texttt{Unit})$. Cette méthode est la même pour les cas T-TRUE, T-FALSE, T-INT, T-STRING, T-ADDRESS et T-NONE.
— cas T-FUNC ($\Delta - \Sigma, (x : A) \vdash (\lambda y : C \to t_2) : (C \to D)$)
Notons que nous assumons que y n'est pas libre dans v. Si c'est le cas, il faut renommer y dans $(\lambda y : C \to t_2)$, ce qui n'empêche pas la preuve.
Si $x = y$, le lambda bloque la réduction, nous prouvons alors $(\Delta - \Sigma \vdash (\lambda y : C \to t_2) : (C \to D))$ avec l'hypothèse, en remarquant que le contexte $(x : A)$ est dispensable.
Si $x \neq y$, (et que nous assumons que y n'est pas libre dans v) :
Par T-FUNC, nous avons $(\Delta - \Sigma, (x : A), (y : C) \vdash t_2 : D)$.
Nous voulons appliquer l'hypothèse de récurrence, mais ce n'est pas possible directement car le contexte de t_2 ci-dessus est $(\Sigma, (y : C), (x : A))$, ce qui est différent du contexte dans $(\Delta - \Sigma \vdash v : A)$. Cependant comme y n'est pas libre dans v c'est sans effet de l'ajouter à son contexte. Par hypothèse de récurrence cela nous donne $(\Delta - \Sigma, (y : C) \vdash t_2[x := v] : D)$.
Par T-FUNC, nous obtenons $(\Delta - \Sigma \vdash (\lambda y : C \to t_2[x := v]) : (C \to D))$. C'est équivalent à $(\Delta - \Sigma \vdash (\lambda y : C \to t_2)[x := v] : (C \to D))$.
— cas T-APP ($\Delta - \Sigma, (x : A) \vdash (t_1\ t_2) : B$) :
Par T-APP, nous avons $(\Delta - \Sigma, (x : A) \vdash t_1 : (C \to B))$ et $(\Delta - \Sigma, (x : A) \vdash t_2 : C)$.

Par l'hypothèse de récurrence, cela nous donne $(\Delta - \Sigma \vdash t_1[x := v] : (C \to B))$ et $(\Delta - \Sigma \vdash t_2[x := v] : C)$.
Par T-APP, nous avons $(\Delta - \Sigma \vdash ((t_1[x := v])\ (t_2[x := v])))$, ce qui est équivalent à $(\Delta - \Sigma \vdash (t_1\ t_2)[x := v])$.
Conclusions similaires pour les cas T-IF et T-SOME.

— cas T-SEND $(\Delta\,\Sigma, (x : A) \vdash (\texttt{send-msg}\ t_1\ \texttt{to}\ t_2) : \texttt{Unit})$:
Il n'y a rien de spécifique à `send-msg`, c'est similaire au cas de l'application.

— cas T-SPAWN $(\Delta\,\Sigma, (x : A) \vdash (\texttt{spawn}\ s\ \texttt{to}\ t_1) : \texttt{Address})$
Il n'y a rien de spécifique à `spawn`, la condition sur l'`actordef` n'a pas d'influence, c'est similaire au cas de l'application.

□

A présent nous prouvons la propriété sur les règles d'évaluation :

Proposition 6 (`conservation-type`$_{\lambda a}$)**.** *Tout terme bien typé qui s'évalue conserve son type.*

$$\begin{array}{l}\texttt{pour tout}\ \Delta, t, A, \alpha, \Pi, \alpha', t',\\ \quad \texttt{si}\ (\Delta - \varnothing \vdash t : A)\ \texttt{et}\ (\Delta\alpha - t \xrightarrow{\lambda a} \alpha'\Pi - t'),\\ \quad\quad \texttt{alors}\ (\Delta - \varnothing \vdash t' : A)\end{array}$$

Démonstration. Par récurrence sur la relation de typage $(:)$ avec $(\Delta - \varnothing \vdash t : A)$ et en sachant $(\Delta\alpha - t \xrightarrow{\lambda a} \alpha'\Pi - t')$.

— cas T-IDENT $(\Delta - \varnothing \vdash x : A)$: cas impossible car $(x : A) \notin \varnothing$.

— cas T-UNIT $(\Delta - \varnothing \vdash \texttt{unit} : \texttt{Unit})$: cas impossible car `unit` est une valeur et ne peut s'évaluer.
Même conclusion pour les cas T-TRUE, T-FALSE, T-INT, T-STRING, T-ADDRESS et T-NONE.

— cas T-FUNC $(\Delta - \varnothing \vdash f : (A \to B))$: cas impossible car une fonction est une valeur et ne peut s'évaluer.

— cas T-APP $(\Delta - \varnothing \vdash (t_1\ t_2) : B)$
Par cas sur les trois règles d'évaluation d'une application :

  — cas E-APP-1 :
  Par T-APP, nous avons $(\Delta - \varnothing \vdash t_1 : (A \to B))$.
  Par E-APP-1, nous avons $(\Delta\alpha - t_1 \xrightarrow{\lambda a} \alpha'\Pi - t_1')$.
  Par hypothèse de récurrence nous avons $(\Delta - \varnothing \vdash t_1' : (A \to B)$.
  Par E-APP-1, nous avons $(\Delta\alpha - (t_1\ t_2) \xrightarrow{\lambda a} \alpha'\Pi - (t_1'\ t_2))$.
  Par T-APP, nous avons $(\Delta - \varnothing \vdash (t_1'\ t_2) : B)$.

  — cas E-APP-2 : Similaire à E-APP-1.

  — cas E-BETA :
  Par E-BETA, nous avons $t_1 = (\lambda x : A \to t_3)$ et $t_2 = v_2$.
  Par T-APP puis T-FUNC, nous avons $(\Delta - (x : A) \vdash t_3 : B)$.
  Par T-APP, nous avons $(\Delta - \varnothing \vdash v_2 : A)$.
  Par `conservation-type-beta-reduction`$_{\lambda a}$, nous avons $(\Delta - \varnothing \vdash t_3[x := v_2] : B)$.
  Par E-BETA, nous avons $(\Delta\alpha - (t_1\ t_2) \xrightarrow{\lambda a} \alpha\varnothing - t_3[x := v_2])$.

— cas T-IF : similaire au cas T-APP, la réduction en branche gauche ou droite conserve le type.

— cas T-SOME : similaire au cas T-APP, quand le sous-terme est une valeur le terme ne s'évalue pas.

— cas T-SEND $(\Delta - \varnothing \vdash (\texttt{send-msg}\ t_1\ \texttt{to}\ t_2) : \texttt{Unit})$:
Pour les règles E-SEND-1 et E-SEND-2, c'est similaire à l'application.
Pour la règle E-SEND $(\Delta\alpha_1 - (\texttt{send-msg}\ v_1\ \texttt{to}\ \alpha_2) \xrightarrow{\lambda a} \alpha_1(\texttt{msg}\ v_1\ \texttt{to}\ \alpha_2) - \texttt{unit})$:
Par T-UNIT, $(\Delta - \varnothing \vdash \texttt{unit} : \texttt{Unit})$.
Le message créé est dans le côté `<processus>` donc n'est pas concerné par le typage du terme. Nous y remédierons dans une proposition séparée.

— cas T-SPAWN ($\Delta \varnothing \vdash (\texttt{spawn}\ s\ \texttt{with}\ t_1) : \texttt{Address}$) :
C'est similaire au cas T-SEND, nous utilisons T-ADDRESS, et l'acteur nouvellement créé n'a pas d'influence.

□

Typage de système ($:_s$)

Le système est composé de définitions, d'acteurs, d'acteurs en évaluation et de messages. Typer le système consiste en typer toutes ces entités. Les vérifications sont faites de façon indépendante entre les entités, mis à part le fait que nous avons toujours besoin de connaître les définitions. Nous rappelons que pour le moment nous ne vérifions pas que le type d'un message est compatible avec le type d'un acteur destinataire, ce qui fait que nous pouvons avoir des erreurs à l'exécution. Nous corrigerons ce point à la section 5.f.
Il n'y a pas de type intéressant à donner au système, s'il est correct nous lui donnons le type `Unit`.

Dans la définition suivante, Δ est une méta-variable pour `<definitions>`, α est une méta-variable pour `<address>`, Π est une méta-variable `<processus>`, s est une méta-variable pour `<string>`, A et M sont des méta-variables pour `<type>`, t est une méta-variable pour `<term>`, v est une méta-variable pour `<value>`.

Définition 12 (typage système ($:_s$))**.**

$$
\begin{aligned}
&(\Delta\alpha\Pi :_{\texttt{s}} \texttt{Unit}) \Leftrightarrow \\
&\quad \forall(\texttt{actdef}\ s_1\ A_1\ t_1) \in \Delta, && \text{(i)} \\
&\qquad (\Delta - \varnothing \vdash t_1 : (\texttt{Address} \to A_1 \to M_1 \to \texttt{Option}[A_1])) \\
&\quad \wedge \forall(\texttt{actwait}\ s_2\ \alpha_2\ t_2) \in \Pi, && \text{(ii)} \\
&\qquad \exists(\texttt{actdef}\ s_2\ A_2\ t_{2b}) \in \Delta, \\
&\qquad\quad \Delta - \varnothing \vdash t_2 : (M_2 \to \texttt{Option}[A_2]) \\
&\quad \wedge \forall(\texttt{acteval}\ s_3\ \alpha_3\ t_3) \in \Pi, && \text{(iii)} \\
&\qquad \exists(\texttt{actdef}\ s_3\ A_3\ t_{3b}) \in \Delta, \\
&\qquad\quad \Delta - \varnothing \vdash t_3 : \texttt{Option}[A_3] \\
&\quad \wedge \forall(\texttt{msg}\ v_4\ \texttt{to}\ \alpha_4) \in \Pi, && \text{(iv)} \\
&\qquad \Delta - \varnothing \vdash v_4 : M_4
\end{aligned}
$$

Un terme t dans une définition `actdef` est typé par une fonction à trois arguments : l'adresse de l'acteur, la mémoire initiale, et le message. C'est pour ça que le retour de la fonction est $\texttt{Option}[A]$: soit c'est un `some` qui contient une valeur de même type que la mémoire initiale, soit c'est un `none`.
Un `actor` attend un message, c'est une fonction de `actordef` qui a déjà reçu l'adresse et la mémoire initiale.
Un `actorineval` est en cours d'évaluation et a aussi reçu le message.
Pour les messages, nous vérifions que la valeur transportée est typée. Pas besoin de vérifier α car la proposition d'inversion nous garantit déjà qu'une `<address>` est de type `Address`.

Comme pour les termes d'acteurs, un système bien typé soit est une valeur (système), soit n'est pas une forme normale (système) :

Proposition 7 (`progression-type`$_{\texttt{s}}$)**.** *Tout système bien typé soit est une valeur système, soit peut s'évaluer avec* $\xrightarrow{\texttt{s}}$.

$$
\begin{aligned}
&\texttt{pour tout}\ \Delta, \alpha, \Pi, \\
&\quad \texttt{si}\ (\Delta\alpha\Pi :_s \texttt{Unit})\ \texttt{alors} \\
&\qquad \texttt{soit}\ \texttt{Value}_{\texttt{s}}(\Delta\alpha\Pi) && \text{(i)} \\
&\qquad \texttt{soit}\ \neg\texttt{FN}_{\texttt{s}}(\Delta\alpha\Pi) && \text{(ii)}
\end{aligned}
$$

Démonstration. Nous utilisons la définition de typage des systèmes afin de montrer que chaque élément (acteur en mode attente ou évaluation, message en cours de transmission) respecte la propriété par le cas (i) ou/et (ii). Ensuite, il apparaît qu'il suffit d'un seul élément en cas (ii) pour que l'ensemble du système soit dans ce cas (ii) aussi.

— Pour tout ($\texttt{actwait}\ s\ \alpha\ t_1$) $\in \Pi$:
Soit il existe un message à destination d'α, et alors nous validons (ii) car la règle ES-MSG est applicable, soit il n'existe pas de tel message, et nous validons (i) selon la définition de $\texttt{Value}_s$.
— Pour tout ($\texttt{acteval}\ s\ \alpha\ t$) $\in \Pi$:
Par le typage système et la progression λa, nous savons que t soit est une valeur, soit peut s'évaluer.
Soit t est une valeur de la forme `(none of <type>)`, et nous validons (ii) par la règle ES-CLEAN-ACTOR.
Soit t est une valeur de la forme `(some of <value>)`, nous validons (ii) par la règle ES-RESPAWN : ($:_s$) nous dit que la définition d'acteur existe.
Soit t est un terme qui peut s'évaluer et nous validons (ii) par la règle ES-EVAL.
— Pour tout ($\texttt{msg}\ v\ \texttt{to}\ \alpha$) $\in \Pi$:
Soit il existe un destinataire `actwait` correct et nous validons (ii) par la règle ES-MSG.
Soit il existe un destinataire correct non prêt `acteval` et nous venons de montrer que ce dernier valide toujours (ii). Le système peut donc s'évaluer, et notre message sera transmis lorsque l'`acteval` sera prêt.
Soit il n'existe aucun destinataire correct, et nous validons (ii) par la règle ES-CLEAN-MSG.

□

Prouver la conservation du typage système pendant l'évaluation système n'est pas possible avec les définitions actuelles. La valeur d'un message doit être typée, mais avec n'importe quel type. Or la fonction de réception du destinataire accepte un type de message bien précis. Rien ne garantit que les deux types sont identiques.

5.f Typage des envois de message

Nous voulons vérifier que le type de message est bien celui accepté par le destinataire. Nous voulons vérifier à l'endroit du `send-msg` , au moment du typage, pour cela nous allons paramétrer les adresses d'acteurs par le nom de la définition utilisée pour les créer :

⟨*type*⟩ ::= ...
| Address[⟨**string**⟩]

Une fonction de réception se type à présent ainsi :

$$\forall(\texttt{actdef}\ s_1\ A_1\ t_1), \Delta - \varnothing \vdash t : (\texttt{Address}[s_1] \to A_1 \to M \to \texttt{Option}[A_1])$$

Pour maintenir le typage sans inférence que nous utilisons, nous devons ajouter le nom aux valeurs d'adresse, comme nous le faisons pour `none`, mais notons que dans le programme nous manipulons des identifiants d'adresses et rarement des adresses directement, qui apparaissent plutôt pendant l'évaluation.

⟨*value*⟩ ::= ...
| ⟨*address*⟩[⟨**string**⟩]

$$\text{T-ADDRESS}\ \frac{\varnothing}{\Delta - \Sigma \vdash \alpha[s] : \texttt{Address}[s]} \qquad \text{T-SPAWN}\ \frac{\Delta - \Sigma \vdash t_1 : A \quad (\texttt{actdef}\ s\ A\ t_2) \in \Delta}{\Delta - \Sigma \vdash (\texttt{spawn}\ s\ \texttt{with}\ t_1) : \texttt{Address}[s]}$$

Voici les règles d'évaluation pour l'envoi de message, et pour la création d'acteur. Notons que la présence du nom s dans les termes n'a pas d'influence sur l'évaluation.

E-SEND

$$\frac{\varnothing}{\Delta\alpha_1 - (\texttt{send-msg}\ v\ \texttt{to}\ \alpha_2[s]) \xrightarrow{\lambda_a} \alpha_1\,(\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s]) - \texttt{unit}}$$

E-SPAWN

$$\frac{(\texttt{actdef}\ s\ A\ t) \in \Delta \qquad \alpha_2 = next(\alpha_1)}{\Delta\alpha_1 - (\texttt{spawn}\ s\ \texttt{with}\ v) \xrightarrow{\lambda_a} \alpha_2\,(\texttt{actwait}\ s\ \alpha_1[s]\ (t\ \alpha_1[s]\ v)) - \alpha_1[s]}$$

Les autres règles sont à considérer aussi réécrites en transformant α en $\alpha[s]$.

Afin d'obtenir la vérification, nous précisons dans la définition d'acteur le type du message attendu :

⟨*actor-def*⟩ ::= (actordef ⟨*string*⟩ ⟨*type*⟩ ⟨**type**⟩ ⟨*term*⟩)

Nous pouvons à présent vérifier les envois de messages lors du typage :

T-SEND

$$\frac{\Delta - \Sigma \vdash t_1 : M \quad \Delta - \Sigma \vdash t_2 : \texttt{Address}[s] \quad (\texttt{actdef}\ s\ A\ M\ t_3) \in \Delta}{\Delta - \Sigma \vdash (\texttt{send-msg}\ t_1\ \texttt{to}\ t_2) : \texttt{Unit}}$$

Lorsqu'un nouveau message apparaît dans le système, nous savons qu'il a un type correspondant à la fonction de réception du destinataire. Nous l'exprimons en modifiant le typage système ainsi :

Définition 13 (typage système ($:_s$) version messages vérifiés)**.** *Les modifications sont soulignées.*

$$\begin{aligned}
&(\Delta\alpha\Pi :_{\texttt{s}} \texttt{Unit}) \Leftrightarrow \\
&\quad \forall(\texttt{actdef}\ s_1\ A_1\ \underline{M_1}\ t_1) \in \Delta, && \text{(i)} \\
&\qquad (\Delta - \varnothing \vdash t_1 : (\texttt{Address}\underline{[s_1]} \to A_1 \to M_1 \to \texttt{Option}[A_1])) \\
&\quad \wedge \forall(\texttt{actwait}\ s_2\ \alpha_2\underline{[s_2]}\ t_2) \in \Pi, && \text{(ii)} \\
&\qquad \exists(\texttt{actdef}\ s_2\ A_2\ \underline{M_2}\ t_{2b}) \in \Delta, \\
&\qquad\quad \Delta - \varnothing \vdash t_2 : (M_2 \to \texttt{Option}[A_2]) \\
&\quad \wedge \forall(\texttt{acteval}\ s_3\ \alpha_3\underline{[s_3]}\ t_3) \in \Pi, && \text{(iii)} \\
&\qquad \exists(\texttt{actdef}\ s_3\ A_3\ \underline{M_3}\ t_{3b}) \in \Delta, \\
&\qquad\quad \Delta - \varnothing \vdash t_3 : \texttt{Option}[A_3] \\
&\quad \wedge \forall(\texttt{msg}\ v_4\ \texttt{to}\ \alpha_4\underline{[s_4]}) \in \Pi, && \text{(iv)} \\
&\qquad \underline{\exists(\texttt{actdef}\ s_4\ A_4\ M_4\ t_{4b}) \in \Pi,} \\
&\qquad\quad \Delta - \varnothing \vdash v_4 : M_4
\end{aligned}$$

Avant de prouver la conservation du type nous prouvons deux propositions qui comblent les manques de la conservation $_{\lambda a}$. Cette dernière se préoccupe uniquement du terme, et ne donne aucune information sur l'éventuel message ou acteur créé pendant l'évaluation.

Proposition 8 (`nouveau-message-correct`)**.** *Si un message a été créé lors de l'évaluation λa d'un terme clos bien typé, la définition d'acteur associée à l'adresse existe, et le type du message est égal à celui inscrit dans la définition.*

$$\begin{aligned}
&\texttt{pour tout}\ \Delta, t_1, A, \alpha_1, \alpha_1', v, \alpha_2, s, t_1' \\
&\quad \texttt{si}\ (\Delta - \varnothing \vdash t_1 : A)\ \texttt{et}\ (\Delta\alpha_1 - t_1 \xrightarrow{\lambda_a} \alpha_1'\,(\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s]) - t_1'), \\
&\quad \texttt{alors il existe}\ B, M, t_2\ \texttt{tels que} \\
&\qquad (\texttt{actdef}\ s\ B\ M\ t_2) \in \Delta\ \texttt{et}\ (\Delta - \varnothing \vdash v : M)
\end{aligned}$$

Démonstration. Par récurrence sur la définition de $\xrightarrow{\lambda_a}$:
- cas E-APP-1 $(\Delta\alpha_1 - (t_{1a}\ t_{1b}) \xrightarrow{\lambda_a} \alpha_1'\ (\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s]) - (t_{1a}'\ t_{1b}))$:
  Par E-APP-1, nous avons $\Delta\alpha - t_{1a} \xrightarrow{\lambda_a} \alpha_1'\ (\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s]) - t_{1a}'$.
  Par T-APP, nous avons $\Delta - \varnothing \vdash t_{1a} : (B \to A)$.
  Par hypothèse de récurrence, nous avons $(\texttt{actdef}\ s\ A\ M\ t_a) \in \Delta$ et $(\Delta - \varnothing \vdash v : M)$.
  Nous avons terminé puisque le même message est utilisé dans la conclusion de E-APP-1.
- cas E-APP-2 : Similaire à E-APP-1.
- cas E-BETA : Cas impossible car il ne permet pas d'avoir le message dans les processus de sortie.
- cas E-IF-1 : Similaire à E-APP-1.
- cas E-IF-T, E-IF-F : Similaire à E-BETA.
- cas E-SOME, E-SEND-1, E-SEND-2 : Similaire à E-APP-1.
- cas E-SEND $(\Delta\alpha_1 - (\texttt{send-msg}\ v\ \texttt{to}\ \alpha_2[s]) \xrightarrow{\lambda_a} \alpha_1\ (\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s]) - \texttt{unit})$:
  Par T-SEND (version modifiée), nous avons $(\Delta - \varnothing \vdash v : M)$ et $(\Delta - \varnothing \vdash \alpha_2[s] : \texttt{Address}[s])$ et $(\texttt{actdef}\ s\ B\ M\ t_3) \in \Delta$. C'est exactement ce que nous avons besoin pour la propisition.
- cas E-SPAWN-1 : Similaire à E-APP-1.
- cas E-SPAWN : Cas impossible, similaire à E-BETA.

□

Proposition 9 (`nouvel-acteur-correct`)**.** *Si l'évaluation d'un terme clôt bien typé génère un nouvel acteur, alors celui-ci a une définition associée, et il est le résultat de l'application de la fonction de réception à a propre adresse et à une valeur (mémoire) du bon type.*

$$\begin{array}{l}\texttt{pour tous}\ \Delta, t_1, A, \alpha_1, \alpha_1', s, \alpha_2, t_1', t_2,\\ \quad \texttt{si}\ (\Delta - \varnothing \vdash t_1 : A)\ \texttt{et}\ (\Delta\alpha_1 - t_1 \xrightarrow{\lambda_a} \alpha_1'\ (\texttt{actwait}\ s\ \alpha_2[s]\ t_2) - t_1')\\ \quad \texttt{alors il existe}\ B, M, t_3, v\ \texttt{tels que}\\ \qquad (\texttt{actdef}\ s\ B\ M\ t_3) \in \Delta\ \texttt{et}\ (t_2 = (t_3\ \alpha_2[s]\ v))\ \texttt{et}\ (\Delta - \varnothing \vdash v : B)\end{array}$$

Démonstration. Par récurrence sur la définition de $\xrightarrow{\lambda_a}$:
- cas E-APP-1 $(\Delta\alpha_1 - (t_{1a}\ t_{1b}) \xrightarrow{\lambda_a} \alpha_1'\ (\texttt{actwait}\ s\ \alpha_2[s]\ t_2) - (t_{1a}'\ t_{1b}))$:
  Par E-APP-1 nous avons $(\Delta\alpha_1 - t_{1a} \xrightarrow{\lambda_a} \alpha_1'\ (\texttt{actwait}\ s\ \alpha_2[s]\ t_2) - t_{1a}')$.
  Par T-APP nous avons $\Delta \varnothing \vdash t_{1a} : (C \to A)$.
  Par hypothèse de récurrence nous avons $(\texttt{actdef}\ s\ B\ t_3) \in \Delta$ et $(t_2 = (t_3\ \alpha_2[s]\ v))$ et $(\Delta - \varnothing \vdash v : B)$.
  C'est le même acteur utilisé dans la conclusion d'E-APP-1, donc nous avons terminé.
- cas E-APP-2 : Similaire à E-APP-1.
- cas E-BETA : Il n'y a pas d'acteur créé donc ce cas est impossible.
- cas E-IF-1 : Similaire à E-APP-1.
- cas E-IF-T et E-IF-F : Similaire à E-BETA.
- cas E-SOME, E-SEND-1, E-SEND-2 : Similaire à E-APP-1.
- cas E-SEND : Similaire à E-BETA.
- cas E-SPAWN-1 : Similaire à E-APP-1.
- cas E-SPAWN $(\Delta\alpha_1 - (\texttt{spawn}\ s\ \texttt{with}\ v) \xrightarrow{\lambda_a} \alpha_2\ (\texttt{actwait}\ s\ \alpha_1[s]\ (t_3\ \alpha_1[s]\ v)) - \alpha_1[s])$:
  Par T-SPAWN (version modifiée) nous avons $(\Delta - \varnothing \vdash v : B)$ et $(\texttt{actdef}\ s\ B\ M\ t_3) \in \Delta$.
  C'est le même acteur utilisé dans la conclusion d'E-SPAWN donc nous avons terminé.

□

Nous pouvons à présent définir la proposition de conservation pour les systèmes :

Proposition 10 (`conservation-type`$_s$)**.** *Tout système bien typé qui s'évalue est encore bien typé.*

$$\begin{array}{l}\texttt{pour tout}\ \Delta, \alpha, \Pi, \alpha', \Pi'\\ \quad \texttt{si}\ (\Delta\alpha\Pi :_s \texttt{Unit})\ \texttt{et}\ (\Delta\alpha\Pi \xrightarrow{s} \Delta\alpha'\Pi')\\ \quad \texttt{alors}\ (\Delta\alpha'\Pi' :_s \texttt{Unit})\end{array}$$

Démonstration. Par cas sur les évaluations systèmes :

— cas ES-EVAL $(\Delta\alpha\Pi \parallel (\texttt{acteval}\ s\ \alpha[s]\ t) \xrightarrow{s} \Delta\alpha'\Pi \parallel \Pi' \parallel (\texttt{acteval}\ s\ \alpha[s]\ t')$:
Par $(:_s)$, nous avons $(\Delta - \varnothing \vdash t : \texttt{Option}[A])$.
Par ES-EVAL, nous avons $(\Delta\alpha - t \xrightarrow{\lambda a} \alpha'\Pi' - t')$.
Par la conservation λa, nous avons $(\Delta - \varnothing \vdash t' : \texttt{Option}[A])$.
Il nous reste à vérifier Π'. Par $\xrightarrow{\lambda a}$, nous montrons facilement que soit Π' est vide, soit il contient un message, soit il contient un acteur. Vérifions ces deux derniers cas :
— cas $\Pi' = (\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s_2])$:
Par `nouveau-message-correct`, nous avons $(\texttt{actdef}\ s_2\ B\ M\ t_2) \in \Delta$ et $(\Delta - \varnothing \vdash v : M)$.
C'est ce qui est demandé par $(:_s)$.
— cas $\Pi' = (\texttt{actwait}\ s_2\ \alpha_2[s_2]\ t_2)$:
Par `nouvel-acteur-correct`, nous avons $(\texttt{actdef}\ s_2\ B\ M\ t_3) \in \Delta)$ et $(t_2 = (t_3\ \alpha_2[s_2]\ v))$ et $(\Delta - \varnothing \vdash v : B)$.
Par $(:_s)$, nous avons $(\Delta - \varnothing \vdash t_3 : (\texttt{Address}[s_2] \to B \to M \to \texttt{Option}[B]))$.
Par T-ADDRESS, nous avons $(\Delta - \varnothing \vdash \alpha_2[s_2] : \texttt{Address}[s_2])$.
Par deux fois T-APP, nous avons $(\Delta - \varnothing \vdash (t_3\ \alpha_2[s_2]\ v) : (M \to \texttt{Option}[B]))$.
C'est ce qui est demandé par $(:_s)$.

— cas ES-RESPAWN $(\Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ \alpha_2[s]\ (\texttt{some}\ v)) \xrightarrow{s} \Delta\alpha_1\Pi \parallel (\texttt{actwait}\ s\ \alpha_2[s]\ (t\ \alpha_2[s]\ v))$:
Par $(:_s)$, nous avons $(\texttt{actdef}\ s\ A\ M\ t) \in \Delta$ et $(\Delta - \varnothing \vdash (\texttt{some}\ v) : \texttt{Option}[A])$ et $(\Delta - \varnothing \vdash t : \texttt{Address}[s] \to A \to M \to \texttt{Option}[A])$.
Par T-SOME, nous avons $(\Delta - \varnothing \vdash v : A)$.
Par T-ADDRESS, nous avons $(\Delta - \varnothing \vdash \alpha_2[s] : \texttt{Address}[s])$.
Par deux fois T-APP, nous avons $(\Delta - \varnothing \vdash (t\ \alpha_2[s]\ v) : M \to \texttt{Option}[A])$.
C'est ce qui est demandé par $(:_s)$.

— cas ES-MSG $(\Delta\alpha_1\Pi \parallel (\texttt{actwait}\ s\ addr_2[s]\ t) \parallel (\texttt{msg}\ v\ \texttt{to}\ addr_2[s])) \xrightarrow{s} \Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ addr_2[s]\ (t\ v))$:
Par $(:_s)$, nous avons $(\texttt{actdef}\ s\ A\ M\ t_2) \in \Delta$ et $(\Delta - \varnothing \vdash t : M \to \texttt{Option}[A])$ et $(\Delta - \varnothing \vdash v : M)$.
Par T-APP, nous avons $(\Delta - \varnothing \vdash (t\ v) : \texttt{Option}[A])$.
C'est ce qui est demandé par $(:_s)$.

— cas ES-CLEAN-ACTOR $(\Delta\alpha_1\Pi \parallel (\texttt{acteval}\ s\ \alpha_2\ (\texttt{none of}\ A)) \xrightarrow{s} \Delta\alpha_1\Pi)$:
Le système était bien typé. Il l'est toujours car il n'a fait que perdre un acteur en évaluation, dont ne dépendait aucun autre élément des processus pour le typage.

— cas ES-CLEAN-MSG $(\Delta\alpha_1\Pi \parallel (\texttt{msg}\ v\ \texttt{to}\ \alpha_2[s]) \xrightarrow{s} \Delta\alpha_1\Pi)$:
Même conclusion que le cas ES-CLEAN-ACTOR.

□

petit programme

Voici un exemple de petit programme, avec un acteur "Main" qui crée un acteur "Compteur" et lui envoie le message "increment".

```
∅
‖ (actdef ``Compteur'' Int
    (λself:Address[“Compteur”]→ (λcompteur:Int→ (λmsg:String→
        (if (msg = ``increment'') then (some (compteur + 1))
         else (if (msg = ``stop'')
                then (none of Int)
                else (some compteur)))))))
‖ (actdef ``Main'' Unit
    (λself:Address[“Main”]→ (λmem:Unit→ (λmsg:Unit→
        let addrOfCompteur:Address[``Compteur''] = (spawn ``Compteur'' with 0) in
        let unit:Unit = (send-msg ``increment'' to addrOfCompteur) in
        (none of Unit)))))
α1
∅
‖ (actwait ``Main'' α0[``Main''] t)
‖ (msg unit to α0[``Main''])
```

Figure 5.f.1 – *Exemple de programme avec acteurs typés*

Typage et expressivité

Le typage n'a pas une précision parfaite, il interdit des termes et des systèmes qui s'évaluent sans erreur. Il existe des exemples peu importants, comme le fait de donner un argument du mauvais type à une fonction qui n'utilise jamais son argument : $((\lambda \texttt{x} : \texttt{Int} \rightarrow 3)\ \texttt{"a"})$. En revanche, il y a par exemple la question du polymorphisme, où la même fonction pourrait servir à plusieurs types :

```
let f :(Int → Int) = (λx:Int → x) in
let n :Int = (f 33) in
let s :String = (f "a")
```

Nous prenons un exemple simple, mais il existe des usages plus utiles de polymorphie, par exemple les fonctions des bibliothèques de structures de données : `map`, `iter`... qui sont définies une seule fois et sont bien typées pour n'importe quel type d'éléments de la structure.
Nous pouvons aussi penser à des cas où plusieurs types sont possibles, et aux types union, par exemple :

```
let x :Int = (if cond then 3 else "a") in
let y :Int = (if cond then x + 1 else "b")
```

Cela vaut aussi pour les systèmes d'acteurs : le type de la fonction de réception détermine le type de message accepté par l'acteur. C'est très restrictif : un acteur doit choisir entre recevoir `Int` et `String`. Il faut ajouter un mécanisme de types union, ou de sous-typage.

Il y a un point de différence entre les fonctions de réception d'acteur et les fonctions classiques. Une fonction classique a un type d'entrée et un type de sortie. Cela suffit à décrire le contexte dans lequel elle sera utilisée, et elle se contente de respecter ces deux types. Les fonctions de réception ont toujours `Unit` en type de sortie : pour communiquer une valeur "en réponse", elle doivent envoyer un message à une adresse. Dans Akka non typé, il existe un mot-clé `sender` qui s'évalue en l'adresse de l'émetteur du message. Dans Akka typé, il faut explicitement donner une adresse (typée) dans le contenu du message lui-même. Pour retrouver l'expressivité de `return`, qui exprime "n'importe quel contexte qui accueuille le type de sortie", il faut utiliser un paramètrage des adresses par "message" au lieu de "nom de définition". C'est ainsi que fonctionne Akka typé, qui utilise par exemple un type comme `Address[String]` au lieu de `Address[Compteur]`, ce qui exprime tout acteur pouvant recevoir un message de type `String`.
Ensuite, ceci peut être augmenté par de la polymorphie bornée par exemple : $\forall A <: \texttt{String}.\texttt{Address}[A]$.

En conclusion, en définissant les acteurs avec un langage lambda calcul augmenté, et en utilisant la notion de *système* (ensemble d'acteurs, de définitions et de messages), nous avons pu appliquer une vérification simple des types d'envois de messages. Celle-ci fonctionne de façon similaire à la vérification des types sur le lambda calcul basique.
En utilisant un modèle où les acteurs sont créés à partir d'une *définition*, où la réception de message se

fait une seule fois et n'est pas un mot-clé accessible au programmeur, nous avons simplifié le typage de réception de l'acteur en le concentrant à un seul endroit. Nous interdisons ainsi un type "à états" (ce qu'on peut trouver dans d'autres modèles d'acteurs) et le mot-clé *become* disponible dans la bibliothèque *Akka* version non typée. C'est un choix dont les conséquences restent à évaluer : les acteurs avec "types à état" sont plus faciles à définir puisqu'il y a moins de situations erronées à gérer. Ils sont plus difficiles à vérifier : il faut savoir dans quel état se situe un acteur pour vérifier s'il peut recevoir un certain type de message. C'est le travail effectué sur les *types sessions* [NY14], que nous n'abordons pas ici. Nous assumons qu'il est déjà efficace, même si moins expressif, de vérifier les envois de messages dans les définitions d'acteurs typées "à un seul état". La bibliothèque *Akka* a déjà largement expérimenté dans ce sens, et un avantage supplémentaire par rapport aux types sessions est que ce typage s'exprime aisément dans le typage existant du langage *Scala*.
Il reste aussi à ajouter formellement au langage de types des mécanismes supplémentaires : polymorphie, bornes, sous-typage, etc.

Chapitre 6

Réalisation : plateforme de simulations

Nous avons travaillé à la réalisation d'une "plateforme de simulations". Il s'agit d'un logiciel capable de lancer des simulations de façon simultanée et de communiquer leurs résultats. Le logiciel peut s'étendre sur plusieurs machines, étant donné les ressources potentiellement nécessaires pour lancer chaque simulation. Les programmes de simulateurs sont fixés à l'avance par l'administrateur, les utilisateurs ne peuvent pas en ajouter. En revanche, ils peuvent demander à exécuter une simulation en indiquant des paramètres initiaux. Le logiciel est pensé pour fonctionner avec des simulations à période, ce qui permet à l'utilisateur d'indiquer une période maximum, et au logiciel de découper les périodes en paquets. Le logiciel a un côté "serveur" : l'ensemble de machines qui répond aux requêtes et exécute des simulations, et un côté "client" : un (ou des) utilisateur avec un navigateur qui communique avec le serveur. L'utilisateur indique un simulateur (parmi une liste), une période maximale, et des paramètres de départ. Le serveur exécute la simulation, et envoie au fur et à mesure les résultats au client. Le client affiche les résultats (les paquets de périodes) au fur et à mesure de leur réception. Pour être plus précis, le côté serveur utilise une base de données pour pérériser les données des simulations. La première fois, les données sont ajoutées à la base de données, et toute lecture interroge directement la base. Même quand les données sont déjà en base, le concept d'envoi en paquets reste pertinent, car l'envoi de toutes les périodes en une seule fois peut prendre du temps, et donc abîmer la qualité de l'interface client.
Côté client, le logiciel présente donc une interface permettant de faire un choix parmi les simulations disponibles et les paramètres. Éventuellement, il peut connaître quelles sont les configurations déjà présentes en base de données. Lorsqu'il a fait un choix, l'interface est capable de recevoir les données sur les périodes en paquet, et les afficher dans des graphiques et tableaux. Elle est capable d'observer que les prochaines données ne sont pas arrivées, sans bloquer l'interface complète. Elle présente des fonctionnalités de "zoom" sur les graphiques, et autres informations en "overlay", tout en gérant la mise à jour de ces graphiques lors de la réception d'un paquet de périodes.
Côté serveur, le logiciel possède des exécutables de simulateurs à lancer, un service HTTP et un service de base de données. Lorsqu'il reçoit une demande initiale, contenant un nom de simulateur et des paramètres, il regarde si elle existe déjà en base. Sinon, il lance la simulation. Comme il s'agit de HTTP, l'initiative des communications appartient au client, qui demande les paquets de périodes. Le serveur indique au client quand un paquet n'est pas prêt.

À titre d'exemple, le simulateur *Jamel* nécessite en général un millier de périodes, et possède une cinquantaine de paramètres. Le temps d'exécution est entre dix et trente secondes. Il y a une cinquantaine de valeurs enregistrées à chaque période.

Pour la partie serveur, nous avons utilisé la bibliothèque *Akka* : il y a un acteur pour chaque client, et un acteur pour chaque exécution de simulation. Pour le serveur nous avons utilisé *Spray*, un outil qui est à présent directement intégré à la bibliothèque *Akka*. Pour la communication de paquets de données, nous utilisons le format *JSON*. Pour l'affichage des graphiques côté client, nous utilisons la bibliothèque *D3JS*.
Dans la version courante du logiciel, nous n'avons que le simulateur fonctionnel. Dans une version pré-

cédente, nous avions aussi intégré *Jamel*, mais nous ne l'avons pas encore repris. Pour le paramétrage, le système est capable de distinguer deux demandes de simulations différentes, au niveau du lancement d'acteurs et d'enregistrement en base de données, mais ne permet pas encore d'appliquer les paramètres. Pour la base de données, nous avons utilisé un *Redis* simple, ce qui ne passe pas à l'échelle et nécessite du travail. Le projet actuel comporte environ 1500 lignes de code (*Scala*, *Javascript*, *Html*, *Css*). Il est consultable à l'adresse `https://github.com/antoinekagit/plateforme-sim`.

Les informations à récupérer de la simulation sont les états entre chaque période, ainsi que certains évènements survenus pendant les périodes. Elles sont réduites de façon "macro" : par exemple, le salaire moyen des travailleurs durant la période. Nous pouvons aussi extraire quelques informations "micro" : par exemple un suivi complet d'une entreprise en particulier (ses variables, ses choix, etc..).

Conclusion de la thèse

En prenant comme point de départ les simulateurs d'économie, nous nous plaçons dans le contexte de programmes complexes. Le nombre d'agents, le nombre d'interactions, la grande taille des données générées, rendent difficile la résolution d'erreurs.

Le renforcement de propriétés, par exemple avec un type abstrait, semble exagéré lorsque nous le voyons sur de petits exemples. Mais il permet de réduire les possibilités du programme, le rendant plus simple à comprendre, et plus facile à vérifier. Il augmente la séparation entre les concepts à l'intérieur du programme. Il oblige le programmeur à gérer les erreurs lorsqu'il crée une valeur du type abstrait, ce qui est un avantage pour la sécurité, et un désavantage pour la facilité d'utilisation de la bibliothèque.

La programmation fonctionnelle, soit l'absence de mutation, supprime une partie des erreurs possibles, en simplifiant les scénarios d'évolution de valeur d'une variable. Les modifications sont locales, et ce qui n'est pas local est dans la clôture et donc est constant.
Avec une variable mutable, nous pouvons ne pas bien comprendre son évolution, par exemple après avoir appelé plusieurs fonctions qui ont accès à la variable. Si nous voulons vérifier que la variable n'a pas changé, il faut lire le code des fonctions appelées. Avec une variable immutable, nous savons qu'elle n'a pas changé. Mais avec le mécanisme de surcharge, nous attribuons de nouvelles valeurs au même nom. Nous introduisons donc un nouveau type d'erreur : quand la variable mutable avait une valeur unique (la dernière en date), la variable immutable conserve plusieurs de ses valeurs en même temps. Il est donc possible de se tromper en croyant utiliser la dernière valeur.
La monade *state* est particulière, puisqu'en un sens, elle rétablit presque l'unicité de la valeur, en rendant par défaut implicite la variable auxiliaire. Cependant elle laisse la possibilité de contrôler : il est plus facile de relancer deux fois ou de jeter l'exécution d'une valeur monadique *State*, que de dupliquer dans le code une variable mutable.

Les monades sont une structure ambitieuse, qui vise à doter la programmation fonctionnelle de concepts non fonctionnels, de modifier les capacités du langage de travail lui-même. La nécessaire expressivité de la fonction `bind`, avec la fusion de contextes et la règle d'associativité à respecter, réduit en même temps les possibilités d'une monade. La tendance à utiliser une monade pour générer des symboles syntaxiques, qui sont plus tard interprétés par une fonction `run`, revient à créer un langage à l'intérieur du langage. C'est intéressant dans un contexte de création de *DSL* [1], mais ce n'est peut-être pas pertinent en général, car le langage de base n'est sans doute pas prévu et optimisé dans ce but. L'idée directrice semble être la création d'un langage encore plus haut niveau, plus séparé de la machine. Nous avons vu qu'il existe encore des limites qu'il faut prendre en compte, lorsqu'il s'agit de la mémoire disponible.
Les monades seront encore plus puissantes lorsqu'il sera facilement possible de désigner une monade et d'abstraire la combinaison de monades dans laquelle elle est plongée. La valeur monadique est une véritable boîte noire, et il faut utiliser l'interface fournie pour la manipuler. C'est à la fois un renforcement de la sûreté du programme, et la mise en place d'une structure déjà prévue pour évoluer.

La solution des concrets monadiques dévelopée dans ce document prend le parti de l'activation manuelle,

1. /Domain Specific Language/ : un langage créé pour une application spécifique. Autrement dit il n'est pas prévu pour écrire n'importe quel type de programme (jeu, calculcateur, application web...)

comme l'est la solution d'appel terminal. Nous aurions bien sûr préféré réussir à définir une structure qui soit efficace dans tous les contextes. Les concrets monadiques ne s'inscrivent pas exactement dans le courant des monades. Nous nous sommes concentrés sur un ensemble de monades qui sont celles dont nous avions besoin pour les programmes de simulateurs. C'est donc un point de vue qui ne cherche pas à être valide pour toute monade. Aussi, nous avons écarté la solution de séparation entre construction syntaxique et interprétation. C'était dans une logique de rester proche des structures du langage de base, et de leur performance, ce qui va à l'encontre de l'abstraction complète. Il est difficile de dire si c'est la bonne approche, nous verrons comment évoluent les monades. Il semble que bien qu'elles soient apparues dans d'autres langages que *Haskell*, leur utilisation soit encore très restreinte aux structures de données (listes, etc), et que toute valeur du programme ne soit pas monadique. Notons que nous avons toutefois pris soin de conserver la modularité en définissant des solutions qui s'adaptent aux transformateurs de monades.

En conclusion, en définissant les acteurs avec un langage lambda calcul augmenté, et en utilisant la notion de *système* (ensemble d'acteurs, de définitions et de messages), nous avons pu appliquer une vérification simple des types d'envois de messages. Celle-ci fonctionne de façon similaire à la vérification des types sur le lambda calcul basique.
En utilisant un modèle où les acteurs sont créés à partir d'une *définition*, où la réception de message se fait une seule fois et n'est pas un mot-clé accessible au programmeur, nous avons simplifié le typage de réception de l'acteur en le concentrant à un seul endroit. Nous interdisons ainsi un type "à états" (ce qu'on peut trouver dans d'autres modèles d'acteurs) et le mot-clé *become* disponible dans la bibliothèque *Akka* version non typée. C'est un choix dont les conséquences restent à évaluer : les acteurs avec "types à état" sont plus faciles à définir puisqu'il y a moins de situations erronées à gérer. Ils sont plus difficiles à vérifier : il faut savoir dans quel état se situe un acteur pour vérifier s'il peut recevoir un certain type de message. C'est le travail effectué sur les *types sessions*, que nous n'abordons pas ici. Nous assumons qu'il est déjà efficace, même si moins expressif, de vérifier les envois de messages dans les définitions d'acteurs typées "à un seul état". La bibliothèque *Akka* a déjà largement expérimenté dans ce sens, et un avantage supplémentaire par rapport aux types sessions est que ce typage s'exprime aisément dans le typage existant du langage *Scala*.
Il reste aussi à ajouter formellement au langage de types des mécanismes supplémentaires : polymorphie, bornes, sous-typage, etc.

Dans la partie Akka, nous avons suivi le même objectif de sûreté : donner de meilleures garanties de succès aux programmes. Cela s'est traduit par l'ajout de typage des envois de messages, dans un système qui ne le faisait pas initialement. Ce dernier est une modélisation sous forme de lambda calcul d'un extrait de la bibliothèque *Akka* version non typée. Comme exprimé en conclusion de cette partie, bien des travaux abordent la sûreté des acteurs par le *typage session*. C'est une approche différente, qui permet un typage plus précis car possédant la capacité d'exprimer des types "à état", c'est à dire des types qui évoluent au fur et à mesure de l'exécution. Le typage que nous proposons ne possède pas cette caractéristique : le type d'un acteur est constant. Cela oblige à gérer tous les cas possibles, y compris ceux qui en fait n'arrivent jamais dans certaines situations (c'est à dire que l'analyse du programmeur est plus précise que celle du compilateur). Nous pensons néanmoins que cela reste possible de programmer dans ce contexte, et nous mettons en avant que ce typage s'intègre facilement dans les systèmes de types existants, contrairement aux types sessions. Ceci est montré par les expérimentations de la bibliothèque *Akka*. Aussi, il serait intéressant d'explorer une combinaison des deux typages, qui ont des niveaux de précisions différents, et sont donc peut-être complémentaires.

Bibliographie

[Arm03] Joe Armstrong. « Making reliable distributed systems in the presence of software errors ». Thèse de doct. 2003.

[Bja12] Rúnar Bjarnason Óli. « Stackless Scala With Free Monads ». url : http://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf, dernière visite le 27 août 2019. Avr. 2012.

[BKP15] Pierre Boudes, Antoine Kaszczyc et Luc Pellissier. « Monetary Economics Simulation : Stock-Flow Consistent Invariance, Monadic Style ». In : *11th Artificial Economics Conference*. Faculdade de Economia do Porto. Porto, Portugal, sept. 2015. url : https://hal.archives-ouvertes.fr/hal-01181278.

[Bla11] Mario Blažević. « Coroutine Pipelines ». In : *The Monad Reader* 19 (2011). url : https://themonadreader.files.wordpress.com/2011/10/issue19.pdf, dernière visite le 26 août 2019.

[Erk02] Levent Erkok. « Value Recursion in Monadic Computations ». AAI3063791. Thèse de doct. 2002. isbn : 0-493-82294-1.

[Fow19] Simon Fowler. « Typed Concurrent Functional Programming with Channels, Actors, and Sessions ». Thèse de doct. 2019.

[Fre15] Phil Freeman. « Stack Safety for Free ». sur le site personnel de Phil Freeman. document obtenu sur le site internet personnel de Phil Freeman, url : http://functorial.com/stack-safety-for-free/index.pdf, dernière visite le 19 Juillet 2019. Phil Freeman a présenté ce travail à *Code Mesh 2016*. Août 2015. url : http://functorial.com/stack-safety-for-free/index.pdf.

[HBS73] Carl Hewitt, Peter Bishop et Richard Steiger. « A Universal Modular ACTOR Formalism for Artificial Intelligence ». In : *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA : Morgan Kaufmann Publishers Inc., 1973, p. 235-245. url : http://dl.acm.org/citation.cfm?id=1624775.1624804.

[KI15] Oleg Kiselyov et Hiromi Ishii. « Freer Monads, More Extensible Effects ». In : *SIGPLAN Not.* 50.12 (2015), p. 94-105. issn : 0362-1340. doi : 10.1145/2887747.2804319. url : http://doi.acm.org/10.1145/2887747.2804319.

[LHJ98] Sheng Liang, Paul Hudak et Mark Jones. « Monad Transformers and Modular Interpreters ». In : *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (1998). doi : 10.1145/199448.199528.

[Nca] *ncatlab : monad* (*in computer science*). url : https://ncatlab.org/nlab/show/monad+%28in+computer+science%29, dernière visite le 27 août 2019.

[NY14] Rumyana Neykova et Nobuko Yoshida. « Multiparty Session Actors ». In : *16th International Conference on Coordination Models and Languages* (*COORDINATION*). Sous la dir. de David Hutchison et al. T. LNCS-8459. Coordination Models and Languages. Berlin, Germany : Springer, 2014, p. 131-146. doi : 10.1007/978-3-662-43376-8_9. url : https://hal.inria.fr/hal-01290073.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. isbn : 978-0-521-66350-2.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PK14] Atze van der Ploeg et Oleg Kiselyov. « Reflection Without Remorse : Revealing a Hidden Sequence to Speed Up Monadic Reflection ». In : *SIGPLAN Not.* 49.12 (sept. 2014), p. 133-144. issn : 0362-1340. doi : 10.1145/2775050.2633360. url : http://doi.acm.org/10.1145/2775050.2633360.

[PZ17] Jennifer Paykin et Steve Zdancewic. « The Linearity Monad ». In : *SIGPLAN Not.* 52.10 (sept. 2017), p. 117-132. issn : 0362-1340. doi : 10.1145/3156695.3122965. url : http://doi.acm.org/10.1145/3156695.3122965.

[Sep14] Pascal Seppecher. « Pour une macroéconomie monétaire dynamique et complexe ». In : *Revue de la régulation. Capitalisme, institutions, pouvoirs* 16 | 2e semestre / Autumn 2014 (déc. 2014), 2-35 (revue en ligne). doi : 10.4000/regulation.10977. url : https://hal.archives-ouvertes.fr/hal-01097473.

[Sep16] Pascal Seppecher. « Modèles multi-agents et stock-flux cohérents : une convergence logique et nécessaire ». working paper or preprint. Avr. 2016. url : https://hal.archives-ouvertes.fr/hal-01309361.

[SO11] Tom Schrijvers et Bruno C.d.S. Oliveira. « Monads, Zippers and Views : Virtualizing the Monad Stack ». In : *SIGPLAN Not.* 46.9 (sept. 2011), p. 32-44. issn : 0362-1340. doi : 10.1145/2034574.2034781. url : http://doi.acm.org/10.1145/2034574.2034781.

[Wad95] Philip Wadler. « Monads for functional programming ». In : *Advanced Functional Programming*. Sous la dir. de Johan Jeuring et Erik Meijer. Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, p. 24-52. isbn : 978-3-540-49270-2.

Annexes

Annexe A

Définitions OCaml basiques de monades usuelles

A.1 Monade `State`

```
module type TYPE = sig
    type t
end

module State (Var : TYPE) = struct
    type 'a monad = (Var.t -> ('a * Var.t))

    let return a var = (a, var)

    let bind f ma var0 =
        let (a, var1) = (ma var0) in
        (f a var1)
end
```

A.2 Monade `Reader`

```
module type TYPE = sig
    type t
end

module Reader (Par : TYPE) = struct
    type 'a monad = (Par.t -> 'a)

    let return a par = a

    let bind f ma par = (f (ma par) par)
end
```

A.3 Monade `Writer`

```
module type MONOID = sig
    type t
    val nil : t
    val add : (t -> t -> t)
end

module Writer (Var : MONOID) = struct
    type 'a monad ('a * Var.t)

    let return a = (a, Var.nil)

    let bind f ma =
        let (a, var1) = ma in
        let (b, var2) = (f a) in
        (b, Var.add var1 var2)
end
```

A.4 Monade `Option`

```
module Option = struct
    type 'a monad = ('a option)

    let return a = (Some a)

    let bind f ma =
        match ma with
        | (Some a) -> (f a)
        | None -> None
end
```

A.5 Monade `List`

```
module List = struct
    type 'a monad = ('a list)

    let return a = [a]

    let bind f ma =
        (List.flatten (List.map f ma))
end
```

A.6 Monade `Continuation`

```
module Continuation = struct
    type ('a,'r) monad = (('a -> 'r) -> 'r)

    let return a rA = (rA a)

    let bind f ma rB =
        (ma (fun a -> (f a rB)))
end
```

Annexe B

Preuves

B.1 Preuve d'équivalence des deux façons d'écrire les règles des monades

Nous travaillons avec les définitions suivantes :

```
(ma >>= f)   ===   (bind f ma)                                  (* définition de >>= *)

((ma >>= f) >>= g)   ===   (ma >>= (fun a -> (f a) >>= g))   (* associativité >>= *)
                           (* a n'est pas libre dans f et g *)

((return a) >>= f)   ===   (f a)                                 (* neutralité gauche >>= *)

(ma >>= return)   ===   ma                                       (* neutralité droite >>= *)

(f >>> g)   ===   (fun a -> bind g (f a))                       (* définition de >>> *)
                  (* a n'est pas libre dans f et g *)

((f >>> g) >>> h)   ===   (f >>> (g >>> h))                      (* associativité >>> *)

(return >>> f)   ===   f   ===   (f >>> return)          (* neutralité gauche et droite >>= *)
```

Nous voulons montrer qu'en assumant les définitions de >>= et >>>, les propriétés de neutralité et d'associativité de >>= sont équivalentes à celles de >>>.

Neutralité gauche »=

En supposant les définitions de >>= et >>>, et les propriétés d'associativité et de neutralité sur >>>, nous montrons la propriété de neutralité gauche sur >>=.

```
((return a) >>= f)
------------------ (* définition de >>= *)
(bind f (return a))
------------------------------- (* expansion *)
((fun a -> bind f (return a)) a)
------------------------------- (* définition de >>> *)
((return >>> f) a)
------------------ (* neutralité gauche >>> *)
(f a)
```

Neutralité droite »=

En supposant les définitions de >>= et >>>, et les propriétés d'associativité et de neutralité sur >>>, nous montrons la propriété de neutralité droite sur >>=.

```
ma >>= return
--------------- (* définition de >>= *)
bind return ma
------------------------------- (* expansion (x et y non libres dans ma) *)
bind return ((fun x -> ma) y)
------------------------------------------ (* expansion *)
(fun y -> bind return ((fun x -> ma) y)) y
------------------------------------------ (* définition de >>> *)
((fun x -> ma) >>> return) y
--------------------------- (* neutralité droite >>> *)
(fun x -> ma) y
-------------- (* beta reduction *)
ma
```

Associativité »=

En supposant les définitions de >>= et >>>, et les propriétés d'associativité et de neutralité sur >>>, nous montrons la propriété d'associativité sur >>=.

```
(ma >>= f) >>= g
----------------- (* définition >>= x2 *)
bind g (bind f ma)
------------------------------- (* expansion *)
bind g (bind f ((fun x -> ma) y))
------------------------------------------- (* expansion *)
bind g ((fun y -> bind f ((fun x -> ma) y)) y)
------------------------------------------- (* définition >>> *)
bind g (((fun x -> ma) >>> f) y)
------------------------------------------ (* expansion *)
(fun y -> bind g (((fun x -> ma) >>> f) y)) y
------------------------------------------ (* définition >>> *)
(((fun x -> ma) >>> f) >>> g) y
------------------------------ (* associativité >>> *)
((fun x -> ma) >>> (f >>> g)) y
------------------------------------------ (* définition >>> *)
(fun y -> bind (f >>> g) ((fun x -> ma) y)) y
------------------------------------------ (* beta reduction x2 *)
bind (f >>> g) ma
------------------------------ (* définition >>> *)
bind (fun a -> bind g (f a)) ma
------------------------------ (* définition >>= x2 *)
ma >>= (fun a -> (f a) >>= g)
```

Neutralité gauche »>

En supposant les définitions de >>= et >>>, et les propriétés d'associativité et de neutralité sur >>=, nous montrons la propriété de neutralité gauche sur >>>.

```
(return >>> f)
---------------------------- (* définition >>> *)
(fun a -> bind f (return a))
---------------------------- (* définition >>= *)
(fun a -> (return a) >>= f)
---------------------------- (* neutralité gauche >>= *)
(fun a -> (f a))
---------------- (* eta expansion *)
f
```

Neutralité droite »>

En supposant les définitions de >>= et >>>, et les propriétés d'associativité et de neutralité sur >>=, nous montrons la propriété de neutralité droite sur >>>.

```
(f >>> return)
---------------------------- (* définition >>> *)
(fun a -> bind return (f a))
---------------------------- (* définition >>= *)
(fun a -> (f a) >>= return)
---------------------------- (* neutralité droite >>= *)
(fun a -> (f a))
---------------- (* eta expansion *)
f
```

Associativité »>

En supposant les définitions de >>= et >>>, et les propriétés d'associativité et de neutralité sur >>=, nous montrons la propriété d'associativité sur >>>.

```
((f >>> g) >>> h)
------------------------------ (* définition >>> *)
(fun a -> bind h ((f >>> g) a)
---------------------------------------------- (* définition >>> *)
(fun a -> bind h ((fun a -> bind g (f a)) a))
---------------------------------------------- (* beta reduction *)
(fun a -> bind h (bind g (f a)))
------------------------------- (* définition >>= x2 *)
(fun a -> ((f a) >>= g) >>= h)
---------------------------------------- (* associativité >>= *)
(fun a -> (f a) >>= (fun b -> (g b) >>= h))
---------------------------------------- (* définition >>= x2 *)
(fun a -> bind (fun b -> bind h (g b)) (f a))
---------------------------------------------- (* définition >>> *)
(fun a -> bind (g >>> h) (f a))
------------------------------ (* définition >>> *)
(f >>> (g >>> h))
```

B.2 Preuve que `WriterM` est une monade pour toute monade `M`

Règle d'associativité

```
bind g (bind f ma)
------------------------------------------------------------------- deplier bind
M.bind (fun (b, logs12) ->
           M.map (fun (c, logs3) -> (c, logs12 ++ logs3))
                 (g b))
       (M.bind (fun (a, logs1) ->
                    M.map (fun (b, logs2) -> (b, logs1 ++ logs2))
                          (f a))
               ma
```

```
------------------------------------------------------------------ remplacer M.map (2x)
M.bind (fun (b, logs12) ->
            M.bind (fun (c, logs3) -> M.return (c, logs12 ++ logs3))
                   (g b))
       (M.bind (fun (a, logs1) ->
                    M.bind (fun (b, logs2) -> M.return (b, logs1 ++ logs2))
                           (f a))
               ma
------------------------------------------------------------------ associativite M.bind (2x)
M.bind (fun (a, logs1) ->
            M.bind (fun (b, logs2) ->
                         M.bind (fun (b, logs12) ->
                                      M.bind (fun (c, logs3) -> M.return (c, logs12 ++ logs3))
                                             (g b))
                                (M.return (b, logs1 ++ logs2)))
                   (f a))
       ma
------------------------------------------------------------------ neutralite gauche M.return
M.bind (fun (a, logs1) ->
            M.bind (fun (b, logs2) ->
                         M.bind (fun (c, logs3) ->
                                      M.return (c, (logs1 ++ logs2) ++ logs3))
                                (g b))
                   (f a))
       ma
------------------------------------------------------------------ associativite (++)
```

(suite)

```
M.bind (fun (a, logs1) ->
            (M.bind (fun (b, logs2) ->
                         M.bind (fun (c, logs3) ->
                                     M.return (c, logs1 ++ (logs2 ++ logs3)))
                                (g b))
                    (f a)))
       ma
------------------------------------------------------------------ neutralite gauche M.return
M.bind (fun (a, logs1) ->
            (M.bind (fun (b, logs2) ->
                         M.bind (fun (c, logs3) ->
                                     M.bind (fun (c, logs23) -> M.return (c, logs1 ++
                                     ↪  logs23))
                                            (M.return (c, logs2 ++ logs3)))
                                (g b))
                    (f a)))
       ma
------------------------------------------------------------------ associativite M.bind (2x)
M.bind (fun (a, logs1) ->
            M.bind (fun (c, logs23) -> M.return (c, logs1 ++ logs23))
                   (M.bind (fun (b, logs2) ->
                                M.bind (fun (c, logs3) -> M.return (c, logs2 ++ logs3))
                                       (g b))
                           (f a)))
       ma
------------------------------------------------------------------ replacer M.map (2x)
M.bind (fun (a, logs1) ->
            M.map (fun (c, logs23) -> (c, logs1 ++ logs23))
                  (M.bind (fun (b, logs2) ->
                               M.map (fun (c, logs3) -> (c, logs2 ++ logs3))
                                     (g b))
                          (f a)))
       ma
------------------------------------------------------------------ replier bind
bind (fun a -> bind g (f a)) ma
```

Neutralité gauche

```
bind f (return a)
------------------------------------------------------------------ deplier bind et return
M.bind (fun (a, logs1) ->
            M.map (fun (b, logs2) -> (b, logs1 ++ logs2))
                  (f a))
       (M.return (a, logs_nil))
------------------------------------------------------------------ neutralite gauche M.bind
 M.map (fun (b, logs2) -> (b, logs_nil ++ logs2))
       (f a)
------------------------------------------------------------------ neutralite gauche (++)
 M.map (fun (b, logs2) -> (b, logs2))
       (f a)
------------------------------------------------------------------ neutralite M.map id
(f a)
```

Neutralité droite

```
bind return ma
------------------------------------------------------------------ deplier bind et return
M.bind (fun (a, logs1) ->
            M.map (fun (b, logs2) -> (b, logs1 ++ logs2))
                  (M.return (a, logs_nil)))
       ma
------------------------------------------------------------------ remplacer bind
M.bind (fun (a, logs1) ->
            M.bind (fun (b, logs2) -> M.return (b, logs1 ++ logs2))
                   (M.return (a, logs_nil)))
       ma
------------------------------------------------------------------ neutralite gauche M.bind
M.bind (fun (a, logs1) ->
            M.return (a, logs1 ++ logs_nil))
       ma
------------------------------------------------------------------ neutralite droite (++)
M.bind (fun (a, logs1) ->
            M.return (a, logs1))
       ma
------------------------------------------------------------------ neutralite droite M.bind
ma
```

Annexe C

Tests de performance

C.1 Procédure de test

Trois scripts bash sont utilisés. Leur but est de générer un tableau de tests pour chaque exécutable dans la liste donnée.

Un test limite soit la taille de la pile, soit la taille de la mémoire virtuelle. Un tableau de test fait varier la taille (pile ou mémoire virtuelle) dans les ordonnées, et fait varier l'argument de l'exécutable testé en abscisses (cet argument est défini comme étant un entier).

Chaque case d'un tableau contient le temps d'exécution. Celui-ci est obtenu par la sortie "temps user" de la commande `/bin/time`. C'est une moyenne sur dix essais. Chaque case indique aussi si une erreur a eu lieu (`stack overflow` ou `out of memory`).

La commande lancée est `/bin/bash lancer_serie_tableaux_tests.bash serie1.txt`. Les scripts et les fichiers de valeurs sont donnés ci-après.

C.2 Caractéristiques de la machine de test

```
$ ocamlopt -version
4.02.3

$ /usr/bin/timeout --version
timeout (GNU coreutils) 8.23

$ /usr/bin/time --version
GNU time 1.7

$ bash --version
GNU bash, version 4.3.30(1)-release (x86_64-pc-linux-gnu)

$ uname -srvmpio
Linux 3.16.0-4-amd64 #1 SMP Debian 3.16.48-1 (2017-09-28) x86_64 unknown unknown GNU/Linux

$ cat /proc/meminfo
MemTotal:       16375920 kB
SwapTotal:      15624188 kB

$ lscpu
Architecture :        x86_64
Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
```

```
Processeur(s) :       4
Thread(s) par cœur : 1
Cœur(s) par socket : 4
Socket(s) :           1
Nœud(s) NUMA :       1
Identifiant constructeur : GenuineIntel
Famille de processeur : 6
Modèle :             60
Nom de modèle :      Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz
Révision :           3
Vitesse du processeur en MHz : 871.171
Vitesse maximale du processeur en MHz : 3900,0000
Vitesse minimale du processeur en MHz : 800,0000
BogoMIPS :            6984.05
Virtualisation :      VT-x
Cache L1d :           32K
Cache L1i :           32K
Cache L2 :            256K
Cache L3 :            6144K
Nœud NUMA 0 de processeur(s) : 0-3
```

C.3 Scripts de tests

Script 1 : lancer un test

Ce script limite soit le stack soit la mémoire virtuelle autorisée, puis lance l'exécutable donné avec l'argument entier donné. Il utilise la commande `ulimit` pour limiter la mémoire et la commande `time` pour mesurer le temps CPU.

Nom : `lancer_test.bash`

```
#!/bin/bash
set -o nounset

# warning : beware of your ram !
# this program will set stack or virtual memory limit
# to the param $2 in kbytes.
# be sure to have enough ram
# or the results will be meaningless


# params :
# $1 mode : "stack" or "virtualmem"
# $2 mode size in kbytes
# $3 executable to launch
# $4 integer argument given to executable

if [[ $# -ne 4 ]]
then
    echo "mauvais nombre d'arguments pour $0"
    exit 1
else
    mode="$1"
    size=$"$2"
    executable="$3"
    n="$4"
fi

if [ ! -f "$executable" ]
then
    echo "$executable is not file"
    exit 1
```

```
fi

if [ "$mode" = "stack" ]
then
    ulimit -v "$((3 * 1000 * 1000))" # kbytes
    ulimit -s "$(($size))" # kbytes
fi

if [ "$mode" = "virtualmem" ]
then
    ulimit -s "$((1 * 1000 * 1000))" # kbytes
    ulimit -v "$((size))" # kbytes
fi

res=$(/usr/bin/time -f "user %U" "$executable" "$n" 2>&1)
echo "$res"
```

Script 2 : tableau de tests

Ce script appelle le script 1 plusieurs fois, en faisant varier la taille de la pile (ou de l'environnement) et le nombre entier donné en argument de l'exécutable. Si le programme dure moins de 5 minutes, il va pour chaque case du tableau faire la moyenne du temps passé sur 10 essais. Si le programme dure plus de 5 minutes, il ne fait qu'un seul essai. Attention, le script est prévu pour détecter les chaînes de caractères d'erreurs données par l'exécutable ocaml. Il ne permet pas de gérer les différents messages d'erreurs donnés par d'autres langages.

Notons que pour obtenir une meilleure précision, il aurait fallu lancer plus de dix fois les petits exemples. Cela justifie un temps nul dans la première colonne des résultats. Ce n'est pas gênant car nous nous intéressons aux grandes valeurs.

Nom : `lancer_tableaux_tests.bash`

```
#!/bin/bash
set -o nounset

# params :
# $1 ymode : "stack" or "virtualmem"
# $2 yvalues : a file containing pairs
#              of lines : number (in kbytes) and label
# $3 executable to test
# $4 nvalues : a file containing pairs
#              of lines : number and label

if [[ $# -ne 4 ]]
then
    echo "mauvais nombre d'arguments pour $0"
    exit 1
else
    ymode="$1"
    mapfile -t yvalues < "$2"
    ylength="${#yvalues[@]}"
    executable="$3"
    mapfile -t nvalues < "$4"
    nlength="${#nvalues[@]}"
fi

if [ ! -f "$executable" ]
then
    echo "$executable is not file"
    exit 1
fi

awkOOM='
```

```
BEGIN { oom = 0 }
/out of memory/ { oom = 1 }
END {  print oom }'

awkSO='
BEGIN { so = 0 }
/Stack_overflow/ { so = 1 }
END { print so }'

awkTimeout='
BEGIN { to = 1 }
/^user/ { to = 0 }
END { print to }'

awkTimeUser='/^user/ { sub(/^0m/, "", $2) ; print $2 }'

awkSumFloat='{ print $1 + $2 }'
awkDivFloat='{ print ($1 / $2) }'

awkFormatTime='
{
    if ($1 >= 60) {
        printf "%dm%ds \n", int($1 / 60), ($1 % 60) ;
    }
    else { printf "%.2fs\n", $1 ; }
}'
```

Suite :

```
# header
echo "|---"
if [ "$ymode" = "stack" ]
then echo -n "| taille de pile | "
else echo -n "| taille de l'environnement | "
fi
for (( x = 1 ; x <= $(($nlength - 1)) ; x += 2 ))
do
    echo -en "n = ${nvalues[$x]} | "
done
echo ""
echo "|---"

# body
for (( y = 0 ; y <= $(($ylength - 2)) ; y += 2 ))
do
    currentY="$((${yvalues[$y]}))"
    echo -en "| ${yvalues[$(($y + 1))]} | "

    for (( x = 0 ; x <= $(($nlength - 2)) ; x += 2 ))
    do
        currentN="$((${nvalues[$x]}))"

        nbExec=10
        to=$(
            /usr/bin/timeout --foreground 5m \
            /bin/bash ./lancer_test.bash "$ymode" "$currentY" "$executable" "$currentN")
        if [ "$to" = "1" ] ; then nbExec=1 ; fi
        sumTU=0
        atleastoneSO="0"
        atleastoneOOM="0"
        for ((k = 0 ; k < $nbExec ; k ++))
        do
            res=$(/bin/bash ./lancer_test.bash "$ymode" "$currentY" "$executable"
            ↪  "$currentN")
```

```
            tu=$(echo "$res" | awk "$awkTimeUser")
            oom=$(echo "$res" | awk "$awkOOM")
            so=$(echo "$res" | awk "$awkSO")
            sumTU=$(echo "$sumTU $tu" | awk "$awkSumFloat")
            if [ "$so"  = "1" ] ; then atleastoneSO="1"  ; fi
            if [ "$oom" = "1" ] ; then atleastoneOOM="1" ; fi
        done

        avg=$(echo "$sumTU $nbExec" | awk "$awkDivFloat")
        avgF=$(echo "$avg" | awk "$awkFormatTime")
        echo -en "$avgF"
        if [ "$atleastoneSO"  = "1" ] ; then echo -en " (SO)"  ; fi
        if [ "$atleastoneOOM" = "1" ] ; then echo -en " (OOM)" ; fi

        echo -en " | "
    done

    echo ""
done

echo "|---"
```

Script 4 : lancer une série de tableaux

Ce script permet simplement de lancer en série le script 2.

Nom : `lancer_serie_tableaux_tests.bash`

```
#!/bin/bash
set -o nounset

# params :
# $1 tableaux : a file containing quintuplets of lines of :
#               - "---" (a visual separator line)
#               - ymode ("stack" or "virtualmem")
#               - yvalues file
#               - executable to test
#               - nvalues file

if [[ $# -ne 1 ]]
then
    echo "mauvais nombre d'arguments pour $0"
    exit 1
else
    mapfile -t tableaux < "$1"
    tableauxlength="${#tableaux[@]}"
fi

for (( i = 0 ; i <= $(($tableauxlength - 5)) ; i += 5 ))
do
    ymode="${tableaux[$(($i + 1))]}"
    yvalues="${tableaux[$(($i + 2))]}"
    executable="${tableaux[$(($i + 3))]}"
    nvalues="${tableaux[$(($i + 4))]}"

    echo "======================================================================="
    echo "$executable"
    /bin/bash lancer_tableau_tests.bash "$ymode" "$yvalues" "$executable" "$nvalues"
    echo "======================================================================="
done
```

Fichiers donnés aux scripts

Nom : `serie1.txt`

```
---
stack
stackvalues1.txt
codes/taille_list_lineaire.exe
nvalues1.txt
---
stack
stackvalues1.txt
codes/taille_list_constant.exe
nvalues1.txt
---
virtualmem
virtualmemvalues1.txt
codes/m_for_l_state_virtualmem.exe
nvalues1.txt
---
stack
stackvalues1.txt
codes/m_for_l_state_stack.exe
nvalues1.txt
---
virtualmem
virtualmemvalues1.txt
codes/m_for_r_state_virtualmem.exe
nvalues1.txt
---
stack
stackvalues1.txt
codes/m_for_r_state_stack.exe
nvalues1.txt
---
virtualmem
virtualmemvalues1.txt
codes/m_for_r_writer.exe
nvalues1.txt
---
stack
stackvalues1.txt
codes/m_for_r_writer.exe
nvalues1.txt
---
virtualmem
virtualmemvalues1.txt
codes/m_for_l_statewriter_virtualmem.exe
nvalues1.txt
---
stack
stackvalues1.txt
codes/m_for_l_statewriter_stack.exe
nvalues1.txt
---
virtualmem
virtualmemvalues1.txt
codes/m_for_r_statewriter_virtualmem.exe
nvalues1.txt
---
stack
stackvalues1.txt
codes/m_for_r_statewriter_stack.exe
nvalues1.txt
```

Nom : `virtualmemvalues1.txt`

```
10 ** 7
10G
5 * (10 ** 6)
5G
10 ** 6
1G
10 ** 5
100M
```

Nom : `stackvalues1.txt`

```
10 ** 7
10G
5 * (10 ** 6)
5G
10 ** 6
1G
10 ** 5
100M
10 ** 4
10M
```

Nom : `nvalues1.txt`

```
10 ** 5
100K
10 ** 6
1M
10 ** 7
10M
10 ** 8
100M
5 * (10 ** 8)
500M
10 ** 9
1G
```

C.4 Taille de liste sans récursion terminale

Pile

```
let make_liste_zero n =
  let rec loop l i =
    if i < n
    then loop (0 :: l) (i + 1)
    else l
  in
  loop [] 0


let rec taille_liste l =
  if l = []
  then 0
  else 1 + (taille_liste (List.tl l))


let n = int_of_string Sys.argv.(1)
let lc = make_liste_zero n
let tlc = taille_liste lc
```

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.03s	0.48s	4.31s (SO)	5.03s (OOM)	5.00s (OOM)
5G	0.00s	0.03s	0.47s	4.31s (SO)	5.00s (OOM)	5.00s (OOM)
1G	0.00s	0.03s	0.48s	4.33s (SO)	5.00s (OOM)	5.04s (OOM)
100M	0.00s	0.04s	0.41s (SO)	4.41s (SO)	5.23s (OOM)	5.25s (OOM)
10M	0.00s	0.03s (SO)	0.40s (SO)	4.32s (SO)	4.97s (OOM)	5.02s (OOM)

C.5 Taille de liste avec récursion terminale

Pile

```
let make_liste_zero n =
  let rec loop l i =
    if i < n
    then loop (0 :: l) (i + 1)
    else l
  in
  loop [] 0

let rec taille_liste l t =
  if l = []
  then t
  else (taille_liste (List.tl l) (1 + t))

let n = int_of_string Sys.argv.(1)
let lc = make_liste_zero n
let tlc = taille_liste lc 0
```

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.03s	0.42s	4.41s	4.99s (OOM)	5.03s (OOM)
5G	0.00s	0.03s	0.41s	4.41s	5.00s (OOM)	5.00s (OOM)
1G	0.00s	0.03s	0.41s	4.43s	5.01s (OOM)	5.24s (OOM)
100M	0.00s	0.03s	0.43s	4.61s	5.24s (OOM)	5.21s (OOM)
10M	0.00s	0.03s	0.43s	4.61s	5.21s (OOM)	5.20s (OOM)

C.6 Monade *State* avec récursion gauche

Mémoire virtuelle

```
let return a s = (a, s)

let bind f ma =
  fun aux0 ->
  let (a, aux1) = (ma aux0) in
  (f a aux1)

let m_for_l n f a =
  let rec loop i ma =
    if i < n
    then loop (i + 1) (bind f ma)
    else ma
  in
  loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = fun i -> (a, i + 1)
let bigFunction = m_for_l n f 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.05s	0.63s	6.95s	16.55s (OOM)	16.46s (OOM)
5G	0.00s	0.05s	0.61s	6.79s	8.05s (OOM)	8.13s (OOM)
1G	0.00s	0.05s	0.62s	1.51s (OOM)	1.50s (OOM)	1.49s (OOM)
100M	0.00s	0.05s	0.12s (OOM)	0.12s (OOM)	0.12s (OOM)	0.12s (OOM)

Notons que nous n'avons pas lancé le calcul avec un état initial, afin d'étudier uniquement l'espace occupé par les fonctions. Si nous lançons le calcul, nous obtenons un tableau qui contient aussi des erreurs `SO`, comme dans le test suivant.

Pile

```
let return a s = (a, s)

let bind f ma =
  fun aux0 ->
  let (a, aux1) = (ma aux0) in
  (f a aux1)

let m_for_l n f a =
  let rec loop i ma =
    if i < n
    then loop (i + 1) (bind f ma)
    else ma
  in
  loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = fun i -> (a, i + 1)
let bigFunction = m_for_l n f 'a'
let res = (bigFunction 0)
```

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.10s	4.22s	4.70s (OOM)	4.68s (OOM)	4.69s (OOM)
5G	0.00s	0.09s	4.21s	4.72s (OOM)	4.69s (OOM)	4.71s (OOM)
1G	0.00s	0.09s	4.22s	4.79s (OOM)	4.83s (OOM)	4.87s (OOM)
100M	0.00s	0.10s	0.65s (SO)	4.88s (OOM)	4.87s (OOM)	4.82s (OOM)
10M	0.00s	0.05s (SO)	0.64s (SO)	4.83s (OOM)	4.84s (OOM)	4.84s (OOM)

Pour rappel, ici les erreurs `OOM` arrivent plus tôt car quand nous testons la pile, nous fixons la taille d'environnement à `3G`.

C.7 Monade *State* avec récursion droite

Mémoire virtuelle

```
let return a s = (a, s)

let bind f ma =
  fun aux0 ->
  let (a, aux1) = (ma aux0) in
  (f a aux1)

let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a = fun i -> (a, i + 1)
let bigFunction = m_for_r n f 'a'
let res = bigFunction 0
```

Notons que contrairement au test avec récusion gauche, ici nous exécutons le programme, sans quoi le temps est toujours nul puisque l'exécution attend l'état initial pour effectuer la récursion.

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.08s	0.87s	4.33s	8.68s
5G	0.00s	0.00s	0.08s	0.87s	4.30s	8.61s
1G	0.00s	0.00s	0.08s	0.85s	4.29s	8.59s
100M	0.00s	0.00s	0.08s	0.85s	4.28s	8.60s

Pile

```
let return a s = (a, s)

let bind f ma =
  fun aux0 ->
  let (a, aux1) = (ma aux0) in
  (f a aux1)

let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a = fun i -> (a, i + 1)
let bigFunction = m_for_r n f 'a'
let res = bigFunction 0
```

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.08s	0.89s	4.46s	8.89s
5G	0.00s	0.00s	0.08s	0.88s	4.44s	8.87s
1G	0.00s	0.00s	0.08s	0.88s	4.45s	8.90s
100M	0.00s	0.00s	0.08s	0.88s	4.46s	8.88s
10M	0.00s	0.00s	0.08s	0.88s	4.46s	8.94s

C.8 Monade *State* avec `m_while`

Mémoire virtuelle

```
let m_while test f a =
  let rec loop (a, st) =
    if (test a)
    then loop (f a st)
    else (a, st)
  in
  (fun st -> loop (a, st))

let n = int_of_string Sys.argv.(1)
let test i = n < n
let f n aux = (n + 1, aux)
let bigf = m_while test f 0
let res = bigf 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.04s	0.46s	2.30s	4.61s
5G	0.00s	0.00s	0.04s	0.47s	2.32s	4.57s
1G	0.00s	0.00s	0.04s	0.46s	2.28s	4.65s
100M	0.00s	0.00s	0.04s	0.45s	2.31s	4.59s

Pile

Le code est identique au précédent.

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.04s	0.46s	2.29s	4.60s
5G	0.00s	0.00s	0.04s	0.46s	2.29s	4.65s
1G	0.00s	0.00s	0.04s	0.46s	2.31s	4.60s
100M	0.00s	0.00s	0.04s	0.46s	2.32s	4.74s
10M	0.00s	0.00s	0.04s	0.46s	2.32s	4.66s

C.9 Monade *State* avec concret monadique

Mémoire virtuelle

```
let apply f (a, st) = (f a st)
let with_mconc fconc a st = fconc (a, st)

let std_for n f a =
  let rec loop i a =
    if i < n
    then loop (i + 1) (f a)
    else a
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a aux = (a, aux + 1)
let res = with_mconc (std_for n (apply f)) 'a' 0
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.03s	0.34s	1.75s	3.50s
5G	0.00s	0.00s	0.03s	0.34s	1.76s	3.50s
1G	0.00s	0.00s	0.03s	0.34s	1.75s	3.50s
100M	0.00s	0.00s	0.03s	0.34s	1.75s	3.50s

Pile

Le code est identique au précédent.

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.03s	0.35s	1.76s	3.54s
5G	0.00s	0.00s	0.03s	0.35s	1.77s	3.55s
1G	0.00s	0.00s	0.03s	0.33s	1.74s	3.52s
100M	0.00s	0.00s	0.03s	0.35s	1.76s	3.54s
10M	0.00s	0.00s	0.03s	0.35s	1.77s	3.55s

C.10 Monade *Writer* avec récursion gauche

Mémoire virtuelle

```
let return a = (a, 0)

let bind f ma =
  let (a, auxA) = ma in
  let (b, auxB) = (f a) in
  (b, auxA + auxB)

let m_for_l n f a =
    let rec loop i ma =
        if i < n
        then loop (i + 1) (bind f ma)
        else ma
    in
    loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = (a, 1)
let res = m_for_l n f 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.04s	0.40s	2.05s	4.11s
5G	0.00s	0.00s	0.04s	0.40s	2.04s	4.09s
1G	0.00s	0.00s	0.04s	0.41s	2.05s	4.10s
100M	0.00s	0.00s	0.04s	0.40s	2.04s	4.12s

Pile

Le code est identique au précédent.

Le tableau est identique au précédent.

C.11 Monade *Writer* avec récursion droite

Mémoire virtuelle

```
let return a = (a, 0)

let bind f ma =
  let (a, auxA) = ma in
  let (b, auxB) = (f a) in
  (b, auxA + auxB)

let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a = (a, 1)
let res = m_for_r n f 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.19s	13.46s	6m22s (SO)	6m20s (SO)	6m22s (SO)
5G	0.00s	0.19s	13.21s	6m20s (SO)	6m18s (SO)	6m17s (SO)
1G	0.00s	0.19s	13.25s	50.38s (SO)	51.39s (SO)	50.09s (SO)
100M	0.00s	0.19s	0.50s (OOM)	0.49s (OOM)	0.49s (OOM)	0.49s (OOM)

Pile

Le code est identique au précédent.

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.19s	13.22s	6m55s (OOM)	6m54s (OOM)	6m55s (OOM)
5G	0.00s	0.19s	13.29s	6m59s (OOM)	7m1s (OOM)	6m57s (OOM)
1G	0.00s	0.19s	13.34s	6m22s (SO)	6m19s (SO)	6m21s (SO)
100M	0.00s	0.19s	4.16s (SO)	4.15s (SO)	4.16s (SO)	4.15s (SO)
10M	0.00s	0.07s (SO)	0.07s (SO)	0.07s (SO)	0.07s (SO)	0.07s (SO)

C.12 Monade *Writer* avec `m_while`

Mémoire virtuelle

```
let m_while test f a =
  let rec loop (a, logs) =
    if (test a)
    then let (a2, logs2) = (f a) in
         loop (a2, logs + logs2)
    else (a, logs)
  in
  loop (a, 0)

let n = int_of_string Sys.argv.(1)
let test i = i < n
let f n = (n + 1, 1)
let res = m_while test f 0
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.03s	0.38s	1.89s	3.80s
5G	0.00s	0.00s	0.03s	0.38s	1.90s	3.79s
1G	0.00s	0.00s	0.03s	0.38s	1.90s	3.81s
100M	0.00s	0.00s	0.03s	0.38s	1.90s	3.79s

Pile

Le code est identique au précédent.

Le tableau est identique au précédent.

C.13 Monade *Writer* avec concret monadique

Mémoire virtuelle

```
let return a = (a, 0)

let bind f (a, logsA) =
  let (b, logsB) = (f a) in
  (b, logsA + logsB)

let apply = bind
let with_mconc f_mconc a = f_mconc (return a)

let std_for n f a =
  let rec loop i a =
    if i < n
    then loop (i + 1) (f a)
    else a
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a = (a, 1)
let res = with_mconc (std_for n (apply f)) 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.04s	0.46s	2.30s	4.61s
5G	0.00s	0.00s	0.04s	0.46s	2.30s	4.60s
1G	0.00s	0.00s	0.04s	0.46s	2.30s	4.63s
100M	0.00s	0.00s	0.04s	0.46s	2.30s	4.64s

Pile

Le code est identique au précédent.

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.04s	0.46s	2.31s	4.62s
5G	0.00s	0.00s	0.04s	0.46s	2.32s	4.64s
1G	0.00s	0.00s	0.04s	0.46s	2.36s	4.67s
100M	0.00s	0.00s	0.04s	0.47s	2.33s	4.65s
10M	0.00s	0.00s	0.04s	0.46s	2.37s	4.70s

C.14 Monade *StateWriter* avec récursion gauche

Mémoire virtuelle

```
let return a s = ((a, s), 0)

let bind f ma =
  fun staux0 ->
  let ((a, staux1), wtaux1) = (ma staux0) in
  let ((b, staux2), wtaux2) = (f a staux1) in
  ((b, staux2), wtaux1 + wtaux2)

let m_for_l n f a =
  let rec loop i ma =
    if i < n
    then loop (i + 1) (bind f ma)
    else ma
  in
  loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = fun i -> ((a, i + 1), 1)
let bigFunction = m_for_l n f 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.05s	0.61s	6.71s	16.23s (OOM)	16.26s (OOM)
5G	0.00s	0.05s	0.61s	6.76s	7.99s (OOM)	8.02s (OOM)
1G	0.00s	0.05s	0.61s	1.49s (OOM)	1.49s (OOM)	1.48s (OOM)
100M	0.00s	0.05s	0.12s (OOM)	0.11s (OOM)	0.12s (OOM)	0.11s (OOM)

Pile

```
let return a s = ((a, s), 0)

let bind f ma =
  fun staux0 ->
  let ((a, staux1), wtaux1) = (ma staux0) in
  let ((b, staux2), wtaux2) = (f a staux1) in
  ((b, staux2), wtaux1 + wtaux2)

let m_for_l n f a =
  let rec loop i ma =
    if i < n
    then loop (i + 1) (bind f ma)
    else ma
  in
  loop 0 (return a)

let n = int_of_string Sys.argv.(1)
let f a = fun i -> ((a, i + 1), 1)
let bigFunction = m_for_l n f 'a'
let res = (bigFunction 0)
```

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.21s	14.74s	4.67s (OOM)	4.68s (OOM)	4.64s (OOM)
5G	0.00s	0.20s	14.87s	4.61s (OOM)	4.66s (OOM)	4.62s (OOM)
1G	0.00s	0.21s	14.75s	4.63s (OOM)	4.59s (OOM)	4.60s (OOM)
100M	0.00s	0.20s	0.61s (SO)	4.62s (OOM)	4.59s (OOM)	4.60s (OOM)
10M	0.00s	0.05s (SO)	0.60s (SO)	4.59s (OOM)	4.60s (OOM)	4.62s (OOM)

C.15 Monade *StateWriter* avec récursion droite

Mémoire virtuelle

```
let return a s = ((a, s), 0)

let bind f ma =
  fun staux0 ->
  let ((a, staux1), wtaux1) = (ma staux0) in
  let ((b, staux2), wtaux2) = (f a staux1) in
  ((b, staux2), wtaux1 + wtaux2)

let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a


let n = int_of_string Sys.argv.(1)
let f a = fun i -> ((a, i + 1), 1)
let bigFunction = m_for_r n f 'a'
let res = bigFunction 0
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.01s	0.44s	32.92s	4m8s (SO)	4m20s (SO)	4m6s (SO)
5G	0.01s	0.44s	31.80s	4m7s (SO)	4m5s (SO)	4m23s (SO)
1G	0.01s	0.44s	32.03s	36.57s (SO)	35.98s (SO)	37.87s (SO)
100M	0.01s	0.49s	0.43s (OOM)	0.43s (OOM)	0.43s (OOM)	0.43s (OOM)

Pile

```
let return a s = ((a, s), 0)

let bind f ma =
  fun staux0 ->
  let ((a, staux1), wtaux1) = (ma staux0) in
  let ((b, staux2), wtaux2) = (f a staux1) in
  ((b, staux2), wtaux1 + wtaux2)

let m_for_r n f a =
  let rec loop i a =
    if (i < n)
    then bind (loop (i + 1)) (f a)
    else (return a)
  in
  loop 0 a


let n = int_of_string Sys.argv.(1)
let f a = fun i -> ((a, i + 1), 1)
let bigFunction = m_for_r n f 'a'
let res = bigFunction 0
```

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.01s	0.51s	35.99s	4m42s (OOM)	4m55s (OOM)	4m46s (OOM)
5G	0.01s	0.45s	33.80s	4m44s (OOM)	4m39s (OOM)	4m39s (OOM)
1G	0.01s	0.49s	34.61s	4m15s (SO)	4m30s (SO)	4m19s (SO)
100M	0.01s	0.45s	2.93s (SO)	2.92s (SO)	2.93s (SO)	2.91s (SO)
10M	0.01s	0.06s (SO)	0.06s (SO)	0.06s (SO)	0.06s (SO)	0.06s (SO)

C.16 Monade *StateWriter* avec `m_while`

Mémoire virtuelle

```
let m_while_writer test f a =
  let rec loop (a, logs) =
    if (test a)
    then let (a2, logs2) = (f a) in
         loop (a2, logs + logs2)
    else (a, logs)
  in
  loop (a, 0)

let m_while_statewriter test f a aux =
  m_while_writer
     (fun (a, aux) -> test a)
     (fun (a, aux) -> f a aux)
     (a, aux)

let n = int_of_string Sys.argv.(1)
let test i = i < n
let f n aux = ((n + 1, aux), 1)
let bigf = m_while_statewriter test f 0
let res = bigf 'a'
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.06s	0.62s	3.12s	6.26s
5G	0.00s	0.00s	0.06s	0.62s	3.12s	6.28s
1G	0.00s	0.00s	0.06s	0.60s	3.12s	6.28s
100M	0.00s	0.00s	0.06s	0.63s	3.14s	6.23s

Pile

Le code est identique au précédent.

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.06s	0.62s	3.12s	6.26s
5G	0.00s	0.00s	0.06s	0.63s	3.11s	6.24s
1G	0.00s	0.00s	0.06s	0.62s	3.11s	6.23s
100M	0.00s	0.00s	0.06s	0.61s	3.07s	6.23s
10M	0.00s	0.00s	0.06s	0.62s	3.11s	6.23s

C.17 Monade *StateWriter* avec concret monadique

Mémoire virtuelle

```
let return_writer a = (a, 0)

let bind_writer f (a, logsA) =
  let (b, logsB) = (f a) in
  (b, logsA + logsB)

let apply_writer = bind_writer
let with_mconc_writer f_mconc a = f_mconc (return_writer a)

let apply_state_t f ta = apply_writer (fun (a, st) -> (f a st)) ta
let with_mconc_state_t f_mconc a st = with_mconc_writer f_mconc (a, st)

let std_for n f a =
  let rec loop i a =
    if i < n
    then loop (i + 1) (f a)
    else a
  in
  loop 0 a

let n = int_of_string Sys.argv.(1)
let f a aux = ((a, aux + 1), 1)
let res = with_mconc_state_t (std_for n (apply_state_t f)) 'a' 0
```

taille de l'environnement	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.06s	0.70s	3.50s	6.91s
5G	0.00s	0.00s	0.06s	0.69s	3.45s	6.91s
1G	0.00s	0.00s	0.06s	0.69s	3.45s	7.01s
100M	0.00s	0.00s	0.06s	0.68s	3.43s	6.90s

Pile

Le code est identique au précédent.

taille de pile	n = 100K	n = 1M	n = 10M	n = 100M	n = 500M	n = 1G
10G	0.00s	0.00s	0.06s	0.69s	3.50s	6.91s
5G	0.00s	0.00s	0.06s	0.69s	3.46s	6.95s
1G	0.00s	0.00s	0.06s	0.70s	3.49s	6.96s
100M	0.00s	0.00s	0.06s	0.68s	3.47s	7.07s
10M	0.00s	0.00s	0.06s	0.68s	3.46s	7.08s

Annexe D

Chapitre optimisation de monades

D.1 Récursion de valeur monadique

Il existe un autre type de "récursion monadique" que celui que nous utilisons dans ce document : la récursion "de valeur monadique", qui est décrite en détail en référence [Erk02] par Erkok Levent. Cela consiste à effectuer une récursion sur la valeur principale, sans dupliquer les effets. Voici un exemple pour *State*, en *Haskell* car c'est plus pertinent d'utiliser l'appel par besoin :

```haskell
type MyMonad a = Aux -> (a, Aux)

monadic_fix :: (a -> MyMonad a) -> (MyMonad a)

monadic_fix f aux0 = (a, aux1)
    where (a, aux1) = (f a aux0)
```

Code D.1.1 – *Définition de récursion de valeur monadique pour* State *en* Haskell *(appel par besoin)*

La valeur de `(a, aux1)` est récursive, elle dépend du résultat de la fonction `f`, auquel nous avons donné la valeur `a` (récursive). Ce deuxième `a` est donc lui-même le fruit d'une application de la fonction `f` à un autre `a`. Il est entendu que cette expression ne termine pas en appel par valeur. En revanche, en appel par besoin, le paramètre peut être utilisé sans être complètement évalué. Par exemple, voici une utilisation de `monadic_fix` qui construit une liste inifinie avec la monade *State* :

```haskell
from :: Int -> ([Int] * Int) -- MyMonad [Int] avec Aux = Int

from = monadic_fix (\suite n -> (n : suite, n + 1))

(suite5, aux) = (from 5)
```

Code D.1.2 – *Exemple d'utilisation de* `monadic_fix`

`(from 5)` ne calcule rien tant que rien ne lui est demandé. Son type est `([Int], Int)`. `suite5` contient une liste infinie de `5` (non construite), et `aux` contient `5 + 1`. Cela illustre le comportement de `monadic_fix` qui ne répète pas l'effet monadique : il réutilise toujours la valeur auxiliaire initiale. L'infinité de `suite5` ne rend pas nécessairement l'évaluation du programme infinie, seulement dans le cas où nous cherchons à la parcourir complètement.

Tables des figures, tableaux, codes et ressources

Table des figures

Liste des tableaux

Table des codes

Liste des ressources

— Simulation fonctionnelle : dépôt github `https://github.com/antoinekagit/sim-func`, visité le Vendredi 27 Septembre 2019
— Bibliothèque de monades : dépôt github `https://github.com/antoinekagit/concrete-monads`, visité le Vendredi 27 Septembre 2019
— Plateforme de simulation : dépôt github `https://github.com/antoinekagit/plateforme-sim`, visité le Mardi 29 Octobre 2019
— Projet *Jamel* de Pascal Seppecher : dépôt github `https://github.com/pseppecher/jamel`, visité le Mercredi 30 Octobre 2019