Résumé

L'accroissement incessant des **performances** des micro-processeurs donne à ces circuits électroniques une complexité telle qu'il est impossible d'en garantir la **sécurité** de fonctionnement. Le modèle des processeurs n'a pas changé depuis sa création : il est bâti sur une architecture de VON-NEUMANN, typiquement **séquentielle**.

Le **parallélisme** semble être une solution pour augmenter, à priori facilement, les performances des architectures matérielles. Il existe principalement deux sortes de parallélisme : le parallélisme à grain fin, ou parallélisme d'instructions, et le parallélisme à gros grain, ou parallélisme de tâches. Ce dernier concerne plutôt la répartition de processus sur un réseau d'ordinateurs et il ne joue aucun rôle dans l'augmentation des performances des processeurs.

Le parallélisme à grain fin, dont on espère beaucoup pour augmenter les performances des micro-processeurs semble en partie **inefficace**. D'une part, les architectures massivement parallèles sont difficiles à programmer efficacement d'une manière automatique et d'autre part, les temps de communication des architectures basées sur des réseaux d'inter-connections sont prohibitifs par rapport aux temps d'exécution des instructions utiles.

Nous proposons dans notre **thèse** un modèle d'architecture parallèle appelée **architecture parallèle statique**. Il s'agit d'une architecture construite à l'aide de processeurs communs, de type contrôleur ou DSP, partageant une mémoire globale. Son originalité est d'annuler les temps de communication entre les processeurs et la mémoire, puisque les processeurs sont directement connectés au bus, via un circuit d'interface à logique trois états.

Cette architecture n'est utilisable que si les conflits d'accès à la mémoire globale sont résolus au moment de la **compilation** des applications. Son exploitation dépend donc fortement des outils de développement associés.

Les outils développés durant la thèse, baptisés λ-outils, permettent une **programmation graphique** et hiérarchique des applications, une programmation à l'aide d'un **langage de haut niveau**, leur exécution sur le simulateur de l'architecture parallèle statique.

D'une manière informelle, toute application modélisable à l'aide d'un graphe ou d'un schéma de boîtes peut être programmée avec cette chaîne d'outils. Ainsi toute application reposant sur un **modèle à état** est-elle programmable, ce qui couvre en particulier les filtres numériques et les circuits électroniques numériques.

Abstract

Performance improvement of the micro-processors gives to these chips such a complexity that it is impossible to warrant the **quality** of their functionning. Since their creation, processors have been based on the **von Neumann** architecture, mainly **sequential** and **imperative.**

Parallelism seems to be a solution to ealily increase the hardware architecture performances. Mainly, there are exists two kinds of parallelism: the instruction level parallelism and the task/process parallelism, which does not play any role in the processor improvement.

Instruction-level parallelism, which is expected to significantly improve the processor performances, seems to be partially **inefficient**. Mainly, it is because Wide parallel architectures are difficult to program efficiently, and communication times in an inter-connection network are prohibitive, comparared to the useful instructions.

We propose in our thesis a parallel architecture model named the **static parallel architecture**. It is built with common processors, such as controllers or DSP, which share a global memory. Its originality is to avoid communication times, because the processors are directly plugged onto the main bus via a three states logic interface.

This architecture could be only used with the assertion that all the memory access **conflicts** are solved at **compile**-time. So, its exploitation is closely linked to the associated development tools.

The tools developed during the thesis, named λ-tools, allow a graphical and hierarchical programming of the applications, a high level language programming, and the execution on the parallel architecture or its simulator.

In an informal way, every application that can be modeled with a graph or a diagram could be programmed with this tool chain. Thus, all state based applications could be implemented, which covers digital filters or digital electronic diagrams.

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS
École Doctorale des Sciences Pour l'Ingénieur

MODÉLISATION FONCTIONNELLE ET GRAPHIQUE DES APPLICATIONS À ÉTAT EN VUE DE LEUR RÉALISATION SUR ARCHITECTURES PARALLÈLES STATIQUES.

Thèse

présentée pour obtenir le titre de

Docteur en Sciences

mention
Sciences de l'Ingénieur

par
Guilhem de Wailly

Soutenue à Nice, le 13 mars 1997 devant le jury composé de :

Messieurs	Charles André	Président
	Paul Caspi	Rapporteur
	Daniel Dours	Rapporteur
	Jean-Louis Cavaréro	Examinateur
	Jean-François Duboc	Examinateur
	Fernand Boéri	Directeur de thèse

Je dédie cette thèse à Nathalie, ma fiancée, et à la mémoire de Wermond.

Je voudrais exprimer ma reconnaissance aux membres du jury pour avoir accepter de lire et critiquer de manière constructive mon mémoire de thèse.

Je remercie vivement :

- Mr Charles André, Professeur à l'Université de Nice-Sophia Antipolis, pour avoir accepté de présider mon jury.
- Mr Paul Capsi, Professeur à l'Institut Polytechnique de Grenoble, pour avoir rapporté sur mon manuscrit et pour ses remarques instructives.
- Mr Daniel Dours, Professeur à l'Université Paul Sabatier de Toulouse, qui a également accepté de rapporter sur mon manuscrit et pour ses remarques précieuses.
- M. Jean-Louis Cavaréro, Professeur à l'Institut Universitaire de Technologie de Nice Sophia Antipolis, pour l'intérêt qu'il a porté à mes travaux.
- M. Jean-François Duboc, Ingénieur VLSI à Sophia Antipolis, à qui je témoigne ma plus sincère sympathie.
- M. Fernand Boéri, directeur de thèse, Professeur à l'Institut Universitaire de Technologie de Nice Sophia Antipolis, que je ne saurais jamais assez remercier pour la confiance inconditionnelle qu'il m'a toujours témoigné, pour sa disponibilité de tous les instants, malgré ses nombreuses responsabilités, ses conseils toujours avisés, qui ont su me guider et orienter ma recherche.

Je voudrais aussi remercier M. Jean Demartini, Professeur à l'Institut Universitaire de Universitaire de Technologie de Nice Sophia Antipolis, qui m'a permis de connaître le laboratoire I3S et m'a initié à différents aspects de l'informatique.

Je remercie aussi Mme Claude Jourdan, Professeur d'Anglais à l'Institut Universitaire de Technologie de Nice Sophia Antipolis, qui a montré une disponibilité sans limites pour m'aider dans les rédactions en langue anglaise.

J'associe également à ces remerciements l'ensemble des personnes du site Fabron du Laboratoire Informatique Signaux et Systèmes, pour leur soutien amical tout au long de cette thèse.

De plus, je tiens à remercier tous les informaticiens qui donnent leurs logiciels, particulièrement le concepteur de LINUX et de STK, les programmeurs du GNU de TEX et LATEX. Sans eux, il n'aurait pas été possible de disposer de cet ensemble de logiciels de qualité.

Table des matières

Chapitre 1

Introduction

1.1 L'ordinateur

L'ordinateur est une machine conçue pour effectuer des travaux automatiques. Plus il est puissant, plus les tâches qu'il peut effectuer sont complexes : des problèmes insolubles il y a vingt ans peuvent maintenant être traités. Un ordinateur est schématiquement constitué d'un cœur et d'organes. Les organes sont les périphériques d'entrée / sortie tels l'écran et le clavier. Son cœur est constitué de toutes les composantes qui participent aux calculs, comme la mémoire et le micro-processeur. Les performances du cœur des ordinateur a augmenté de manière vertigineuse, par rapport à celles des organes.

Jusqu'alors, le cœur de l'ordinateur a évolué sous le signe du «plus». La capacité d'intégration des circuits électroniques a permis d'énormes progrès dans leur conception. En effet, entre la simple porte logique qui contient quelques dizaines de transistors et les micro-processeurs actuels, qui en contiennent plusieurs millions, la capacité de placer des transistors sur une plaque de silicium a été considérablement multipliée. Plus de transistors dans un micro-processeur se traduit en particulier par plus d'instructions en moins de temps, ce qui permet d'augmenter sa capacité de traitement.

Cette inflation du nombre de transistor augmente de manière considérable la compléxité des circuits et aussi les risque d'erreurs. La complexité des circuits électroniques est devenue telle qu'on ne peut affirmer si un circuit est sans erreurs. Il est courant de dire que la correction d'une erreur rend seulement la correction de l'erreur suivante plus difficile.

Cependant, la capacité d'intégration se heurte à la limite incontournable de la physique. Pour intégrer sur une même plaque de silicium plus de transistors, il est nécessaire d'en réduire la taille. Or cette taille ne peut être réduite au delà d'une certaine limite sans perdre les caractéristiques électriques du transistor. Or, dès que l'on augmente la taille des circuits, leur compléxité de fabrication entraine une augmentation de coûts inacceptable. De plus, en augmentant la taille des circuits, nous augmentons les distances entre les composants, ce qui augmente les durées de transmission et diminue les performances du circuit.

Plutôt que de placer plus de transistors dans un circuit, l'évolution amène les concepteurs à optimiser leurs circuits. Jusqu'alors, il était peu rentable d'investir dans des méthodes nouvelles de conception des circuits, car le gain de performances obtenu alors était systématiquement balayé par les performances de la génération suivante de processeur.

Maintenant que la physique impose ces limites, l'évolution du cœur de l'ordinateur semble se placer sous le signe du «mieux» : la question est de savoir comment tirer un meilleur parti des techniques existantes de conception des circuits.

1.2 Parallélisme

A priori, il est évident que deux unités de calculs calculent plus vite qu'une seule : c'est le parallélisme des calculs. Les concepteurs de machines ont depuis longtemps conçu des

machines parallèles plus ou moins complexes. Mais la simplicité de ce raisonnement cache une réalité toute autre.

Les machines parallèles actuelles sont inefficaces, comparativement à leur coût. De plus, il n'existe pas d'outils logiciels pour les programmer efficacement et simplement. La difficulté de programmation est due essentiellement à la gestion des ressources et des conflits d'accès aux ressources.

Cette liaison entre le matériel et le logiciel est déjà apparue avec les architectures RISC où la complexité est passée du processeur à son compilateur: le matériel impose ses conditions, au logiciel qui l'exploite d'en tenir compte. Cette idée est viable car le logiciel évolue beaucoup plus rapidement que le matériel, et à moindre coût. La cause de l'inefficacité des architectures parallèles serait donc due à une mauvaise coordination entre l'architecture et le logiciel d'exploitation.

Notre étude se place à ce niveau: concevoir un environnement de programmation pour exploiter une architecture parallèle. Notre choix s'oriente vers une machine parallèle à faible coût, alliant simplicité de conception et efficacité.

1.3 Matériel : l'architecture parallèle statique

Une architecture parallèle à faible coût est nécessairement simple et utilise des composants communs. Ces composants pourraient être des contrôleurs actuels, qui sont de véritables petits ordinateurs avec une mémoire vive, une mémoire morte, un micro-processeur, des dispositifs d'entrée/sortie.

La manière la plus simple d'interconnecter les différents composants de l'architecture, la mémoire centrale et les contrôleurs, est de les rattacher directement à un bus passif, c'est à dire un ensemble de lignes de connexion. Le bus sera qualifié de statique, ainsi que l'architecture.

Des circuits utilisant une logique à trois états font office d'interfaces entre les contrôleurs et le bus commun, jouant le double rôle de portes d'accès au bus et de protection des contrôleurs.

Un modèle d'une telle architecture est présenté à la figure 1.1.

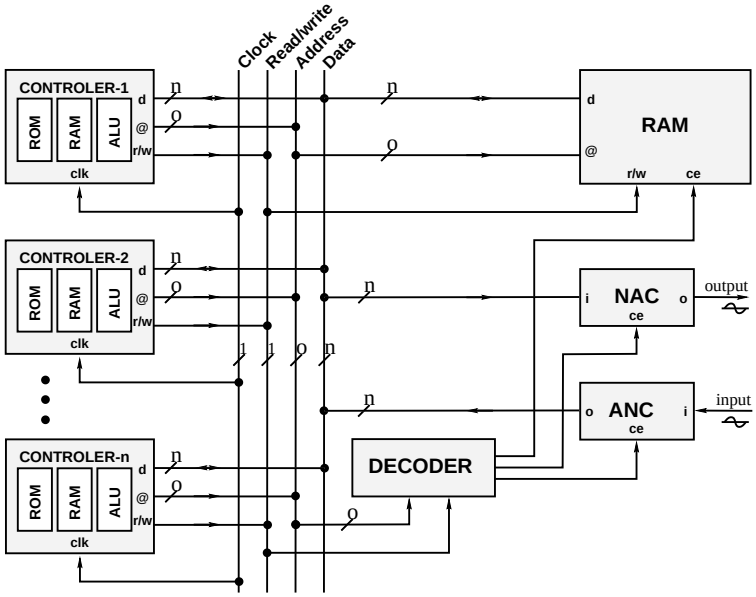


FIG. 1.1 - *Modèle de l'architecture parallèle statique.*

Les contrôleurs sont directement connectés au bus central, lui-même connecté à la mémoire commune. Deux lignes d'entrée/sortie sont décodées à une adresse déterminée du

plan mémoire. Ces lignes sont reliées à des convertisseurs numérique/analogique et analogique/numérique. Chaque contrôleur possède sa mémoire vive, sa mémoire morte contenant son programme et son unité de calculs. Les processeurs peuvent donc communiquer entre eux, via la mémoire centrale.

Dans cette architecture, deux contrôleurs ne peuvent donc accéder au bus central de manière simultanée[1]. La répartition des accès est à la charge du compilateur. C'est ici qu'intervient la coordination entre le logiciel et le matériel.

Si un langage de haut niveau et son compilateur permettent de programmer une telle architecture, cette architecture est très intéressante, et ce pour les raisons suivantes :

- son faible coût de production permet d'envisager une production à grande échelle ;
- l'augmentation des performances est immédiate ;
- l'architecture ne limite pas le nombre de processeurs ;
- une application peut être réalisée à l'aide de plusieurs modules communiquant, chacun basé sur cette architecture ;

Un inconvénient de cette architecture est le fait que l'application doit être recompilée si le nombre de processeur diminue. A priori, la recompilation n'est pas nécessaire si le nombre de processeurs augmente, les processeurs ajoutés restant inactifs.

De plus, cette architecture exploite un parallélisme à grain fin (au niveau des instructions élémentaires, voire au niveau du micro-code)[2]. Elle ne peut pas exploiter un parallélisme massif : dix contrôleurs semble être une limite raisonnable.

1.4 Logiciel : le modèle de programmation

L'intérêt de cette architecture parallèle statique dépend donc de l'éventuel langage de programmation. Une manière simple de programmer cette architecture est d'organiser les programmes de chaque contrôleur en une itération principale, dont le corps ne contiendrait que des instructions dont le temps d'exécution est fixe : nous dirons que le corps de l'itération est déterministe en temps. Chaque programme a la même durée et les accès à la mémoire centrale sont répartis entre les processeurs. Cette répartition statique peut éventuellement utiliser des instructions de remplissage, de type `nop` (*Not Operation*).

La structure du programme de chaque contrôleur est la suivante :

```
 contrôleur c1  |  contrôleur c2   | ... | contrôleur c_n
----------------+------------------+-----+-----------------
 init:...       | init: ...        |     | init: ...
 loop:...       | loop: ...        |     | loop: ...
 next:...       | next: ...        |     | next: ...
      goto loop |       goto loop  |     |       goto loop
                |                  |     |
```

où ... sont des séquences d'instructions. Les programmes communiquent avec la mémoire globale en lecture et écriture, soit avec des instruction spécialisée, soit par un décodage d'adresses. Ils procèdent à une initialisation, puis effectuent le calcul du corps de l'itération, et préparent la prochaine exécution du corps de l'itération.

Existe-t-il des applications qui peuvent être exécutées selon ce modèle, basé essentiellement sur une itération ? Considérons le schéma général d'une application présenté dans la figure 1.2.

1. En fait, plusieurs contrôleurs peuvent lire la même adresse simultanément.
2. Le grain du parallélisme de l'architecture ne limite pas le grain des tâches qui peuvent y être réalisées. Cela signifie seulement que les tâches sont composées d'instructions élémentaires.

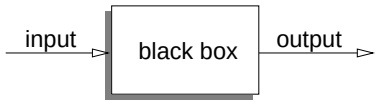


FIG. 1.2 - *Modèle général d'une application informatique.*

Il s'agit d'un filtre recursif[3] du second ordre.

Elle possède une entrée[4] et une sortie. Entre ces deux voies de communication, il y a une boîte noire, qui effectue un calcul. Nous admettrons que si nous fournissons une valeur en entrée, il est possible d'obtenir une valeur en sortie.

L'itération apparaît dès que l'on désire obtenir la liste des valeurs de sortie pour une liste de valeurs d'entrée. La contrainte apportée par le modèle d'architecture parallèle statique porte sur les déterminismes en temps et ressources, ce qui interdit immédiatement toute application utilisant la récursivité et l'allocation dynamique de la mémoire.

Prenons un exemple d'application. Considérons le graphe de la figure 1.3 qui représente un filtre récursif passe-bas du second ordre[5].

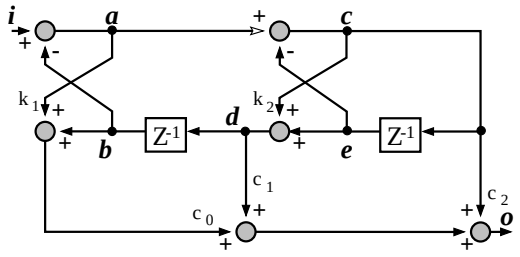


FIG. 1.3 - *Filtre récursif du second ordre.*

Ce filtre utilise une description bien connue dans le traitement du signal, basée sur un modèle à état. Il est possible de programmer ce filtre sous la forme d'un programme en pseudo-assembleur, sous la forme d'une itération :[6]

```
init:
        equ a 1         ; addresse of a
        equ b 2         ; addresse of b
        equ c 3         ; addresse of c
        equ d 4         ; addresse of d
        equ e 5         ; addresse of e

        lda #0          ; A = 0 (A and B are registers)
        mov b           ; b = 0 (a, b, c, d and e are global variables)
        mov e           ; e = 0
loop:
a:
        get b           ; A = b
        exc B           ; B = b
        get 0           ; A = input
        sub B           ; A = input - b
        mov a           ; a = input - b
```

3. Le concept de recursivité va etre utilisé très souvent au long de mémoire. Ici, le filtre est conçu de manière récursive, mais il engendre un processus itératif.

4. Pour simplifier, on considère une seule entrée, qui peut éventuellement être un vecteur de valeurs.

5. Il faut faire ici la distinction entre la récursivité du filtre (transformée en Z possédant un numérateur et un dénominateur tous deux différents de l'identité) qui traduit une équation récurrente de la récursivité opérationnelle qui traduit une équation de point fixe.

6. Pour simplifier le programme, nous considérons que les coefficients c et k valent 1.

```
c:
        get e           ; A = e
        exc B           ; B = e
        get a           ; A = a
        sub B           ; A = a - e
        mov c           ; c = a - e
d:
        get c           ; A = c
        exc B           ; B = c
        get e           ; A = e
        add B           ; A = e + c
        mov d           ; d = e + c
output:
        get a           ; A = a
        exc B           ; B = a
        get b           ; A = b
        add B           ; A = b + a
        exc B           ; B = b + a
        get c           ; A = c
        add B           ; A = c + b + a
        exc B           ; B = c + b + a
        get d           ; A = d
        add B           ; A = d + c + b + a
        mov 0           ; output = d + c + b + a
stop:
next:
b:
        get d           ; A = d
        mov b           ; b = d
e:
        get c           ; A = c
        mov e           ; e = c
        jmp loop
```

Cet assembleur est issu du simulateur de l'architecture parallèle décrit dans la troisième partie (§ III-5). Dans cet assembleur, les étiquettes sont notées `init:` ou a:, les registres du processeur sont `A` et `B`. La mémoire vive est accessible via des adresses symboliques comme `a` ou `b`[7]. Les entrées / sorties s'opèrent via l'adresse externe `0`.

Dès lors que l'application est mise sous la forme d'une itération déterministe en temps et en ressources, il est possible de répartir le programme sur l'architecture parallèle statique. Avec une architecture à trois contrôleurs, nous obtenons le code :

contrôleur 1

```
init:
        equ a 1
        equ b 2
        equ c 3
        equ d 4
        equ e 5
        lda #0
        mov b
        mov e
loop:
a:      get b
        exc B
        get 0
        sub B
        mov a
output: exc B
        get b
        add B
        exc B
        get c
        add B
        exc B
        get d
        add B
        mov 0
stop:
        jmp loop
```

contrôleur 2

```
init:
        equ a 1
        equ b 2
        equ c 3
        equ d 4
        equ e 5
        nop
        nop
        nop
loop:
c:      nop
        get e
        exc B
        nop
        nop
        get a
        sub B
        mov c
e:      nop
        nop
        nop
        mov e
        nop
        nop
        nop
stop:
        jump loop
```

contrôleur 3

```
init:
        equ a 1
        equ b 2
        equ c 3
        equ d 4
        equ e 5
        nop
        nop
        nop
loop:
d:      nop
        get e
        exc B
        nop
        nop
        nop
        nop
        nop
        nop
        get c
        add B
        mov d
b:      nop
        nop
        mov b
        nop
stop:
        jump loop
```

Dans ce code, les accès à la mémoire vive partagée sont notés `mov a` pour un accès en

7. Elles peuvent être remplacées en adresses physiques par un macro-processeur.

écriture, et `get a` pour un accès en lecture. Les entrées/sorties sont effectuées au travers de adresse `0`. Une même adresse dans la mémoire globale peut être accédée simultanément plusieurs fois en lecture. Un accès en écriture est bloquant. De plus, dans ce code, les accès à la mémoire globale sont optimisés de manière à en limiter le nombre.

Bien que l'application soit très petite et fortement séquentielle, le gain en performances dû à l'architecture parallèle statique est appréciable.

La critique de la méthode pour obtenir cette programmation est immédiate : elle n'est pas automatique. D'une part, la transcription de la représentation graphique de haut niveau vers le code organisé en itération n'est pas évidente lorsque la taille de l'application est importante, mais la répartition manuelle des micro-tâches sur les processeurs est fastidieuse. Les temps de programmation deviennent prohibitifs, et les risques d'erreurs importants.

1.5 Position du problème

L'objet de cette thèse est de proposer un environnement de programmation de haut niveau permettant de programmer dans de bonnes conditions l'architecture parallèle statique. Sa structure apparaît dans la figure 1.4.

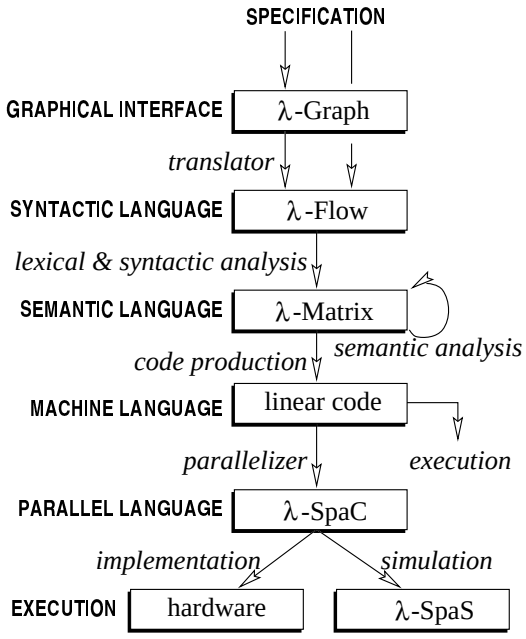


FIG. 1.4 - *Environnement de programmation pour programmer les applications à état sur l'architecture parallèle statique..*

Cet environnement de programmation permet la saisie graphique des applications, avec l'outil λ-GRAPH, et leur réalisation sur l'architecture parallèle statique. Comme la programmation exclusivement graphique est peu commode, l'utilisation conjointe du langage dataflow synchrone λ-FLOW est possible.

Pour permettre de prouver les programmes, le formalisme exclusivement fonctionnel des λ-matrice est proposé. Il permet notamment d'établir les déterminismes temps/ressources des programmes.

Les programmes sont compilés, puis ordonnancés pour fonctionner sur l'architecture parallèle statique, ou son simulateur graphique.

1.6 Organisation de la thèse

I- Étude de l'existant - La première partie de la thèse étudie les formalismes existants, reconnus pour permettre une exploitation du parallélisme. Nous effectuerons une analyse critique qui nous permettra de justifier nos choix pour conduire notre étude.

Le premier formalisme étudié concerne les graphes à flots de données, ou graphes dataflow (§ I-2). Nous montrerons que la représentation graphique est un point fort des graphes dataflow. Par contre, l'absence d'une sémantique précise nous conduira à ne pas utiliser ce formalisme directement.

Les langages fonctionnels sont une forme de graphes dataflow. Le λ-calcul est un outil mathématique qui formalise le concept de fonction (§ I-3). Il est la base des langages fonctionnels, réputés pour leurs propriétés sémantiques. L'outil théorique du λ-calcul ne sera pas utilisé dans notre formalisme car il est beaucoup trop expressif. Il permet d'écrire des programmes indéterministes en temps et en ressources, incompatibles avec le modèle de programmation que nous désirons. Cependant, beaucoup de concepts de notre formalisme reposent sur le λ-calcul, comme la définition d'une machine abstraite donnant la sémantique du langage.

Le langage SCHEME est un λ-calcul en mode applicatif non typé, ou plutôt à typage implicite (§ I-4). Ce langage sera utilisé tout au long de la thèse pour expliquer les définitions formelles et illustrer les exemples. C'est un langage fonctionnel qui possède une construction héritée des langages impératifs, l'affectation. Nous montrerons dans cette section que l'avantage des langages fonctionnels sur les langages impératifs n'est pas aussi évidente qu'il y paraît de prime abord. Il faut cependant garder en mémoire qu'un formalisme peut être étudié de manière formelle beaucoup plus aisément s'il est fonctionnel, car il se transforme en notations mathématiques immédiatement. SCHEME ne pourra pas nous servir de modèle de programmation car, comme le λ-calcul, il permet de construire des programmes indéterministes en temps et en ressources de manière trop évidente. Nous conserverons cependant de SCHEME le typage implicite des données qui libère le programmeur des incessantes déclarations de types de la plupart des langages typés, bien que notre formalisme soit fortement typé.

Nous étudierons ensuite de manière plus rapide les principaux langages fonctionnels dataflow auxquels la littérature prête des capacités à exploiter le parallélisme des applications (§ I-5). Nous montrerons pourquoi ils ne peuvent pas convenir directement, bien qu'ils soient très proches du langage développé au cours de cette thèse.

La dernière section de la première partie fera la synthèse des langages et formalismes présentés (§ I-6).

II- Étude théorique : les λ-matrices - La seconde partie est l'étude théorique du langage développé au cours de la thèse. Son introduction pose les bases du développement qui suivra (§ II-1). Elle montre une modélisation fonctionnelle et algébrique des applications à état. Le langage abstrait introduit les concepts clefs, comme la structure de la machine abstraite de résolution, inspirée du λ-calcul. Cette machine permet de résoudre les applications au moyen d'une itération principale, comme nous le souhaitons. La caractéristique fonctionnelle du formalisme permet d'effectuer des vérifications formelles des programmes.

Le langage introduit dans la première section est étendu à l'aide d'une grammaire : il s'agit du langage des λ-matrices, qui se prononce lambda matrice (§ II-2). Ses expressions sont tout d'abord expliquées de manière informelle. Dès cette partie, le langage SCHEME est utilisé pour définir les constructeurs de ce langage ; l'objectif est de développer en SCHEME la machine abstraite des λ-matrices, sous la forme d'un interprète, puis leur compilateur. Le lecteur pourra vérifier que l'ensemble des outils développés n'utilise jamais l'affectation, car le formalisme est strictement fonctionnel.

Après avoir décrit la grammaire des λ-matrices, le second chapitre décrit leur machine abstraite de résolution (§ II-3). Tout ce qui la compose est décrit formellement, puis réalisé en SCHEME. Les environnements sont étudiés en insistant sur le mode de liaison des variables des λ-matrices. Puis la machine proprement dite est étudiée, avec ses quatre combinateurs

de résolution. Chaque combinateur a un rôle particulier, reprenant le principe énoncé dans l'introduction de cette partie.

Puis les trois fonctions de critères qui permettent d'établir les déterminismes en temps et en ressources des applications sont étudiées (§ II-4). Le critère de la calculabilité garantit qu'une application ne contient pas d'équations de point fixe qui suppriment le déterminisme en temps. Le critère de la fermeture garantit que toutes les variables utilisées dans un programme sont définies. Enfin, le critère de stabilité montre que la consommation en ressources d'une application est constante. Les déterminismes en temps et en ressources sont directement déduits du critère de la stabilité.

Le compilateur formel des λ-matrices est alors introduit (§ II-5). Il permet de transformer une application modulaire en une forme aplatie. Pour pouvoir compiler une λ-matrice, un critère supplémentaire est introduit. Il effectue un contrôle strict des types. Le compilateur formel utilise un producteur de code optimisant qui évite la duplication du code.

La dernière section traite des vérifications formelles (§ II-6). Dans un premier temps, nous examinerons la raison pour laquelle la caractéristique fonctionnelle de la machine permet d'établir des propriétés des programmes. Nous montrerons par exemple, qu'un programme ne peut jamais atteindre un certain état si les valeurs d'entrée ont certaines propriétés. Puis nous montrerons comment extraire l'équation temporelle d'une λ-matrice, qui permet de connaître l'expression temporelle de tous les états, en fonction des entrées du programme. Dans un second temps et pour chaque fonction de critère, nous montrerons que l'expression du critère établit bien la propriété recherchée, puis que le critère est constant.

III- Réalisations pratiques : les λ-outils - La troisième partie de la thèse décrit les outils effectivement réalisés, qui s'appuient sur l'étude théorique de la seconde partie. Dans un premier temps, nous étudierons très succinctement STKLOS, l'interface objet de l'interprète SCHEME utilisé (§ III-2). Ce langage est utilisé pour spécifier les outils réalisés.

Le premier outil décrit est l'interface graphique λ-GRAPH (§ III-3). Elle permet une programmation graphique des applications. La conception d'une application est modulaire et hiérarchique. Les données manipulées et les opérateurs associés sont décrits dans des algèbres lues dynamiquement par l'interface. Les programmes λ-GRAPH sont traduits en λ-FLOW qui est décrit dans la section suivante. Dans ce chapitre, λ-GRAPH est spécifiée à l'aide d'une analyse orientée objet. Puis les méthodes essentielles sont décrites à l'aide de STKLOS.

Nous donnerons ensuite au langage abstrait des λ-matrices une forme concrète : c'est la définition du langage λ-FLOW (§ III-4). Nous décrirons son compilateur et la manière dont il peut être étendu de manière dynamique en le liant aux algèbres de l'utilisateur. Le compilateur est indépendant des cibles : nous en avons défini quatre, fort différentes, comme SCHEME, C, l'assembleur du `intel-386` et le format `SPA` pour le compilateur parallèle. L'utilisateur pourra, au moyen d'une vingtaine de lignes de définitions, définir autant de cibles qu'il le souhaite.

L'une des cibles de λ-FLOW, le format `SPA` permet de programmer l'architecture parallèle statique. Pour connaître précisément la structure interne de cette architecture, son simulateur λ-SPAS est modélisé à l'aide d'une analyse orientée objet (§ III-5). Dans ce chapitre, la partie non graphique du simulateur est spécifiée. Elle permet de réaliser un simulateur complet. Avec l'interface graphique, il est possible d'exécuter un programme en mode pas à pas ou en mode animation, de poser des points d'arrêt dans les contrôleurs, de modifier dynamiquement le contenu des mémoires locales, de la mémoire globale ou des registres.

Enfin, Le compilateur parallèle λ-SPAC permet de répartir une application compilée à l'aide de λ-FLOW sur les contrôleurs de l'architecture parallèle statique (§ III-6). Le compilateur est spécifié à l'aide de SCHEME. Nous analyserons aussi dans ce chapitre les performances de la chaîne d'outils, en le comparant à une architecture classique mono-processeur. Nous montrerons ensuite l'impact des optimisations apportées à la production de code parallèle sur les performances.

L'ensemble de ces outils va être utilisé pour traiter une application concrète, ce qui est décrit dans la quatrième partie.

IV- Application concrète : la norme de compression G.726 - La dernière partie de la thèse traite un exemple réel issu de l'industrie. Il s'agit de la norme `G.721` de compression pour les téléphones portables. Cette norme est traitée à partir des recommandations de l'organisme normalisateur, de la représentation graphique et textuelle jusqu'à la compilation sur l'architecture parallèle statique, ou plus exactement, son simulateur.

Il sera alors temps de conclure cette thèse.

Première partie

Étude de l'existant

Chapitre 1

Introduction

Cette étude propose d'étudier les différents outils actuels qui pourraient être utilisés pour programmer l'architecture parallèle statique décrite dans l'introduction.

La programmation de cette architecture repose sur le fait que l'application doit pouvoir être réalisée sous la forme d'une itération. Elle doit être en outre déterministe en temps et en ressources pour obtenir un ordonnancement statique des tâches. Dès lors, il est possible de répartir cette itération sur les différents processeurs.

Les graphes dataflow seront étudiés dans la première partie parce qu'ils sont connus dans la littérature pour permettre une exploitation du parallélisme (§ 2). Nous montrerons leur intérêt et leurs faiblesses.

Le λ-calcul est la fondement des langages fonctionnels qui peuvent être vus comme une variante des graphes dataflow (§ 3). Ce formalisme introduit des concepts importants, que nous utiliserons dans toute l'étude. Cependant, il est trop expressif pour être utilisé directement.

Le langage SCHEME est un langage fonctionnel qui est une sorte de λ-calcul en mode applicatif (§ 4). Ce langage permet une programmation très proche de la spécification des applications, à tel point qu'il est souvent utilisé directement comme langage de spécification.

Enfin, nous examinerons les différents langages dataflow actuels, comme LUCID, VAL, SISAL, LUSTRE et SIGNAL, en montrant pour quelles raisons ils ne peuvent être utilisés comme outils de programmation de l'architecture parallèle statique (§ 5).

La critique apportée à ces différents langages et formalismes justifiera la conception et l'étude théorique qui formera la seconde partie de cette thèse.

Chapitre 2

Les graphes à flots de données

2.1 Principes de base des flots de données

Le mot «dataflow» revêt différents sens dépendant du contexte dans lequel il est utilisé. C'est sans doute en génie logiciel qu'on l'utilise le plus facilement : il fait alors référence à des flots d'informations entre des unités de traitement. KARP et MILLER furent les premiers à utiliser ce concept dans le contexte des calculateurs parallèles [51] en 1966. Les premiers à formaliser un modèle de calcul dataflow furent ADAMS en 1968 [3] et RODRIGUEZ en 1969 [67]. Les modèles dataflow ont pris leur plein essor lorsque KHAN a montré en 1974 qu'un réseau dataflow peut être modélisé avec un ensemble d'équations récurrentes [50]. Au même moment, DENNIS proposait la première architecture permettant de mettre en œuvre un programme dataflow [31, 35]. C'est le modèle, où plus précisément cette famille de modèles que nous allons étudier ici.

Un modèle de résolution est une méthode qui décrit comment un programme doit être évalué. Il est lié au modèle de programmation utilisé. Il est abstrait dans la mesure où l'on considère une réalisation idéale. La méthode de résolution la plus courante est le modèle de VON NEUMANN basé sur un flot de contrôle. Ce modèle suppose qu'un programme est une série d'instructions, effectuant soit un transfert de la mémoire centrale vers les registres de traitements ou inversement, soit un traitement au niveau des registres. La méthode pour exécuter un tel programme est d'exécuter la première instruction, puis la suivante dans le cas où la première ne modifie pas le flot de contrôle [71]. Par exemple, considérons le programme suivant :

```
c = if n == 0
      then a + b
      else a - b
    fi
```

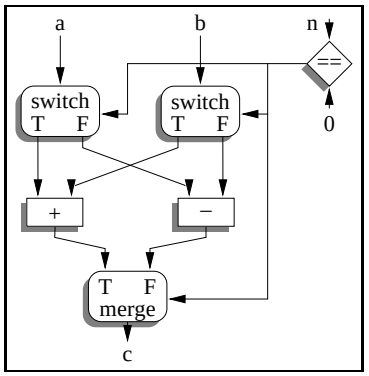


FIG. 2.1 - *Graphe dataflow.*

En utilisant un modèle de résolution basé sur un flot de contrôle, le programme va être converti en une série d'instructions débutant par une instruction qui compare `n` et `0` et transfère le contrôle soit vers une instruction qui calcule `a+b` soit vers une instruction qui calcule `a-b`, puis dans les deux cas vers une instruction qui range le résultat dans `c`. Ce modèle précise essentiellement la prochaine instruction à exécuter en fonction du résultat de l'instruction courante.

Un modèle dataflow ne repose pas sur un flot de contrôle mais sur un flot de données. Contrairement au modèle basé sur un flot de contrôle, il suppose que le programme est un graphe dépendant des données (ou graphe dataflow) dans lequel les nœuds sont des opérations et les liens sont des dépendances entre ces opérations. Le graphe dataflow du programme ci-dessus est donné dans la figure 2.1.

Lorsque ce graphe est évalué avec un modèle de résolution dataflow, il exécute toutes les opérations dénotées par un nœud dès que les liens en entrées ont suffisamment de données. En particulier, si `n` est disponible, l'opération `==` peut être appliquée à ces opérandes `n` et à la constante `0`. De même, si `a` et `b` sont tous deux disponibles, les deux opérations `+` et `-` peuvent être appliquées aux opérandes `a` et `b`, même si seul l'un des deux résultats est nécessaire. En réalité, ces dernières pourraient être exécutées avant même que le résultat de la comparaison de `n` et `0` ne soit obtenu.

La différence essentielle entre le modèle à flot de contrôle et le modèle à flot de données est que dans le premier, l'exécution d'un programme correspond aux instructions à effectuer sur les données restantes, alors que dans le second, un programme correspond aux données à traiter par les opérations restantes.

2.2 Les différentes sortes de Dataflow

Le modèle de programmation des graphes dataflow pilotés par les données est basé sur les graphes des programmes. Il existe plusieurs variantes de langages de graphes que nous allons sommairement décrire dans les sections suivantes.

2.2.1 Graphes dataflow pilotés par les données

Un graphe dataflow est un graphe direct dans lequel un nœud (aussi appelé acteur) dénote une opération et un lien, une dépendance entre deux opérations dénotées par des nœuds. Des valeurs sont consommées et produites par les nœuds ; elles sont alors transportées d'un bout à l'autre des liens vers l'opération suivante.

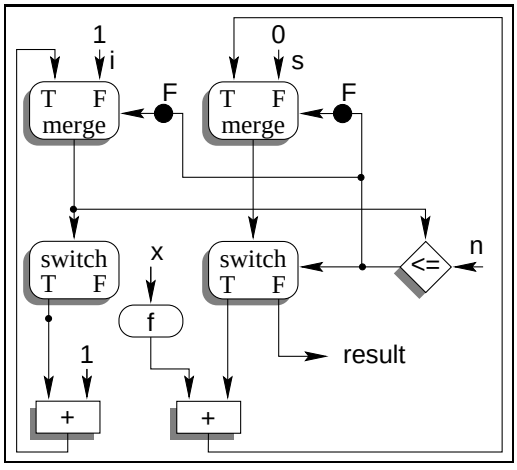


FIG. 2.2 - *Exemple de programme dataflow.*

Nous avons vu plus haut qu'il est naturel de représenter les expressions comme des graphes dataflow. En plus des opérateurs arithmétiques et logiques, il existe un certain nombre d'opérateurs, connus sous le nom d'opérateurs spéciaux, nécessaires pour exprimer autre chose que de simples expressions, comme par exemple des alternatives ou des boucles.

Le premier nœud de contrôle est le nœud `switch` qui aiguille la valeur en entrée sur l'une de ses sorties en fonction de la valeur présente sur sa seconde entrée. Le second nœud de contrôle est `merge`, utilisé pour sélectionner l'une ou l'autre de ses deux entrées principales vers sa sortie, en fonction de la valeur booléenne présente sur sa troisième entrée. Le troisième noeud de contrôle `apply` permet d'utiliser des fonctions de l'utilisateur. L'une des entrées de ce noeud est la fonction à invoquer, les autres entrées étant les arguments de cette fonction. Sa sortie est le résultat de l'application de la fonction aux arguments.

Nous pouvons voir dans la figure 2.2 un exemple de programme en graphe dataflow pour la formule $s = \sum_{i=1}^{n} f(x_i)$ qui illustre l'usage des opérateurs de contrôle.

2.2.2 Modèle statique classique

Le modèle dataflow statique classique [31, 35] repose sur la simple règle suivante pour décider si un opérateur est prêt à être exécuté :

> *Un opérateur dénoté par un nœud est exécutable lorsqu'une valeur est présente sur chacune de ses entrées.*

L'exécution d'un nœud consomme les valeurs en entrée et produit une valeur en sortie. Cette règle apporte deux contraintes :

- chaque lien ne peut transporter qu'une seule donnée en même temps. Cela signifie qu'un opérateur exécutable ne peut s'exécuter que si son lien de sortie ne contient pas de valeur. Cette contrainte impose l'usage de signaux de réception entre nœud ;
- certains opérateurs peuvent être exécutés même si certaines de leurs entrées ne contiennent pas de valeur. Par exemple, pour l'opérateur `merge`, si la valeur présente sur son entrée booléenne est `true`, seule la valeur sur sa première entrée est nécessaire. La valeur de la seconde entrée peut alors être supprimée lorsqu'elle arrive.

Ce modèle dataflow peut exploiter le parallélisme structurel (car des opérateurs sans dépendance peuvent être exécutés en même temps) et le parallélisme pipeline (car différentes parties d'un graphe consomment des valeurs d'un même flot à des instants différents).

Une seule itération d'une boucle, ou une seule invocation d'une même fonction peut être active au même moment, car les liens ne peuvent contenir qu'une seule valeur au même moment. Par conséquent, ce modèle ne peut exploiter les formes dynamiques de parallélisme, comme le parallélisme des boucles (en exécutant simultanément des parties sans dépendance du corps d'une boucle) ou bien le parallélisme récursif (en exécutant simultanément plusieurs appels à une fonction récursive).

Ce modèle ne peut pas non plus traiter les flots intermittents qui sont des flots dans lesquels certaines données sont indéfinies, car il ne fournit aucun moyen à un opérateur d'accéder au passé d'une donnée présente sur l'une de ses entrées.

Ce modèle est bien adapté pour les applications possédant une structure opérationnelle régulière, comme les applications de traitement du signal, ou du traitement des images [34] qui n'utilisent pas de structures de programme itératives ou récursives.

E.A. Lee et D.G. Messerschmitt ont étudié la manière de réaliser des graphes dataflow statiques multi-cadencés[57]. Il s'agit de graphes dataflow dont la circulation des données est soumise à des horloges différentes. Les auteurs établissent les propriétés que doivent posséder les graphes pour concerver le déterminisme en ressources. Les cannaux de communication entre nœuds sont alors des tampons de données de taille fixe.

2.2.3 Modèle dynamique classique

Bien qu'il y ait plusieurs variantes de dataflow dynamique, nous décrirons ici le modèle classique, en utilisant la plupart des constructions et notations de Arvind et Gostelow [9]. Dans ce modèle, les valeurs sont associées à un indicateur qui identifie de façon unique sa position conceptuelle dans le lien.

Un indicateur possède quatre champs $< c, i, b, a >$ où c est le numéro d'invocation, i le numéro d'itération, b l'adresse du bloc de code et a le numéro d'instruction dans le bloc de code. Le numéro d'invocation permet de distinguer deux données dans un appel de fonction récursif. Le numéro d'itération permet de distinguer deux données dans une itération. L'adresse du bloc de code ainsi que le numéro d'instruction permettent d'identifier la destination de la donnée.

La règle de déclenchement dans le modèle dynamique de dataflow est la suivante :

> *L'opérateur associé à un nœud est exécutable lorsque des données avec un indicateur identique sont présentes sur chacune de ses entrées.*

Un lien peut donc contenir plusieurs données au même instant, et ce nombre n'est pas limité. Lorsqu'un nœud est exécuté, les données présentes sur ses entrées sont enlevées et une donnée avec un indicateur approprié est produite sur sa sortie.

Ce modèle est qualifié de dynamique car la présence des indicateurs autorise l'exploitation du parallélisme issu des itérations et des fonctions récursives (qui survient dynamiquement,

au moment de l'exécution). Le coût de cette exploitation est la comparaison des indicateurs. Ceci est le point faible de ce modèle.

Le problème principal de ce modèle est l'exploitation excessive du parallélisme qui pénalise les performances. La plupart des réalisations consomment la mémoire de manière intensive, ce qui aboutit souvent à un blocage du système [8, 19, 68] ou à une sous-utilisation des processeurs [73]. Plusieurs approches ont été proposées pour éviter ces écueils, mais leur étude sort du cadre de cette introduction [15, 39, 45, 73].

2.2.4 Dataflow piloté par la demande

Dans le modèle dataflow piloté par les données, l'exécution de l'opérateur associé à un nœud n'est déclenchée que si une valeur est présente sur chacune de ses entrées et s'il existe une demande en aval pour le résultat. La motivation essentielle de ce modèle est de ne déclencher que les opérations strictement nécessaires. Les modèles pilotés par les données vus précédemment en sont incapables.

Il existe deux modèles de programmation pour ce type de graphe : les réseaux d'opérateurs [11] très semblables aux graphes dataflow, et les programmes fonctionnels [6] basés sur le λ-calcul. Nous décrirons les réseaux d'opérateurs et les modèles pour leur exécution en mode piloté par la demande. La programmation fonctionnelle sera l'objet des chapitres suivants.

2.2.5 Les réseaux d'opérateurs

Un réseau d'opérateurs est un modèle simple et graphique de programmation dans lequel les programmes ne prennent un sens mathématique qu'associés à des algèbres et des ensembles particuliers de valeurs en entrée.

Bien que les réseaux d'opérateurs apparaissent identiques aux graphes dataflow, il y a une différence majeure entre les deux modèles. La première différence est que les réseaux d'opérateurs ne font aucune hypothèse quant à leur mode d'évaluation. Par exemple, ils peuvent être soit pilotés par la demande, soit pilotés par les données. La seconde différence réside dans le fait qu'un réseau d'opérateurs dénote toujours une fonction, qui est la composition des fonctions correspondantes aux sous-réseaux.

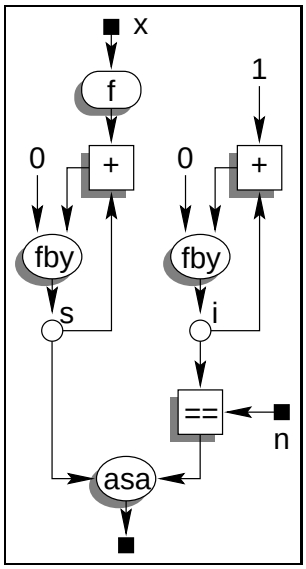


FIG. 2.3 - *Exemple d'un réseau d'opérateurs.*

La syntaxe associée aux réseaux d'opérateurs est la suivante. Le réseau d'opérateurs principal peut contenir plusieurs sous-réseaux utilisés comme des définitions de fonctions. Un réseau peut avoir plusieurs nœuds en entrée et seulement un seul nœud en sortie. Les nœuds d'entrée/sortie sont des nœuds spéciaux. L'ensemble des nœuds d'entrée et du nœud de sortie est l'interface du réseau avec l'extérieur qui peut être soit l'utilisateur lui-même dans le cas du réseau principal, soit l'environnement d'appel dans le cas d'un réseau définissant une fonction. Les nœuds d'entrée n'ont pas de lien d'entrée et un lien de sortie alors que les nœuds de sortie ont un lien d'entrée et pas de lien de sortie.

Il existe en plus des nœuds d'entrée / sortie, deux autres nœuds spéciaux : le nœud de duplication et le nœud applicateur. Le nœud de duplication possède un seul lien en entrée et un ou plusieurs liens de sortie. Le nœud applicateur possède un nom qui est soit le nom d'un opérateur provenant d'une algèbre, soit le nom d'une fonction qui correspond à un sous-réseau. Ce type de nœud possède un ou plusieurs liens en entrée et un seul lien en sortie. Le nombre de liens d'entrée doit correspondre soit à l'arité de l'opérateur, soit au nombre de nœuds d'entrée du sous-réseau. Les nœuds d'entrée/sortie peuvent être inter-connectés de toutes les manières, pourvu que les interfaces soient respectées. La figure 2.3 montre un réseau d'opérateurs pour évaluer la formule donnée précédemment.

D'un point de vue opérationnel, un lien dans un réseau d'opérateurs dénote une séquence de *datons* où un daton est un indicateur associé à une valeur et qui est identifié de manière unique sur le lien. Cet identificateur contient la référence du lien et la position logique de la donnée dans le lien. Chaque nœud dénote une fonction qui est appliquée aux datons en entrée pour produire un daton en sortie, que la fonction soit primitive ou pas.

Les réseaux d'opérateurs apparaissent donc comme un moyen de connecter des opérateurs entre eux, sans connaître leur nature. Les seuls opérateurs spéciaux sont l'opérateur applicateur, l'opérateur duplicateur et les opérateurs d'entrée/sortie. Ils permettent une construction hiérarchique des applications car un nœud applicateur peut faire référence soit à un opérateur d'une algèbre, soit à un sous-réseau.

2.2.6 Modèle à «éduction»

Le modèle à éduction est un modèle de graphe dataflow piloté seulement par la demande. La structure des programmes est la même que pour les autres modèles de graphes dataflow, avec la règle de déclenchement suivante :

> *Un opérateur dénoté par un nœud peut être évalué seulement lorsqu'une demande particulière survient sur sa sortie.*

La seule manière d'obtenir une valeur d'un opérateur est de la lui demander explicitement. la demande est répercutée aux entrées de l'opérateur, puis elle se propage le long du graphe jusqu'à ce qu'elle soit satisfaite. Lorsque l'opérateur a toutes ses valeurs en entrée, il effectue l'opération et produit la valeur demandée sur sa sortie. Il y a deux manières d'obtenir une valeur :

- lorsque la demande provient d'un lien d'entrée ou qu'elle porte sur une constante ;
- lorsque la valeur est demandée à un nœud duplicateur ayant déjà calculé et produit une seule valeur.

L'un des désavantages du modèle piloté par la demande est que la propagation de la demande doit avoir lieu avant la propagation des données. Par conséquence, le temps de calcul est multiplié par deux, par rapport au modèle piloté par les données.

Par contre l'avantage est de permettre une évaluation paresseuse en ne calculant que ce qui est nécessaire.

2.3 Synthèse

Les graphes dataflow proposent deux modèles principaux d'évaluation, le modèle piloté par la demande et le modèle piloté par les données. L'avantage du premier modèle est de permettre une évaluation paresseuse des applications, en ne calculant que le strict minimum. Par contre, son désavantage est de dupliquer le temps de propagation des données en nécessitant la propagation préalable de la demande. Le modèle piloté par les données évite la propagation de la demande, mais risque de traiter certaines expressions inutilement, comme les deux branches d'une alternative.

Les architectures parallèles basées sur les graphes dataflow sont inefficaces. Nous supposons que cela provient du grain trop fin des opérateurs. En effet, les temps de gestion des différents indicateurs est beaucoup plus important que le temps de calcul des opérations utiles. Il en résulte des architectures extrêmement gourmandes en temps de calcul et en ressources. L'autre critique est le manque de sémantique des graphes dataflow, qui ne sont en fait qu'un modèle d'exécution.

Le principal avantage des graphes dataflow est de permettre une programmation graphique des applications. Une telle programmation est facile d'accès, et en général, appréciée

des utilisateurs non spécialistes, si l'on en juge par le développement des interfaces graphiques, ces dernières années. Cependant, une programmation exclusivement graphique aboutit à des programmes illisibles car trop volumineux. La première solution est de permettre une programmation graphique hiérarchique, un peu à la manière des réseaux d'opérateurs, où un nœud est soit un opérateur primitif soit un sous-réseau. Une autre solution consiste à permettre une programmation mixte alliant un langage graphique et un langage syntaxique.

L'interface graphique λ-GRAPH développée au cours de notre étude tient compte de ces remarques (§ III-3). Elle permet une programmation hiérarchique et peut utiliser des parties écrites à l'aide du langage λ-FLOW (§ III-4).

Chapitre 3

Le λ-calcul

Le λ-calcul est un outil mathématique de formalisation du concept de fonction. Il définit le moyen d'écrire des fonctions, et les règles qui les manipulent. Il existe beaucoup d'ouvrages qui lui sont entièrement consacrés [55, 66] et d'ouvrages qui l'utilisent [47, 59, 60, 65].

Nous présentons ici le λ-calcul et les règles de réductions les plus importantes. Cette présentation montre bien les concepts de ce formalisme et introduit des concepts essentiels, comme les notions de variables libres ou liées, de réductions, de combinateurs.

Dans la dernière section, nous montrerons dans une synthèse les raisons qui nous poussent à ne pas utiliser directement le λ-calcul comme langage de modélisation. Mais cette section donnera aussi les aspects importants retenus pour la conception de notre formalisme.

Bien qu'il puisse sembler sortir du cadre de ce mémoire, il nous a semblé important d'y include ce chapitre pour deux raisons.

D'une part, ce mémoire s'adresse à des lecteurs qui pourraient éprouver le besoin de ce rappel sur le λ-calcul. D'autre part, bien que n'utilisant pas directement le λ-calcul, cette étude utilise constamment des concepts dont il est l'origine. Ce chapitre permet donc de définir implicitement l'acception des termes que nous utiliserons par la suite.

3.1 Fonctions mathématiques

En mathématique, il existe deux moyens de décrire une fonction. Le premier moyen est la définition en *extension*. Elle présente une fonction comme un ensemble explicite de couples. Par exemple :

$$f = \{(a, 1), (b, 2), (c, 3), (d, 4)\} \tag{3.1}$$

Cette méthode convient aux fonctions finies simples. Mais dans le cas de fonctions moins élémentaires, il est nécessaire de définir une fonction de façon plus abstraite en donnant ses propriétés plutôt qu'en énumérant de manière exhaustive ses couples. Il s'agit alors d'une définition en *compréhension*. Il existe une notation dite la notation lambda permettant de telles définitions. Cette théorie porte le nom de λ-calcul, qui se prononce lambda-calcul.

3.2 Syntaxe du λ-calcul

Sa syntaxe est très simple : les termes du λ-calcul sont des variables (noms), des abstractions (fonctions) ou des applications (appels de fonction). En notant ν l'ensemble des

variables utilisables, l'ensemble Λ des termes du λ-calcul peut être défini récursivement par[1] :

$$\begin{array}{lll} \text{variable :} & \forall x \in \nu, & x \in \Lambda \\ \text{abstraction :} & \forall x \in \nu, \forall M \in \Lambda, & \lambda x.M \in \Lambda \\ \text{application :} & \forall M, N \in \Lambda, & (M)N \in \Lambda \end{array} \tag{3.2}$$

Cette définition est une grammaire abstraite. Elle permet de construire des expressions du λ-calcul par induction sur des expressions plus simples. Les expressions les plus simples ne sont pas composées et elles sont appelées *atomes*. Ici, les atomes sont les variables.

3.2.1 Applications et curryfication

L'écriture $(f)x$ signifie *la fonction f appliquée à l'argument x.*

Comment exprimer l'application d'une fonction à plusieurs arguments ? Une nouvelle notation pourrait être utilisée, comme $(+)1, 3$, si $+$ est une fonction qui attend deux paramètres.

Cependant l'écriture $((+)1)3$ sera préférée. L'expression $(+)1$ représente *la fonction qui ajoute 1 à son argument.* L'expression entière dénote donc *la fonction $+$ appliquée à l'argument 1, dont le résultat est une fonction que l'on applique à 3.*

3.2.2 Abstraction

L'autre construction syntaxique du λ-calcul est l'abstraction, qui permet de construire de nouvelles fonctions. Par exemple :

$$\lambda x.((+)1)x \tag{3.3}$$

Le λ signifie «voici une fonction», et il est immédiatement suivi par une variable, x, puis par le corps de la fonction, $((+)1)x$. La variable est appelée *paramètre*, et nous disons que λ le lie. Nous pouvons lire l'expression de la manière suivante :

$$\begin{array}{cccccc} \lambda & x & . & ((+) & 1) & x \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ \text{La fonction} & \text{de } x & \text{qui} & \text{ajoute} & 1 & \text{et } x \end{array}$$

Une λ-abstraction comporte toujours ces quatre composantes : le λ, le paramètre formel, le point et le corps. Une abstraction avec plusieurs paramètres formels s'écrit :

$$\lambda x.\lambda y.((+)x)y \tag{3.4}$$

3.3 Sémantique opérationnelle du λ-calcul

Cette première étape de la description du λ-calcul ne s'est intéressée qu'à la syntaxe, c'est à dire aux outils pour écrire les programmes. Cette syntaxe est extrêmement simple puisqu'elle repose sur trois formes syntaxiques : la variable, l'abstraction et l'application.

Ici, le sens des écritures syntaxiques va être étudié : c'est l'aspect sémantique du λ-calcul. L'intérêt de cette étude est de montrer que la sémantique du langage est donnée à l'aide de fonctions qui forment une machine abstraite de résolution du langage. Ainsi, le langage est entièrement défini par le couple formé d'une grammaire et d'une machine abstraite.

Mais tout d'abord une terminologie importante est introduite.

1. Le λ-calcul n'a pas une seule syntaxe. Nous utilisons la syntaxe Revesz [66] qui offre moins de libertés que d'autres, évitant les confusions.

3.3.1 Variables libres et liées

Considérons l'expression :

$$(\lambda x.(((+)x)y)4 \tag{3.5}$$

Elle définit tout d'abord une fonction $\lambda x.((+)x)y$ à un paramètre x qui ajoute ce paramètre à la variable y, et applique cette fonction à 4 avec $(\lambda \ldots)4$.

Les variables x et y jouent des rôles différents. La variable x est destinée à prendre la valeur de l'argument 4 lors de l'application ; nous disons qu'il s'agit d'une variable liée.

Par contre, la valeur de y est inconnue. Le résultat de l'évaluation de cette expression est une fonction qui dépend de y. Pour évaluer complètement cette expression, il est nécessaire de définir une *variable globale* y et de lui donner une valeur.

On dit que y est une *variable libre* et que x est une *variable liée*. Les variables liées peuvent être vues comme des «trous» à remplir avec des arguments lors de l'application de la fonction. Nous remarquons aussi que leurs noms a peu d'importance et que moyennant quelques précautions, ils pourraient être changés :

$$(\lambda x.((+)1)x)3 \equiv (\lambda a.((+)1)a)3 \tag{3.6}$$

On appelle *combinateur* toute λ-abstraction qui ne possède aucune variable libre. Les combinateurs sont très utilisés dans la réduction des λ-expressions.

Ce qui suit s'occupe essentiellement de réduire les expressions en remplaçant les paramètres formels par les arguments de l'application : on dira que la réduction «évalue» l'expression. Ces réductions sont complexes car elles doivent tenir compte des éventuels télescopages entre les noms des variables et opérer à des changements de noms les cas échéants.

3.3.2 β-réduction

Comme une λ-abstraction représente une fonction, il est nécessaire de décrire le moyen de l'appliquer à un argument. Par exemple :

$$(\lambda x.((+)1)x)3 \tag{3.7}$$

est la juxtaposition de la λ-abstraction $\lambda x.((+)1)x$ et de l'argument 3. La règle pour une telle application de fonction est très simple :

> *Le résultat de l'application d'une λ-abstraction à un argument est une instance du corps de la λ-abstraction dans laquelle les occurrences du paramètre sont remplacées par des copies de l'argument.*

Donc le résultat de l'application de la λ-abstraction $\lambda x.((+)1)x$ à l'argument 4 est $((+)1)4$. Cette expression est une instance du corps $((+)1)x$ dans laquelle les occurrences du paramètre x sont remplacées par l'argument 4. La réduction s'écrit en utilisant la flèche $\longrightarrow$:

$$(\lambda x.((+)1)x)4 \longrightarrow ((+)1)4 \longrightarrow 5 \tag{3.8}$$

Cette opération de remplacement est appelée β-réduction.

3.3.3 β-abstraction

La règle β peut être utilisée à l'envers, pour construire de nouvelles abstractions, comme :

$$((+)4)1 \longleftarrow (\lambda x.((+)x)1)4 \tag{3.9}$$

Cette opération est appelée β-abstraction, notée avec une flèche de réduction inversée.

3.3.4 β-conversion

Une β-conversion est soit une β-réduction soit une β-abstraction, et elle est notée avec une double flèche $\longleftrightarrow_\beta$. Nous écrivons donc :

$$((+)4)1 \longleftrightarrow_\beta (\lambda x.((+)x)1)4 \tag{3.10}$$

La β-conversion peut être vue comme exprimant une équivalence de deux expressions «qui semblent différentes» mais qui ont «la même signification». Pour exprimer l'équivalence des expressions, deux règles supplémentaires sont nécessaires.

3.3.5 α-conversion

Considérons les deux abstractions suivantes :

$$\lambda x.((+)1)x \tag{3.11}$$

et :

$$\lambda y.((+)1)y \tag{3.12}$$

Elles *doivent* être équivalentes. Cela peut être établi par la α-conversion qui permet de changer le nom des paramètres formels d'une λ-expression, pourvu que cela soit réalisé de façon cohérente. Ainsi :

$$\lambda x.((+)1)x \longleftrightarrow_\alpha \lambda y.((+)1)y \tag{3.13}$$

La précaution à prendre est que le nom de la variable introduite ne doit pas apparaître libre dans le corps de la λ-abstraction originale, auquel cas elle deviendrait liée.

3.3.6 η-conversion

Une règle de conversion supplémentaire est nécessaire pour exprimer l'équivalence entre λ-abstractions. Considérons les deux expressions :

$$\lambda x.((+)1)x \tag{3.14}$$

et :

$$(+)1 \tag{3.15}$$

Ces expressions se comportent exactement de la même manière lorsqu'elles sont appliquées à un argument : elles lui ajoutent 1. La η-conversion est une règle exprimant une telle équivalence :

$$\lambda x.((+)1)x \longleftrightarrow_\eta (+)1 \tag{3.16}$$

Plus généralement la règle de η-conversion peut être exprimée comme suit :

$$\lambda x.(F)x \longleftrightarrow_\eta F \tag{3.17}$$

si x n'apparaît pas dans F, et F représente une fonction.

La condition de liberté empêche de fausses conversions. Par exemple $\lambda x.((+)x)x$ n'est pas η-convertible avec $+$ car la variable x apparaît libre dans $(+)x$. La condition imposant à F de représenter une fonction empêche de réaliser de fausses conversions de constantes pré-définies. Par exemple 3 n'est pas η-convertible avec $\lambda x.(3)x$. Lorsqu'elle est utilisée de gauche à droite, la η-conversion est appelée η-réduction.

3.3.7 Capture des noms de variables

Le problème de la capture des noms de variables met en évidence le fait que le λ-calcul est plus complexe qu'il n'y paraît au premier abord. Supposons définie une λ-abstraction :

$$twice = \lambda f.\lambda x.(f)(f)x \tag{3.18}$$

Considérons maintenant la β-réduction de l'expression $(twice)twice$:

$$(twice)twice = (\lambda f.\lambda x.(f)(f)x)twice \longrightarrow \lambda x.(twice)(twice)x \tag{3.19}$$

Maintenant, il y a deux radicaux β, $(twice)x$ et $(twice)(twice)x$. Choisissons arbitrairement de réduire le plus interne, en remplaçant d'abord $twice$ par sa λ-expression :

$$\lambda x.(twice)(twice)x \longrightarrow \lambda x.(twice)(\underline{\lambda f.\lambda x.(f)(f)x})x \tag{3.20}$$

Le problème apparaît : pour appliquer $twice$ à x, il est nécessaire de fabriquer une nouvelle instance du corps de $twice$ (souligné) en remplaçant les occurrences du paramètre formel f par l'argument x. Mais x est déjà utilisé comme paramètre formel à l'intérieur du corps de l'abstraction. Il est clairement faux de réduire en :

$$\lambda x.(twice)(\underline{\lambda f.\lambda x.(f)(f)x})x \overset{faux}{\longrightarrow} \lambda x.(twice)\underline{\lambda x.(x)(x)x} \tag{3.21}$$

car dans ce cas, le x substitué à f serait *capturé* par la λ-abstraction la plus interne. Nous appelons cela le problème de la *capture des noms de variables.* Une solution est d'utiliser la α-conversion afin de changer le nom de l'un des λx. Par exemple :

$$\begin{array}{lll} \longrightarrow \lambda x.(twice)(\underline{\lambda f.\lambda x.(f)(f)x})x & \longleftrightarrow_\alpha & \lambda x.(twice)(\underline{\lambda f.\lambda y.(f)(f)y})x \\ & \longrightarrow & \lambda x.(twice)\underline{\lambda y.(x)(x)y} \end{array} \tag{3.22}$$

On peut conclure que :

- la β-réduction n'est valide que si les occurrences libres des variables de l'argument n'entrent en conflit avec aucun paramètre formel du corps de la λ-abstraction ;
- la α-conversion est quelquefois nécessaire afin d'éviter la condition précédente.

3.3.8 δ-conversion

Il est possible de considérer les fonctions pré-définies comme une forme supplémentaire de conversion : la δ-conversion. Pour cette raison, les règles de réductions relatives aux fonctions pré-définies sont quelquefois appelées δ-règles.

Comme nous l'avons vu, l'application des règles de conversion n'est pas toujours immédiate, ce qui conduit à en donner une définition formelle. A cet effet, une notation supplémentaire sera utile.

La notation $E[M/x]$ représente l'expression E dans laquelle M est substituée à toutes les occurrences libres de x.

Il est possible de lire $E[M/x]$ comme E *dans laquelle* M *remplace les* x, de telle sorte que $x[M/x] = M$. Cette notation permet d'exprimer simplement la β-conversion par :

$$(\lambda x.E)M \longleftrightarrow_\beta E[M/x] \tag{3.23}$$

Elle est aussi utilisée pour exprimer la α-conversion.

3.4 Ordre de réduction

Si une expression ne contient aucun radical, alors l'évaluation est terminée et l'expression est dite en *forme normale*. Nous appelons *radical* une expression qui peut être réduite. Plus précisément, un radical possède au moins une application dont le premier terme est un λ. Lorsqu'une expression ne possède plus aucun radical, elle est en forme normale.

Cependant, une expression peut contenir plus d'un radical et dans ce cas la réduction peut s'exécuter selon plusieurs chemins. Par exemple, l'expression ((+) ((∗)3)4) ((∗)7)8 peut être réduite en forme normale par:

$$\begin{array}{lcl} ((+)\ ((*)3)4)\ ((*)7)8 & \rightarrow & ((+)\ 12)\ ((*)7)8 \\ & \rightarrow & ((+)\ 12)\ 56 \\ & \rightarrow & 68 \end{array} \tag{3.24}$$

ou par:

$$\begin{array}{lcl} ((+)\ ((*)3)4)\ ((*)7)8 & \rightarrow & ((+)\ ((*)3)4)\ 56 \\ & \rightarrow & ((+)\ 12)\ 56 \\ & \rightarrow & 68 \end{array} \tag{3.25}$$

Toute expression ne possède pas une forme normale. Considérons par exemple $(D)D$ où $D = \lambda x.(x)x$. L'évaluation de cette expression ne se termine pas car $(D)D$ se réduit en $(D)D$:

$$\begin{array}{lcl} (D)D & \rightarrow & (\lambda x.(x)x)\lambda x.(x)x \\ & \rightarrow & (\lambda x.(x)x)\lambda x.(x)x \end{array} \tag{3.26}$$

De plus, une réduction peut aboutir ou pas à une forme normale selon le chemin de réduction choisi. Par exemple, l'expression $(\lambda x.3)(D)D$ est réduite en une forme normale seulement si nous commençons la réduction par le terme de gauche.

3.4.1 Réduction en ordre normal

Ces considérations posent un problème embarrassant: deux chemins différents de réduction peuvent-ils aboutir à des formes normales différentes? Il est *indispensable* que la réponse soit négative. Heureusement, elle l'est: c'est la conséquence de deux théorèmes puissants connus sous le nom de théorèmes de CHURCH-ROSSER 1 et 2.

Théorème 1 *si $E_1 \longleftrightarrow E_2$, alors il existe une expression E telle que $E_1 \longrightarrow E$ et $E_2 \longrightarrow E$.*

Le corollaire suivant est une conséquence immédiate de ce premier théorème:

- corollaire: aucune expression ne peut être convertie en deux formes normales distinctes (i.e qui ne sont pas α-convertibles);
- preuve: supposons que $E \longleftrightarrow E_1$ et que $E \longleftrightarrow E_2$ où E_1 et E_2 sont en forme normale. Alors $E_1 \longleftrightarrow E_2$, et, d'après le théorème 1, il doit exister une expression F telle que $E_1 \longrightarrow F$ et $E_2 \longrightarrow F$. Mais E_1 et E_2 ne contiennent pas de radicaux, donc $E_1 = F = E_2$.

Le second théorème de CHURCH-ROSSER concerne un ordre particulier de réduction appelé l'*ordre normal*:

Théorème 2 *si $E_1 \longleftrightarrow E_2$ et si E_2 est en forme normale, alors il existe une suite de réductions de E_1 vers E_2 suivant l'ordre normal.*

Ce théorème indique qu'il y a au plus un résultat, et que l'ordre normal de réduction y aboutira si ce résultat existe. Remarquons aussi qu'aucune suite de réductions ne peut aboutir à un résultat incorrect ; le pire qu'il puisse arriver est la non terminaison de la réduction.

La réduction en ordre normal spécifie que le radical le plus extérieur et le plus à gauche doit être réduit en premier. Cela correspond à une évaluation paresseuse peu mise en œuvre dans les langages de programmation actuels : les arguments des fonctions sont évalués à la demande, si le corps de la fonction en a besoin.

3.4.2 Ordre optimal de réduction

L'ordre normal de réduction garantit de trouver une forme normale, si elle existe, mais ne garantit pas de la trouver par un nombre minimal de réductions. En fait, pour la réduction d'arbre, nous montrons que c'est la moins favorable. Mais il semble que si l'on considère la réduction des graphes, l'ordre normal est presque optimal, et il est probablement plus coûteux de trouver le radical préservant l'optimalité que de suivre l'ordre normal.

3.5 Les combinateurs

Lorsqu'une abstraction n'a pas de variables libres, elle est appelée combinateur. Le rôle joué par les combinateurs est essentiel dans le λ-calcul, car leur comportement ne dépend que de leur paramètres. Il peuvent donc être modélisés de manière mathématique. Voici un certain nombre d'exemples de combinateurs classiques :

$$\begin{array}{rcl}
true & = & \lambda x.\lambda y.x \\
false & = & \lambda x.\lambda y.y \\
nand & = & \lambda x.\lambda y.((x)false)((y)false)true \\
0 & = & \lambda f.\lambda x.x \\
1 & = & \lambda f.\lambda x.(f)x \\
2 & = & \lambda f.\lambda x.(f)(f)x \\
3 & = & \lambda f.\lambda x.(f)(f)(f)x \\
\ldots & & \\
succ & = & \lambda n.\lambda f.\lambda x.(f)((n)f)x \\
pred & = & \lambda n.(((n)\lambda p.\lambda z.((z)(succ)(p)true) \\
 & & (p)true)\lambda z.((z)0)0)false \\
if & = & \lambda cond.\lambda alors.\lambda sinon.((cond)alors)sinon \\
then & = & \lambda alors.alors \\
else & = & \lambda sinon.sinon
\end{array} \tag{3.27}$$

On voit qu'avec les combinateurs, il est possible de définir des données, comme $false$ et $true$, et même les entiers naturels avec les deux fonctions $succ$ et $pred$ qui retournent respectivement le successeur et le prédécesseur d'un entier.

3.6 Structure de données

Bien évidemment, le λ-calcul n'offre aucun moyen de structurer les données, puisqu'il ne définit même pas les données. Cependant, il est possible de définir une structure de liste à l'aide de l'application et de l'abstraction.

La structure de base dans une liste est la paire. Une paire assemble dans une structure logique deux éléments à l'aide du constructeur de paire. Il est aussi nécessaire de définir les sélecteurs qui permettent d'accéder aux éléments constituant une paire.

Une liste est alors définie comme une paire dont l'élément de gauche contient une valeur et dont l'élément de droite contient une liste. C'est une structure définie de manière récursive,

dont la terminaison aboutit à la définition d'un élément connu d'entres tous, et reconnu comme étant la liste vide :

$$\begin{array}{rcl} cons & = & \lambda a.\lambda b.\lambda z.((z)a)b \\ head & = & \lambda l.(l)true \\ tail & = & \lambda l.(l)false \\ nil & = & false \end{array} \tag{3.28}$$

Une paire est définie comme étant une fonction z qui retourne soit l'élément de droite, soit l'élément de gauche.

3.7 Fonctions récursives

Le λ-calcul permet de modéliser tous les langages fonctionnels. Or la récursion est une caractéristique très utilisée dans ce type de langages et le λ-calcul ne possède pas de construction ressemblant à la récursion.

Dans ce qui suit, il est montré comment le λ-calcul est capable d'exprimer des fonctions récursives sans aucune extension. Cette possibilité est tout à la fois un atout, car elle renseigne sur la puissance expressive du λ-calcul, et un désavantage, car elle permet dans certain cas un comportement aberrant. La suppression de ce comportement aberrant conduit à l'étude du λ-calcul typé, un sous-ensemble du λ-calcul.

3.7.1 Le combinateur Y

Considérons la définition récursive de la fonction factorielle :

$$\begin{array}{rcl} fact & = & \lambda n. \\ & & (((if)\ ((=)n)0) \\ & & (then)1) \\ & & (else)((*)n)(fact)((-)n)1 \end{array} \tag{3.29}$$

Cette définition repose sur le fait de pouvoir nommer une λ-abstraction et de faire référence à ce nom dans son corps : ici, la définition de l'abstraction nommée $fact$ utilise $fact$ dans son corps. Il n'y a pas de telle construction dans le λ-calcul. Comme les λ-abstractions sont des fonctions anonymes, elles ne peuvent pas se nommer et donc s'auto-référencer.

Ici, seul le cas des récursions simples sera étudié. Soit une définition récursive de type :

$$fact = \lambda n.(\ldots fact \ldots) \tag{3.30}$$

Cette définition peut être écrite sous la forme :

$$fact = (H)fact \tag{3.31}$$

où :

$$H = (\lambda fact.(\lambda n.(\ldots fact \ldots)))fact \tag{3.32}$$

La définition de H est simple. C'est une λ-abstraction ordinaire et elle n'utilise pas de récursion. La récursion est exprimée uniquement dans la définition 3.30 ci-dessus. La définition 3.30 ressemble à une équation mathématique. Par exemple, afin de résoudre l'équation :

$$x^2 - 2 = x \tag{3.33}$$

Nous cherchons des valeurs de x qui satisfont l'équation. De la même manière, pour résoudre 3.30, nous cherchons pour $fact$ une λ-expression qui satisfait 3.30. Comme pour les équations mathématiques, il peut exister plusieurs solutions. L'équation 3.30 :

$$fact = (H)fact \tag{3.34}$$

spécifie que le résultat de l'application de la fonction H à $fact$ est $fact$ elle-même. Nous disons que $fact$ est *une équation de point-fixe*. Une fonction peut avoir plusieurs point-fixes.

Nous recherchons donc un point-fixe de H. Clairement, cela ne peut dépendre que de H ; il est possible d'inventer une fonction Y qui à une fonction associe un point-fixe de cette fonction. La fonction Y a donc le comportement suivant :

$$(Y)H = (H)(Y)H \tag{3.35}$$

Y est appelé *un combinateur de point-fixe*. Si un tel Y existe, le problème est résolu et l'équation 3.30 deviendrait :

$$fact = (Y)H \tag{3.36}$$

c'est à dire une définition non-récursive de $fact$. Afin de nous convaincre que $fact$ possède le comportement attendu, calculons $(fact)1$. Les définitions de $fact$ et H sont rappelées :

$$\begin{array}{rcl} fact & = & (Y)H \\ H & = & \lambda fact.\lambda n.(((if)\ ((=)n)0)\ (then)1)\ (else)((*)n)(fact)((-)n)1 \end{array} \tag{3.37}$$

Donc :

$$\begin{array}{l} (fact)1 \\ =((Y)H)1 \\ =((H)(Y)H)1 \\ =((\lambda fact.\lambda n.(((if)\ ((=)n)0)\ (then)1)\ (else)((*)n)(fact)((-)n)1)(Y)H)1 \\ =(\lambda n.(((if)\ ((=)n)0)\ (then)1)\ (else)((*)n)((Y)H)((-)n)1)1 \\ =((*)1)((Y)H)0 \\ =((*)1)((H)(Y)H)0 \\ =((*)1)((\lambda fact.\lambda n.(((if)\ ((=)n)0)\ (then)1)\ (else)((*)n)(fact)((-)n)1)(Y)H)0 \\ =((*)1)(\lambda n.(((if)\ ((=)n)0)\ (then)1)\ (else)((*)n)((Y)H)((-)n)1)0 \\ =((*)1)1 \\ =1 \end{array} \tag{3.38}$$

3.7.2 Y est une λ-expression

Il est donc possible de transformer la définition récursive de $fact$ en une définition non récursive, mais en utilisant un combinateur de point-fixe Y. Cette fonction doit satisfaire la propriété suivante :

$$(Y)H = (H)(Y)H \tag{3.39}$$

Cette expression semble exprimer la récursivité dans sa forme la plus pure car elle peut exprimer toutes les autres fonctions récursives. Nous pouvons définir Y comme une λ-abstraction sans utiliser la récursion :

$$Y = \lambda h.(\lambda x.(h)(x)x)\lambda x.(h)(x)x \tag{3.40}$$

Afin de montrer que Y satisfait la propriété attendue, évaluons :

$$\begin{array}{rcl} (Y)H & = & (\lambda h.(\lambda x.(h)(x)x)\lambda x.(h)(x)x)H \\ & = & (\lambda x.(H)(x)x)\lambda x.(H)(x)x \\ & = & (\lambda u.(H)(u)u)\lambda x.(H)(x)x \\ & = & (H)(\lambda x.(H)(x)x)\lambda x.(H)(x)x \\ & = & (H)(Y)H \end{array} \tag{3.41}$$

Ce qui produit le résultat escompté.

3.7.3 Récursion et aberration

L'idée de définir la sémantique du λ-calcul en réduisant d'abord toute expression à sa forme normale par une suite de réductions est intéressante mais ne fonctionne pas toujours. Comme chaque étape de réduction supprime un λ, il a été conclu implicitement que le nombre total de λ en était automatiquement réduit. En fait, une étape de réduction peut ajouter un ou plusieurs λ quand elle en réduit un. Cette désagréable possibilité peut effectivement se présenter, comme le montre l'exemple suivant. Soit la définition de *AutoAuto*, fortement inspirée de Y :

$$AutoAuto = (\lambda x.(x)x)\lambda x.(x)x \tag{3.42}$$

Soit *Auto* la fonction à appliquer :

$$Auto = \lambda x.(x)x.\ Auto \tag{3.43}$$

prend un argument x et l'applique à lui-même. Cet argument doit être une fonction. *AutoAuto* peut paraître un cas extrême car elle est elle-même l'application d'une fonction *Auto* à l'argument *Auto*. Pourtant, rien dans la définition des λ-expressions n'exclut des expressions de ce genre.

Si nous essayons de réduire *AutoAuto*, nous remplacerons chaque occurrence de x dans le corps de *Auto* par l'argument *Auto*. Cependant comme ce corps est :

$$(x)x \tag{3.44}$$

Nous obtenons comme résultat :

$$(Auto)Auto \tag{3.45}$$

qui n'est rien d'autre que l'expression originale *AutoAuto*. D'un point de vue opérationnel, le «calcul» de la réduction de cette expression ne se termine jamais. Avec *AutoAuto*, toutes les réductions renvoient l'expression originale, tout en conservant constant le nombre de λ. Dans d'autres cas, une réduction peut en réalité augmenter ce nombre, de sorte que l'expression «grossit» indéfiniment.

Où est le problème? C'est la récursivité. Dans la plupart des cas, la récursivité est bien fondée. Mais dans certains cas, comme celui de *AutoAuto*, la possibilité qu'a une expression de s'auto-référencer conduit à une augmentation de la taille de l'expression, en terme de λ, en cours de réduction.

L'un des moyens de contrecarrer le problème général de la récursivité est de contraindre les λ-expressions en leur ajoutant un type : cela définit le λ-calcul typé.

En mathématique, décider si une fonction récursive est bien définie n'est pas une chose facile [40]. En règle générale, une fonction récursive est définissable si les expressions en argument sont de plus en plus «simples», ce qui nécessite de définir la relation d'ordre «plus simple». Pour les entiers, cette notion peut se traduire par le fait que les arguments doivent être de plus en plus petits. Pour les réels, ce ne peut pas être un critère valide car il existe une infinité de réels plus petits qu'une valeur donnée. Ces considérations ont abouti à l'étude de la sémantique dénotationnelle [59, 60, 70] construite sur des ensembles ordonnés appelés treillis.

Pour un langage de programmation défini par une grammaire inductive, une récursion qui décompose l'expression en sous-expressions est généralement acceptée. En effet, une grammaire inductive construit des objets qui «ne peuvent se contenir eux-mêmes», donc les composantes sont nécessairement plus simples que l'objet lui-même.

Dans cette étude, lorsqu'une fonction récursive nous semblera diffilement definissable, ou que l'étude de cette propriété sort du cadre de ce mémoire, elle sera appelé «axiome». Rappelons qu'un axiome est une propriété, ou définition, acceptée[2].

2. En traitement du signal, il est courant de concevoir des systèmes récursifs sans fin. Comment montrer qui sont définissables?

3.8 Synthèse

Le λ-calcul est la base théorique des langages de programmation fonctionnelle. Il est extrêmement simple, et son pouvoir de description est important. De plus, il repose sur les mathématiques, ce qui lui confère des propriétés théoriques intéressantes.

Mais il ne peut servir directement d'outil de modélisation car le pouvoir expressif du λ-calcul est trop important. Or, nous désirons montrer les déterminismes en temps et en ressources des programmes. Comment le faire lorsque l'utilisateur peut réaliser *cons*, le constructeur de cellules ? De plus, l'existence d'expressions non réductibles, comme *AutoAuto*, est très gênante.

Par contre, nous retiendrons certaines caractéristiques particulièrement attrayantes. Le fait de définir un langage à la fois statiquement avec une grammaire, et dynamiquement, avec une machine abstraite, nous semble indispensable.

L'utilisation d'un formalisme fonctionnel semble incontournable. Il permet de donner une équivalence mathématique aux programmes, et donc de vérifier leurs propriétés formelles.

De plus, l'utilisation de combinateurs pour définir la machine abstraite de résolution est très avantageuse car ils introduisent le déterminisme de la résolution.

Chapitre 4

Le langage scheme

Le λ-calcul vu dans le chapitre précédent est un modèle mathématique des langages fonctionnels. Sa grande simplicité facilite son étude théorique. Cependant, il est peu commode de programmer une application réelle dans ce formalisme, car d'une part, il existe peu d'environnements de développements basés directement sur le λ-calcul, et d'autre part, la méthode d'évaluation en mode normal est très coûteuse en temps de calcul.

Tout au long de notre développement, nous utiliserons SCHEME comme un support destiné à faciliter la compréhension du lecteur. De plus, ce langage introduit des concepts clefs de la programmation fonctionnelle, concepts auxquels nous ferons abondamment allusion au cours de notre développement, dans la seconde partie.

Vers la fin de cette présentation, nous mettrons en évidence que la suprématie des langages fonctionnels sur les langages impératifs ne peut être établie de manière évidente. Nous montrerons comment transformer un programme dit impératif en programme fonctionnel, et que cette transformation peut être automatisée.

La dernière section nous présentera une synthèse de SCHEME.

Bien qu'il puisse sembler sortir du cadre de ce mémoire, il nous a semblé important d'y inclure ce chapitre pour deux raisons.

D'une part, ce mémoire s'adresse aussi bien à des lecteurs universitaires qui connaissent en général les langages de la famille de LISP qu'à des lecteurs issus de l'industrie qui pourraient éprouver le besoin de cette présentation à SCHEME.

D'autre part, cette étude est conçue de manière à n'utiliser qu'un seul langage de programmation. Ce chapitre pourra donc être consulté à la manière d'un manuel de référence à l'aide de l'index lorsque le lecteur en éprouvera la besoin.

4.1 Introduction

Le langage SCHEME peut être vu comme un λ-calcul en mode applicatif. Il existe d'excellents interprètes SCHEME dans le domaine public, basés sur la norme du MIT [18]. Nous utilisons STK de ERIC GALLESIO [37] qui a l'avantage de permettre la programmation d'applications graphiques utilisant la boîte à outils TK [64].

SCHEME est un descendant de LISP [65]. Cependant, il y a un certain nombre de différences majeures, comme la portée statique des liaisons des variables, le fait de considérer les fonctions comme les autres objets (on dit qu'elles sont des citoyens de première classe). Les langages impératifs de la famille de PASCAL orientent le programmeur à construire des structures de données, puis à réaliser les fonctions qui vont les manipuler. SCHEME permet de s'attacher davantage aux fonctionnalités de l'application. Ceci provient en grande partie du mécanisme automatique de la gestion de la mémoire de SCHEME.

La cellule est la structure élémentaire de SCHEME. Il n'y a pas de structure qui ne puisse être décrite à l'aide d'un ensemble de cellules. Par rapport à PASCAL, l'atome des structures est toujours le même : il s'agit de la cellule. Là où il suffit d'invoquer trois fois le constructeur

de cellules, il faut en PASCAL définir une structure de données, la créer, la gérer pour enfin la détruire. De plus, l'environnement d'exécution d'un programme SCHEME se charge totalement de la gestion de la mémoire, soulageant grandement l'effort de programmation.

Les concepts utilisés dans SCHEME sont extrêmement simples et faciles à comprendre. Cependant, nous montrerons que les notions sémantiques sont assez subtiles et reposent sur le λ-calcul. SCHEME ne dispose que de peu de moyens pour construire les expressions composées et de pratiquement aucune structure syntaxique. Le *manuel du programmeur* est un excellent livre écrit par ABELSON et SUSSMAN [1], et le *manuel de référence* est la norme du langage du MIT [18].

Nous devons définir un certain nombre de points syntaxiques. Puis nous présenterons le langage comme si nous disposions d'un interprète SCHEME. Enfin, nous ferons la synthèse du langage.

4.2 Éléments de base

Voici des expressions SCHEME :

```
STk> ; ceci est un commentaire
STk> 21
21
STk> (+ 1 2 3)
6
```

où l'invite de l'interprète est simulée par `STk>`. Chaque ligne représente une commande donnée à l'interprète, et sa réponse apparaît sur les lignes qui commencent sans son invite. Une réponse peut tenir sur plusieurs lignes.

4.2.1 Type des données

Par opposition au *type de données manifeste* où le type de la donnée est explicitement déclaré, les données de SCHEME possèdent un type latent. Les types sont associés aux valeurs et non aux variables (pour l'instant, considérons une variable comme étant un emplacement nommé contenant une valeur). Une variable pourra donc changer plusieurs fois de type, puisque le type est associé à la valeur qu'elle contient. Le type de la donnée est déduit de son écriture. Ainsi, la chaîne de caractère `123` est interprétée comme un entier.

4.2.2 Appel de fonction

La syntaxe pour appeler une fonction (on dira appliquer une fonction) est : `(opérateur arg-1 arg-2 ...arg-n)`. L'opérateur prend la première position et les arguments de l'application occupent les positions suivantes. Cette syntaxe est équivalente à la syntaxe λ-calcul : $(((operateur)arg_1)\ldots)arg_n$, qui est sous une forme curryfiée (§ 3.2.1).

4.2.3 Arguments

Les arguments des applications sont passés par valeur. De plus, l'évaluateur fonctionne en mode applicatif, c'est à dire que les arguments sont pré-évaulés et que les résultats de ces évaluations sont passés à la fonction[1].

4.2.4 Définition

SCHEME permet de nommer des objets. Cela s'écrit :

```
STk> (define nom-objet 21)
```

1. Le mode d'évaluation applicatif de SCHEME s'oppose au mode normal du λ-calcul où les arguments ne sont pas pré-évalués.

```
#unspecified
```

Cette écriture donne à la variable `mon-objet` la valeur `21` de type entier. La réponse renvoyée par `define` est `#unspecified`, qui indique qu'elle n'est pas spécifiée. Maintenant, nous pouvons utiliser `mon-objet` comme synonyme de `21`, comme dans l'expression :

```
STk> (+ mon-objet 4)
25
```

Remarquons que les noms de variable peuvent utiliser le signe moins. En fait, tous les caractères sont utilisables pour définir les noms de variables, à l'exception des «blancs» que sont l'espace, la tabulation et le retour à la ligne.

La fonction `define` est le moyen d'abstraction le plus simple du langage, car il permet de nommer des résultats d'opérations composées. Il serait extrêmement pénible d'écrire un programme sans pouvoir donner un nom aux expressions. Cette possibilité d'associer un nom à une valeur impose à l'environnement d'exécution des programmes SCHEME de tenir un dictionnaire des symboles. Ce dictionnaire est appelé *environnement*, et plus précisément dans ce cas, *environnement global*.

Un environnement contient des associations entre des noms et des valeurs. Cette structure permet de retrouver la valeur avec le nom.

4.2.5 Formes spéciales

Nous avons vu précédemment que lors d'une application d'une fonction à des arguments, les arguments sont préalablement évalués, et c'est le résultat de cette évaluation qui est passé comme argument à la fonction. Dans l'écriture `(define mon-objet 21)`, l'entier `21` est évidemment évalué en l'entier 21. Mais quelle est la valeur retournée par l'évaluation du symbole `mon-objet` ?

Il est ici évident que certaines fonctions, dont `define`, ne peuvent être appliquées à leurs arguments de manière classique. Ici, lorsque l'environnement d'exécution se rend compte que la fonction est `define`, il n'évalue pas les arguments. Ces fonctions sont appelées *formes spéciales*. Concernant la règle d'évaluation, SCHEME est en mode applicatif pour les fonctions (évaluation préalable des arguments) et en mode normal pour les formes spéciales (pas d'évaluation des arguments).

4.2.6 Durée de vie des objets

La définition de la variable `mon-objet` crée quelque part dans l'environnement d'exécution une association entre l'identificateur `mon-objet` et la valeur `21`. À quel moment cet emplacement sera-t-il libéré ? La réponse est jamais, ou plutôt, lorsque l'on quittera l'environnement d'exécution du programme.

Par contre l'emplacement occupé par la valeur 21 dans l'association sera libéré lorsque l'on redéfinira la variable `mon-objet`. Ce mécanisme est connu sous le nom de *ramasse miettes* : lorsque l'espace disponible pour de nouvelles définitions se fait rare, ce mécanisme est chargé de récupérer l'espace occupé par toutes les valeurs qui ne sont pas attachées à une variable. Le programmeur est de ce fait libéré de cette tâche qui est une source de bien des difficultés dans les langages classiques, même évolués, comme C++, dès que les structures manipulées sont cycliques.

4.2.7 Évaluation des expressions

L'interprète SCHEME utilise une forme spéciale chargée d'évaluer les expressions, qui

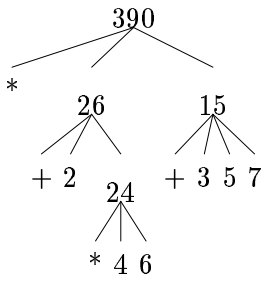
- évalue les sous-expressions de l'expression principale,
- applique l'opérateur (premier terme à gauche) aux arguments (autres termes) de l'expression obtenue.

Bien entendu, ce fonctionnement ne tient pas compte des formes spéciales, qui sont évaluées par un autre moyen.

Nous voyons qu'avant d'appliquer effectivement l'opérateur, il faut évaluer chacun des éléments de l'expression. La règle d'évaluation est donc récursive par nécessité. Par exemple :

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

impose d'appliquer la règle d'évaluation à quatre formes différentes. La figure suivante donne une image de ce calcul sous la forme d'un arbre. Chaque forme est représentée par un nœud dont les branches correspondent à l'opérateur et aux opérandes :



Cet arbre se parcourt en suivant les sous-arbres de gauche à droite, puis la racine. Les résultats intermédiaires sont conservés dans chaque nœud.

4.2.8 Représentation externe

Nous avons vu au cours des exemples précédents que l'interprète donne toujours une expression en retour. Cette expression est la *représentation externe* d'un objet. Une liste est représentée par un ensemble de valeurs entre parenthèses. Un couple est représenté par deux expressions séparées par un point, le tout entre parenthèses. Tout objet SCHEME a une représentation externe. Ainsi, si nous entrons :

```
STk> +
#procedure
```

l'interprète nous retourne `#procedure` qui indique que l'objet évalué est une fonction. Dans certains cas, comme celui-ci, la forme de la valeur de retour dépend de l'interprète utilisé.

4.2.9 L'alternative : `if`

Son écriture générale est :

```
(if condition clause-alors clause-sinon)
```

où `clause-alors` et `clause-sinon` sont des expressions. Le résultat de l'écriture dépend du résultat de l'évaluation de la `condition`. S'il est faux, l'évaluation de `clause-sinon` est retournée, et dans le cas contraire, l'évaluation de `clause-alors` est retournée. La `clause-sinon` est facultative ; si elle est omise et si l'évaluation de la `condition` est fausse, la valeur `#undefined` est retournée.

L'alternative de SCHEME est une forme spéciale dans la mesure où l'une seulement des clauses est évaluée, en fonction du résultat de l'évaluation de la condition. Cette définition du `if` est une sorte d'évaluation paresseuse, où seules les clauses nécessaires sont évaluées. Mais la nécessité de cette forme spéciale provient du moteur d'évaluation de SCHEME qui est, nous l'avons vu, en mode applicatif. Si cette forme spéciale n'existait pas, les fonctions récursives ne pourraient pas être écrites.

Les clauses de `if` sont de simples expressions. Lorsque plusieurs actions doivent être effectuées pour une seule clause, il est nécessaire de les rassembler dans une construction `begin`.

4.3 Les fonctions

4.3.1 Définition de fonction

L'opérateur spécial `lambda`, qui est à rapprocher de λ du λ-calcul, permet de définir de nouvelles fonctions. Cet opérateur attend une liste d'identificateurs qui seront les paramètres de la fonction, et une ou plusieurs instructions qui forment le corps de la fonction. Par exemple :

```
STk> (lambda (a b) (+ a b 10))
#procedure
```

est une fonction à deux paramètres qui retourne la somme des deux paramètres et de dix. Bien évidemment, il est possible d'appliquer une telle fonction à des arguments, comme dans :

```
STk> ((lambda (a b) (+ a b 10)) 3 4)
17
```

L'opérateur spécial `define` permet de nommer une fonction et de la conserver dans l'environnement global. Par exemple, la fonction d'incrémentation peut s'écrire :

```
STk> (define inc (lambda (n) (+ 1 n)))
#unspecified
```

Nous pouvons maintenant utiliser la nouvelle fonction `inc` :

```
STk> (inc 7)
8
```

Il existe une forme de `define` spécialement conçue pour simplifier les définitions de fonction :

```
STk> (define (inc n) (+ 1 n))
#unspecified
```

4.3.2 Statut des fonctions

SCHEME a une particularité très intéressante qui le distingue de la plupart des autres langages : les fonctions sont des citoyens de première classe, ce qui signifie qu'elles sont des objets de même nature que les entiers ou tout autre objet, aux vues de l'interprète[2].

Notamment, une fonction peut être passée en paramètre, ou être retournée par une autre fonction. Considérons par exemple la fonction de dérivation mathématique. La proposition «la dérivée de x^3 est $3x^2$» signifie que la dérivée de la fonction dont la valeur en x est x^3 est une autre fonction, et plus précisément la fonction dont la valeur en x est $3x^2$.

Plus généralement, la dérivation peut être vue comme un opérateur qui, appliqué à une fonction f, donne pour résultat une fonction Df. Pour décrire la dérivée, nous pouvons dire que si f est une fonction et dx un nombre, alors la dérivée Df de f est la fonction dont la valeur au point x est donnée par :

$$Df(x) = \frac{f(x+dx)-f(x)}{dx}$$

lorsque dx tend vers zéro. Traduite en SCHEME, cette expression devient :

```
(lambda (x)
  (/ (- (f (+ x dx)) (f x))
     dx)))
```

où dx est un nombre.

2. Les deux autres langages possédant cette propriété sont COMMON LISP et ML [18].

Nous pouvons utiliser ceci pour exprimer l'idée de la dérivée elle-même dans la fonction :

```
(define (dérivée f dx)
    (lambda (x)
       (/ (- (f (+ x dx)) (f x))
          dx)))
```

La fonction `dérivée` prend pour argument une fonction `f` et retourne une fonction (produite par `lambda` qui, appliquée à un nombre `x`, fournit une valeur approchée de la dérivée de `f` en `x`.

Nous pouvons par exemple utiliser la fonction `dérivée` pour calculer une valeur approchée de la dérivée de la fonction `cube` en `5` :

```
STk> (define (cube x) (*  x x x))
#unspecified

STk> ((dérivée cube .001) 5)
75.015
```

La valeur exacte attendue est 75.

4.3.3 Fonction récursive

Considérons la fonction factorielle définie par :

$$n! = n.(n-1).(n-2)\ldots 3.2.1$$

Il y a plusieurs façons de calculer une factorielle. L'une d'entre elles part de l'observation que $n!$ est égal à n fois $(n-1)!$. Il est donc possible de calculer $n!$ en calculant $(n-1)!$ et en multipliant le résultat par n. En sachant que $1! = 1$, nous obtenons :

```
STk> (define (fact n)
       (if (= n 1)
          1
          (* n (fact (- n 1)))))
#unspecified
```

Voici comment se déroule le calcul de `(fact 4)` :

```
(fact 3)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

Nous pouvons calculer la factorielle d'une autre façon. Une règle de calcul de $n!$ peut consister à multiplier d'abord 1 par 2, puis à multiplier le résultat par 3, puis par 4, jusqu'à n.

Plus concrètement, nous mettons à jour le produit courant et un compteur allant de 1 à n. Nous pouvons décrire le calcul en disant que le compteur et le produit varient en même temps d'une étape à l'autre en suivant la règle :

$$\begin{aligned} \mathit{produit} &\longleftarrow \mathit{compteur} \times \mathit{produit} \\ \mathit{compteur} &\longleftarrow \mathit{compteur} + 1 \end{aligned}$$

en précisant que $n!$ est la valeur du produit lorsque le compteur dépasse n. Cette description donne aussi une fonction de calcul de la factorielle :

```
STk> (define (fact-iter produit compteur maximum)
       (if (> compteur maximum)
         produit
         (fact-iter (* compteur produit)
                    (+ compteur 1)
```

```
                    maximum)))
#unspecified

STk> (define (fact n) (fact-iter 1 1 n))
#unspecified
```

Comme précédemment, examinons le processus d'évaluation de l'application `(fact 4)` :

```
(fac 4)
(fact-iter  1 1 4)
(fact-iter  1 2 4)
(fact-iter  2 3 4)
(fact-iter  6 4 4)
(fact-iter 24 5 4)
24
```

Comparons les deux processus. Ils calculent la même fonction mathématique, en un même nombre de pas de calcul. Mais le premier processus utilise visiblement une zone de stockage des résultats intermédiaires différente du premier.

Le premier processus est récursif. Cette forme de programmation existe dans tous les langages évolués. Ceux-ci utilisent une structure de pile pour créer l'environnement local à chaque fonction qui reçoit la valeur des arguments. A chaque appel de la fonction, un environnement local est créé. Si une fonction récursive ne possède pas de condition d'arrêt, l'espace occupé par les environnements locaux correspondant à chaque appel augmente sans cesse, ce qui aboutit tôt ou tard à un blocage du programme.

Le second processus est itératif. Il s'agit d'une boucle qui s'arrête selon une certaine condition. Si la condition d'arrêt est supprimée, le programme ne se terminera jamais. Cependant, le programme ne sera jamais interrompu par manque de ressource car l'espace occupé reste constant.

Les fonctions récursives gérèrent donc un indéterminisme en ressources. Mais certaines fonctions récursives utilisent un espace constant. Ce sont les fonctions à récursion terminale.

4.3.4 Récursion terminale

Considérons la fonction suivante :

```
STk> (define (pour-chaque fonction liste)
       (if (not (null? liste))
         (begin
           (fonction (car liste))
           (pour-chaque (cdr liste) fonction))))
#unspecified
```

Cette fonction applique une fonction à tous les éléments d'une liste. Comme nous pouvons le voir, elle est définie de manière récursive, puisqu'elle s'appelle elle-même. Mais considérons cette même fonction programmée en C, en nous interdisant l'appel récursif :

```
void pour_chaque (void (* fonction)(), TypeListe liste) {
  boucle:
    if (! null (liste)) {
      fonction (car (liste));
      liste = cdr (liste);
      goto boucle;
    }
}
```

Cette fonction fonctionne parfaitement en consommant un espace constant. Cette traduction itérative de la fonction `pour-chaque` est possible parceque le résultat d'un appel à cette fonction est le résultat de l'appel suivant, avec des arguments modifiés. Il n'y a pas de stockage des résultats intermédiaires. Ce type de fonction est détecté par les interprètes SCHEME, et la récursion est traduite en itération dans la machine d'exécution. Cette transposition existe aussi dans certains compilateurs de langages impératifs tel que le C.

4.4 Les environnements

4.4.1 Variables libres et liées

Nous avons déjà abordé le sujet au cours de notre introduction au λ-calcul. Rappelons simplement que dans une fonction, il existe deux sortes de variables, les variables libres et les variables liées. Les variables liées sont les noms donnés aux paramètres des fonctions, et les variables libres ne font pas partie des paramètres. Lorsque nous écrivons :

```
STk> ((lambda (a b) (+ a b c)) 3 4)
```

les variables `a` et `b` sont liées car elles font partie de la liste des paramètres. Par contre, la variable `c` doit être trouvée ailleurs. Dans ce cas, l'interprète la cherchera dans l'environnement global. Si `c` a été définie par `(define c 31)` par exemple, le résultat affiché sera `38`. Si `c` n'a pas été précédemment définie, l'interprète affichera un message d'erreur indiquant que `c` n'est pas définie. Mais que se passe-t-il dans le cas suivant :

```
STk> ((lambda (a b)                ;(1)
        ((lambda (a b) (+ a b))    ;(2)
           (+ a b)
           (* a b))) 3 4)
```

Il y a visiblement un conflit entre les variables `a` et `b` liées dans les deux fonctions `lambda`.

Dans la réalité, l'opérateur `lambda` crée un environnement local dans lequel il place les associations entre les paramètres et les arguments. Dans le cas de la première fonction (1), cet environnement a pour parent l'environnement global. L'environnement local de la seconde fonction (2) a pour parent l'environnement local de la première fonction. Il existe donc une hiérarchie des environnements, dont le père est l'environnement global.

4.4.2 Définir des symboles : `define`

Cette fonction que nous avons déjà vue permet de définir des symboles dans l'environnement en cours. Si le symbole existe, sa valeur associée est remplacée, sinon, il est créé. L'utilisation la plus naturelle de `define` est dans l'environnement global.

Cette fonction agit par effet de bord et en séquence dans l'environnement global. C'est justement l'effet recherché. Mais il y a une utilisation plus subtile de cette fonction, dans le corps des fonctions :

```
(define (une-fonction)
   (define a 3)
   (display a)
   (define a 300)
   a)
```

Le `lambda` crée un nouvel environnement. Le premier `define` ajoute une association dont le symbole est `a` dans cet environnement. Le second modifie la valeur liée à `a`. Cela signifie que `a` n'a pas toujours la même valeur dans le corps de `une-fonction`.

Le second `define` effectue en fait un `set!`.

4.4.3 Affectation : `set!`

Cette fonction permet de modifier la valeur associée à un symbole dans l'environnement courant. Si l'association n'existe pas, une erreur se produit. Cette fonction est généralement utilisée dans le corps des abstractions pour modifier la valeur des arguments.

Toute écriture qui contient un `set!` n'est pas fonctionnelle.

4.4.4 Variables locales : `let`

Cette fonction permet de définir des variables locales à un certain nombre d'expressions. Elle s'utilise comme suit :

```
(let ((nom-1 valeur-1)
      (nom-2 valeur-2)
      ...
      (nom-n valeur-n))
   expression-1
   expression-2
   ...
   expression-n)
```

où les valeurs sont des expressions de SCHEME. Cependant, elles ne peuvent faire référence à aucun des `nom-i`, qui leur sont invisibles.

Il faut remarquer que `let` est simplement une écriture qui peut être remplacée par la primitive `lambda` comme suit :

```
((lambda (nom-1 nom-2 ... nom-n)
    expression-1
    expression-2
    ...
    expression-n)) valeur-1 valeur-2 ... valeur-n)
```

Ainsi, nous comprenons mieux pourquoi les valeurs ne peuvent utiliser les noms.

4.4.5 `let` nommé

Il existe une variante du `let` appelée `let` nommé. Elle est expliquée dans la section 4.6.17.

4.4.6 `let*`

Il existe une autre forme de `let` notée `let*`. Son écriture est la même que pour le `let`. Mais dans cette écriture, les valeurs peuvent utiliser tous les noms précédemment définis :

```
(let* ((a  3)
       (b (+ a 15))
       (c (+ a  b)))
   (* a b c))
```

Le `let*` peut lui aussi se ramener à une expression purement fonctionnelle utilisant `lambda` :

```
(let ((a 3))
  (let* ((b (+ a 15))
         (c (+ a  b)))
    (* a b c)))
```

4.4.7 `letrec`

La syntaxe de cette expression est la même que pour les deux précédentes. Par contre, les valeurs peuvent utiliser tous les noms définis. Nous avons par exemple :

```
(letrec ((f (lambda (x) (g x))
         (g (lambda (y) (f y)))
   (g 15))
```

`letrec` est la contraction de *let recursively*. Cet opérateur introduit des liaisons éventuellement récursives pour les variables définies. Nous pouvons donner une traduction de cet opérateur avec des opérateurs primitifs avec :

```
(letrec ((v1 E1))
  E)
```

qui est équivalent à :

```
(let ((v1 #unspecified))
  (let ((temp1 E1))
    (set! v1 temp1))
  E)
```

Il faut remarquer la présence du `set!` qui enlève la propriété fonctionnelle de la construction.

4.5 Structure de données

4.5.1 Quotation

La quotation est un mécanisme indiquant à l'interprète de ne pas évaluer l'expression qui suit. Par exemple :

```
STk> '(+ 1 2)
(+ 1 2)
```

Le résultat de cette expression est la liste formée des identificateurs `+`, `1` et `2`. La liste vide est notée :

```
STk> '()
#Nul-object
```

4.5.2 Quasi-quotation

La quasi-quotation permet de construire des expressions non évaluées, dont certaines parties sont évaluées dans l'environnement courant. Par exemple :

```
STk> `(+ 1 2)
(+ 1 2)
```

Le résultat de cette expression est la liste formée des identificateurs `+`, `1` et `2`. Nous avions obtenu ce résultat avec la quotation. Considérons maintenant :

```
STk> `(+ 1 ,(+ 2 3))
(+ 1 5)
```

L'expression introduite par une virgule dans une expression quasi-quotée est évaluée, et le résultat de cette évaluation est ajouté dans l'expression quotée. La quasi-quote est beaucoup utilisée dans les techniques de compilation, ou de remplacement de chaînes de caractères.

4.5.3 Cellule

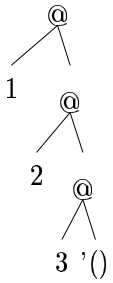
La cellule est la structure essentielle de SCHEME. Une cellule se compose d'une tête et d'une queue. La tête et la queue peuvent être elles-mêmes des cellules. Il existe trois opérateurs pour les manipuler, `cons` qui est le constructeur, `car` qui retourne la tête de la cellule et `cdr` qui retourne la queue.

Nous avons vu dans le chapitre consacré au λ-calcul des opérateurs similaires nommés *cons*, *head* et *tail* qui sont définis par des λ-expressions. Il en est de même avec SCHEME, et `cons`, `car` et `cdr` peuvent être écrits en SCHEME.

Les cellules permettent de construire des listes. Par exemple, nous voulons former la liste des trois premiers entiers naturels. Pour cela, nous écrivons :

```
STk> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

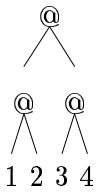
Une représentation schématique de cette liste peut être :



Il est important de noter la présence de l'élément spécial `'()`. Il représente la cellule vide, et pour conserver une structure homogène à la liste, il en est le dernier élément. À l'aide des cellules, il est aussi possible de construire la structure suivante :

```
STk> (cons (cons 1 2) (cons 3 4))
((1 . 2)(3 . 4))
```

Cette structure peut être représentée schématiquement par :



Nous voyons donc qu'avec les seules cellules, il est possible de construire des structures complexes telles que des listes de couples, des arbres, etc. Une autre façon de construire les listes est d'utiliser la quotation, comme nous l'avons vu plus haut.

4.5.4 `cons`, `car` et `cdr`

Nous avons dit précédemment que ces opérateurs ne sont pas essentiels dans la mesure où ils peuvent être écrits à l'aide de primitives de SCHEME. Voici ces fonctions :

```
STk> (define (cons tête queue)
       (let ((aiguiller (lambda (message)
              (cond ((eq? message 'donne-tête)  tête)
                    ((eq? message 'donne-queue) queue)
                    (else (display "erreur: message inconnu"))))))
       aiguiller))
#unspecified

STk> (define (car liste) (liste 'donne-tête))
#unspecified

STk> (define (cdr liste) (liste 'donne-queue))
#unspecified
```

La valeur retournée par `cons` est la fonction `aiguiller` qui a un paramètre `message`. Cette fonction retourne soit `tête` soit `queue`, en fonction de la valeur du message.

Cette définition des fonctions de manipulation des listes est valide, mais elle est totalement inefficace. Elle permet cependant de montrer que ces fonctions ne sont pas essentielles, ce qui peut être important pour formaliser le langage, et qu'elles sont fonctionnelles.

4.6 Autres expressions usuelles

4.6.1 Comparateurs logiques

Pour comparer des atomes du langage, comme les entiers ou les identificateurs, il existe la fonction `eq?`. De plus, pour comparer des nombres, entiers ou réels, nous disposons de `>`, `>=`, `<=` et `<`.

4.6.2 Comparaison d'expressions complexes

SCHEME propose la fonction `equal?` qui permet de comparer des objets complexes qui peuvent être structurés à l'aide des cellules.

Par exemple :

```
STk> (equal? '(1 2 (3 4)) '(1 2 (3 4))
#t
```

4.6.3 Opérateur booléen d'interrogation

Comme SCHEME possède un système de typage dynamique, où le type des valeurs est attaché aux valeurs elles-mêmes, il est nécessaire de disposer d'un certain nombre de fonction d'interrogation, comme `symbol?` qui retourne vrai si son argument est un identificateur, `number?` qui retourne vrai si son argument est un nombre, `extact?` qui retourne vrai si son argument est un nombre entier, `pair?` qui retourne vrai si l'argument est une cellule et enfin `null?` qui retourne vrai si son argument est la cellule vide `'()`.

4.6.4 Opérateurs booléens

Il existe un certain nombre d'opérateurs booléens, comme `and` et `or` qui peuvent avoir plusieurs arguments, et `not`, à un seul argument. Nous pouvons par exemple définir l'opérateur dyadique `nand` par :

```
(define (nand a b)
  (not (and a b)))
```

4.6.5 Le bloc d'instructions : `begin`

Il est parfois nécessaire de rassembler plusieurs instructions pour former un bloc. Ceci est particulièrement utile avec la construction `if`. L'écriture d'un bloc est :

```
(begin
  expression-1
  expression-2
  ...
  expression-n)
```

À l'intérieur de ce bloc, les expressions sont évaluées en séquence, et le résultat de l'évaluation du bloc est le résultat de la dernière évaluation.

4.6.6 Choix multiple : `cond`

Il est parfois très utile de disposer d'une structure à choix multiple. Bien sûr, une telle construction peut être obtenue à l'aide de plusieurs `if` imbriqués, mais cela est peu commode. Aussi, `cond` a-t-il été défini. Son écriture est :

```
(cond ((condition-1 expressions-1)
       (condition-2 expressions-2)
       ...
```

```
        (condition-n expression-n)))
```

où `expressions-i` représente une séquence d'expressions (l'usage de `begin` n'est donc pas nécessaire). Chacune des `conditions` est évaluée tour à tour. Lorsque le résultat est vrai, l'évaluation de l'`expression` est retournée. Si aucune `condition` n'est vraie, `#undefined` est retournée.

4.6.7 Choix multiple simplifié : `case`

La forme `cond` est parfois trop complexe pour définir certaines sélections. Il existe une forme simplifiée appelée `case`. Sa syntaxe est :

```
(case valeur
  ((liste-de-cas-1) liste-actions-1)
  ((liste-de-cas-2) liste-actions-2)
  ...
  ((liste-de-cas-n) liste-actions-n))
```

On pourrait écrire par exemple :

```
(case index
  ((1 2 3) (display "ok"))
  (4       (display "erreur"))
  (else    (display "indéfini")))
```

4.6.8 Accéder aux éléments d'une liste

En plus des opérateurs `car` et `cdr`, SCHEME offre des extensions de ces opérateurs. Il existe un certain nombre de fonctions de la forme `cadd...r` et `cddd...r`. La première lettre qui suit le `c` indique la dernière action à entreprendre sur la liste. Par exemple `(caddr liste)` est en fait `(car (cdr (cdr liste)))`.

4.6.9 Constructeur de liste : `list`

Pour construire une liste directement, sans passer par une succession de `cons`, SCHEME définit la fonction `list`. Nous avons :

```
STk> (list 1 2 3)
(1 2 3)
```

Il faut faire attention car les éléments de la liste résultante sont le résultat de l'évaluation des arguments de `list`.

4.6.10 Concaténation de deux listes : `append`

SCHEME définit un opérateur qui permet de concaténer deux listes. Par exemple :

```
STk> (append '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)
```

4.6.11 Recherche dans une liste : `member`

Il est possible d'extraire d'une liste la liste dont le premier élément est égal à l'élément donné en argument. La fonction `member` réalise cette opération :

```
STk> (member 3 '(1 2 3 4 5 6))
(3 4 5 6)
```

Si l'élément n'est pas trouvé, la valeur `#f` est retournée, et non la liste vide `'()`. `member` utilise pour tester l'égalité la fonction `equal?`. La fonction `memq` utilise quant à elle `eqv?`.

4.6.12 Accès indexé à un élément d'une liste : `list-ref`

Pour accéder directement à un élément d'une liste par son index, nous pouvons utiliser la fonction **`list-ref`** qui prend une liste et un index qui doit être un entier. Nous avons par exemple :

```
STk> (list-ref '(1 2 3) 1)
2
```

Le premier élément de la liste possède l'index `0`.

4.6.13 Longueur d'une liste : `length`

Avant d'utiliser la fonction **`list-ref`**, il est souvent utile de connaître la longueur de la liste. Cela peut être obtenu avec la fonction **`length`** qui retourne le nombre d'éléments de la liste en argument. Par exemple :

```
STk> (length '(1 2 3))
3
```

4.6.14 Parcourir une liste : `for-each`

Pour effectuer une action sur tous les éléments d'une liste, SCHEME définit la fonction **`for-each`** qui a comme arguments une fonction à un argument et une liste. Nous avons par exemple :

```
STk> (for-each (lambda (arg) (display arg)) '(1 2 3 4))
1234
```

Le même résultat peut être obtenu avec **`(for-each display '(1 2 3 4))`**.

4.6.15 Autre parcours d'une liste : `map`

La fonction **`map`** permet de construire la liste des résultats d'une fonction appliquée à tous les éléments d'une liste. Par exemple :

```
STk> (map (lambda (arg) (+ arg 10)) '(1 2 3 4))
'(11 12 13 14)
```

4.6.16 Nombre variable de paramètres

SCHEME permet de définir des fonctions ayant un nombre variable de paramètres. Dans ce cas, les paramètres supplémentaires sont placés dans une liste. La syntaxe est **`(lambda (par-1 par-2 . liste-par) ...)`**. Par exemple, nous avons :

```
STk> (define f (lambda (a b . l)
                 (display (+ a b))
                 (display l)))
STk> (f 1 2 3 4)
3'(3 4)
```

Il est bien sûr possible de définir une fonction ayant zéro, un ou plusieurs paramètres avec l'écriture **`(lambda (. list-par)...)`**. Cette écriture est équivalente à **`(lambda list-par ...)`**.

4.6.17 Parcours d'une liste avec un `let` nommé

Parfois, la fonction **`for-each`** n'offre pas un contrôle du processus suffisant. Dans ce cas, nous utilisons plutôt le **`let`** nommé. Voici un exemple de son utilisation :

```
(let loop ((liste    '(1 2 3 4 5))
```

```
        (résultat 0))
    (if (null? liste)
      résultat
      (loop (cdr liste)
            (+ résultat (car liste)))))
```

4.6.18 Appliquer un opérateur : `apply`

Enfin, la dernière fonction que nous utiliserons est **`apply`**. Elle permet d'appliquer explicitement une fonction à une liste d'arguments. Nous avons par exemple :

```
STk> (apply + '(1 2 3 4 5))
15
```

qui est possible car **+** est une fonction dont le nombre de paramètres est variable.

4.7 Opérations ensemblistes avec scheme

Dans cette section, nous présentons un jeu de fonctions permettant de réaliser des opérations ensemblistes, comme l'intersection et l'union. Cette bibliothèque permet de mesurer la puissance expressive de SCHEME et elle sera utilisée dans la réalisation de λ-GRAPH dans la troisième partie (§ III-3).

4.7.1 Sélection

Un ensemble est une liste. La première opération consiste à sélectionner des éléments répondant à un critère dans un ensemble :

```
(define (select test set)
  (let loop ((subset set)
             (result '()))
    (if (null? subset)
        result
        (loop (cdr subset)
              (if (test (car subset))
                  (cons (car subset) result)
                  result)))))
```

Il s'agit d'une fonction à deux arguments, une fonction booléenne **`test`** et un ensemble. Elle est bâtie autour d'un **`let`** nommé définissant le reste de l'ensemble **`subset`** et l'ensemble résultat **`result`**. Si le test est positif pour le premier élément du sous-ensemble, l'élément est ajouté au résultat. L'itération se poursuit jusqu'à ce que le sous-ensemble soit vide.

4.7.2 Union

L'union de deux ensembles est l'ensemble constitué du second ensemble et des éléments du premier ensemble qui ne sont pas dans le second. Cela se traduit par :

```
(define (union set-1 set-2)
  (append (select (lambda (elem-1)
                    (not (member elem-1 set-2)))
                  set-1)
          set-2))
```

4.7.3 Intersection

L'intersection de deux ensembles est l'ensemble des éléments du premier appartenant au second. Il vient :

```
(define (inter set-1 set-2)
  (select (lambda (elem-1)
```

```
            (member elem-1 set-2))
          set-1))
```

4.7.4 Ajout d'un élément

On ne peut ajouter un élément dans un ensemble que s'il n'y est pas déjà, ce qui donne :

```
(define (addset elem set)
  (if (member elem set)
      set
      (cons elem set)))
```

4.7.5 Retrait d'un élément

Extraire un élément d'un ensemble revient à sélectionner tous les éléments de l'ensemble qui sont différents de l'élément à extraire :

```
(define (subset elem set)
  (select (lambda (elem-2)
            (not (equal? elem elem-2)))
          set))
```

4.7.6 Sous ensemble

Il est aussi possible de définir un prédicat qui retourne la valeur vraie si un ensemble est un sous-ensemble d'un autre. Nous avons :

```
(define (subset? set-1 set-2)
  (null? (select (lambda (elem-1)
                   (not (member elem-1 set-2)))
                 set-1)))
```

4.8 Programmation fonctionnelle avec scheme

Une fonction ne crée pas d'effet de bord. Un effet de bord est une modification de l'environnement dans laquelle la fonction est définie. Par exemple :

```
STk> (define compteur 0)
#unspecifed

STk> (define (compte)
       (set! compteur (+ 1 compteur))
        compteur)
#unspecified
```

définit une fonction `compte` qui compte le nombre de fois qu'elle est invoquée en créant un effet de bord. Ici, l'effet de bord modifie la variable global `compteur` à chaque appel. Pour connaître la valeur de retour d'un appel à `compte`, il est nécessaire de connaître la valeur de la variable globale `compteur`. Elle n'est donc pas une fonction.

On peut transformer `compte` en une définition fonctionnelle en lui ajoutant un environnement global comme paramètre supplémentaire, et en lui ajoutant l'environnement modifié comme valeur de retour supplémentaire :

```
STk> (define (compte env)
       (let ((compteur (valueOf 'compteur env)))
         (cons (+ 1 compteur)
               (set 'compteur compteur env))))
#unspecified
```

La différence fondamentale entre les deux définitions de `compte` est que la première est impérative alors que la seconde est fonctionnelle.

On peut facilement transformer la première version de `compte` en la seconde, de manière automatique. Il suffit d'ajouter à toutes les fonctions l'environnement courant comme paramètre, et de remplacer les accès à l'environnement par des appels de fonctions. De plus, les fonctions retourneront l'environnement comme valeur de retour supplémentaire.

D'ailleurs, cette transformation est faite dans la machine abstraite associée à SCHEME [18]. Cette machine est décrite à l'aide de la sémantique dénotationnelle qui est une forme de λ-calcul typé permettant de spécifier les langages de programmation. Cette machine abstraite est parfaitement fonctionnelle, et elle décrit tout aussi parfaitement le fonctionnement de `set!` [3].

Quel est donc l'avantage de la programmation fonctionnelle?

A cette question, il semble que la réponse ne puisse être aussi évidente que l'on pourrait s'y attendre, puisque tout programme impératif peut être transformé de manière automatique en un programme fonctionnel.

Il est cependant clair que pour effectuer des vérifications formelles sur un programme, il est très utile qu'il soit fonctionnel, ainsi que la machine abstraite qui va le résoudre. S'il ne l'est pas, il faudra au préalable le transformer en un programme fonctionnel. Pour prouver un programme, il est nécessaire de le traduire préalablement en notations mathématiques ou de traiter le programme comme un objet statique auquel des assertions mathématiques sont attachées.

Quoi qu'il en soit, une programmation fonctionnelle avec SCHEME utilise un sous ensemble de ce langage dans lequel l'instruction `set!` est absente, ainsi que toutes les instructions l'utilisant. Par exemple, `define` sur des valeurs déjà définies, `letrec` et toutes les fonctions se terminant par '`!`' sont supprimées.

4.9 Synthèse de scheme

L'étude du langage SCHEME est riche en enseignements. Il peut être défini comme un λ-calcul en mode applicatif. Il permet des constructions impératives comme l'affectation. Il met en évidence que la suprématie des langages fonctionnels n'est pas aussi franche que l'on pouvait s'y attendre.

Cependant, ce langage ne peut être utilisé dans le cadre de notre étude pour modéliser les applications à réaliser sur l'architecture parallèle statique. En effet, SCHEME est trop riche. De plus, il permet à l'utilisateur d'abstraire des expressions, ce qui donne la possibilité de construire des programmes indéterministes en temps et en ressources. Les limitations qu'il faudrait apporter au langage le dénatureraient trop pour qu'il soit encore appelé SCHEME.

Mais certains concepts sont à garder en mémoire. Le typage implicite des données, basé sur leur syntaxe, est une caractéristique intéressante car elle simplifie grandement le langage. En effet, lorsque l'utilisateur écrit 123, il sait que c'est un entier. L'obligation de préciser sans cesse le type des données, comme dans les langages de la famille de PASCAL, est très contraignante [4].

Dans certains cas, il serait cependant utile de pouvoir préciser le types des expressions, notamment dans l'écriture des abstractions. Nous aimerions pouvoir écrire :

```
(lambda (a:integer b:real) ...)
```

Ceci permettrait de s'affranchir de contrôler sans cesse les données des fonctions.

De plus, SCHEME n'effectue le contrôle des types qu'au moment de l'exécution des expressions. Une partie incohérente d'un programme peut donc exister à l'insu du programmeur tant qu'elle n'est pas exécutée. Cette caractéristique est un effet pervers du typage dynamique adopté par SCHEME, et qui est incompatible avec une programmation sûre.

3. L'autre caractéristique principale des langages impératifs est l'existence d'instructions de branchement de type `goto`. Là encore, la sémantique dénotationnelle, outil fonctionnel par excellence, permet de modéliser ce comportement [59, 70].

4. Le système de vérification des types des langages recents, comme ML est beaucoup plus puissant et souple à l'utilisation.

Dans ce qui suit, SCHEME va servir à réaliser la machine abstraite de notre langage décrite à l'aide de définitions formelles. Leur lecture s'en trouvera grandement facilitée.

Chapitre 5

Les langages à flots de données

Dans cette section, nous présentons les principaux langages dataflow synchrones actuels, c'est à dire LUCID, VAL, SISAL, LUSTRE et SIGNAL. Pour chaque langage, nous tenterons d'apporter une critique par rapport à l'utilisation que nous voulons en faire, c'est à dire pour programmer l'architecture parallèle statique.

5.1 Le langage lucid

LUCID fut l'un des premiers langages dataflow conçu [10, 12, 17, 72]. Il est basé sur le langage abstrait de LANDIN [56], appelé ISWIM[1] et le langage LUSWIM.

LUCID fut le premier à montrer comment remplacer les itérations des langages impératifs par des définitions de flots de données. C'est un langage non procédural, c'est à dire qu'il ne contient aucune caractéristique impérative, comme l'affectation.

5.1.1 Éléments de base

LUCID est un langage abstrait dans le sens où il ne spécifie pas les données manipulées et les opérateurs qui s'y rapportent. L'interprète LUCID propose quant à lui les types de données classiques, comme les caractères, les chaînes de caractères, les entiers et les réels. Tous les opérateurs standards se rapportant à ces types de données sont aussi définis. Par la suite, nous appelerons « algèbres » un type de donnée élémentaire est l'ensemble des opérateurs qui s'y rapporte.

LUCID définit une donnée particulière pour chaque algèbre, appelée `bottom`.[2] Cette donnée permet d'obtenir une relation d'ordre complète sur les éléments qui composent une algèbre.

5.1.2 Construction `where`

Ce type de construction provient du langage ISWIM. A l'époque, cette construction a été perçue comme une grande avancée des langages fonctionnels. En effet, LISP était le principal langage fonctionnel, mais son manque de lisibilité du fait de l'usage intensif des parenthèses lui était reproché.

La construction `where` s'utilise comme suit :

```
fact(6)
  where
    fact(x) = if n < 2
                then 1
```

1. ISWIM est l'acronyme de *If you See What I Mean*, qui signifie *si vous voyez ce que je veux dire.*

2. L'existence de cette donnée indique le courant de pensée qui a donné naissance à LUCID. C'est ce même courant qui a engendré les langages fonctionnels et la sémantique dénotationnelle [59, 60, 70]. Cette donnée est nécessaire pour obtenir des ensembles de données ordonnés, et ainsi rendre les définitions récursives du langage mathématiquement définissables.

```
              else n * fact (n - 1)
            fi;
  end
```

On remarquera que la construction `where` permet des définitions récursives (`letrec` en SCHEME). Nous nous souvenons que ce type de constructions est dangereux car il met en péril la propriété fonctionnelle des langages et engendre des processus indéterministes en temps.

Le langage LUCID propose l'alternative `if ... then ... else ... fi` qui s'utilise comme dans les langages classiques.

5.1.3 Fonctions

LUCID est un langage de premier ordre, c'est à dire que les fonctions ne peuvent pas être des arguments de fonctions. Ceci n'est pas très gênant d'un point de vue programmation et la limitation peut toujours être contournée.

Il est basé sur une évaluation paresseuse, c'est à dire que les expressions ne sont évaluées que lorsque cela est strictement nécessaire, comme pour la condition d'une alternative. Considérons l'expressions suivante :

```
f(3,4)
  where
    f(X,Y) = if X<1
               then 0
               else f(X-1, f(X, Y+1))
             fi;
  end
```

Avec un mode d'évaluation applicatif, comme celui de SCHEME, cette évaluation ne se termine jamais, alors que l'interprète LUCID, qui utilise une évaluation paresseuse, retournera `0` comme résultat.

Il donc possible de définir sa propre alternative `cond`, comme dans :

```
fact(3)
  where
    fact(n) = cond(n<1, 1, n=fact(n-1))
    where
      cond(p,x,y) = if p then x else y;
    end;
  end
```

Ceci est impossible dans les langages en mode applicatif, comme C ou SCHEME, car l'invocation de `cond` aurait préalablement évalué ses arguments, ce qui engendre un processus infini.

L'évaluation paresseuse de LUCID peut être rapprochée des graphes dataflow pilotés par la demande (§ I-2).

5.1.4 Environnement et liaison des variables

LUCID utilise un mode statique de liaison des variables, c'est à dire que les variables libres des expressions sont liées où l'expression est *définie*, et non pas où l'expression est *utilisée*.

Considérons :

```
f(3, U)
  where
    B = 7;
    C = 5;
    f(X,Y) = X + (B * Y) + C;
    U = P - Q
      where
        Q = 2;
        B = 3;
        P = f(B,Q);
      end;
  end
```

La définition de la fonction `f` possède deux variables libres `B` et `C`. La variable `B` lors du deuxième appel est liée à sa première définition. le résultat de cette expression est `148` car `B` vaut toujours `7`.

5.1.5 Flot et itération

Les caractéristiques de LUCID décrites jusqu'à maintenant ne lui sont pas propres. Nous les retrouvons plus ou moins dans tous les langages fonctionnels. Ce qui suit constitue sa caractéristique essentielle.

LUCID fut conçu pour montrer qu'une itération dans un langage impératif peut être aisément transformée en un flot de données. L'avantage d'une telle transformation est que la mathématisation du programme est grandement facilitée, et donc que les preuves de programmes sont presque évidentes [12].

Un flot est défini à l'aide de la construction `fby` pour *followed by*[3]. Pour définir cet opérateur, considérons les flots $x = \langle x_0, x_1, \ldots \rangle$ et $y = \langle y_0, y_1, \ldots \rangle$. Nous avons alors :

$$x \; fby \; y = \langle x_0, y_0, y_1 \ldots \rangle \tag{5.1}$$

On peut alors définir le flot :

```
i
  where
    i = 1 fby 2 * i + 1
```

qui crée la suite de valeur $i = \langle 1, 3, 7, 15, \ldots \rangle$. Les définitions de flots peuvent être cycliques[4]. Par exemple :

```
f
  where
    f = 1 fby f + g;
    g = 0 fby f;
  end
```

construit les flots d'entiers :

$$\begin{array}{rcl} f & = & \langle \; 1, \; 1, \; 2, \; 3, \; 5, \; \ldots \rangle \\ g & = & \langle \; 0, \; 1, \; 1, \; 2, \; 3, \; \ldots \rangle \end{array} \tag{5.2}$$

La traduction d'un tel programme dans un langage impératif n'est pas évidente. Par exemple, les deux premiers programmes suivants sont inexacts, alors que le troisième est exact :

incorrect

```
main() {
 int f = 1, g = 0;
 while (1) {
  printf ("%d ", f);
  f = f + g;
  g = f;
 }
}
```

incorrect

```
main() {
 int f = 1, g = 0;
 while (1) {
  printf ("%d ", f);
  g = f;
  f = f + g;
 }
}
```

correct

```
main() {
 int f = 1, g = 0,
     new_f, new_g;
 while (1) {
  printf ("%d ", f);
  new_f = f + g;
  new_g = f;
  f = new_f;
  g = new_g;
 }
}
```

3. Pour etre plus précis, toute expression dans LUCID est un flot. Par exemple, `1` est le flot $\langle 1, 1, \ldots \rangle$ et `1 + 2` est le flot $\langle 3, 3, \ldots \rangle$.

4. On dira par la suite qu'elles définissent des équations récurrentes.

Les variables `f` et `g` sont parfois appelées variables d'itération.

La définition d'un flot peut aussi contenir des variables instantanées. Ces variables sont évaluées avec les valeurs instantanées des autres variables.

Considérons :

```
v
  where
    v = 1 fby v + w;
    w = v + 1;
  end
```

La variable `w` est calculée avec la valeur instantanée de `v`, qui donne les flots :

$$\begin{array}{lcl} v & = & \langle\ 1,\ 3,\ 7,\ \ldots\rangle \\ w & = & \langle\ 2,\ 4,\ 8,\ \ldots\rangle \end{array} \tag{5.3}$$

5.1.6 Décomposition de `fby`

L'opérateur **`fby`** de LUCID est en fait un synonyme de deux opérateurs plus primitifs, **`first`** et **`next`**.

L'écriture :

```
i = 1 fby i + 1
```

peut être remplacée par :

```
first i = 1;
next  i = i + 1;
```

Si le flot x est défini par $x = \langle x_0, x_1, \ldots\rangle$, ces deux opérateurs peuvent être formalisés par :

$$\begin{array}{lcl} first\ x & = & \langle\ x_0,\ x_0,\ \ldots\rangle \\ next\ x & = & \langle\ x_1,\ x_2,\ \ldots\rangle \end{array} \tag{5.4}$$

Ils peuvent être utilisés à droite ou à gauche du signe =. Ils ont longtemps fait penser que LUCID était finalement un langage impératif dont ces deux opérateurs représentaient l'affectation. Il n'en est rien, car l'expression **`next`** ne peut être utilisée qu'une seule fois pour définir le futur d'une variable.

Pour comprendre leur utilité, considérons le programme suivant qui effectue la somme des factorielles des entiers naturels :

```
s
  where
    i = 1 fby i + 1;
    f = 1 fby i * (i + 1);
    s = 1 fby s + f * (i + 1);
  end
```

L'exécution de ce programme produit les flots :

$$\begin{array}{lcl} i & = & \langle\ 1,\ 2,\ 3,\ 4,\ 5,\ \ldots\rangle \\ f & = & \langle\ 1,\ 2,\ 6,\ 24,\ 120,\ \ldots\rangle \\ s & = & \langle\ 1,\ 3,\ 9,\ 33,\ 153,\ \ldots\rangle \end{array} \tag{5.5}$$

On remarque que l'expression du futur de `f` est exprimée en fonction du futur de `i`, c'est à dire `i+1` et non `i`. Cette remarque est aussi vraie pour `s`. L'opérateur **`next`** permet de s'affranchir de cette contrainte :

```
s
  where
    i = 1 fby i + 1;
    f = 1 fby i * next i;
```

```
    s = 1 fby s + next f;
  end
```

La traduction en C de ce programme est :

```
main() {
  int i = 1, f = 1, s = 1;
  while (1) {
    printf ("%d ", s);
    i = i + 1;
    f = f * i;
    s = s + f;
  }
}
```

L'opérateur `next` complique la traduction d'un programme LUCID en un langage impératif. Les auteurs de LUCID font remarquer que l'opérateur `next` peut être réalisé simplement de manière syntaxique en remplaçant l'expression par le futur du flot concerné. Par exemple :

```
s
  where
    i = 1 fby next-i;
    f = 1 fby next-f;
    s = 1 fby s + next-f;
    where
      next-i = i + 1;
      next-f = i * next-i;
    end;
  end
```

Ceci peut être réalisé car les variables instantanées sont mises à jour avec les valeurs instantanées des flots.

De la même manière, nous pouvons aisément simuler l'expression `first x` à l'aide d'une constante.

5.1.7 Synthèse de lucid

LUCID est l'un des premiers langages dataflow. Ces auteurs ont utilisé des outils modernes pour décrire sa sémantique, comme la sémantique dénotationnelle. Ils ont étudié les possibilités de ce langage dans les moindre retranchements, en gardant comme contrainte de ne pas ajouter au langage de caractéristiques impératives, comme ce fut le cas pour LISP.

Cependant, dans le souci de le comparer aux langages impératifs, LUCID s'est vu ajouter des possibilités non essentielles, comme les instructions `whenever` ou `upon`. LUCID permet de plus de créer des itérations imbriquées et des fonctions récursives, qui sont indéterministes en temps.

Le langage abstrait développé dans la thèse peut être vu comme un sous ensemble de LUCID, dans lequel les possibilités indéterministes en temps ont été supprimées, comme les définitions de fonctions récursives et la possibilité de créer des itérations au sein d'un programme. Cependant, leurs sémantiques sont très semblables, notament en ce qui concerne la gestion des environnements.

5.2 Le langage val

VAL est un langage dataflow développé au MIT par, entre autres, J. DENNIS [2, 32]. Ce langage fut conçu à l'origine pour fonctionner sur une architecture dataflow statique [33, 34, 35]. Il ne supporte donc aucune caractéristique des graphes dataflow dynamiques, comme la récursion. VAL a été méticuleusement défini à l'aide de la sémantique axiomatique et avec la sémantique dénotationnelle. VAL est fonctionnel par nature, donc les effets de bord y sont absents. Cependant, VAL est fortement typé et il ressemble intentionnellement aux langages conventionnels. En fait, il ressemble beaucoup à ML. Par exemple, la fonction VAL suivante calcule la moyenne et l'écart type de ses paramètres :

```
function stats (X,Y,Z: real): real, real;
```

```
  let
    Mean: real := (X + Y + Z) / 3;
    SD  : real :=   sqrt ((X - Mean) ** 2
                  + (Y - Mean) ** 2
                  + (Z - Mean) ** 2) / 3;
  in
    Mean, SD
  end
end
```

Dans VAL, les fonctions retournent plusieurs valeurs, ici `Mean` et `SD`. Cette syntaxe simplifie la composition des fonctions car les valeurs retournées peuvent immédiatement être utilisées comme arguments.

VAL possède les types de données standards, comme les entiers, les réels, les booléens et les caractères. De plus, il possède le type tableau, le type choix et la structure. VAL propose beaucoup d'opérations sur les tableaux, comme la concaténation, l'extension et la contraction. Lorsqu'un tableau ou une structure sont construits, toutes les valeurs sont évaluées simultanément, autorisant leur parallélisation.

Il semble que VAL définisse l'affectation, mais ceci est faux : les identificateurs ne peuvent être associés qu'à une seule valeur dans un bloc. Cela peut poser un problème lors d'une itération où des valeurs peuvent être modifiées entre deux itérations. VAL crée de nouvelles associations à chaque itération, où les valeurs sont soit recopiées soit calculées à partir des associations de l'itération précédente.

Les valeurs structurées, comme les tableaux, sont manipulées comme des valeurs uniques. Leurs composantes ne peuvent donc pas être modifiées individuellement. Pour modifier l'une des composantes d'un tableau, il est nécessaire de créer un nouveau tableau différent de l'original seulement par la composante modifiée.

5.2.1 Parallélisme implicite

VAL exprime le parallélisme implicite. Les opérations qui peuvent être exécutées séparément sont parallèles, sans la nécessité d'utiliser une notation supplémentaire. VAL exploite le parallélisme implicite car il est fonctionnel et que les expressions ne peuvent avoir d'effets de bord. Si deux expressions sont indépendantes, elles peuvent être exécutées de manière parallèle. Une source d'effet de bord dans les langages conventionnels provient de l'utilisation d'alias : une même zone de la mémoire possède plus d'une référence. Le passage des arguments par référence et les pointeurs peuvent créer des alias. Les alias ne font pas partie de VAL.

Le fait que VAL exploite le parallélisme implicite est justifié par le fait que ce parallélisme est souvent à grain très fin, au niveau des opérations arithmétiques. Il est exclu d'imposer à l'utilisateur de spécifier le parallélisme à ce niveau.

5.2.2 Parallélisme explicite : `forall`

En plus du parallélisme implicite, VAL permet d'exploiter le parallélisme de manière explicite, avec l'expression `forall`. Cette construction répète une opération pour un certain nombre de valeurs. La largeur de ce parallélisme dépend des valeurs que peut prendre un identificateur de contrôle dans un intervalle donné. VAL fournit deux moyens pour manipuler les résultats de chaque itération. L'opérateur `construct` permet de construire un tableau contenant les résultats de chaque itération et `accumulate` qui permet d'appliquer un opérateur sur tous les résultats. Nous avons par exemple :

```
forall i in [1,100] do
  left   : real    := point[i].x_low;
  bottom : real    := point[i].y_low;
  right  : real    := point[i].x_high;
  top    : real    := point[i].y_high;

  area   : real    := (right - left) * (top - bottom);
  okay   : Boolean := acceptable (area);
  abort  : Boolean := erroneous (area);
```

```
  accumulate +  if okay then area else 0.0  end;
  accumulate or abort;
  construct  if okay then area else 0.0 end;
end
```

Ce fragment de programme est exécuté avec 100 flots d'exécution parallèles. Leurs résultats sont additionnés avec `accumulate +` s'ils sont acceptables. La variable `abort` indique si l'un au moins des résultats n'est pas acceptable. Enfin, ces résultats sont utilisés pour construire un tableau. Le résultat de ce fragment de programme est le triplet des trois valeurs.

5.2.3 Séquence : `for`

L'instruction `for` réalise une itération qui ne peut être exécutée en parallèle car les valeurs produites par une itération sont directement utilisées par l'itération suivante. La décision de continuer l'itération est prise dans une alternative :

```
for a:real := 0.0;
    b:real := 1.0; do
  let
    c:real,
    done:Boolean := Compute (a, b);
  in
    if done then c
    else
      iter
        a := NewA (a, b);
        b := NewB (a, b);
      end
    end
  end
end
```

Les identificateurs `a` et `b` sont les paramètres de l'itération. Pour la première itération, ils ont la valeurs `0.0` et `1.0`. La fonction `Compute` retourne deux valeurs qui sont placées dans les variables `c` et `done`. Si la variable `done` est fausse, une nouvelle itération commence avec `iter`. Les nouvelles valeurs de `a` et `b` sont calculées pour cette nouvelle itération avec les anciennes valeurs de `a` et `b`. Lorsque `done` est vraie, l'expression retourne la valeur de la variable `c`.

5.2.4 Type choix : `choice`

Un type choix est construit avec :

```
type list =
  choice [
    empty : void;
    noempty : recors [ item:real; rest:list ];
  ]
```

`void` est un type pré-défini indiquant l'absence de valeur. Cet exemple définit `list` comme une liste liée ordinaire, avec une différence intéressante : elle est définie de manière récursive, sans utiliser la notion de pointeur. En effet, VAL ne définit pas les pointeurs car ils permettent la définition d'alias qui supprime le caractère fonctionnel des langages. Une liste est donc une structure récursive plutôt qu'une séquence d'éléments individuels liés entre eux.

Une valeur d'un type choix est créée en utilisant le constructeur `make`, comme :

```
make list[empty : nil]
```

Pour garantir la compatibilité des types, le contenu d'un choix n'est accessible qu'avec l'opérateur `tagcase`, comme :

```
function IsEmpty (L: list): Boolean;
  tagcase L of
    when tag empty   => true
```

```
        when tag noempty => false
      end
    end
```

VAL propose aussi des constructions conditionnelles. Comme les effets de bord sont absents de VAL, le langage permet l'évaluation spéculative[5]. Cependant, dans certains cas, VAL n'utilise pas l'évaluation spéculative pour éviter les évaluations inutiles. En particulier, dans les expression `if`, `tagcase` et `for`, les évaluations ne sont commencées que lorsque l'expression de contrôle est évaluée (la condition d'une alternative, par exemple). Dans ce cas, VAL utilise une méthode d'évaluation paresseuse. Par contre, les composantes des expressions ordinaires sont supposées être pilotées par les données, et elles sont évaluées de manière spéculative. De même, les paramètres des fonctions sont toujours pré-évalués avant que la fonction ne soit invoquée.

5.2.5 Synthèse de val

VAL fut l'une des premières tentatives de définition d'un langage dataflow. Il possède une sémantique bien définie et peut être exécuté sur une machine réelle [33, 34, 35].

Le modèle de l'architecture repose sur un parallélisme dynamique activé par un mécanisme de messages [33]. L'architecture possède une liste des tâches restantes. Chacune de ces tâches génère des tâches qui s'ajoute à la liste. Cette exploitation du parallélisme s'est avérée peu efficace car les temps de gestion des messages sont trop importants par rapport à la granularité de ces tâches [7, 8, 19, 20].

Le fait que le parallélisme est exploitable dynamiquement se retrouve dans la définition du langage. VAL propose un opérateur d'itération pour exploiter dynamiquement le parallélisme. Le nombre d'itérations peut être constant ou variable. Dans le second cas, il peut dépendre d'une entrée principale du programme, et donc être inconnu à l'avance. Ceci n'est pas gênant, parceque l'architecture dataflow repose sur une gestion dynamique du parallélisme.

De plus, VAL permet de créer dynamiquement des structures de données, ce qui implique une gestion dynamique de la mémoire.

Notre modèle d'architecture repose sur une exploitation statique du parallélisme, ce qui contraint les programmes générés à être déterministes en temps et en ressources. Le langage VAL ne convient donc pas pour le modèle d'architecture proposé dans cette thèse.

5.3 Le langage sisal

Le langage SISAL [16] est basé sur VAL, vu dans la section précédente. C'est l'un des projets les plus avancés en matière d'exploitation du parallélisme d'un langage dataflow.

L'acronyme SISAL signifie *Streams and Iteration in a Single Assignement Language*. C'est un langage applicatif à usage général défini pour des architectures classiques ou des architectures parallèles spécifiques. Il est fortement typé et il utilise une syntaxe proche de celle de PASCAL, pour faciliter la compréhension des programmeurs.

Il possède des caractéristiques sémantiques importantes. Premièrement, les fonctions sont mathématiquement correctes et n'ont pas d'effets de bord. Deuxièmement, les programmes SISAL sont transparents par référence. Comme les identificateurs sont liés à des valeurs et non à des emplacements en mémoire, il n'y a pas d'effets de synonymie, plus connus sous l'anglicisme *aliasing*. Troisièmement, SISAL est un langage à affectation unique, ce qui signifie qu'un identificateur n'est associé à une valeur qu'une seule fois. En général, un programme SISAL est un ensemble d'équations mathématiques. Ces propriétés permettent de transformer un programme SISAL en un graphe dataflow de manière triviale.

5. Les deux clauses d'une alternative sont évaluées, et un seul des deux résultats sera utilisé, en fonction de la condition.

5.3.1 Fonctions

Une fonction peut avoir plusieurs paramètres et plusieurs valeurs de retour. Le type des paramètres est déclaré dans l'entête de la fonction. Les seules variables utilisables dans le corps de la fonction sont les paramètres et les variables locales. SISAL ne permet pas la définition de variables globales.

5.3.2 Types de données

SISAL définit les types de données standards comme les booléens, les caractères, les entiers et les réels à simple ou double précision. Il définit aussi des types de données composées, comme les tableaux, les structures, les unions et les flots.

5.3.3 Tableau

Les opérations sur les tableaux sont les suivantes :

Opérations	Commentaires
`array [1: 1, 2, 3]`	création
`array_fill (1, N, 0)`	création d'un tableau de `N` zéros
`A[1: 0]`	remplace `A[1]` par `0`
`A || B`	concatène `A` et `B`

5.3.4 Flot

Un flot est une séquence de valeurs de même type. Ces valeurs peuvent être lues de manière séquentielle seulement. Les opérations possibles sur les flots sont les suivantes :

Opérations	Commentaires
`stream_append (A, B)`	ajoute `B` à `A`
`stream_first (A)`	premier élément de `A`
`stream_rest (A)`	tous les éléments suivants de `A`
`stream_empty (A)`	le flot `A` est-il vide?
`A || B`	concatène `A` et `B`

Un flot ne peut avoir qu'un seul producteur et plusieurs consommateurs. Par définition, les flots de SISAL sont non stricts, ce qui signifie que les éléments sont disponibles dès qu'ils sont produits. L'environnement d'exécution doit permettre l'exécution parallèle du producteur et des consommateurs. Ainsi, les flots mettent-ils en œuvre le parallélisme *pipeline*.

5.3.5 Itération parallèle : `for ... initial`

SISAL permet de réaliser des itérations séquentielles et des itérations parallèles. L'expression `for initial` ressemble à une itération des langages impératifs, mais utilise en fait la sémantique de l'affectation unique. Par exemple :

```
for initial
  i := 1;
  x := Y[1];
while i < n repeat
  i := old i + 1;
  x := old x + Y[1];
returns array of x
end for
```

Cette expression possède quatre champs : l'initialisation, l'itération, les tests de terminaison et la clause résultat. L'initialisation définit toutes les variables d'itération et leur donne

leur valeur initiale. Le corps de l'itération calcule les valeurs suivantes des variables d'itération. Ce calcul peut accéder aux valeurs des variables d'itération de l'itération précédente, à l'aide du préfixe `old`. Le test de la terminaison peut apparaître avant ou après le corps de l'itération. La clause résultat définit le résultat de l'expression et son arité. Les résultats sont soit la valeur finale d'une variable d'itération, soit une réduction de toutes les valeurs de chaque itération d'une variable d'itération. SISAL propose sept types de réductions : `array of`, `stream of`, `catenate`, `sum`, `product`, `least` ou `greatest`.

5.3.6 Itération séquentielle : `for`

La construction `for` est utilisée comme suit :

```
for i in 1, N
  x := A[i] * B[i]
returns value of sum x
end for
```

Elle permet de définir des itérations séquentielles. Sa sémantique interdit les références aux valeurs des variables d'itération des autres itérations. L'expression `for` comporte trois parties : le générateur de l'index, le corps de l'itération et la clause résultat. Le générateur d'index peut avoir les formes suivantes :

Opérations	Commentaires
`for x in A`	dispersé
`for i in 1, n`	séquence
`for x in A dot y in B`	composition `dot` de deux index dispersés
`for x in 1, n cross j in 1, m`	composition `cross` de deux séquences

Une itération est exécutée pour chaque valeur de l'index. La clause résultat définit les valeurs et l'arité du résultat de l'expression. Chaque valeur de ce résultat est une réduction des valeurs à chaque itération d'une des variables d'itération.

5.3.7 Synthèse de sisal

SISAL est un langage basé sur VAL. Il permet d'exploiter le parallélisme des itérations. Il est fonctionnel et possède un certain nombre de caractéristiques des langages fonctionnels.

Le principal reproche que l'on peut faire à SISAL est l'usage d'une syntaxe très obscure où certaines expressions ne sont utilisables que de manière très localisée. De plus, certaines déclarations prennent des formes différentes selon le contexte où elles sont utilisées, comme par exemple, la déclaration des types.

En ce qui concerne l'architecture parallèle statique, SISAL ne convient pas pour les mêmes raisons que VAL, car il est basé sur un parallélisme dynamique.

5.4 Le langage lustre

Cette introduction au langage LUSTRE est extraite d'un certain nombre de documents le concernant [41, 42, 43].

Le langage LUSTRE est un langage de spécification adapté aux phases amont de la conception et de la synthèse d'architectures, car il possède les qualités requises : clarté, concision, puissance expressive, abstraction et transparence par référence. Il s'agit historiquement d'un formalisme utilisé par les automaticiens : son expression est donc naturelle, et elle peut être graphique.

LUSTRE est relativement proche du langage SIGNAL (dont nous parlerons pas ici, du fait de cette ressemblance), et il a reçu une sémantique formelle. Ces bonnes propriétés formelles ont permis la définition d'outils de vérifications des propriétés fonctionnelles et temporelles

des programmes. L'outil de transformation TRANSE est basé également sur des propriétés mathématiques du langage.

Le langage LUSTRE est initialement conçu pour concevoir des systèmes réactifs. Nous désignons sous le nom de systèmes réactifs les systèmes dont le rôle est de réagir de manière continue à leur environnement physique, à une vitesse déterminée par cet environnement. Cette classe de systèmes est à comparer aux systèmes *transformationnels* disposant de leurs entrées à l'initialisation et délivrant leurs résultats à leur terminaison, et aux systèmes interactifs qui réagissent de manière continue et à leur vitesse à leur environnement.

5.4.1 Présentation du langage

Flots de données et horloges En LUSTRE, toute variable ou expression désigne un flot, c'est à dire un couple formé d'une suite éventuellement vide de valeurs, et d'une horloge représentant une suite d'instants. Un flot possède la n-ième valeur de sa suite de valeurs au n-ième instant de son horloge.

Tout programme a un comportement cyclique qui définit son horloge de base, celle à partir de laquelle toutes les autres horloges sont dérivées. L'unité de l'horloge de base est donc le cycle d'exécution du programme.

Équations Les variables de LUSTRE représentent donc des flots de données. Chaque variable est déclarée avec un type. Le type est soit un type de base (LUSTRE définit les entiers, les booléens et les réels, et il permet d'importer des types externes). Une variable qui n'est pas une entrée du programme est définie une seule fois par une équation. Une équation est de la forme `x=e` où `x` est un nom de la variable et `e` une expression. LUSTRE est transparent par référence, c'est à dire que `x` et `e` sont identiques et peuvent être mutuellement inter-changés. De plus, `x` est entièrement défini par e, c'est à dire que le comportement de `e` ne dépend pas de son environnement d'utilisation.

Les opérateurs primitifs de LUSTRE se rapportent aux types de base: ce sont les opérateurs arithmétiques et booléens, ainsi que les opérateurs externes relatifs aux types importés. Ces opérateurs ne peuvent opérer que sur des données de même horloge. Les constantes représentent des flots de valeurs constantes soumises à l'horloge de base.

Opérateurs temporels LUSTRE définit quatre opérateurs pour gérer le temps dans un programme. Ils opèrent sur des flots.

L'opérateur `pre` (pour *précédent*) mémorise la valeur d'une expression à l'instant précédent de son horloge: si $\langle e_1, e_2, \ldots, e_n, \ldots \rangle$ est la suite des valeurs d'une expressions `E`, alors `pre(E)` est une expression de même horloge, et dont la suite des valeurs est $\langle nil, e_1, e_2, \ldots, e_{n-1}, \ldots \rangle$, où *nil* représente une valeur indéfinie, correspondant à l'absence d'initialisation.

L'opérateur `->` qui se lit *suivi de* sert à définir une valeur initiale: si `E` et `F` sont des expressions ayant même horloge, et dont les suites de valeurs sont $\langle e_1, e_2, \ldots, e_n, \ldots \rangle$ et $\langle f_1, f_2, \ldots, f_n, \ldots \rangle$, alors `E -> F` est une expression de même horloge que `E` et `F`, et dont la suite de valeurs est $\langle e_1, f_2, \ldots, f_n, \ldots \rangle$. Autrement dit, `E -> F` est toujours égale à `F`, sauf à l'instant initial de son horloge.

L'opérateur `when` sert à sous-échantillonner une expression avec une horloge plus lente: si `E` est une expression, et `B` une expression booléenne sur la même horloge que `E`, alors `E when B` est une expression sur l'horloge définie par `B`, et dont la suite des valeurs est extraite de celle de `E` en ne conservant que les valeurs dont l'indice correspond à `vrai` dans la suite des valeurs de `B`. Autrement dit, il s'agit de la suite des valeurs de `E` quand `B` est `vrai`.

L'opérateur `current` est utilisé pour sur-échantillonner une expression avec une horloge plus rapide. Si `E` est une expression ayant une horloge différente de l'horloge de base, et `B` l'expression booléenne ayant généré cette horloge, alors `current E` est une expression ayant même horloge que `B`, et dont la valeur à chaque instant est la valeur prise par `E` au dernier instant où `B` valait `vrai`.

Assertion Le corps d'un programme LUSTRE est composé d'un système d'équations et d'assertions. Les assertions sont une généralisation des équations : elles consistent en des expressions booléennes qui sont supposées toujours vraies. Ce sont des indications données au compilateur pour optimiser le code produit. Les assertions jouent un rôle capital dans la vérification des programmes.

Structuration des programmes De manière à structurer une application, LUSTRE définit les nœuds. Un nœud est le paquetage d'un ensemble d'équations, avec un nom, des entrées et des sorties.

Un nœud est donc défini par une interface donnant la liste des paramètres en entrée, avec leur type et éventuellement leur horloge, la liste des valeurs de retour associées à leur type, et l'ensemble des équations définissant ces valeurs de retour. Par exemple, la définition suivante définit un compteur général, paramétré par une valeur initiale, une valeur d'incrément et un événement de ré-initialisation :

```
node Compteur (init, incrément : int, raz : bool)
     returns (n : int);
  let
    n = init -> if raz
                then init
                else pre (n) + incrément;
  tel.
```

Un tel nœud peut être instancié à la manière d'une fonction dans n'importe qu'elle expression. Nous pouvons écrire :

```
Pair    = Compteur (0, 2, false);
Modulo5 = Compteur (0 ,1, pre (Modulo5) = 4);
```

qui définissent respectivement la suite des nombres pairs et la suite cyclique des entiers modulo 5.

En ce qui concerne les horloges, et conformément au point de vue «flot de données», l'horloge de base d'un nœud est déterminée par l'horloge de ses paramètres.

Un nœud peut retourner plusieurs paramètres ; dans ce cas, le résultat est un tuple. Nous pouvons donc facilement concevoir un additionneur à 1 bit :

```
node Add1 (a, b, carryIn : bool)
  returns (result, carryOut : bool);
let
  result   = a xor b xor carryIn;
  carryOut =    (a and b)
             or (b and carryIn)
             or (a and carryIn);
tel.
```

Pour utiliser cet additionneur à un bit, nous pourrions écrire :

```
(res, cary) = add1 (e1, e2, 0);
```

où `(res, carry)` sont deux variables locales qui valent le résultat de `result` et `carryOut` de l'instanciation de `Add1`.

5.4.2 Le compilateur lustre

Le compilateur LUSTRE commence son travail par une vérification statique des programmes, puis il transforme le programme en une forme aplatie en effectuant une expansion des nœuds. Le code produit est structuré soit en simple itération, soit en automate à état fini.

Vérification statique La détection d'incohérences à la compilation est un objectif particulièrement important dans un langage prétendant améliorer la fiabilité des programmes,

sans en pénaliser les temps d'exécution. Hormis les vérifications classiques de cohérence des types, la compilateur LUSTRE assure :

- la vérification que toute variable qui n'est pas une entrée du programme fait l'objet d'une et une seule définition ;
- la vérification qu'un nœud n'est jamais instancié de manière récursive ;
- le calcul et la vérification de cohérence des horloges sur lesquelles nous allons revenir ;
- la vérification qu'aucune expression non initialisée (valeur `nil`) n'influence la valeur des horloges, des sorties ou des assertions ;
- l'absence de définition cyclique : une variable ne peut intervenir dans sa propre définition qu'au travers de valeurs du passé strict. Le compilateur ne tente pas de donner un sens à une équation de la forme `x = 3 * x + 1`. Une telle définition est assimilable à un inter-blocage. La détection de ces blocages est effectuée par une simple analyse de dépendance statique : LUSTRE refuse également les faux blocages comme le cas suivant :

  ```
  x = if c then y else z;
  y = if c then z else x;
  ```

  Il est clair que l'acceptation de telles expressions dans un programme pose des problèmes insolubles de manière automatique.

Le traitement des horloges constitue un élément original de la compilation de LUSTRE. Ce traitement consiste à associer une horloge à toutes les expressions du programme et à vérifier que tout opérateur est appliqué à des expressions ayant des horloges convenables, c'est à dire :

- tout opérateur de base ayant plus d'un argument est appliqué à des opérandes ayant des horloges identiques. Établir l'identité des horloges est un problème difficile. LUSTRE considère deux horloges comme identiques si elles peuvent être ramenées à la même expression, par des substitutions syntaxiques successives ;
- toute opérande d'un opérateur `current` possède une horloge différente de l'horloge de base du nœud où il apparaît ;
- les horloges des paramètres effectifs d'un nœud satisfont les contraintes spécifiées dans la déclaration d'interface du nœud.

Expansion des nœuds Le compilateur LUSTRE produit un code séquentiel. Cette production de code passe par le traitement des nœuds qui sont «instanciés», c'est à dire que leur corps remplace leur invocation dans le programme, après traitement des paramètres. Cette méthode d'instanciation est proche des macro-langages, comme le pré-processeur du langage C, par exemple. Cette méthode permet, par exemple, de définir des cycles, qui, en fait, n'en sont pas, après instanciation. Considérons l'exemple suivant qui illustre cette propriété :

```
node Copie (a, b : int)
    return (x, y : int);
  let
    x = a;
    y = b;
  tel
```

Ce nœud possède deux entrées et deux sorties, qui sont la recopie directe des entrées correspondantes. Il est évident qu'il existe deux possibilités pour le code séquentiel produit à partir d'un tel programme : soit `x=a; y=b;` soit `y=b; x=a;`. Le problème est que le choix entre l'une et l'autre des deux formes n'est pas indifférent et peut dépendre de la manière dont le nœud est appelé. Par exemple, pour l'appel suivant :

```
(i,j) = Copie (a, i);
```

seule la première forme convient. Avant de générer le code, le compilateur réalise donc l'expansion des appels de nœuds dans le programme source : tout appel de nœud est remplacé par son corps, après avoir renommé les paramètres et traité les horloges. La génération de code se fera donc à partir d'un programme «aplati».

Production de code Le compilateur peut produire deux sortes de code : un codage basé sur une itération infinie, et un codage basé sur un automate à états finis.

L'itération infinie est composée de deux parties : une partie concernant l'initialisation des variables, et une partie concernant le fonctionnement du programme.

Les auteurs de LUSTRE ont préféré concentrer leurs efforts sur la production de code en automate à états finis, basé sur le compilateur du langage ESTEREL.

5.4.3 Synthèse de lustre

Le langage LUSTRE est un langage qui pourrait convenir pour notre modélisation fonctionnelle. En effet, il impose aux applications les déterminismes en temps et en ressources et il est modulaire.

Cependant, la résolution d'un programme LUSTRE ne peut être modélisée directement de manière fonctionnelle. En effet, une transformation préalable des programmes est nécessaire.

Deux types de transformations sont possibles. La première consiste à instancier les nœuds, c'est à dire à remplacer leurs invocations par leur corps dans lequel les paramètres ont été remplacés par la valeur des arguments. La secondes transformation possible consiste a ajouter un paramètre aux nœuds. Ce paramètre représente l'état du nœud au moment de l'invocation. Bien sûr, le nœud retourne alors une valeur suplémentaire qui est son nouvel état. Un tel mécanisme est décrit dans la présentation du langage SCHEME (§ 4.8).

De plus, la puissance de LUSTRE s'exprime par le fait qu'il gère parfaitement les horloges. Nous pensons que cette possibilité est de trop haut niveau dans le cadre de notre modélisation fonctionnelle. De plus, les horloges peuvent être réalisées par une séries d'alternatives (`if ... then ... else`). Notre langage primitif possédera donc les alternatives mais ne contiendra pas la notion d'horloges multiples, ce qui simplifiera sa formalisation. La définition des horloges pourrait être ajoutée au langage au moyen d'un pré-processeur.

Enfin, LUSTRE ne gère les environnements qu'au premier niveau. Cela signifie qu'un nœud ne peut pas contenir de variables libres. Dans notre modélisation, le langage primitif supporte pleinement la modularisation (équivalent des nœuds) avec la possibilité de gérer les variables libres. Cela se traduit par une gestions hiérarchique des environnements.

La gestion hierarchique des environnements est intéressante d'un point de vue de la puissance expressive d'un langage. Elle permet notamment de définir des variables globales, qui représentent par exemple les options de l'application. Cette possibilité diminue considérablement le nombre des paramètres des modules d'un programme.

Par contre, la gestion hiérarchique des environnements se traduit par un accroissement de la complexité des définitions formelles donnant la sémantique du langage. Notamment, la construction des environnements et le mode de liaison des variables doivent être modélisés. Dans notre étude, la gestion hiérarchique des environnements est modélisée de manière fonctionnelle. Nous pouvons considérer notre étude comme une extention de LUSTRE qui apporte la gestion hiérarchique des environnements modélisée de manière fonctionnelle.

5.5 Le langage signal

Les langages SIGNAL [13, 38] et LUSTRE se ressemblent beaucoup, quant aux concepts qu'ils manipulent. Bien que SIGNAL soit antérieur à LUSTRE, nous avons présenté celui-ci en premier car sa syntaxe est plus accessible, et il est plus proche de notre travail de thèse.

SIGNAL est un langage synchrone de type équationnel construit autour d'un noyau minimum de constructeurs de bases. Il n'est pas fonctionnel mais relationnel, dans la mesure où

il définit des relations entre les flots d'entrées et les flots de sorties. La manière d'utiliser un flot de sortie influence sa définition. Il s'agit d'une programmation par contraintes.

Un programme SIGNAL décrit les relations entre signaux. Un signal est une suite non bornée de valeurs typées. Nous associons à chaque signal son horloge qui détermine les instants où les valeurs sont disponibles. Il est possible d'exprimer des contraintes de synchronisation entre les différents signaux. Le compilateur se charge alors de déterminer si les contraintes sont vérifiées ou pas.

SIGNAL utilise l'hypothèse de synchronisme fort : toute action (calcul ou communication) est instantanée. Cette hypothèse a l'avantage de permettre au concepteur de se concentrer sur les aspects logiques du problème sans considération pour les détails de mise en œuvre (tels que les vitesses de traitement et les délais de communication). En effet, ces détails compliquent le problème dans sa réalisation et ajoutent des informations qui dépendent de la spécificité du matériel hôte. Cependant, ces problèmes de mise en œuvre ne sont pas ignorés, mais ils sont traités dans une phase ultérieure.

Il existe deux types d'opérateurs de base dans SIGNAL : ceux qui expriment des équations entre signaux synchrones (expressions fonctionnelles et retards) et ceux qui composent des signaux d'horloges différentes (conditions et fusions). Ces opérateurs de base servent à décrire des équations qui sont combinées au moyen d'un opérateur de composition associatif et commutatif.

SIGNAL utilise un système de preuves interactif appelé SIGALI. Ce dernier permet de vérifier les propriétés dynamiques d'un programme. Nous pouvons faire de la preuve par comparaison ou *model checking*. La spécification peut être décrite en SIGNAL ou bien en logique temporelle.

SIGNAL permet une conception mixte matérielle-logicielle. Il est alors utilisé comme langage de spécification, en association avec le langage VHDL qui permet d'accéder à des bibliothèques existantes. Un système décrit en SIGNAL peut donc être réalisé soit de manière logicielle, soit de manière matérielle, soit mixte.

L'adéquation de SIGNAL pour la spécification de systèmes réactifs ainsi que son utilisation comme base pour la réalisation en circuit ou en système mixte se justifie par ce qui suit :

- SIGNAL est un langage déclaratif indépendant de toute réalisation particulière. Il se prête aussi bien à une mise en œuvre logicielle que matérielle ;
- SIGNAL peut décrire des systèmes matériels ou logiciels à différents niveaux d'abstraction. Il permet par exemple de modéliser l'aspect flot de données des circuits. Son aspect multi-horloges lui confère une flexibilité supplémentaire en l'autorisant à modéliser différents types de circuits ;
- à l'inverse des langages tels que VHDL ou C, il possède une sémantique définie formellement et un nombre d'opérateurs réduit. Ceci permet d'envisager des raisonnements formels sur des programmes (preuves, transformations, etc) ;
- la validation des programmes peut être faite soit par simulation soit par vérification formelle ;
- des outils permettant la saisie, la compilation et la réalisation sur des machines mono ou multi-processeurs sont disponibles.

Une méthodologie de conception conjointe se basant sur SIGNAL comme langage de spécification permet de profiter des outils de spécification, d'analyse et de validation au niveau système offert par l'environnement SIGNAL.

5.5.1 Synthèse de signal

L'ensemble des critiques de LUSTRE peut être apporté à SIGNAL. De plus, la syntaxe de ce langage et les concepts utilisés sont très difficiles d'accès, ce qui le met hors de portée d'un utilisateur non averti.

5.6 Synthèse des langages cités

Dans ce chapitre, nous avons tenté de décrire l'essentiel des principaux langages de programmation dataflow. Ils permettent tous de construire les applications de manière modulaire, en définissant les notions de modules et d'interfaces.

Ils se répartissent clairement en deux catégories, les langages basés sur la définition d'itération, comme LUCID[6], VAL et SISAL. Dans ces langages, les calculs sont décrits comme étant des itérations sur des données. La fin de l'itération est un rendez-vous pour continuer l'exécution du programme. Ce type de résolution est fondamentalement indéterministe en temps. En effet, la durée des itérations intérieures ne peut être connue à l'avance car elle peut dépendre des entrées du programme. Parmi ces langages, LUCID définit les flots d'une manière puissante et expressive, alors que VAL et SISAL ont une syntaxe et une sémantique complexes.

L'autre famille de langage est composée de LUSTRE et de SIGNAL. Ici, d'une manière simplificatrice, nous assimilons SIGNAL à LUSTRE. LUSTRE est très intéressant parce qu'il permet d'obtenir les déterminismes en temps et en ressources des applications. Il suit une autre voie que LUCID en associant aux flots des horloges. Il propose une algèbre des horloges et effectue des vérifications de concordance à leur sujet. Cependant, LUSTRE est un langage de trop haut niveau pour notre étude. Enfin, LUSTRE propose une gestion des environnements au premier niveau, sans construction hiérarchique.

Notre étude peut donc être vue comme la tentative d'apporter à LUSTRE une gestion hiérarchique des environnements avec la liaison statique des variables. Cette modélisation est réalisée de manière fonctionnelle.

6. LUCID permet de décrire des itérations au sein d'un programme avec l'opérateur `asa` qui est la principale différence entre lui et le langage LUSWIM.

Chapitre 6

Synthèse et position du problème

Nous avons présenté dans la première section de cette partie ce qui constitue le cahier des charges de notre travail de thèse : il s'agit de programmer avec un formalisme de haut niveau l'architecture parallèle statique présentée dans la figure 1.1.

Une manière simple de programmer cette architecture est que les programmes de chacun des contrôleurs qui la compose soit basé sur une itération. La durée de cette itération doit être constante, ce qui impose les déterminismes en temps et en ressources du programme.

Pour programmer cette architecture, nous nous sommes intéressés aux graphes dataflow. Nous avons constaté qu'ils sont un moyen de décrire les applications et de les résoudre. La description graphique des applications est un point fort de ces formalismes.

Cependant, le manque de sémantique globale des graphes dataflow ne nous permet pas de les utiliser directement. Nous conserverons des graphes dataflow la possibilité de décrire les applications de manière graphique.

Les langages fonctionnels peuvent être vus comme une famille de graphes dataflow. Le point commun à tous les langages fonctionnels est le λ-calcul qui formalise mathématiquement le concept de fonction. Il est décrit à l'aide d'une grammaire abstraite extrêmement simple, puisqu'elle ne définit que trois éléments, et de règles de réductions qui sont en fait une machine abstraite de résolution.

L'utilisation conjointe d'une grammaire et d'une machine abstraite définit complètement le langage de manière précise et non ambiguë. Cependant, le λ-calcul permet de définir des expressions dont le comportement est aberrant, ce qui ne signifie pas que l'ensemble du formalisme est lui-même aberrant. De plus, il est trop expressif et permet de concevoir des programmes indéterministes en temps et en ressources.

Nous conserverons du λ-calcul l'utilisation conjointe d'une grammaire abstraite et d'une machine de résolution pour définir le langage, ainsi que la propriété fonctionnelle.

Le langage SCHEME est une forme de λ-calcul dont le mode d'évaluation est limité, ce qui le rend efficace. C'est un langage fonctionnel basé sur LISP. Il ne possède qu'une seule instruction héritée des langages impératifs : l'affectation.

SCHEME précise les notions d'environnements, de variables libres, liés et globales, ainsi que les fonctions à récursion terminale. Il faut noter que ces notions sont exprimables avec le λ-calcul, mais d'une manière moins accessible.

SCHEME a comme particularité intéressante le typage implicite des données, où le type d'une donnée est déduit par sa syntaxe plutôt que par une déclaration de type. Ce mode de typage a comme avantages de libérer le programmeur des déclarations intempestives des types et de favoriser les définitions polymorphes. L'inconvénient du typage implicite est que les erreurs de types sont détectées à l'exécution du programme, ce qui signifie que des portions de programmes peuvent demeurer incohérentes tant qu'elles ne sont pas exécutées.

Comme le λ-calcul, SCHEME permet d'une manière trop évidente de construire des programmes indéterministes en temps et en ressources. Il ne peut donc être notre langage de description. Nous garderons de SCHEME la gestion hiérarchique des environnements et le ty-

page implicite des données (bien que le compilateur du langage que nous décrirons effectue un contrôle strict de la cohérence des types).

Les langages dataflow sont la dernière catégorie de formalismes étudiés. Ils se décomposent en deux ensembles, les langages basés sur les rendez-vous et qui permettent une exploitation dynamique du parallélisme, et les langages synchrones.

Le premier ensemble contient le langage LUCID qui est pratiquement l'ancêtre de tous les langages dataflow. Le flot de données est la structure qui lui permet de donner une vision fonctionnelle des variables. Il permet de construire de nouveau flots de données d'une manière très intéressante, à l'aide du seul opérateur `fby`. De plus, les environnements sont gérés de manière totalement hiérarchique, à l'aide de la construction `where`.

Cependant, LUCID possède une construction qui permet de définir des itérations au sein même des programmes, ce qui les rend indéterministes en temps.

L'autre ensemble de langages dataflow est représenté par LUSTRE. Ce langage déclaratif permet d'obtenir le déterminisme en temps des programmes. Cependant, sa définition ancienne le limite sérieusement.

Nous retiendrons de LUCID sa définition des flots extrêmement simple et accessible, ainsi que leur sémantique. De LUSTRE nous garderons les définitions acycliques qui permettent d'obtenir le déterminisme en temps des programmes.

La deuxième partie de la thèse va donc décrire un formalisme qui retiendra tout ce que nous avons énoncé dans cette synthèse, c'est à dire :

- définition abstraite du langage à l'aide d'une grammaire et de sa machine de résolution ;
- définition du langage exclusivement fonctionnelle pure ;
- indépendance du langage avec les données manipulées, qui sont définies dans des algèbres, formées d'un type de données et des opérateurs qui s'y rapporte ;
- typage implicite déduit de la syntaxe des données ;
- gestion hiérarchique des environnements ;
- langage dataflow ;
- sémantique des flots inspirée de LUCID ;
- déterminismes en temps en ressources ;
- définitions de l'interprète et du compilateur de ce langage.

Bien évidemment, la résolution de ce langage sera basée sur une itération qui pourra par la suite être réalisée sur l'architecture parallèle statique.

La troisième partie de la thèse montrera comment obtenir une représentation graphique des applications et comment extraire le parallélisme de ce langage.

Deuxième partie

Étude théorique : les λ-matrices

Chapitre 1

Introduction

Le point de départ de ce travail de thèse repose sur une modélisation fonctionnelle des applications à état (§ I-3).

Nous espèrons tirer profit de cette modélisation de deux manières. D'une part, la modélisation fonctionnelle repose sur des définitions formelles qui devraient permettre d'effectuer des vérifications formelles des programmes plus facilement qu'avec un formalisme non fonctionnel [40]. D'autre part, la programmation fonctionnelle semble faciliter l'exploitation du parallélisme des applications. Nous verrons qu'effectivement, le formalisme fonctionnel développé dans cette thèse facilite à la fois les vérifications formelles et l'exploitation du parallélisme.

Au cours de ce chapitre, un formalisme abstrait va être développé. Conjointement aux définitions formelles, les définitions SCHEME correspondantes seront décrites, construisant au fur et à mesure l'interprète puis le compilateur du langage développé. SCHEME est utilisé à la fois pour clarifier les expressions algébriques parfois complexes, et pour donner des exemples *réels*. Le lecteur notera que l'affectation, qui se traduit en SCHEME par `set!` ou `define` d'un symbole déjà défini, n'est utilisée nulle part. Ceci confirme le caractère fonctionnel du formalisme.

Une application à état est un système qui réagit à son environnement par des entrées et qui agit sur lui par ses sorties. A un instant donné, la valeur des sorties ne peut être déterminée seulement à l'aide de la valeur des entrées : le système possède un état interne qui évolue avec le temps.

Une modélisation fonctionnelle d'une application à état repose sur deux points clefs : la représentation de l'application, généralement avec un langage, et sa résolution, à l'aide d'une machine de résolution fonctionnelle.

Le langage de description peut être concret ou abstrait. Un langage concret ne contient aucune ambiguïté et peut être traité avec des moyens informatiques. Il est souvent décrit à l'aide d'une grammaire. Un langage est abstrait lorsqu'il est partiellement défini. Par exemple, les langages abstraits ne précisent pas nécessairement la priorité des opérateurs. Ils ont l'avantage d'une grande simplicité, et c'est pourquoi nous les utiliserons. Nous pouvons dire, d'une manière imagée, qu'une grammaire abstraite est formée d'une collection d'arbres d'analalyse du langage plutot que d'une collection de chaînes de caractères.

Une application décrite à l'aide d'un langage est destinée à être résolue. Cette résolution peut être concrète ou abstraite. La résolution concrète d'une application informatique passe généralement par l'utilisation d'un ordinateur. Comme l'ordinateur ne «comprend» que son propre langage machine, une traduction préalable est nécessaire. Cette traduction peut être effectuée immédiatement, au moment de la résolution, en faisant appel à un interprète. Un interprète est un programme écrit en langage machine qui «comprend» le langage source et le traduit au fur et à mesure en séquences d'instructions. Le programme peut aussi être pré-traduit totalement en langage machine, puis directement exécuté par l'ordinateur. Nous faisons alors appel à un compilateur.

L'interprète facilite la mise au point des programmes en affranchissant le concepteur

des phases de traductions préalables. Cependant, le programme sera moins efficace du fait que la traduction fait partie de l'exécution. Le compilateur oblige le concepteur à traduire son programme avant de l'exécuter. Mais alors le programme traduit est très efficace car il n'effectue aucune traduction pendant l'exécution. Le formalisme développé ici fera à la fois appel à un interprète et à un compilateur.

Une résolution est abstraite lorsqu'elle fait appel à une machine abstraite de résolution. Une machine abstraite n'a pas nécessairement d'existence physique mais propose plutôt un modèle de résolution. L'avantage d'utiliser une machine abstraite est de proposer un modèle de résolution indépendant de tout ordinateur. De plus, elle est généralement beaucoup plus simple à décrire qu'une machine réelle.

Cette introduction présente un langage abstrait primitif permettant la description des applications à état ainsi que la machine abstraite de résolution correspondante. Cette approche préliminaire n'est pas le cœur du travail de thèse mais introduit les principes et les méthodes qui seront utilisées par la suite.

La représentation usuelle des applications à état est le point de départ de cette étude préliminaire. La figure 1.1 montre une représentation graphique usuelle d'une application à état. Les entrées sont situées sur la gauche et les sorties sont situées sur la droite. Une partie des sorties est ramenée en entrée, définissant un vecteur d'état. La boîte grisée représente la partie fonctionnelle du modèle : chaque sortie est une fonction des entrées.

Les fonctions de la partie grisée sont supposées calculer les sorties en permanence. De manière à synchroniser l'ensemble, il existe trois horloges. La première permet de fixer[1] les entrées, la seconde permet de fixer les sorties, et la troisième permet de recopier les sorties correspondantes au vecteur d'état sur les entrées correspondantes et de les fixer.

Le modèle fonctionne alors sur un cycle infini des trois horloges h_1, h_2, h_3, dans cet ordre. L'existence de ce cycle immuable nous incite à poursuivre l'étude de ce modèle, car l'architecture parallèle statique nécessite des applications basées sur ce type de cycle (§ 1-1.3).

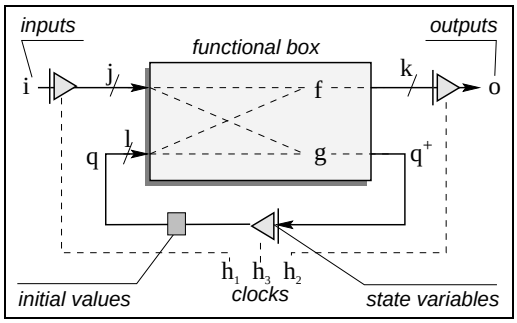


FIG. 1.1 - Représentation usuelle d'un modèle d'état.

Deux remarques peuvent être formulées. Premièrement, le devenir des sorties ne concerne pas le modèle et, deuxièmement, les entrées sont supposées disponibles lorsque l'horloge h_1 les fixe. A l'intérieur du système, l'unité de temps est donnée par un cycle des trois horloges. Il existe donc «un temps extérieur» au système qui anime les trois horloges, et «un temps intérieur» dont l'unité est un cycle des trois horloges. La seule condition pour que cette hypothèse soit valable est que le temps de réponse de la boîte fonctionnelle soit inférieure à un cycle des trois horloges. Cette manière d'appréhender le temps est la base des langages synchrones, comme ESTEREL [14] ou LUSTRE [43].

Dans cette introduction, nous proposons un modèle fonctionnel pour décrire et résoudre les applications à état. Le point important de cette description est la modélisation fonctionnelle de l'état, chose qui semble habituellement dans la littérature incompatible, ou tout du moins, peu naturelle.

Le modèle proposé ici utilise des notions simples de l'algèbre linéaire. Les structures algébriques sont utilisées comme un langage abstrait de description. Les fonctions algébriques définissent quant à elles la machine abstraite de résolution.

Pour simplifier, les données manipulées par le modèle sont des entiers relatifs de l'ensemble $\mathbb{Z}$. Nous utilisons l'ensemble $\mathbb{F}_k$ des fonctions f à k paramètres telles que :

$$\begin{aligned} &f : \mathbb{Z}^k \longrightarrow \mathbb{Z} \\ &\bar{n} = (n_1, n_2, \ldots, n_k) \mapsto f(\bar{n}) = f(n_1, n_2, \ldots, n_k) \end{aligned} \tag{1.1}$$

1. Le verbe fixer peut être entendu comme «échantillonner».

On utilise l'ensemble $\mathbb{F}_k^n$ des fonctions vectorielles $\bar{g}$ de dimension n telles que:

$$\begin{array}{l} \bar{g} : \mathbb{Z}^k \longrightarrow \mathbb{Z}^n \\ \bar{u} = (u_1, u_2, \ldots, u_k) \mapsto (\bar{g}_1(\bar{u}), \bar{g}_2(\bar{u}), \ldots, \bar{g}_n(\bar{u})) \end{array} \tag{1.2}$$

On définit maintenant l'ensemble $\mathbb{S}^n$ des systèmes à n composantes par:

$$\forall s \in \mathbb{S}^n, s = (\bar{e}, \bar{f}), \bar{e} \in \mathbb{Z}^n, \bar{f} \in \mathbb{F}_n^n \tag{1.3}$$

qui signifie que quelque soit l'élément particulier s des systèmes à n composantes, s est un couple formé d'un n-uplet d'entiers et d'un n-uplet de fonctions à n arguments.

Nous appellerons ce couple *modèle d'état*. Les valeurs de E représentent les valeurs initiales du vecteur d'état. Les fonctions de F représentent le moyen d'obtenir un nouvel ensemble de n valeurs représentant le nouveau vecteur d'état.

Notons que dans cette description, les entrées ne sont pas représentées. Pour les prendre en compte, il suffirait d'étendre les domaines de définition des fonctions f. Mais cette extension n'est pas utile dans cette étude préliminaire qui se focalise sur la modélisation fonctionnelle des variables d'état.

Un système est donc le couple d'un ensemble de valeurs initiales et d'un ensemble de fonctions. Cette structure algébrique est obtenue à l'aide de structures plus simples qui jouent le rôle d'un langage abstrait. Il s'agit là de la partie descriptive ou statique du formalisme car nous ne connaissons pas encore la façon de résoudre de telles applications.

La résolution d'une application est l'aspect dynamique du modèle. Résoudre une application revient à obtenir la succession des vecteurs d'état. Sans condition d'arrêt, cette succession est composée d'un nombre infini de vecteurs d'états.

Cette résolution s'effectue à l'aide de quatre opérateurs. Le premier permet d'appliquer une fonction de l'ensemble $\mathbb{F}_n$ à n arguments entiers de manière à obtenir un résultat entier. Nous appellerons cet opérateur *opérateur de réduction* car il réduit une application[2] en un résultat. Il est défini par:

$$\triangleleft : \mathbb{F}^n \times \mathbb{Z}^n \to \mathbb{Z} \quad | \quad \triangleleft (f_n, \bar{a}) = \triangleleft(f_n, a_1, \ldots, a_n) = f_n(a_1, \ldots, a_n) \tag{1.4}$$

Le second opérateur utilisé par la résolution des systèmes est appelé *opérateur d'évaluation* car il donne une valeur au système. Il prend comme argument un système S à n composantes et il retourne un ensemble à n valeurs entières. Ces valeurs sont obtenues en réduisant par $\triangleleft$ l'application des fonctions de S à son état. Il est défini par:

$$\triangle : \mathbb{S}^n \times \mathbb{Z}^n \to \mathbb{Z}^n \quad | \quad \triangle (s) = (\triangleleft(\bar{f}_1, \bar{e}), \ldots, \triangleleft(\bar{f}_n, \bar{e})) \tag{1.5}$$

Le troisième opérateur participant à la résolution des systèmes permet d'obtenir un système avec un nouveau vecteur d'état à partir d'un système original. L'unité de temps à l'intérieur d'un système est un cycle de résolution. Ce cycle de résolution est généré par cet opérateur qui est appelé *opérateur de régénération*. Il prend comme argument un système à régénérer pour rendre le système régénéré. Ce système est constitué du même ensemble de fonctions, mais son état est le résultat de l'évaluation du système. Il est défini par:

$$\bigtriangledown : \mathbb{S}^n \to \mathbb{S}^n | \bigtriangledown (s) = (\bar{e}', \bar{f}) \text{ avec } \bar{e}' = (\triangle(s), \bar{f}) \tag{1.6}$$

Enfin, l'*opérateur de résolution* peut être défini. Il prend comme argument un système S à résoudre et un entier i qui va servir d'index pour la condition d'arrêt dans le système. Cet opérateur est une fonction à récursion terminale bâtie autour d'une alternative. Si la valeur à l'index i dans le système est différente de zéro, alors le processus de résolution est arrêté,

2. Le terme application peut dénoter un programme, ou l'application d'un opérateur à des arguments. La distinction sera précisée à chaque ambiguïté.

et son résultat est, par convention, le système S. Si cette valeur vaut zéro, alors le processus continue avec le système régénéré à partir de S. L'expression de cet opérateur est :

$$\triangleright : \mathbb{S}^n \times \mathbb{Z} \to \mathbb{S}^n \mid \triangleright_i (S) = \begin{cases} S, \text{ si } e_i \neq 0, \text{ avec } S = (\{e_1, \ldots, e_n\}, F) \\ \triangleright_i(\bigtriangledown(s)), \text{ sinon.} \end{cases} \tag{1.7}$$

Pour définir un système de manière plus confortable qu'avec la structure algébrique précédente, il est possible de définir un langage primitif formé des objets suivants :

- les vecteurs jouent le rôle de structure d'accueil et ils sont notés $\left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix} \right]$, où c_1 et c_n sont des définitions,
- les définitions permettent d'associer un nom à une valeur. Elles sont notées $s := v$ où s est un identificateur et v un flot,
- les flots associent un état et une fonction, et ils sont notés $e\, f_{by}\, c$, où e est un entier jouant le rôle de l'état et c une application,
- des applications, notées $f(i_1, i_2, \ldots, i_n)$ où f est un opérateur et i des identificateurs.
- des identificateurs qui sont des symboles dont la syntaxe peut être celle utilisée dans les autres langages.

Ce langage est primitif. Par exemple, il n'est pas possible de donner à une définition une valeur autre qu'un flot. Mais il permet déjà de décrire une application à état. Par exemple, le système défini par $S_n = \{\{e_1, e_2, ..., e_n\}, \{f_1, f_2, ..., f_n\}\}$, est «programmé» avec :

$$\left[\begin{array}{l} i_1 := e_1\, f_{by}\, f_1(i_1, i_2, ..., i_n) \\ i_2 := e_2\, f_{by}\, f_2(i_1, i_2, ..., i_n) \\ ... \\ i_n := e_n\, f_{by}\, f_n(i_1, i_2, ..., i_n) \end{array} \right] \tag{1.8}$$

Ce formalisme introductif montre qu'il est possible de modéliser de manière absolument fonctionnelle une application à état [22]. Dans les chapitres qui vont suivre, nous allons construire un langage abstrait sur ces bases. Ce langage possède une grammaire (§ 2), une machine abstraite comportant les mêmes opérateurs de résolution que dans cette introduction (§ 3), et trois fonctions de critères permettant d'établir les déterminismes en temps et en ressources (§ 4). Nous montrerons ensuite comment compiler d'une manière formelle un programme (§ 5). Enfin, nous vérifierons certaines propriétés de ce modèle (§ 6).

Mais quel est l'intérêt d'une telle modélisation fonctionnelle? L'atout majeur est la possibilité d'établir des vérifications formelles d'une manière plus simple qu'avec un formalisme non fonctionnel. Ceci provient du fait que les règles de transformations des programmes sont des combinateurs, c'est à dire des fonctions définies sur les expressions du langage dont la valeur dépend seulement de leurs paramètres. Il en découle que les démonstrations suivent simplement une méthode inductive sur les différentes expressions du langage. Dans ce rapport, nous donnons des exemples de déductions par induction qui nous permettent de vérifier les déterminismes des applications (§ 6). L'utilisateur pourra à sa convenance, établir d'autres propriétés, comme par exemple dans le chapitre (§ 6.1).

Cependant, l'utilisation d'une méthode purement fonctionnelle peut sembler coûteuse en temps de calcul. Notamment, le lecteur aura remarqué que l'opérateur de régénération duplique la totalité du système pour fournir un nouveau système dont les états des flots sont mis à jour. Cette duplication est propre aux modélisations fonctionnelles et elle en est leur principale critique et leur principale source d'inefficacité.

A ce stade, il faut faire la différence entre le modèle abstrait de calcul et la réalisation de ce modèle. Lorsque nous chercherons à réaliser une application, nous utiliserons certaines

propriétés du modèle afin d'éviter cette duplication. La duplication sera en fait remplacée par des variables à affectation[3] unique ou *single assignments*.

Cette affectation unique est la traduction dans un formalisme impératif d'un modèle fonctionnel. Il peut être résumé ainsi : une seule valeur est affectée à une variable, et avant cette affectation, aucune expression n'utilise cette variable. Dans ce cas, la variable est en fait une définition, au sens des formalismes fonctionnels. Le modèle de résolution proposé ici permet d'exploiter au mieux cette particularité et d'obtenir ainsi des réalisations très efficaces.

L'autre inconvénient de la modélisation fonctionnelle provient toujours de la duplication du programme. Nous allons définir des combinateurs permettant d'établir des propriétés, comme les déterminismes en temps et en ressources. Ces combinateurs vont être appliqués au programme initial. Or la résolution de ce programme est basée sur l'opérateur de régénération qui duplique le programme original pour en fournir un nouveau dont les états des flots sont mis à jour. Cette régénération est répétée jusqu'à ce que la condition arrêt soit atteinte. La question qui se pose est la suivante : si le programme original possède une propriété, le programme régénéré la possède-t-il aussi ? Pour répondre à cette question, nous utiliserons une méthode inductive sur les expressions du langage, ou bien des déductions simples, lorsque cela est possible (§ 6.1).

3. Les langages impératifs reposent sur la notion de variables, directement liée à l'architecture de VON NEUMANN des ordinateurs actuels. Une variable est une case de la mémoire de l'ordinateur. Nous disons alors que l'on affecte une valeur à une variable lorsque l'on place une valeur dans cette case.

Chapitre 2

Langage abstrait

Ce chapitre décrit un langage fonctionnel abstrait. Ce langage est voulu aussi simple que possible : il ne contient aucune construction superflue qui pourrait être obtenue autrement, ce que LANDIN appelle du *sucre syntaxique*. Donc, pas de sucre !

Ce langage permet de modéliser les applications à état avec la structure de flot de données. Il autorise la programmation structurée à l'aide de modules primitifs.

Dans une première section, le langage est décrit à l'aide d'une grammaire abstraite (§ 2.1), puis les objets de la grammaire sont expliqués d'une manière informelle (§ 2.2).

Les constructeurs SCHEME de ces objets sont aussi décrits et utilisés en exemple (§ 2.3). Ils permettront de construire, tout au long du développement, un interprète SCHEME des λ-matrices.

2.1 Grammaire abstraite

Cette section décrit les acteurs[1] du langage abstrait des λ-matrices. Ils sont construits par induction sur une grammaire abstraite [5]. Le lecteur pourra avantageusement consulter la section suivante qui donne une explication informelle des acteurs (§ 2.2). En première approximation, une grammaire abstraite est formée d'une collection d'arbres d'analyse plutôt qu'une collection de chaînes de caractères.

Premièrement, l'ensemble des atomes est défini :

$$
\begin{array}{lcll}
a \in \mathcal{A} & \longrightarrow & d \in \mathcal{D} & \text{: donnée} \\
 & | & i \in \mathcal{I} & \text{: identificateur} \\
 & | & z \in \mathcal{Z} & \text{: entier} \\
 & | & o \in \mathcal{O} & \text{: opérateur} \\
 & | & \bot & \text{: bottom} \\
 & | & \top & \text{: top}
\end{array}
\tag{2.1}
$$

Le symbole $\longrightarrow$ peut être lu comme *est* ou *peut être remplacé par*. Le | signifie *ou bien*. Donc un élément particulier a de l'ensemble des atomes A est soit un élément particulier d de l'ensemble D des données, soit un élément particulier i de l'ensemble des identificateurs, et ainsi de suite.

Les atomes sont l'alphabet du langage. L'ensemble des données et l'ensemble des opérateurs sont incomplètement définis, et ils contiennent des éléments définis par l'utilisateur dans des structures appelées *algèbres*. Cette possibilité est importante ; en effet, le langage n'impose pas la nature des données manipulées. Seuls les entiers et les opérateurs associés sont pré-définis. Les indicateurs $\bot$ et $\top$ sont inaccessibles à l'utilisateur et ils sont utilisés de manière interne.

1. Un acteur dans un langage est une expression de ce langage.

Les atomes sont des objets insécables, c'est à dire qu'ils ne sont pas composés par d'autres objets du langage. L'existence de tels objets est indispensable pour que la grammaire soit définissable mathématiquement [40]. Voici maintenant l'ensemble de tous les acteurs du langage, contenant les atomes et les objets composés :

$$\begin{array}{rcll} x \in \mathcal{X} & \longrightarrow & a & : \text{atome} \\ & | & x \to x, x \quad \in \mathcal{T} & : \text{alternative} \\ & | & o(x_1,\, x_2,\, \ldots,\, x_n) \quad \in \mathcal{P} & : \text{application} \\ & | & x\, f_{by}\, x \quad \in \mathcal{S} & : \text{flot} \\ & | & i := x \quad \in \mathcal{F} & : \text{définition} \\ & | & x \cdot x \quad \in \mathcal{E} & : \text{extraction} \\ & | & \left[\begin{array}{c} x_1 \\ \vdots \\ x_n \end{array}\right] \quad \in \mathcal{V} & : \text{vecteur} \end{array} \tag{2.2}$$

Il existe un sous-ensemble des acteurs appelé *ensemble des acteurs figés* et défini par :

$$\begin{array}{rcll} x' \in \overline{\mathcal{X}} & \longrightarrow & d|n|o|\bot|\top \quad \in \mathcal{A}-\mathcal{I} & : \text{atome – identificateur} \\ & | & \left[\begin{array}{c} x'_1 \\ \vdots \\ x'_n \end{array}\right] \quad \in \mathcal{V} & : \text{vecteur d'acteurs figés} \end{array} \tag{2.3}$$

Cet ensemble d'acteurs aura son importance par la suite, lors des vérifications formelles (§ 6).

Dans ce qui suit, les lettres utilisées dans les expressions ne feront pas référence à l'ensemble d'appartenance de l'expression. Par exemple, une alternative sera notée $c \to a, s$, où c pour *condition*, a pour *alors* et s pour *sinon* sont tous trois des acteurs de $\mathcal{X}$. D'une manière plus rigoureuse, nous devrions la noter $x \to x, x$, sans différencier les composantes.

La grammaire permet de construire des expressions reconnues comme des acteurs de ce langage. Par exemple :

$$1\, f_{by}\, +(1, 3) \tag{2.4}$$

est une expression valide du langage. Il faut noter qu'il s'agit d'une grammaire abstraite. Une grammaire abstraite peut laisser apparaître dans certains cas des expressions ambiguës. Par exemple:

$$1\, f_{by}\, 2\, f_{by}\, 3 \tag{2.5}$$

peut être compris de deux manières, selon que l'opérateur f_{by} est associatif à droite ou à gauche. Nous pouvons donc lire l'expression (2.5) des deux manières suivantes :

$$(1\, f_{by}\, 2)\, f_{by}\, 3 \tag{2.6}$$

ou :

$$1\, f_{by}\, (2\, f_{by}\, 3) \tag{2.7}$$

Une grammaire abstraite est donc incomplètement définie, mais elle est simple : elle se focalise plus sur les concepts manipulés que sur la rigueur de sa syntaxe. C'est pour cette raison qu'un langage défini avec une telle grammaire est dit «langage abstrait». Une grammaire abstraite, ambiguë, doit être enrichie pour devenir une grammaire concrète, et c'est ce que nous ferons dans les sections consacrées à la définition du langage concret associé (§ III-4).

La grammaire définie ici permet de construire des expressions. Les objets construits peuvent être associés à une construction algébrique basée sur les familles indexées [25]. Les objets mathématiques créés possèdent une propriété importante : ils sont acycliques, ce qui signifie

qu'un objet ne peut se contenir lui-même, directement ou indirectement. Cette propriété est conservée avec l'utilisation d'une grammaire inductive[2].

Nous utiliserons aussi par la suite une notation générale pour les acteurs. La notation $\{c_1, c_2, ..., c_n\}_t$ dénote un acteur de type t et de composantes c_i. Cette notation est une réminiscence de la structure algébrique initialement utilisée pour construire les acteurs, abandonnée au profit de la grammaire. Elle permet de manipuler les composantes d'un acteur, indépendamment de son type.

Une grammaire permet de construire les expressions d'un langage. Elle ne nous renseigne en rien sur le sens de ces expressions. Le sens des expressions est défini par la sémantique du langage. Dans la section suivante, nous donnerons de manière informelle cette sémantique à l'aide du langage naturel. Par la suite, une définition formelle du langage sera décrite à l'aide de la machine abstraite de résolution (§ 3).

2.2 Acteurs

La section précédente a présenté la grammaire abstraite du langage abstrait des λ-matrices. Cette grammaire permet de construire des expressions dont le sens n'est pas encore connu, à ce point du mémoire. Cette section décrit la sémantique des expressions de manière informelle. Par la suite, elle sera étudiée plus précisément à l'aide des combinateurs de la machine abstraite de résolution (§ 3).

Les λ-matrices sont composées d'un ensemble d'acteurs. Ces acteurs permettent de décrire ou programmer une application à état de manière modulaire.

Les acteurs se répartissent en trois catégories : les atomes, les objets de base (alternative, application, définition et flot) et les objets primitifs de la modularité (extraction et vecteur).

Les paragraphes suivants décrivent d'une manière informelle chaque acteur et donnent son constructeur en SCHEME. Par la suite, nous allons construire un interprète en SCHEME des λ-matrices. L'usage de SCHEME facilitera la compréhension du lecteur.

Les atomes sont l'alphabet du langage. Les entiers relatifs et les opérateurs associés, les données de l'utilisateur et les opérateurs associés[3], les identificateurs, les indicateurs $\bot$ et $\top$ sont des atomes.

La syntaxe des atomes est celle habituellement utilisée dans les autres langages, mis à part $\bot$ et $\top$ qui ne sont pas accessible à l'utilisateur.

La transcription en SCHEME des atomes est très simple. Les identificateurs sont des symboles quotés, comme `'foo` ou `'baz`, les entiers sont les entiers de SCHEME, comme 123. Les deux indicateurs sont notés `'top` et `'bottom`. Les opérateurs sur les entiers sont ceux de SCHEME, comme `+`. Enfin, les données de l'utilisateur ne sont pas précisées.

L'alternative est un choix entre deux expressions dépendant d'une condition. Elle est écrite : $condition \rightarrow alors, sinon$. Si l'évaluation[4] de *condition* est différente de l'entier 0 alors le résultat est l'évaluation de la clause *alors*, sinon le résultat est l'évaluation de la clause *sinon*[5].

Dans la modélisation SCHEME, une alternative est une liste dont le premier élément est le symbole quoté `'alternative`, et les éléments suivants sont la *condition*, la clause *alors* et la clause *sinon* qui sont tous trois des acteurs du langage.

2. Nous avons abordé succinctement les problèmes liés à la récursivité dans les expressions du λ-calcul (§ I-3.7.3). Il s'agit en fait d'une question très générale étudiée de manière très abstraite par de nombreuses équipes de mathématiciens [40].

3. Les objets de l'utilisateur et leurs opérateurs forment des algèbres. L'algèbre des entiers relatifs est définie par défaut. Plusieurs algèbres peuvent coexister simultanément.

4. L'évaluation fait partie de la machine abstraite de résolution décrite en section (§ 3).

5. Pour simplifier le langage, les entiers sont aussi utilisés comme booléens.

Les constructeurs des acteurs sont des fonctions SCHEME dont nous faisons terminer le nom par '`:`'[6], comme `alternative:`. Trois sélecteurs sont aussi définis, permettant l'accès à la *condition*, à la clause *alors* et à la clause *sinon* d'une alternative.

Nous avons donc les fonctions SCHEME suivantes :

:

◇ **Actors/alternative**

```
(define (alternative: cond then else)
  (list 'alternative cond then else))

(define condition  cadr)

(define clauseThen caddr)

(define clauseElse cadddr)
```

L'application applique un opérateur à des arguments.

Elle s'écrit : $op(arg_1, arg_2, \ldots, arg_n)$, qui applique l'opérateur *op* aux acteurs arg_i.

Il faut noter que le contrôle des types des arguments est dynamique. Il n'est pas pris en charge par la machine abstraite mais par chaque opérateur (§ 3.3.4). Par la suite, lorsque nous définirons le compilateur de λ-matrice, un contrôle statique des types des arguments sera effectué (§ 5).

En SCHEME, un constructeur et deux sélecteurs manipulent cet acteur. La difficulté avec cet acteur est que le nombre d'arguments est variable. La construction SCHEME `(lambda (x . y) ...)` qui permet de définir des fonctions avec un nombre de paramètres non fixé est utilisée. L'acteur application est une liste formée du symbole quoté `'application`, de l'opérateur et des arguments :

◇ **Actors/application**

```
(define (application: operator . arguments)
  (cons 'application (cons operator arguments)))

(define operator cadr)

(define arguments  cddr)
```

Par convention, dans ce qui suit, l'emploi d'un `'s'` à la fin des symboles indique que ces symboles font référence à des listes. Par exemple, `arguments` dénote la liste des arguments.

La définition permet une association entre un identificateur et une valeur. Elle est écrite : *ident*:=*valeur*, où *ident* est un identificateur et *valeur* un acteur.

Dès lors qu'il existe dans un langage ce type d'association, la notion d'environnement apparaît. Cette notion dynamique est intimement liée à la machine abstraite de résolution et sera décrite dans la section (§ 3.2).

Il faut bien noter qu'une définition dans les λ-matrices n'est pas une affectation : il s'agit clairement d'associer de manière unique un identificateur à une valeur. Ainsi, l'ordre d'apparition des définitions dans un programme n'a pas d'importance et un identificateur peut être utilisé *avant* d'être défini, car la notion *avant* n'a pas de sens ici.

Pour manipuler les définitions, le constructeur et les deux sélecteurs associés sont :

◇ **Actors/definition**

```
(define (definition: identifier value)
  (list 'definition identifier value))

(define identifier cadr)

(define value caddr)
```

6. Ceci n'est pas une contrainte imposée par SCHEME, mais une convention qui permet d'identifier la nature de la fonction avec son nom.

Le flot permet de prendre en compte le temps dans une expression. Il s'écrit : *état* f_{by} *contrat* où *état* et *contrat* sont tous deux des acteurs.

Ses deux constituants sont l'état qui est la valeur initiale du flot et le contrat qui permet de calculer les valeurs suivantes de l'état.

Le flot est la transcription fonctionnelle de la notion de variable des langages impératifs. Dans une même expression il rassemble la valeur de la variable et son devenir. Dans les langages impératifs, l'affectation de la valeur initiale et le devenir de la variable sont répartis dans l'ensemble du programme, rendant la manipulation de la variable complexe. En fait, il est possible de connaître l'état d'un programme impératif, mais cette connaissance engendre un processus de calcul np-complet. La structure de flot permet naturellement de s'affranchir de cette complexité, car à chaque instant, les variables et leur devenir sont parfaitement connus et localisés.

Les fonctions SCHEME associées sont :

◇ **Actors/stream**

```
(define (stream: state contract)
  (list 'stream state contract))

(define state cadr)

(define contract caddr)
```

Le vecteur est l'objet de structuration qui rassemble dans une même structure plusieurs acteurs. Il est écrit : $\left[\begin{smallmatrix} c_1 \\ \dots \\ c_n \end{smallmatrix}\right]$, où $c_1 \dots c_n$ sont des acteurs. Un vecteur a au moins une composante.

Un vecteur joue aussi le rôle d'un cadre d'environnement (§ 3.2) où les associations entre identificateur et valeurs sont créées par les définitions qu'il contient.

Le constructeur SCHEME du vecteur reçoit donc en paramètre la liste des composantes du vecteur. Comme leur nombre n'est pas connu, la notation `(lambda (first . next) ...)` est utilisée car elle impose au moins un argument :

◇ **Actors/vector**

```
(define vector: (lambda (first . next)
        (cons 'vector (cons first next))))

(define components cdr)
```

Une extraction accède à une composante d'un vecteur à l'aide d'un index numérique. Elle est écrite : $indexed \cdot index$.

Le vecteur est soit directement donné, soit nommé à l'aide d'un identificateur. L'index doit être évalué comme un entier. Si la valeur évaluée de l'index est hors limite, la première composante du vecteur est systématiquement retournée, ce qui donne un caractère fonctionnel à l'extraction.

Les fonctions SCHEME associées sont :

◇ **Actors/extraction**

```
(define (extraction: indexed index)
  (list 'extraction indexed index))

(define indexed cadr)

(define index caddr)
```

Bien que la définition suivante concerne tous les acteurs, nous préférons la placer ici, du fait de son caractère général. Par la suite, il sera souvent nécessaire de connaître le type d'un acteur. Pour cela, une fonction est définie en SCHEME par :

◇ **Miscellaneous/type**

```
(define (type actor)
```

```
(cond [(pair? actor) (car actor)]
      [(symbol? actor) (case actor
                         ((bottom top) 'flag)
                         (else 'identifier))]
      [(and (number? actor)
            (exact? actor)) 'integer]
      [(procedure? actor) 'operator]
      [else 'error]))
```

La fonction `type` retourne le type d'un acteur sous la forme d'un symbole quoté. Il détecte si l'acteur est une liste, et dans ce cas, il retourne la tête de la liste qui est le type de l'acteur.

Si l'acteur n'est pas une liste, et s'il est un symbole, l'opérateur retourne `'identifier` si l'acteur n'est ni `'top` ni `'bottom`, et `'flag` dans le cas contraire. Si l'acteur est un entier, `'integer` est retourné et s'il est une fonction SCHEME, `'operator` est retourné.

◇ **Miscellaneous/atom**

```
(define (atom? actor)
  (case (type actor)
   [(flag identifier integer operator) #t]
   [else #f]))
```

La fonction `atom?` retourne `#t` si l'acteur est un atome, et `#f` s'il est un acteur composé.

Dans la section suivante, un exemple simple est programmé à l'aide des λ-matrices, puis en utilisant les constructeurs SCHEME.

2.3 Exemple

Cette section montre l'utilisation des λ-matrices pour programmer une petite application de traitement du signal. L'exemple traité ici est un filtre récursif du second ordre dont le schéma est dans la figure 2.1. Cet exemple sera traité tout au long de la thèse, pour aboutir finalement à sa réalisation sur l'architecture parallèle statique.

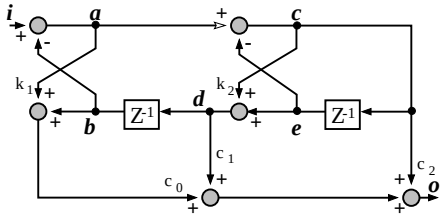


FIG. 2.1 - *Filtre récursif du second ordre.*

A partir de ce diagramme, les équations en Z du filtre sont obtenues après une simple lecture. Les coefficients k_1, k_2, c_1 et c_2 valent 1, pour simplifier. Il vient :

$$\begin{cases} o = a + b + c + d \\ a = i - b \\ b = Z^{-1}d \\ c = a - e \\ d = c + e \\ e = Z^{-1}c \end{cases} \tag{2.8}$$

La fonction de transfert du filtre est déduite des équations (2.8). Elle est :

$$H(Z) = \frac{O(Z)}{I(Z)} = \frac{3+3Z^{-1}+Z^{-2}}{1+2Z^{-1}+Z^{-2}}. \tag{2.9}$$

La λ-matrice correspondant à ce filtre est la suivante :

$$\left[\begin{array}{l} o{:=}+(+(+(a,\,b),\,c),\,d) \\ a{:=}-(i,\,b) \\ b{:=}0\,f_{by}\,d \\ c{:=}-(a,\,e) \\ d{:=}+(c,\,e) \\ e{:=}0\,f_{by}\,c \end{array}\right]. \tag{2.10}$$

Elle est aussi obtenue après simple lecture du diagramme. L'opérateur de retard unitaire Z^{-1} est remplacé par un flot de valeur initiale 0. Il faut noter la grande similitude entre la λ-matrice et les équations (2.8).

En utilisant les constructeurs SCHEME définis précédemment, le programme suivant est obtenu :

◇ **example/filter**

```
(define filter
  (vector:
    (definition: 'o (application: + 'a
                      (application: + 'b
                        (application: + 'c 'd))))
    (definition: 'a (application: - 'i 'b))
    (definition: 'b (stream: 0 'd))
    (definition: 'c (application: - 'a 'e))
    (definition: 'd (application: + 'c 'e))
    (definition: 'e (stream: 0 'c))))
```

Nous avons défini la fonction SCHEME `convert` qui permet d'obtenir la représentation syntaxique d'un programme à partir de sa représentation interne. Par exemple, il vient :

```
STk> (convert filter)
[
  o := + (a, + (b, + (c, d)))
  a := - (i, b)
  b := 0 fby d
  c := - (a, e)
  d := + (c, e)
  e := 0 fby c
]
```

`STk` représente l'invite de l'interprète SCHEME utilisé (ici, STK de ERIC GALLESIO [37]). `(convert filter)` est la commande donnée à l'interprète, suivie de la touche de validation, qui apparaît comme un retour à la ligne. Les lignes suivantes sont la réponse de l'interprète. Nous reconnaissons les acteurs décrits dans la section précédente, comme la définition, l'application ou bien le flot.

Dans les chapitres qui suivront, nous verrons comment résoudre ce filtre (§ 3), comment établir les déterminismes temps/ressources (§ 4), et le compiler (§ 5).

Chapitre 3

Résolution

Ce chapitre définit la machine fonctionnelle abstraite de résolution des λ-matrices. Cette machine est l'aspect dynamique de ce formalisme, et le langage en est l'aspect statique.

La machine est construite à l'aide de définitions récursives dans le sens où le terme défini apparaît dans le corps de l'expression. Elles restent cependant valides, d'un point de vue mathématique, car le terme définissant est «plus simple» que le terme défini [40].

Dans un premier temps, trois opérateurs de base sont définis (§ 3.1), ainsi que l'importante notion des environnements (§ 3.2). Enfin, les quatre opérateurs de résolution sont décrits, ainsi que leur équivalent en SCHEME (§ 3.3).

3.1 Opérateurs de base

Les trois combinateurs définis dans cette section permettent d'obtenir une composante d'un vecteur, la dimension d'un acteur et l'appartenance d'un acteur à l'ensemble des acteurs figés. Ils représentent bien la manière avec laquelle cette étude est menée.

D'un point de vue mathématique, ces opérateurs récursifs sont-ils définissables [40] ? Pour qu'ils le soient, il est nécessaire que les arguments de chaque récursion soient de «plus en plus simples». Or ces opérateurs manipulent des acteurs construits à l'aide de la grammaire inductive des λ-matrices. Il est acquis qu'une telle grammaire construit des objets mathématiques ordonnés par une relation de composition. Nous avons donc, en première approximation, dire que ces opérateurs sont définissables, au sens mathématique. Une étude poussée serait intéressante, mais elle sort du cadre de cette thèse. Cependant, nous nous poserons quand même la question d'une existence mathématique de chaque opérateur récursif défini dans ce travail.

3.1.1 Composante d'un vecteur

L'opérateur décrit ici permet d'accéder à une composante d'un vecteur en donnant son index. Cette fonction est définie par cas. Deux cas sont distingués : 1– l'acteur indexé est un vecteur et l'index est un entier, et 2– les autres cas :

$$\delta : \mathcal{X} \times \mathcal{X} \longrightarrow \mathcal{X} \quad | \tag{3.1}$$

$$\delta(\begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}, i) = \begin{cases} c_i, \text{ si } i \in \mathcal{Z} | i \in [1, n], \\ \bot, \text{ sinon.} \end{cases} \quad \text{(vecteur)}$$

$$\delta(x, i) = \bot \quad \text{(cas général)}$$

Ce format pour définir les opérateurs algébriques va être utilisé tout au long de cette étude théorique. Dans la partie haute de la définition, il y a la signature de l'opérateur : dans ce cas, il opère sur deux acteurs ($\mathcal{X} \times \mathcal{X}$) et son résultat est un acteur ($\longrightarrow \mathcal{X}$).

Ensuite, nous avons une succession de cas dont l'intitulé apparaît entre parenthèses sur la droite. Lorsque qu'un acteur répond à plus d'un cas, par exemple avec les cas *atome* et *identificateur*, le cas le plus spécialisé doit être choisi.

Le premier cas est défini lorsque le premier argument est un vecteur de la forme $\begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$. Dans ce cas, le résultat dépend de la nature de l'index i : s'il est un entier compris entre 1 et le nombre de composantes du vecteur, le résultat est la i^e composante, et l'indicateur $\perp$ sinon. Le second cas concerne tous les autres acteurs et le résultat est toujours $\perp$.

La définition de cet opérateur en SCHEME est :

◇ **Miscellaneous/indexOf**

```
(define (indexOf indexed index)
  (if (and (eq? (type indexed) 'vector)
           (eq? (type index)   'integer)
           (> index 0)
           (<= index (length (components indexed))))
   (list-ref indexed index)
   'bottom))
```

Étant donné le nombre peu important de cas distingués dans cette fonction, la construction `if` de SCHEME est préférée aux constructions `cond` ou `case`.

La fonction `type` définie ultérieurement (§ 2.2) retourne le type d'un acteur sous la forme d'un symbole quoté. Le type de l'acteur est comparé à un symbole quoté comme `'vector` avec la fonction SCHEME standard `eq?`. Le fait que l'opérateur `and` de SCHEME soit une forme spéciale est utilisé ici pour ne comparer que des index entiers à `0` ou au nombre des composantes du vecteur. L'indicateur `'bottom` est retourné si une composante ne peut être retournée.

3.1.2 Dimension d'un acteur

La dimension d'un acteur représente l'espace abstrait nécessaire pour le décrire. Par convention, la dimension d'un atome est 1. La dimension d'un objet composé est la somme des dimensions de ses composantes. Ainsi, la dimension de $e \to 0, (123\ f_{by}\ 456)$ est 4, la somme des dimensions de tous les atomes de l'expression. L'opérateur est défini par :

$$
\begin{aligned}
&|\,|\,: \mathcal{X} \longrightarrow \mathcal{Z} \quad | && (3.2)\\
&|x| = 1, && \text{(atome)}\\
&|\{c_1, c_2, \ldots, c_n\}_t| = \sum_{i=1}^{n} |c_i|\,, && \text{(cas général)}
\end{aligned}
$$

Il s'agit d'une fonction à un argument qui retourne un entier. La dimension d'un atome est 1, même s'il s'agit d'une donnée définie par l'utilisateur. Dans le cas général, cette définition utilise l'écriture générique $\{c_1, c_2, \ldots, c_n\}_t$ qui indique qu'il s'agit d'un objet de type t et dont les composantes sont $c_1, c_2, \ldots, c_n$ (§ 2.1). Dans ce cas, le résultat est la somme des dimensions des composantes. Ici, la nature algébrique des acteurs leur conférant la propriété d'être acycliques est importante pour que cette expression soit calculable.

La définition SCHEME de la dimension est :

◇ **Miscellaneous/dimension**

```
(define (dimension actor)
  (if (atom? actor)
      1
      (apply + (map dimension (cdr actor)))))
```

Dans cette fonction, la structure de liste des acteurs composés est utilisée (§ 2.2). En effet, un acteur est une liste dont le premier élément est un type, et dont les autres éléments sont des composantes (§ 2.2). La fonction standard SCHEME `map` construit la liste des résultats de l'application de son premier argument (`dimension`) à chaque élément de la liste en second

argument (`cdr actor`). Le résultat est donc la liste des dimensions des composantes de l'acteur dont il reste à effectuer la somme en lui appliquant l'opérateur `+` à l'aide de la fonction standard `apply`.[1]

3.1.3 Appartenance à l'ensemble des acteurs figés

L'ensemble des acteurs figés $\overline{\mathcal{X}}$ contient les atomes sans les identificateurs, et les vecteurs dont les composantes sont des acteurs figés (§ 2.1). Dans le formalisme fonctionnel, ceci s'écrit simplement avec le symbole d'appartenance, comme $x \in \overline{\mathcal{X}}$. Cependant, nous avons besoin de définir une fonction SCHEME vérifiant cette appartenance :

◇ **Miscellaneous/frozen**

```
(define (frozen? actor)
  (case (type actor)
    [(integer flag operator) 1]
    [vector
     (apply * (map frozen? (components actor)))]
    [else 0]))
```

Cet opérateur retourne `1` si l'acteur est figé et `0` s'il ne l'est pas. Cette fois-ci, la construction SCHEME `case` est utilisée. Elle permet de discerner simplement les différents cas.

3.2 Environnement

Les environnements sont définis dans les λ-matrices de manière fonctionnelle. Ils donnent au langage une intéressante possibilité de structuration et de programmation modulaire, bien que primitive.

La programmation modulaire nécessite les notions d'identificateur et d'environnement. D'une manière générale, un environnement est une structure qui contient des associations entre des identificateurs et des valeurs. Un sous-environnement est un environnement inclus dans un autre, appelé environnement parent. Une valeur peut être retrouvée avec un identificateur si l'association existe dans l'environnement courant ou dans les environnements parents.

Les identificateurs peuvent être liés aux valeurs de deux manières, par liaison dynamique ou statique. Dans le cas d'une liaison dynamique, la valeur liée à un identificateur est évaluée dans l'environnement où l'identificateur est utilisé. Avec une liaison statique, la valeur liée à un identificateur est évaluée dans l'environnement qui contient l'association. Les langages LISP ou TEX utilisent la première méthode, qui ne permet pas une compilation efficace [5, 47]. Les λ-matrices utilisent, quant à elles, la liaison statique, comme SCHEME [1, 18, 65] ou C.

Les environnements sont construits par une succession de cadres qui contiennent les associations identificateurs-valeurs : un environnement est une liste de cadres. Une bonne explication de cette structure peut être trouvée dans le livre de ABELSON et de SUSMAN [1].

L'objectif de cette section est de donner une description purement fonctionnelle des environnements et des opérateurs associés.

Tous les opérateurs sont mathématiquement définissables bien qu'ils soient récursifs. En effet, les arguments de chaque récursion sont «de plus en plus simples» par rapport à la relation d'ordre introduite par la grammaire inductive des acteurs.

3.2.1 Liaison statique contre liaison dynamique

Avant d'entamer la description des environnements, précisons à l'aide d'un exemple la différence entre les deux modes de liaison des variables dans un environnement.

Soit le programme SCHEME :

```
(define libre 0)
```

1. Cette application est possible car l'opérateur SCHEME `+` a un nombre variable d'arguments.

```
(define (mul a) (* a libre))

(let ((libre 100))
  (display (mul 1)))
```

La définition de la fonction `mul` a un paramètre `a`; elle retourne la multiplication de `a` et de la variable `libre`. La troisième écriture définit la variable `libre` dans l'environnement du `let` et applique l'opérateur standard `display` au résultat de `(mul 1)`.

Bien évidemment, la valeur affichée est `0` car SCHEME utilise la liaison statique. Cela signifie que les variables libres sont liées au plus tôt à une valeur. En l'occurrence, la variable libre de la fonction `mul` est liée à la variable globale `libre`. Le fait de définir `libre` à l'intérieur du `let` n'a aucun effet.

Si SCHEME utilisait la liaison dynamique, le résultat affiché serait 100. Cela signifie que les variables sont liées à une valeur le plus tard possible.

L'avantage de la liaison statique est de permettre une compilation plus efficace [1, 5]. De plus, dans ce cas, la définition d'un module est déterministe car les variables libres doivent être définies quelque part dans l'environnement du module. Avec la liaison dynamique, le comportement d'un module peut être différent selon l'environnement dans lequel il est utilisé.

3.2.2 Définition

Dans les λ-matrices, les vecteurs sont désignés comme étant les cadres des environnements. En effet, pourquoi définir une autre structure pour rassembler des acteurs alors que celle-ci existe déjà. Un environnement est une liste éventuellement vide de cadres. L'ensemble des environnements est défini par :

$$\forall E \in \Gamma, \begin{cases} E = \{v_k, \ldots, v_2, v_1\} \mid v_k, \ldots, v_1 \in \mathcal{V} \\ E = v \in \mathcal{V} \\ E = \bot \end{cases} \tag{3.3}$$

où $\bot$ dénote l'environnement vide.

Pour simplifier les écritures, un environnement E est noté $v_k \star \ldots \star v_2 \star v_1$ où les v_i où les v_i sont ses cadres, qui sont eux-mêmes des vecteurs. Par abus de notation, un vecteur isolé est considéré comme un environnement à un seul cadre.

Si $E_k = v_k \star \ldots \star v_2 \star v_1$ est l'environnement d'un acteur a, alors $E_{k-1} = v_{k-1} \star \ldots \star v_2 \star v_1$ est son environnement parent. L'environnement parent d'un environnement ayant un seul cadre est l'environnement vide, et l'environnement parent de l'environnement vide est l'environnement vide.

La structure SCHEME utilisée pour construire les environnements est bien évidemment la liste, qui sera dans ce cas une liste de vecteurs. L'environnement vide est dénoté par `'bottom`. L'opérateur SCHEME qui permet de construire un environnement est :

◇ **Environment/chain**

```
(define (chain: vector env)
  (cons vector env))
```

Il s'agit simplement du constructeur de liste `cons`. Remarquons qu'aucun contrôle n'est effectué sur la nature du cadre pour garder la simplicité du programme.

Nous aurons aussi besoin d'accéder à l'environnement parent d'un environnement. Nous définissons pour cela :

◇ **Environment/parent**

```
(define (parent env)
  (if (eq? env 'bottom) 'bottom (cdr env)))
```

De plus, il sera souvent nécessaire d'accéder au premier cadre d'un environnement, et nous définissons :

◇ **Environment/frame**

```
(define (frame env)
  (if (eq? env 'bottom) 'bottom (car env)))
```

Ainsi, les environnements définissent un paquetage logiciel indépendant de leur réalisation particulière, dont l'interface est le constructeur `chain` et les deux sélecteurs `parent` et `frame`.

3.2.3 Valeur associée ρ

Étant donnés un identificateur et un environnement, il est possible de retrouver la valeur associée à cet identificateur dans cet environnement. La définition de cet opérateur est :

$$\rho : \mathcal{I} \times \Gamma \longrightarrow \mathcal{X} \quad | \tag{3.4}$$

$$\rho(n, \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_k \end{smallmatrix}\right] \star E) \qquad \text{(cas général)}$$

$$= \begin{cases} v_i, \text{ si } i \text{ est le plus petit } i \text{ dans } [1,k] \text{ tel que } c_i \text{ est } \ n{:=}v_i, \\ \rho(n, E), \text{ sinon,} \end{cases}$$

$$\rho(n, \bot) = \bot. \qquad \text{(env. vide)}$$

Cette fonction attend un identificateur et un environnement, et elle retourne un acteur. Une définition avec l'identificateur correspondant est recherchée dans le cadre le plus à gauche de l'environnement. Si elle est trouvée, sa valeur associée est retournée. Si elle n'est pas trouvée, la recherche se poursuit dans l'environnement parent. Notons que dans cette écriture, l'environnement E peut être l'environnement vide.

On peut remarquer que si un identificateur est défini avec plusieurs définitions dans le même cadre, seule la première est considérée (celle avec le plus petit index). De même, les définitions qui ne sont pas des composantes directes d'un cadre sont elles aussi ignorées. Par exemple, il est absurde d'écrire $0\, f_{by}\ i{:=}3$ car la définition de i est ignorée.

Cela se traduit en SCHEME par deux opérateurs : un opérateur de recherche général qui est aussi utilisé par la fonction définie dans la section suivante, et l'opérateur de recherche proprement dit :

◇ **Environment/find**

```
(define (find commande ident env)
  (if (eq? env 'bottom)
      'bottom
      (let loop ([comps (components (frame env))])
        (if (null? comps)
            (find commande ident (parent env))
            (let ([actor (car comps)])
              (if (and (eq? 'definition (type actor))
                       (eq? ident (identifier actor)))
                  (if (eq? commande 'valueOf)
                      (value actor) env)
                  (loop (cdr comps))))))))
```

Cette fonction générale recherche `ident` dans l'environnement `env`. Le paramètre `command` peut prendre comme valeur les symboles quotés `'valueOf` ou `'assocOf`. La fonction retourne la valeur associée à l'identificateur si elle est trouvée, et l'environnement associé autrement.

Elle est construite autour d'un `let` nommé qui parcourt toutes les composantes du cadre de tête, à la recherche d'une définition ayant l'identificateur `ident`. Si elle est trouvée, la fonction retourne la valeur associée si `command` vaut `valueOf` et l'environnement si `command` vaut `assocOf`. Si le cadre est vide, la recherche se poursuit dans l'environnement parent. Si l'environnement est vide, `'bottom` est retourné.

La définition de la fonction de recherche est très simplifiée par l'utilisation de `find` qui est invoquée en donnant à son paramètre `command` la valeur `'valueOf`. Ceci a pour effet de retourner la valeur associée à l'identificateur.

◇ **Environment/valueOf**

```
(define (valueOf ident env)
  (find 'valueOf ident env))
```

3.2.4 Environnement associé μ

Étant donnés un identificateur et un environnement, cette fonction permet de retrouver l'environnement où l'identificateur a été défini. Cet environnement de définition est appelé environnement lexical d'un identificateur. L'opérateur est défini avec l'expression :

$$\mu : \mathcal{I} \times \Gamma \longrightarrow \Gamma \quad | \tag{3.5}$$

$$\mu(n, \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_k \end{smallmatrix}\right] \star E) \qquad \text{(cas général)}$$

$$= \begin{cases} \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_k \end{smallmatrix}\right] \star E, \text{ si au moins un } c_i \text{ est } n{:=}v, \\ \mu(n, E), \text{ sinon,} \end{cases}$$

$$\mu(n, \bot) = \bot. \qquad \text{(env. vide)}$$

La fonction a comme paramètre un identificateur et un environnement. Elle retourne un environnement. Si le cadre le plus à gauche définit l'identificateur, alors la totalité de l'environnement est retournée. Dans le cas contraire, la recherche se poursuit dans l'environnement parent. Si l'environnement est vide, l'environnement vide est retourné.

La fonction SCHEME définissant cet opérateur est :

◇ **Environment/assocOf**

```
(define (assocOf ident env)
  (find 'assocOf ident env))
```

Là aussi, l'utilisation de la fonction `find` avec `'assocOf` comme argument pour le paramètre `command` simplifie grandement la définition.

3.2.5 Extension d'un environnement η

Cette fonction étend un environnement avec une nouvelle association identificateur-valeur. Si l'environnement est l'environnement vide, un nouvel environnement est créé. Dans le cas contraire, l'association est placée en tête du premier cadre de l'environnement, à l'aide d'une définition. L'ajout de la définition peut occulter une définition précédente dans l'environnement.

La définition est :

$$\eta : \mathcal{I} \times \mathcal{X} \times \Gamma \longrightarrow \Gamma \quad | \tag{3.6}$$

$$\eta(s, v, E) = \left[\begin{smallmatrix} s{:=}v \\ c_1 \\ \cdots \\ c_n \end{smallmatrix}\right] \star E_0, \text{ avec } E = \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix}\right] \star E_0 \qquad \text{(cas général)}$$

$$\eta(s, v, \bot) = [\, s{:=}v] \qquad \text{(env. vide)}$$

Cette fonction attend un identificateur, un acteur et un environnement, et retourne un environnement. La définition composée de l'identificateur et de l'acteur est placée en tête du cadre le plus à gauche, si l'environnement n'est pas vide. Si l'environnement est vide, un nouvelle environnement contenant la définition est retourné.

Le fonction SCHEME de cet opérateur est :

◇ **Environment/extend**

```
(define (extend ident value env)
  (let ([newDef (definition: ident value)])
    (if (eq? env 'bottom)
        (chain: (vector: newDef) 'bottom)
        (let ([comps (components (frame env))])
          (chain: (apply vector: (cons newDef comps))
                  (parent env))))))
```

La fonction commence par créer une nouvelle définition. Si l'environnement est vide, un vecteur contenant la définition comme unique composante est créé, puis il est chaîné à l'environnement vide pour créer un nouvel environnement. Si l'environnement n'est pas vide, la définition est ajoutée en tête des composantes du vecteur le plus à gauche ; le nouveau vecteur ainsi constitué est chaîné à l'environnement parent pour former un nouvel environnement qui est retourné.

3.2.6 Redéfinition dans un environnement $\bar{\eta}$

Cette fonction remplace la valeur associée à un identificateur dans un environnement par une nouvelle valeur et retourne l'environnement ainsi modifié. Il est défini par :

$$\bar{\eta} : \mathcal{I} \times \mathcal{X} \times \Gamma \longrightarrow \Gamma \quad | \tag{3.7}$$

$$\bar{\eta}(s, x, v \star E) = \begin{cases} v \star \bar{\eta}(s, x, E), \text{ si } \rho(s, v \star \bot) = \bot, & \text{(cas général)} \\ v' \star E, \text{ avec } v' = v \text{ où } x \text{ remplace la valeur de } s. \end{cases}$$

$$\bar{\eta}(s, x, \bot) = \bot, \qquad \text{(env. vide)}$$

Cette fonction attend un identificateur, une valeur et un environnement pour retourner un environnement. Si l'environnement est vide, alors l'environnement vide est retourné. Si l'identificateur n'est pas défini dans le cadre le plus à gauche, le remplacement se poursuit dans l'environnement parent. L'environnement retourné est constitué du cadre non modifié et du résultat du remplacement. Si l'identificateur est défini, alors le cadre le plus à gauche est modifié de manière à ce que la valeur associée à l'identificateur soit la valeur en paramètre. L'environnement retourné est constitué du cadre modifié et de l'environnement parent non modifié.

En SCHEME, il est défini par :

◇ **Environment/assign**

```
(define (assign ident value env)
  (if (eq? env 'bottom)
      'bottom
      (if (eq? (valueOf ident (chain: (frame env) 'bottom))
               'bottom)
          (chain: (frame env) (assign ident value (parent env)))
          (chain: (apply vector:
                         (map (lambda (comp)
                                (if (and (eq? (type comp) 'definition)
                                         (eq? (identifier comp) ident))
                                    (definition: ident value)
                                    comp))
                              (components (frame env))))
                  (parent env)))))
```

L'écriture de cette fonction ne présente pas de difficulté particulière : elle est calquée sur sa définition formelle. Nuos noterons que l'opération est purement fonctionnelle et qu'elle est effectuée sans affectation (`set!`).

3.2.7 Recherche de l'identificateur associé

Jusqu'à maintenant, nous nous sommes plus intéressés à la valeur associée à un identificateur, connaissant cet identificateur. Nous aurons besoin de connaître l'identificateur associé à une valeur, connaissant cette valeur lors de la définition du compilateur formel (§ 5).

La définition formelle de cet opérateur est :

$$\kappa : \mathcal{X} \times \Gamma \longrightarrow \Gamma \quad | \tag{3.8}$$

$$\kappa(x, \left[\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}\right] \star E) \qquad \text{(cas général)}$$

$$= \begin{cases} s_i, \text{ si } i \text{ est le plus petit } i \text{ dans } [1,k] \text{ tel que } c_i \text{ est } \ s_i := x, \\ \kappa(x, E), \text{ sinon,} \end{cases}$$

$$\kappa(x, \bot) = \bot, \qquad \text{(env. vide)}$$

Cette fonction recherche dans le cadre le plus à gauche si une définition définit l'acteur passé en argument. Si c'est le cas, elle retourne l'identificateur associé. Dans le cas contraire, la fonction poursuit sa recherche dans l'environnement parent. Si l'environnement est vide, $\bot$ est retourné.

La définition SCHEME de cet opérateur est :

◇ **Environment/identOf**

```
(define (identOf val env)
  (if (eq? env 'bottom)
      'bottom
      (let loop ([comps (components (frame env))])
        (if (null? comps)
            (identOf value (parent env))
            (let ([def (car comps)])
              (if (and (eq? 'definition (type def))
                       (equal? val (value def)))
                  (identifier def)
                  (loop (cdr comps))))))))
```

Cette fonction reprend en tout point la définition formelle. L'opérateur SCHEME `equal?` permet la comparaison des structures élément par élément.

3.3 Machine fonctionnelle abstraite

Nous savons maintenant comment écrire des λ-matrices (§ 2.1), créer et manipuler des environnements (§ 3.2). Mais cela ne nous donne toujours pas la sémantique du langage, c'est à dire le comportement des acteurs.

Cette sémantique est décrite à l'aide d'une machine abstraite agissant un peu à la manière d'un interprète. Comme il est dit dans l'introduction de cette étude théorique (§ 1), la machine est un opérateur fonctionnel à récursion terminale mettant en œuvre trois autres opérateurs.

Ces opérateurs sont dits *dynamiques* car ils agissent sur les acteurs en construisant leur environnement courant au fur et à mesure. L'association entre un acteur dans le programme et son environnement est effectuée par ces opérateurs de manière dynamique.

La résolution d'un programme ne vérifie pas sa justesse : elle agit à la manière d'un interprète. Lorsqu'une erreur est rencontrée, une valeur spéciale est propagée sur tous les résultats intermédiaires. La vérification des programmes sera effectuée par les fonctions de critère définies dans le chapitre (§ 4).

3.3.1 Opérateur de résolution ▷

Un programme écrit à l'aide des λ-matrices est un vecteur dans ce langage. Il est résolu à l'aide d'un combinateur. Ce combinateur engendre un processus itératif, qui peut être

écrit de manière fonctionnelle à l'aide d'une récursion terminale. L'itération est interrompue lorsqu'une condition d'arrêt optionnelle est rencontrée.

La définition du combinateur de résolution est à limite des mathématiques car il s'agit d'une définition récursive dont le terme définissant n'est pas «plus simple» que le terme défini. Nous prendrons cette définition plus comme «un processus de calcul» que comme une définition mathématique. Cependant, le combinateur de résolution repose sur des combinateurs qui sont, eux, mathématiquement définissables bien que récursifs.

Résoudre le système S s'écrit :

$$\triangleright_{stop\ E}\ S, \tag{3.9}$$

où S est le système à résoudre et *stop* un entier jouant le rôle de la condition d'arrêt[2] qui est atteinte lorsque la valeur à l'index *stop* dans le système est nulle, et où E est l'environnement de S. Cela peut être écrit à l'aide d'une alternative :

$$(\triangle_E\ S \cdot stop) \rightarrow S, \text{ continue ...} \tag{3.10}$$

où $\triangle$ est l'opérateur d'évaluation (§ 3.3.3). Cette expression est très proche de la construction `while-do` des langages traditionnels. Par convention, le système lui-même est retourné lorsque la condition d'arrêt est rencontrée.

L'expression *continue* est la résolution du système dans lequel tous les états des flots ont été mis à jour d'une manière synchrone par l'opérateur de régénération (§ 3.3.2). L'expression générale de l'opérateur de résolution est :

$$\begin{array}{l} \triangleright : \mathcal{X} \times \mathcal{Z} \times \Gamma \longrightarrow \mathcal{X} \quad | \\ \triangleright_{i\ E}\ s = (\triangle_E\ s \cdot i) \rightarrow (s), (\triangleright_{i\ E}\ \bigtriangledown_E\ s) \end{array} \tag{3.11}$$

Cette fonction a comme paramètres un acteur, un entier et un environnement, et elle retourne un acteur. Elle est bâtie autour d'une alternative, comme nous l'avons vu au-dessus. La condition de cette alternative est l'évaluation de l'extraction du système s par la condition d'arrêt i, qui s'écrit $\triangle_E\ s \cdot i$. Si l'évaluation de cette condition est différente de l'entier 0, le système s est le résultat de la résolution. Sinon, le résultat est la résolution du système régénéré, qui s'écrit $\bigtriangledown_E\ s$. L'évaluation et la régénération sont décrites dans les sections suivantes.

L'usage de l'environnement E dans l'expression de la résolution est une extension des λ-matrices qui n'est pas actuellement utilisée : elle permettra de résoudre des acteurs à l'intérieur d'un système et donc d'écrire des équations de point-fixe. Dans ce cas, l'acteur doit être considéré dans son environnement courant, qui est noté E. Dans cette étude, ce paramètre sera toujours l'environnement vide, ce qui donne l'écriture :

$$\triangleright_{i\perp}\ s = (\triangle_\perp\ s \cdot i) \rightarrow s, (\triangleright_{i\perp}\ \bigtriangledown_\perp\ s) \tag{3.12}$$

La résolution d'un système s_1 engendre donc les étapes suivantes :

$$\begin{array}{rcl} \triangleright_{i\perp}\ s_1 & = & \triangleright_{i\perp}\ s_2 \\ & = & \triangleright_{i\perp}\ s_3 \\ & = & \ldots \\ & = & \triangleright_{i\perp}\ s_k \\ & = & s_k \end{array} \tag{3.13}$$

2. $\triangleright_{stop\ E}\ S$ peut être lu $\triangleright(stop, E, S)$. Dans ce type d'écriture, l'argument principal est au même niveau que l'opérateur, et les arguments secondaires sont des indices de l'opérateur. Cette écriture évite l'usage intensif des parenthèses qui rendent les écritures difficiles à lire, et attire l'attention sur le paramètre principal, ici S.

avec $s_i = \bigtriangledown_\perp s_{i-1}$, $\bigtriangleup_\perp\ s_{1,\ldots,k-1} \cdot i = 0$ et $\bigtriangleup_\perp\ s_k \cdot i \neq 0$. Notons que k peut être l'infini dans le cas où la condition d'arrêt n'est jamais atteinte. Nous remarquons que ce calcul ressemble fort à la résolution d'une équation de point-fixe, qui converge ou non.

Le monde extérieur ne peut être modélisé de manière fonctionnelle, principalement à cause des effets de bords liés aux entrées/sorties[3]. Cependant, il est possible d'en donner une vision fonctionnelle à l'intérieur d'un système en définissant les entrées globales comme des flots de données, par exemple $input_i := e_{i_0}\ f_{by}\ e_{i_1}\ f_{by}\ \ldots\ f_{by}\ e_{i_\alpha}$.

La traduction en SCHEME de l'opérateur de résolution suit de très près sa définition formelle: nous retrouvons la récursion terminale bâtie autour d'une alternative:

◇ **Machine/solve**

```
(define (solve system stop env)
  (if (zero? (evaluate (extraction: system stop) env))
      (solve (regenerate system env) stop env)
      system))
```

Nous définissons maintenant les autres opérateurs participant à la résolution des systèmes.

3.3.2 Régénération $\bigtriangledown$

Cet opérateur régénère les états des flots d'une manière synchrone. Sa définition est:

$$\bigtriangledown : \mathcal{X} \times \Gamma \longrightarrow \mathcal{X} \quad | \tag{3.14}$$

$$\bigtriangledown_E\ x = x, \tag{atome}$$

$$\bigtriangledown_E\ \{c_1, c_2, ..., c_n\}_t = \{\bigtriangledown_E\ c_1,\ \bigtriangledown_E\ c_2, ...,\ \bigtriangledown_E\ c_n\}_t, \tag{cas général}$$

$$\bigtriangledown_E\ s\ f_{by}\ c = \bigtriangleup_E\ c\ f_{by}\ \bigtriangledown_E\ c, \tag{flot}$$

$$\bigtriangledown_E \begin{bmatrix} c_1 \\ \ldots \\ c_k \end{bmatrix} = \begin{bmatrix} \bigtriangledown_{E'}\ c_1 \\ \ldots \\ \bigtriangledown_{E'}\ c_k \end{bmatrix}, \text{ avec } E' = \begin{bmatrix} c_1 \\ \ldots \\ c_k \end{bmatrix} \star E. \tag{vecteur}$$

Cette fonction a comme paramètre un acteur et un environnement. Elle retourne un acteur. Quatre cas sont distingués: les atomes, les flots, les vecteurs et les autres acteurs. Le régénéré d'un atome est lui-même. Le régénéré d'un flot est un flot dont l'état est le résultat de l'évaluation de son contrat et dont le contrat est le régénéré du contrat. Le régénéré d'un vecteur est un vecteur dont les composantes sont les régénérées de ses composantes dans l'environnement étendu par le vecteur. Le régénéré des autres acteurs[4] sont des acteurs de même type dont les composantes sont régénérées.

La sémantique du flot des λ-matrices apparaît dans l'expression de sa régénération. L'état est remplacé par l'évaluation du contrat dans l'environnement courant, ce qui provoque un retard (§ 2.2). Il faut noter que seuls les états des flots sont modifiés par la régénération, les autres composantes du programme demeurant identiques. Cette remarque aura son importance lors des vérifications formelles (§ 6).

Du point de vue des mathématiques, le combinateur de régénération est définissable si le combinateur d'évaluation l'est aussi. En effet, la récursion apparente dans la définition de ce combinateur opère sur des arguments «de plus en plus simples», ordonnés par la grammaire inductive qui les a construits. L'environnement n'est pas directement utilisé par ce combinateur, mais il l'est par le combinateur d'évaluation, lors de la régénération des flots.

La traduction en SCHEME de l'opérateur de régénération est une fonction par cas:

◇ **Machine/regenerate**

```
(define (regenerate actor env)
  (case (type actor)
    [(flag identifier integer operator)
     actor]
```

3. Cette remarque doit être pondérée par les études actuelles portant sur les *monades* et qui montrent comment modéliser ce type d'effets de bords de manière fonctionnelle [36, 48, 63].

4. Les autres acteurs sont notés $\{c_1, c_2, \ldots, c_n\}_t$, selon la syntaxe introduite dans la section (§ 2.1).

```
[stream
 (stream:
  (evaluate   (contract actor) env)
  (regenerate (contract actor) env))]
[vector
 (apply vector:
   (regenerate (components actor)
     (chain: actor env)))]
[else
 (map (lambda (actor)
   (regenerate actor env)) actor)]))
```

On retrouve les quatre cas de la définition formelle. Remarquons cependant le cas général `else` qui fait appel à la fonction standard `map`. Celle-ci applique son premier argument aux composantes de la liste en second argument et retourne la liste formée des résultats de cette application. La liste est, dans ce cas, un acteur dont la première composante est son type qui est un symbole quoté. Cette utilisation de `map` est possible ici parceque la régénération des symboles a pour résultat ces symboles: la liste reconstruite possède donc toujours en première position le type de l'acteur initial, car le type de l'acteur est considéré comme un symbole.

Le rôle principal des vérifications est de comparer un système avec le résultat de sa régénération dans le but d'établir certaines propriétés. Pour cela, nous utilisons une méthode inductive sur tous les acteurs du langage. L'environnement courant E d'un acteur particulier dans le système initial n'est pas le même que l'environnement courant de l'acteur correspondant dans le système régénéré. Ce dernier est appelé environnement régénéré et il se déduit du premier. Il est noté $\overset{\triangledown}{E}$ et il est défini par:

$$\begin{array}{l} \forall E_n \in \Gamma \quad | \quad E_n = e_n \star \ldots \star e_2 \star e_1, \\ \overset{\triangledown}{E}_n = \overset{\triangledown}{e}_n \star \ldots \star \overset{\triangledown}{e}_2 \star \overset{\triangledown}{e}_1 \quad | \quad \overset{\triangledown}{e}_\alpha = \bigtriangledown_{\overset{\triangledown}{e}_{\alpha-1} \star \cdots \star \overset{\triangledown}{e}_2 \star \overset{\triangledown}{e}_1} e_\alpha. \end{array} \tag{3.15}$$

Les environnements régénérés vont être utilisés pour établir les propriétés des fonctions de critères (§ 4).

3.3.3 Évaluation $\triangle$

L'opérateur d'évaluation donne une valeur à chaque acteur. Son expression formelle est:

$$\triangle : \mathcal{X} \times \Gamma \longrightarrow \overline{\mathcal{X}} \quad | \tag{3.16}$$

$$\triangle_E\ c \to t, e = \begin{cases} \bot, \text{ si } \triangle_E\ c = \bot, \\ \triangle_E\ t, \text{ si } \triangle_E\ c \neq 0, \\ \triangle_E\ e, \text{ sinon}, \end{cases} \quad \text{(alternative)}$$

$$\triangle_E\ o(a_1, a_2, \ldots, a_k) = \triangleleft\, o(\triangle_E\ a_1, \triangle_E\ a_2, \ldots, \triangle_E\ a_k), \quad \text{(application)}$$

$$\triangle_E\ x = x, \quad \text{(atome)}$$

$$\triangle_E\ i{:=}v = \triangle_E\ v, \quad \text{(définition)}$$

$$\triangle_E\ u \cdot i \quad \text{(extraction)}$$

$$= \begin{cases} \delta(i', u'), \text{ si } u' = \triangle_E\ u \in \mathcal{V} \text{ et } i' = \triangle_E\ i \in \mathcal{Z}, \\ \bot, \text{ sinon}, \end{cases}$$

$$\triangle_E\ s\ f_{by}\ c = \triangle_E\ s, \quad \text{(flot)}$$

$$\triangle_E\ i = \triangle_{E_v}\ v, \text{ avec } v = \rho(i, E) \text{ et } E_v = \mu(i, E), \quad \text{(identificateur)}$$

$$\triangle_E \begin{bmatrix} c_1 \\ \vdots \\ c_k \end{bmatrix} = \begin{bmatrix} \triangle_{E'}\ c_1 \\ \cdots \\ \triangle_{E'}\ c_k \end{bmatrix}, \text{ avec } E' = \begin{bmatrix} c_1 \\ \vdots \\ c_k \end{bmatrix} \star E. \quad \text{(vecteur)}$$

C'est une fonction dont le premier paramètre est un acteur, le second un environnement et qui retourne un acteur. Le résultat de l'opérateur d'évaluation dépend de l'acteur évalué,

il est :

- alternative : l'évaluation de la clause *alors* si l'évaluation de la condition *cond* est non nulle, et le résultat de l'évaluation de la clause *sinon* dans le cas contraire. Si l'évaluation de la condition est $\perp$, alors le résultat est $\perp$;
- application : la réduction (§ 3.3.4) de l'application de l'opérateur aux arguments évalués ;
- atome : l'atome lui-même (les identificateurs sont vus plus bas) ;
- définition : l'évaluation de la valeur associée ;
- extraction : la valeur indexée si l'évaluation du premier argument est un vecteur et si l'évaluation du second argument un entier, $\perp$ dans le cas contraire ;
- flot : l'évaluation de son état ;
- identificateur : l'évaluation de la valeur associée dans son environnement associé (§ 3.2) ;
- vecteur : Le vecteur des composantes évaluées dans l'environnement courant étendu par le vecteur (§ 3.2).

La première remarque importante concerne l'évaluation des identificateurs. Dans la définition formelle nous pouvons lire que l'évaluation d'un identificateur retourne l'évaluation de la valeur associée dans l'environnement associé. L'identificateur est le seul acteur dont l'évaluation peut engendrer un cycle.

Considérons par exemple l'acteur $i := f(i)$. L'évaluation de la définition de i conduit à l'évaluation de la valeur associée. Elle est le résultat de la réduction de l'application formée de l'opérateur f et des arguments évalués. Or l'évaluation des arguments, ici i, conduit à l'évaluation de l'application $f(i)$, ce qui crée un cycle sans fin.

Les cycles sont possibles uniquement par l'utilisation des identificateurs. Ils sont indéterministes en temps, ce qui n'est pas acceptable pour le type d'applications traitées. Ils sont détectés par le critère de calculabilité (§ 4.1.1).

On peut aussi remarquer que l'acteur retourné appartient à l'ensemble des acteurs figés (§ 2.3). Rappelons que cet ensemble contient les atomes sans les identificateurs et les vecteurs dont les composantes sont des acteurs figés. Les acteurs figés ont comme propriété d'être identiques à leurs évalués.

D'un point de vue mathématique, ce combinateur soulève un problème. En effet, l'évaluation sans les identificateurs est définissable car la récursion opère sur des arguments «de plus en plus simples», ordonnés par la grammaire inductive qui les a construits.

Ceci est vrai pour tous les acteurs, sauf pour les identificateurs. Dans ce cas, l'environnement est utilisé, et il n'est pas une donnée «plus simple». La fonction d'évaluation reste-elle définissable avec les évaluations des identificateurs ? En première approximation, nous pouvons dire que le critère de la calculabilité empêche l'existence de cycles dans l'évaluation des acteurs, cycles créés par l'utilisation des identificateurs.

Si l'on fait l'hypothèse (vérifiée) que seuls les programmes calculables seront résolus, le combinateur d'évaluation ne manipulera que des acteurs calculables. Or un acteur calculable ne contient pas de cycle, donc l'ensemble des acteurs évalués à partir d'un identificateur contient des acteurs «de plus en plus simples». Bien évidemment, une étude mathématique de ce raisonnement serait intéressante, mais elle sort du cadre de cette thèse.

La traduction SCHEME de l'opérateur d'évaluation est :

◇ **Machine/evaluate**

```
(define (evaluate actor env)
  (case (type actor)
    [alternative
     (let ([econd (evaluate (condition actor) env)])
       (if (eq? econd 'bottom)
        'bottom
```

```
   (if (zero? econd)
            (evaluate (clauseElse actor) env)
            (evaluate (clauseThen actor) env))))]
[application
 (reduce (operator actor)
         (evaluate (arguments actor) env))]
[definition
  (evaluate (value actor) env)]
[extraction
 (indexOf (evaluate (indexed actor) env)
          (evaluate (index   actor) env))]
[(flag integer operator)
 actor]
[identifier
 (evaluate (valueOf actor env)
           (assocOf actor env))]
[stream
 (evaluate (state actor) env)]
[vector
 (apply vector: (evaluate (components actor)
                          (chain: actor env)))]
[else
 (map (lambda (actor)
   (evaluate actor env)) actor)]))
```

La fonction suit la définition formelle. Remarquons cependant l'évaluation d'une liste (cas `else`) qui utilise la fonction standard `map`. L'évaluation d'une liste est utilisée par l'évaluation des vecteurs et des applications.

3.3.4 Réduction $\triangleleft$

L'opérateur de réduction donne une valeur aux applications (il faut entendre les applications d'un opérateur à des arguments). Il est isolé de l'opérateur d'évaluation car son activité dépend des algèbres de l'utilisateur. Notons que les arguments doivent être évalués. Sa définition est :

$$\begin{aligned} &\triangleleft : \mathcal{P} \longrightarrow \mathcal{D} + \mathcal{Z} + \{\bot\} \quad | && (3.17)\\ &\triangleleft\, o(a_1,\, a_2,\, \ldots,\, a_k) = \underline{o(a_1, a_2, \ldots, a_k)}, \forall a_{i,\ldots,k} \neq \bot && \text{(application)}\\ &\triangleleft\, u = \bot, && \text{(cas général)} \end{aligned}$$

Elle prend comme argument une application et rend un entier, une donnée ou l'indicateur $\bot$. Si aucun argument n'est $\bot$, le résultat est le résultat de l'application de l'opérateur aux arguments, noté $\underline{o(a_1, a_2, \ldots, a_k)}$, sinon, il est $\bot$.

0n remarque qu'aucun contrôle de type n'est effectué : c'est à chaque opérateur de contrôler le type des arguments. Ceci ne sera plus vrai lorsque nous traiterons de la compilation des λ-matrices qui effectue un contrôle des types (§ 5).

Le combinateur de réduction est mathématiquement définissable car il n'est pas récursif.

Voici la fonction SCHEME correspondante :

◇ **Machine/reduce**

```
(define (reduce op args)
  (if (let loop ([args args])
        (if (null? args) #t
            (if (eq? (type (car args)) 'integer)
                (loop (cdr args))
                #f)))
      (apply op args)
      'bottom))
```

Elle est volontairement limitée aux entiers. Là, le contrôle des types est effectué par l'opérateur lui-même, contrairement à la définition formelle, dans le but de simplifier le programme. Si tous les arguments sont des entiers, le résultat est l'application de l'opérateur à ses arguments, sinon il est `'bottom`.

3.4 Exemple

Dans cette section, nous allons voir comment manipuler le filtre donné en exemple précédemment (§ 2.3). Ce filtre est construit par la commande SCHEME :

◇ **example/filter**

```
(define filter
  (vector:
    (definition: 'o (application: + 'a
                      (application: + 'b
                        (application: + 'c 'd))))
    (definition: 'a (application: - 'i 'b))
    (definition: 'b (stream: 0 'd))
    (definition: 'c (application: - 'a 'e))
    (definition: 'd (application: + 'c 'e))
    (definition: 'e (stream: 0 'c))))
```

Ce programme crée une λ-matrice. Elle est partiellement définie car l'identificateur i n'est associé à aucune valeur.

Pour utiliser ce filtre, il est nécessaire de l'inclure dans une matrice maîtresse qui va définir la condition d'arrêt et l'entrée i. Nous obtenons le système :

◇ **example/system**

```
(define system
  (vector:
    (definition: 's (alternative: (application: =? 'i 10)
                                  1 0))
    (definition: 'i (stream: 1 (application: + 'i 1)))
    (definition: 'o (extraction: filter 1))))
```

où l'identificateur *filter* doit être remplacé par le filtre défini au-dessus. Ici, l'environnement de SCHEME joue le rôle d'une table de travail où nous pouvons poser les différentes composantes de notre programme.

Afin de simuler l'entrée i du filtre, le système définit l'identificateur i comme le flot des entiers naturels. La condition d'arrêt est définie comme étant s.

Nous pouvons évaluer le système `system` avec la commande :

```
STk> (evaluate system 'bottom)
(vector 0 1 3)
```

Ce résultat est affiché sous la forme d'une liste SCHEME, ce qui n'est pas très explicite. Utilisons la fonction `convert` que nous avons écrite pour faciliter l'affichage des résultats :

```
STk> (convert (evaluate system 'bottom))
[
  0
  1
  3
]
```

Cet exemple montre bien la nature de l'évaluation : elle donne une valeur aux acteurs. Considérons le programme suivant :

```
STk> (define unclosed-filter
       (vector: (definition: 's (application: '+ 1 'x))))
#unspecified
```

Que se passe-t-il lorsque nous tentons de l'évaluer? Entrons la commande suivante :

```
STk> (convert (evaluate unclosed-filter 'bottom))
[
  bottom
]
```

L'évaluation de l'identificateur `'x` retourne `'bottom` car il n'est pas défini. Considérons maintenant le programme suivant :

```
STk> (define uncalculable-filter
```

```
        (vector: (definition: 's (application: '+ 1 's))))
#unspecified
```

qui définit un cycle sur l'identificateur `s` et tentons de l'évaluer :

```
STk> (evaluate uncalculable-filter 'bottom)
```

L'évaluation de ce filtre ne se termine pas, parcequ'elle conduit à une évaluation cyclique du symbole `s`. Ce cycle est détecté par le critère de la calculabilité.

Chapitre 4

Analyse

4.1 Fonctions de critère

Les applications modélisées par les λ-matrices sont résolues par un opérateur basé sur une itération (§ 3.3.1). Le processus engendré par cette résolution doit être déterministe en temps et en ressources, dans le but d'être implanté sur une architecture parallèle statique.

Le déterminisme en temps garantit que les temps de calcul de chaque itération ont une limite supérieure. Dans le cadre des λ-matrices, nous nous posons la question de l'existence de cette limite sans désirer la connaître. Une étude plus poussée permettrait d'utiliser les λ-matrices dans des applications dites « temps réel ». Le déterminisme en ressources garantit que la quantité de mémoire utilisée par le programme est constante à chaque itération. Les déterminismes doivent être vérifiés indépendamment des valeurs des entrées du programme.

Les λ-matrices proposent trois combinateurs qui permettent de vérifier les programmes. Un programme est calculable s'il ne contient pas de cycle. L'absence de cycle garantit que les temps d'évaluation sont déterministes. La calculabilité d'une application est établie par le critère de la calculabilité.

Un programme est fermé s'il est calculable et s'il ne contient aucune variable libre. Là, se pose la question des entrées principales du programme. Nous avons vu qu'il est possible de les définir comme les flots (§ 3.3.1). Ainsi, elles sont définies, et le programme ne contient plus de variables libres.

Un programme est stable s'il est fermé et si sa dimension (§ 3.1.2) reste constante. Comme une λ-matrice contient à la fois les données et le programme, si cette propriété est établie, la taille du programme possède nécessairement une limite supérieure. Comme les temps de calculs dépendent directement de la taille du programme, nous en déduisons le déterminisme en temps du programme.

Les combinateurs des critères sont, comme les combinateurs de résolution (§ 3), des opérateurs dynamiques car ils construisent l'environnement courant des expressions au fur et à mesure. D'une manière générale, ils s'écrivent $\Lambda_{nom}(acteur, environnement)$, où *nom* est le nom du critère, *acteur* l'acteur testé et *environnement* son environnement courant. Ces combinateurs retournent l'entier 0 si le critère n'est pas vérifié et 1 s'il l'est. Par abus de notation, nous exprimerons que le critère t pour x dans E est vérifié par $\Lambda_t(x, E)$, au lieu de $\Lambda_t(x, E) = 1$.

4.1.1 Calculabilité Λ_c

Un système est dit calculable s'il ne contient pas d'équation de point-fixe. Dans les λ-matrices, une équation de point-fixe se manifeste par l'existence d'un cycle dans l'évaluation d'une ou plusieurs composantes du programme.

Un acteur dans son environnement courant est calculable si son évaluation n'est pas une fonction de son évalué. Autrement dit, l'évaluation d'un acteur ne nécessite pas comme résultat intermédiaire l'évaluation de cet acteur.

La nature algébrique des acteurs des λ-matrices indique qu'un acteur ne peut se contenir lui-même [25]. Ainsi, un cycle ne peut être créé que de manière symbolique, à l'aide d'un identificateur. Un programme calculable est caractérisé par :

$$\Lambda_c(x, E) \Rightarrow (x, E) \notin \widehat{\triangle_E}\, x \tag{4.1}$$

où $\widehat{\triangle_E}\, x$ représente l'ensemble des acteurs impliqués par l'évaluation de x dans E. La compréhension intuitive du contenu de cet ensemble est suffisante pour l'instant. Sa définition complète se trouve dans la section (§ 6.3). Ainsi, un acteur est calculable s'il ne participe pas lui-même à son évaluation.

Remarquons que l'évaluation d'un système non-calculable ne crée pas nécessairement un cycle infini et peut retourner une valeur. Mais l'obtention de cette valeur n'est pas déterministe en temps, ce qui signifie que le temps de calcul pour l'obtenir ne peut être borné.

La fonction de critère est définie comme suit :

$$\begin{array}{ll}
\Lambda_c : \mathcal{X} \times \Gamma \longrightarrow \{0,1\} \quad | & (4.2)\\
\Lambda_c(x, E) = 1, & \text{(atome)}\\
\Lambda_c(\ i{:=}v, E) = \Lambda_c(v, E), & \text{(définition)}\\
\Lambda_c(\{c_1, c_2, \ldots, c_n\}_t, E) = \prod_{i=1}^{n} \Lambda_c(c_i, E), & \text{(cas général)}\\
\Lambda_c(s\ f_{by}\ c, E) = \Lambda_c(s, E) \times \Lambda_c(c, \bot), & \text{(flot)}\\
\Lambda_c(i, E) = \left\{ \begin{array}{l} 0, \text{ si } \rho(i,E) = \top, \\ 1, \text{ si } \rho(i,E) = \bot, \\ \Lambda_c(\rho(i,E), \eta(i, \top, \mu(i,E))) \text{ sinon,} \end{array} \right. & \text{(identificateur)}\\
\Lambda_c(\left[\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}\right], E) = \prod_{i=1}^{k} \Lambda_c(c_i, E'), \text{ avec } E' = \left[\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}\right] \star E. & \text{(vecteur)}
\end{array}$$

Cette fonction a comme paramètres un acteur et un environnement. Elle retourne l'entier 0 ou l'entier 1. Dans le cas général, toutes les composantes de l'acteur sont vérifiées, et le résultat est le produit (*et* logique) des résultats. Un atome est toujours calculable, s'il n'est pas un identificateur. Une définition est calculable si sa valeur est calculable. Un vecteur est calculable si toutes ses composantes le sont dans l'environnement étendu par le vecteur.

Un identificateur est calculable s'il n'est pas défini. Il n'est pas calculable si sa valeur associée est l'indicateur $\top$. Dans les autres cas, un identificateur est calculable si sa valeur associée est calculable dans son environnement de définition dans lequel nous associons à l'identificateur l'indicateur $\top$, masquant ainsi la définition précédente. Un exemple de cycle est donné plus loin (§ 4.3).

Un flot est calculable si son état est calculable et si son contrat est calculable dans l'environnement vide. En effet, un contrat ne peut pas créer de cycle avec les acteurs de l'environnement du flot à cause de la récurrence : les expressions sont considérées à des temps différents. Par contre, la régénération d'un flot évalue son contrat. Si le contrat définit un sous-environnement, il peut créer des cycles. La vérification du contrat dans l'environnement vide est une manière de ne prendre en compte que les définitions créées par le contrat. Un exemple de contrat non calculable est donné plus loin (§ 4.3).

Nous montrerons que l'expression de la calculabilité vérifie bien la caractéristique de ce critère et que le régénéré d'un acteur calculable est aussi calculable, ce qui indique que la calculabilité se conserve avec la régénération (§ 6.4).

La fonction SCHEME de la calculabilité est :

◇ **Criteria/calculable**

```
(define (calculable? actor env)
  (case (type actor)
    [(alternative application extraction)
     (calculable? (cdr actor) env)]
    [definition
```

```
  (calculable? (value actor) env)]
[(flag integer operator)
 1]
[identifier
 (let ([value (valueOf actor env)])
   (case  value
     ['bottom 1]
     ['top    0]
     [else
      (calculable? value
                   (extend actor
                           'top
                           (assocOf actor env)))]))]
[stream
 (* (calculable? (state actor)    env)
    (calculable? (contract actor) 'bottom))]
[vector
 (calculable? (components actor)
              (chain: actor env))]
[else
 (apply * (map (lambda (actor)
                 (calculable? actor env))
               actor))]))
```

On retrouve les cas distingués dans la définition formelle du critère. De plus, la fonction sait calculer la calculabilité d'une liste d'acteurs (cas `else`), ce qui est utilisé pour établir la calculabilité dans le cas général (premier cas) où le résultat est le produit de la calculabilité des composantes de l'acteur.

4.1.2 Fermeture Λ_l

Un programme fermé doit d'abord être calculable. De plus, tous les identificateurs d'un programme fermé sont définis dans leur environnement courant (§ 3.2). Autrement dit, un programme fermé ne possède pas de variables libres. Cela s'écrit :

$$\Lambda_l(x, E) \Rightarrow \rho(i, E) \neq \perp, \text{ pour tous les identificateurs } i \text{ dans } x. \tag{4.3}$$

L'expression générale de cette fonction de critère est :

$$\Lambda_l : \mathcal{X} \times \Gamma \longrightarrow \{0, 1\} \quad | \tag{4.4}$$

$$\Lambda_l(x, E) = 1, \tag{atome}$$

$$\Lambda_l(\{c_1, c_2, \ldots, c_n\}_t, E) = \prod_{i=1}^{n} \Lambda_l(c_i, E), \tag{cas général}$$

$$\Lambda_l(\ i{:=}v, E) = \Lambda_l(v, E), \tag{définition}$$

$$\Lambda_l(i, E) = \begin{cases} 0, \text{ si } \rho(i, E) = \perp, \\ 1, \text{ sinon,} \end{cases} \tag{identificateur}$$

$$\Lambda_l(\left[\begin{smallmatrix} c_1 \\ \vdots \\ c_k \end{smallmatrix}\right], E) = \prod_{i=1}^{k} \Lambda_l(c_i, E'), \text{ avec } E' = \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_k \end{smallmatrix}\right] \star E. \tag{vecteur}$$

C'est une fonction à deux paramètres, un acteur et un environnement, qui retourne 0 ou 1. Un atome est toujours fermé, s'il ne s'agit pas d'un identificateur. Une définition est fermée si sa valeur est fermée. Dans le cas général, un acteur est fermé si toutes ses composantes le sont. Un vecteur est fermé si ses composantes sont fermées dans l'environnement courant étendu par le vecteur. Un identificateur est fermé si sa valeur associée existe.

On montrera la relation entre la propriété recherchée (pas de variables libres) et la définition de la fonction de critère. De plus, on montrera que le régénéré d'un programme fermé est fermé, ce qui indique que la fermeture se conserve avec la régénération (§ 6.5).

La fonction SCHEME qui définit la fermeture est :

◇ **Criteria/closed**

```
(define (closed? actor env)
  (case (type actor)
```

```
[(alternative application extraction stream)
 (closed? (cdr actor) env)]
[definition
  (closed? (value actor) env)]
[(flag integer operator)
 1]
[identifier
 (if (eq? (valueOf actor env) 'bottom)
     0 1)]
[vector
 (closed? (components actor)
          (chain: actor env))]
[else
 (apply * (map (lambda (actor)
                 (closed? actor env))
               actor))]))
```

Cette fonction suit la définition formelle et elle a la structure de la fonction SCHEME réalisant le critère de calculabilité décrite plus haut.

4.1.3 Stabilité Λ_s

Un système calculable est dit stable si sa dimension (§ 3.1.2) reste constante avec la régénération (§ 3.2), ce qui s'écrit :

$$\Lambda_s(u, E) \Rightarrow |u| = |\bigtriangledown_E u| \,. \tag{4.5}$$

La définition formelle de cette fonction est :

$$\Lambda_s : \mathcal{X} \times \Gamma \longrightarrow \{0, 1\} \quad | \tag{4.6}$$

$$\Lambda_s(c \rightarrow t, e, E) = \begin{cases} \Lambda_s(c, E).\Lambda_s(t, E).\Lambda_s(e, E), \text{ si } \sigma(t, E) = \sigma(e, E), \\ 0, \text{ sinon,} \end{cases} \quad \text{(alternative)}$$

$$\Lambda_s(x, E) = 1. \quad \text{(atome)}$$

$$\Lambda_s(\{c_1, c_2, \ldots, c_n\}_t, E) = \prod_{i=1}^{n} \Lambda_s(c_i, E), \quad \text{(cas général)}$$

$$\Lambda_s(\ i{:=}v, E) = \Lambda_s(v, E) \quad \text{(définition)}$$

$$\Lambda_s(\ m \cdot i, E) \quad \text{(extraction)}$$

$$= \begin{cases} \Lambda_s(\ \rho(m, E) \cdot i, \mu(m, E)), \text{ si } m \in \mathcal{I}, \\ \Lambda_s(m, E), \text{ si } u \in \mathcal{V} \text{ et } i \in \mathcal{Z}, \\ \Lambda_s(m, E) \times \Lambda_s(i, E), \text{ si } m \in \mathcal{V} \text{ et } i \notin \mathcal{Z}, \text{ et } \forall k \in [1, n], \sigma(c_1, E') = \sigma(c_k, E') \\ \quad \text{avec } m = \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix} \right] \text{ et } E' = m \star E \\ 0, \text{ sinon,} \end{cases}$$

$$\Lambda_s(s\ f_{by}\ c, E) = \begin{cases} \Lambda_s(c, E), \text{ si } s \in \overline{\mathcal{X}} \text{ and } \sigma(s, E) = \sigma(c, E), \\ 0, \text{ sinon,} \end{cases} \quad \text{(flot)}$$

$$\Lambda_s(\left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix} \right], E) = \prod_{i=1}^{n} \Lambda_s(c_i, E'), \text{ avec } E' = \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix} \right] \star E \quad \text{(vecteur)}$$

Ce critère est le plus structuré de tous. Il utilise la norme d'un acteur qui retourne la plus grande dimension possible du résultat de l'évaluation d'un acteur (§ 4.1.4). C'est une fonction à deux paramètres : un acteur et un environnement. Elle retourne soit 0 soit 1. Un système est stable si :

- alternative : la condition *cond*, la clause *alors* et la clause *sinon* doivent être toutes trois stables, et la norme de *alors* et de *sinon* doivent être identiques ;
- extraction : la valeur indexée est soit un identificateur stable soit un vecteur stable. Si l'index n'est pas une constante entière, il doit être stable, et toutes les composantes du vecteur doivent avoir une norme identique ;

– flot : l'état doit appartenir à l'ensemble des acteurs figés (§ 2.3) et sa norme doit être égale à la norme de son contrat ;

– cas général : les composantes des autres acteurs doivent toutes être stables.

On montrera que l'expression de la stabilité correspond bien à la caractéristique. De plus, le régénéré d'un programme stable est aussi stable, ce qui indique que la stabilité se conserve avec la régénération (§ 6.6).

La réalisation en SCHEME de la stabilité est :

◇ **Criteria/stable**

```
(define (stable? actor env)
  (case (type actor)
    [alternative
     (if (eq? (norm (clauseThen actor) env)
              (norm (clauseElse actor) env))
         (* (stable? (condition actor)  env)
            (stable? (clauseThen actor) env)
            (stable? (clauseElse actor) env))
         0)]
    [application
     (stable? (arguments actor) env)]
    [definition
      (stable? (value actor) env)]
    [extraction
     (let ([indexed (indexed actor)]
           [index   (index   actor)])
       (case (type indexed)
         ['vector
          (if (eq? (type index) 'integer)
              (stable? indexed env)
              (if (apply eq? (norm (components indexed)
                                   (chain: indexed env)))
                  (* (stable? indexed env)
                     (stable? index   env))
                  0))]
         ['identifier
          (stable? (extraction: (valueOf indexed env)
                                index)
                   (assocOf indexed env))]
         [else 0]))]
    [(flag integer identifier operator)
     1]
    [stream
     (if (and (eq? (frozen? (state actor)) 1)
              (eq? (norm (state    actor) env)
                   (norm (contract actor) env)))
         (stable? (contract actor) env)
         0)]
    [vector
     (stable? (components actor)
              (chain: actor env))]
    [else
     (apply * (map (lambda (actor)
                     (stable? actor env))
                   actor))]))
```

Cette fonction suit point par point sa définition formelle et elle a la structure des fonctions de critère précédentes.

4.1.4 Norme σ

La norme d'un acteur est un entier représentant la plus grande dimension possible de l'acteur résultant de son évaluation. Par exemple, la dimension de l'évaluation d'une alternative est la dimension de l'évaluation soit de la clause *alors* soit de la clause *sinon*, en fonction du résultat de l'évaluation de la condition. Donc la norme d'une alternative est la plus grande norme de la clause *alors* et de la clause *sinon*.

La définition suivante est obtenue :

$$\sigma : \mathcal{X} \times \Gamma \longrightarrow \mathcal{Z} \quad | \tag{4.7}$$

$$\sigma(c \to t, e, E) = \sup(\sigma(t, E), \sigma(e, E)), \qquad \text{(alternative)}$$

$$\sigma(o(a_1,\, a_2,\, \ldots,\, a_k), E) = 1, \qquad \text{(application)}$$
$$\sigma(x, E) = 1, \qquad \text{(atome)}$$
$$\sigma(\ i{:=}v, E) = \sigma(v, E), \qquad \text{(définition)}$$
$$\sigma(\ u \cdot i, E) \qquad \text{(extraction)}$$
$$= \begin{cases} \left.\begin{array}{r} \sup(\sigma(c_1, E'), \ldots, \sigma(c_k, E')), \text{ si } i \notin \mathcal{Z}, \\ \sigma(\delta(i,u), E'), \text{ sinon,} \end{array}\right\} \text{, si } u = \left[\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}\right], \\ \sigma(\ \rho(u,E) \cdot i, \mu(u,E)), \text{ si } u \in \mathcal{I}, \\ 1, \text{ sinon,} \end{cases}$$
$$\sigma(s\, f_{by}\, c, E) = \sigma(s, E), \qquad \text{(flot)}$$
$$\sigma(i, E) = \sigma(v, E_v), \text{ avec } v = \rho(i, E) \text{ et } E_v = \mu(i, E), \qquad \text{(identificateur)}$$
$$\sigma(\left[\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}\right], E) = \sum_{i=1}^{k} \sigma(c_i, E'), \text{ avec } E' = \left[\begin{array}{c} c_1 \\ \vdots \\ c_k \end{array}\right] \star E. \qquad \text{(vecteur)}$$

La norme est une fonction à deux paramètres, un acteur et un environnement. Elle retourne un entier. La norme d'une alternative est la plus grande norme de la clause *alors* et de la clause *sinon*. La norme d'une application est 1 car son résultat est soit un entier, soit une donnée soit l'indicateur d'erreur $\perp$, qui ont tous trois 1 comme dimension car ce sont des atomes. La norme d'une définition est la norme de sa valeur associée.

Si l'acteur indexé est un vecteur, la norme d'une extraction est soit la plus grande des normes du vecteur si l'index n'est pas un entier, soit la norme de la composante indexée si l'index est un entier. Si l'acteur indexé est un identificateur, la norme de l'extraction est la norme de l'extraction formée de la valeur associée à l'identificateur et de l'index. Dans les autres cas, la norme d'une extraction ne répondant à aucun de ces cas est 1, correspondant à la norme de l'indicateur d'erreur $\perp$.

La norme d'un flot est la norme de son état. La norme d'un identificateur est la norme de sa valeur associée dans son environnement associé. Enfin, la norme d'un vecteur est la somme des normes de ses composantes.

La définition en SCHEME de cette fonction est :

◇ **Miscellaneous/norm**

```
(define (norm actor env)
  (case (type actor)
    [alternative
     (max (norm (clauseThen actor) env)
          (norm (clauseElse actor) env))]
    [application
     1]
    [definition
      (norm (value actor) env)]
    [extraction
     (let ([indexed (indexed actor)]
           [index   (index   actor)])
       (case (type indexed)
         [vector
          (if (eq? (type index) 'integer)
              (norm (indexOf indexed index)
                    (chain: indexed env))
              (apply max (norm (components indexed)
                               (chain: indexed env))))]
         [identifier
          (norm (extraction: (valueOf indexed env)
                             index)
                (assocOf indexed env))]
         [else 1]))]
    [(flag integer operator)
     1]
    [identifier
     (norm (valueOf actor env)
           (assocOf actor env))]
    [stream
     (norm (state actor) env)]
    [vector
     (apply +
            (norm (components actor)
                  (chain: actor env)))]
    [else
     (map (lambda (actor) (norm actor env)) actor)]))
```

La définition de cette fonction suit de prés sa définition formelle.

4.2 Déterminismes

Les λ-matrices modélisent des applications dans le but de les réaliser sur une machine parallèle statique. Une telle architecture impose que le programme qui y est implanté soit déterministe en temps et en ressources.

Les λ-matrices proposent le critère de la stabilité qui indique que la dimension d'un programme est identique à la dimension du régénéré. Nous montrerons par la suite que si un programme est stable, le programme issu de sa régénération est aussi stable (§ 6.6).

La machine de résolution est basée sur un combinateur récursif qui réalise en fait une itération (§ 3). Cette itération régénère le système original jusqu'à ce qu'une condition d'arrêt soit atteinte. Si le régénéré d'un système stable est stable, la machine de résolution engendre un processus stable.

Cette propriété est établie de manière formelle, indépendamment de toute réalisation. Elle reste donc vrai quelque soit la réalisation. Par exemple, elle est vraie dans la réalisation SCHEME proposée tout au long de cette étude.

Une λ-matrice stable est aussi calculable (§ 4.1.1), ce qui indique qu'elle ne contient pas d'équations de point-fixe qui introduiraient un indéterminisme en temps.

Une λ-matrice décrit à la fois le programme et les données, contenus dans le même espace de description. Si la λ-matrice est stable, sa dimension est constante avec les régénérations. Nous pouvons donc dire que la dimension du programme possède une limite supérieure. Les temps de calculs sont directement liés à la taille du programme. Nous concluons donc qu'une λ-matrice stable est déterministe en temps, par rapport à sa machine de résolution.

Les déterminismes en temps et en ressources sont obtenus pour toute λ-matrice stable.

4.3 Exemple

Dans cette section, nous présentons un certain nombre de cas ne répondant pas aux critères. Cette présentation est informelle, mais elle est représentative des cas à tester.

4.3.1 Calculabilité

Le critère de calculabilité interdit la définition d'une équation de point-fixe dans une matrice. Une équation de point fixe survient dans les matrices suivantes :

$$\left[\ \ s := +(1,\ s)\ \ \right] \tag{4.8}$$

et

$$\left[\begin{array}{l} s_1 := +(1,\ s_2) \\ s_2 := -(1,\ s_1) \end{array}\right] \tag{4.9}$$

Ce qui donne en SCHEME :

```
STk> (calculable? (vector: (definition: 's (application: + 1 's)))
                  'bottom )
0
```

et :

```
STk> (calculable? (vector: (definition: 's1 (application: + 1 's2))
                           (definition: 's2 (application: - 1 's1)))
                  'bottom)
0
```

A priori, un cycle intervient dès que la valeur d'une définition utilise l'identificateur qu'elle définit. Ceci est vrai, à condition que le cycle ne se produise pas au travers d'un flot. Par exemple :

$$[\ \ s{:=}0\, f_{by}\ +(1,\ s)\ ] \tag{4.10}$$

est parfaitement calculable et définit le flot des entiers naturels. En SCHEME, nous avons :

```
STk> (calculable? (vector: (definition: 's (stream: 0 (application: + 1 's))))
                  'bottom)
1
```

On pourrait donc penser que la vérification ne traite pas le contrat des flots, mais ceci est faux. Considérons :

$$[\ \ s{:=}[\ 0\ ]\ f_{by}\ [\ \ x{:=}f(x)\ ]\ ] \tag{4.11}$$

le contrat du flot n'est pas calculable car lui-même contient un cycle. Vérifions avec SCHEME :

```
STk> (calculable? (vector: (definition: 's
                     (stream: (vector: 0)
                              (vector: (definition: 'x (application: + 'x))))))
                  'bottom)
0
```

4.3.2 Fermeture

Une λ-matrice n'est pas fermée lorsqu'un identificateur qu'elle contient n'est associé à aucune valeur. Par exemple :

$$[\ +(1,\ s)\ ] \tag{4.12}$$

n'est pas fermée car l'identificateur s n'est associé à aucune valeur. Testons ce fragment avec SCHEME :

```
STk> (closed? (vector: (application: + 1 's))
              'bottom )
0
```

Par contre les expressions :

$$\left[\begin{array}{l} s{:=}3 \\ +(1,\ s) \end{array}\right] \tag{4.13}$$

et

$$\left[\begin{array}{l} s{:=}3 \\ [\ +(1,\ s)\ ] \end{array}\right] \tag{4.14}$$

sont fermées. Avec SCHEME, nous obtenons :

```
STk> (closed? (vector: (application: + 1 's)
                       (definition: 's 3))
              'bottom )
1
```

et :

```
STk> (closed? (vector: (vector: (application: + 1 's))
                       (definition: 's 3))
              'bottom )
1
```

4.3.3 Stabilité

La stabilité d'une matrice garantit que sa dimension est constante. Voici un exemple d'une matrice instable :

$$\left[\ s := \left[\begin{array}{c} 2 \\ 1 \end{array}\right] f_{by} \left[\begin{array}{c} 3 \\ s \end{array}\right] \right] \tag{4.15}$$

Cette matrice est calculable et fermée. Pourtant, la matrice issue de sa régénération est :

$$\left[\ s := \left[\begin{array}{c} 3 \\ \left[\begin{array}{c} 2 \\ 1 \end{array}\right] \end{array}\right] f_{by} \left[\begin{array}{c} 3 \\ s \end{array}\right] \right] \tag{4.16}$$

dont la dimension n'est pas identique à celle de la matrice originale.

Reprenons cet exemple à l'aide de SCHEME :

```
STk> (define test (vector: (definition: 's (stream: (vector: 2 1) (vector: 3 's)))))
STk> (stable? test 'bottom)
0
```

et :

```
STk> (convert (regenrerate test 'bottom))
[
  s := [3 [2 1]] fby [3 s]
]
```

4.3.4 Exemple du filtre

Poursuivons avec l'exemple du filtre introduit dans les chapitres précédents (§ 2.3). La vérification des critères pour le filtre s'écrirait :

```
STk> (calculable? filter 'bottom)
1
```

où `'bottom` représente l'environnement vide. Nous voyons que le filtre est calculable, bien qu'il contienne une variable libre i. Qu'en est-il avec la fermeture ?

```
STk> (closed? filter 'bottom)
0
```

Comme nous le savions, le filtre n'est pas fermé car l'identificateur i n'est pas défini. Il n'est donc pas utile de poursuivre nos investigations sur la stabilité, car un programme stable doit tout d'abord être fermé.

Par contre, le système est lui parfaitement fermé. Vérifions tout d'abord sa calculabilité :

```
STk> (calculable? system 'bottom)
1
```

ce qui nous permet de tester sa fermeture :

```
STk> (closed? system 'bottom)
1
```

et donc la stabilité :

```
STk> (stable? system 'bottom)
1
```

Chapitre 5

Compilation

La compilation transforme un programme source pour une machine d'exécution [5]. Bien souvent, la machine cible est un langage d'un niveau moins élevé que le langage source ; on l'appellera de manière générale la cible. Ainsi, les premiers compilateurs C++ transformaient les programmes en langage C, lui même compilé par un compilateur. A chaque étape de la transformation, un niveau d'abstraction du langage source est supprimé et traduit en une séquence d'instructions de la cible.

Le compilateur des λ-matrices transforme une λ-matrice en une λ-matrice «dégradée» dite *linéaire*. Cette transformation supprime les aspects modulaires de la λ-matrice source, c'est à dire les vecteurs et les extractions. De plus, le compilateur évite la duplication de code, ce qui est une sorte d'optimisation.

Le compilateur formel est un combinateur sans effet de bord. L'utilisation d'un combinateur pour la compilation n'est pas chose courante. Elle permettrait des vérifications formelles du compilateur.

L'avantage de conserver le même langage pour la source et pour la cible est de permettre de résoudre la source et la cible par la même machine d'exécution, la machine fonctionnelle abstraite décrite précédemment (§ 3). Or, cette machine donne la sémantique au langage. Il devient donc possible de comparer de manière formelle les sémantiques de la source et de la cible.

Le processus de compilation nécessite un critère supplémentaire, en plus de la calculabilité, de la fermeture et de la stabilité (§ 4) : il s'agit du critère de la constance, qui effectue un contrôle des types. En effet, comme nous l'avons vu dans la définition de la machine abstraite, aucun contrôle de type n'est effectué, notamment pour les applications où les opérateurs sont chargés de vérifier dynamiquement le type de leurs arguments. Or ce contrôle ne peut être effectué de manière dynamique dans une réalisation concrète, pour des raisons d'efficacité.

5.1 Signature

Cette section définit la notion de type de donnée [21, 52, 61, 69] en l'appliquant aux λ-matrices. Dans le formalisme, un type de donnée est appelé *signature*.

5.1.1 Signatures des données τ

La signature d'une donnée est une fonction définie par :

$$\begin{array}{ll} \tau : \mathcal{D} \longrightarrow \Theta_d \quad | & (5.1) \\ \tau(d) = t_d, & (\text{donnée}) \\ \tau(i) = n. & (\text{entier}) \end{array}$$

Cette fonction a un argument, une donnée, qui peut être un entier ou une donnée introduite par l'utilisateur à l'aide d'une algèbre[1]. Le résultat est une signature appartenant à l'ensemble Θ_d des signatures des données. L'ensemble t_d doit être redéfini par l'utilisateur à chaque algèbre introduite.

La fonction SCHEME qui réalise cette définition formelle est :

◇ **Signature/sign-data**

```
(define (sign-data data)
  (if (eq? (type data) 'integer) 'n '*))
```

Il s'agit simplement de tester si la donnée est un entier. Dans ce cas, la fonction retourne la signature des entiers `'n` et dans le cas contraire, la signature générique `'*`. Rappelons que la réalisation SCHEME des λ-matrices ne propose que les entiers pour conserver un logiciel simple.

5.1.2 Construction des signatures Θ

Après l'introduction des signatures atomiques, il est possible de définir les signatures composées. Elles sont construites à l'aide de deux constructeurs, la règle $\longrightarrow$ utilisée pour les opérateurs, et $\times$ pour les vecteurs. Il vient :

$$\left.\begin{array}{ll} * \in \Theta & : \text{générique} \\ \Theta_d \subset \Theta & : \text{donnée} \\ \alpha \rightarrow \beta \in \Theta & : \text{opérateur} \\ \alpha \times \beta \in \Theta & : \text{vecteur} \end{array}\right\}, \text{ avec } \alpha, \beta \in \Theta. \tag{5.2}$$

où $\alpha \rightarrow \beta$ dénote *la fonction qui attend un argument de type α et dont le type du résultat est β* et $\alpha \times \beta$, *le vecteur dont la première composante est de type α et la seconde, de type β.* La signature $*$ dénote le type générique, utilisé lorsque le type d'une expression ne peut être déterminé.

5.1.3 Signature des opérateurs ϑ

Pour chaque opérateur que l'utilisateur introduit, il doit donner sa signature sous la forme $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_0$. Cette signature indique que la valeur de retour a pour signature t_0, et que les arguments doivent être respectivement de signature t_1 à t_n. Cette section introduit une fonction qui retourne la signature des opérateurs :

$$\begin{array}{l} \vartheta : \mathcal{O} \longrightarrow \Theta \quad | \\ \quad \vartheta(o) = t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_0, \text{ avec } t_{0,\ldots,n} \in \Theta_d. \end{array} \tag{5.3}$$

Remarquons que les opérateurs de l'utilisateur ne peuvent pas manipuler les données génériques de type $*$ car les arguments et la valeur de retour ont une signature appartenant à l'ensemble Θ_d. De même, ils ne peuvent pas manipuler des objets construits, comme les vecteurs.

Le constructeur SCHEME réalisant cette fonction est :

◇ **Signature/sign-op**

```
(define (sign-op op)
  (cond [(or (eq? op +)
             (eq? op *)
             (eq? op -)
             (eq? op /)
             (eq? op =?)) (list 'sign-op 'n 'n 'n)]
        [else '*]))
```

Si l'opérateur est `+`, `-`, `*`, `/` ou `=?`, la signature retournée est une liste, correspondant à la signature $n \rightarrow n \rightarrow n$ d'un opérateur à deux paramètres entiers qui retourne un entier.

1. Comme nous l'avons vu dans la définition du langage (§ 2.1), une partie des acteurs ne peut être connue à l'avance car elle dépend des algèbres de l'utilisateur. Ce fait se retrouve dans la définition des signatures des données.

5.1.4 Signature des expressions θ

A ce stade, il est possible d'obtenir la signature des données et des opérateurs qui s'y rapportent. Cette section définit une fonction qui retourne la signature de tout acteur des λ-matrices.

Son expression formelle est :

$$
\begin{array}{ll}
\theta : \mathcal{X} \times \Gamma \longrightarrow \Theta \quad | & (5.4)\\
\theta(c \to a, s, E) = \left\{ \begin{array}{l} \theta(a,E) \text{ si } \theta(a,E) = \theta(s,E) \text{ et } \theta(c,E) = n, \\ *, \text{ sinon,} \end{array} \right. & \text{(alternative)}\\
\theta(o(a_1,\ a_2,\ \ldots,\ a_n), E) & \text{(application)}\\
\quad = \left\{ \begin{array}{l} t_o, \text{ si } \vartheta(o) = t_1 \to \ldots \to t_n \to t_o \text{ et } \theta(a_i,E) = t_i, \\ *, \text{ sinon,} \end{array} \right. & \\
\theta(x, E) = \tau(x), & \text{(atome)}\\
\theta(\ i{:=}v, E) = \theta(v, E), & \text{(définition)}\\
\theta(\ u \cdot i, E) & \text{(extraction)}\\
\quad = \left\{ \begin{array}{r} \alpha_i, \text{ si } i \in [1,n] \\ \alpha_1, \text{ si } i \notin [1,n] \\ \alpha_1, \text{ si } \alpha_i \text{ identiques} \\ *, \text{ sinon,} \end{array} \right\} \left\{ \begin{array}{l} \text{si } \theta(u,E) = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \\ \text{et si } u \in \mathcal{I} \text{ ou } u \in \mathcal{V} \end{array} \right. & \\
\theta(i, E) = \theta(v, E_v), \text{ avec } v = \rho(i,E) \text{ et } E_v = \mu(i,E), & \text{(identificateur)}\\
\theta(f, E) = * & \text{(indicateur)}\\
\theta(e\ f_{by}\ c, E) = \theta(e, E), & \text{(flot)}\\
\theta(o, E) = \vartheta(o) & \text{(opérateur)}\\
\theta(\left[\begin{array}{c} c_1 \\ \vdots \\ c_n \end{array} \right], E) = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_n, \text{ avec } \alpha_i = \theta(c_i, E') \text{ et } E' = \left[\begin{array}{c} c_1 \\ \vdots \\ c_n \end{array} \right] \star E. & \text{(vecteur)}
\end{array}
$$

C'est une fonction à deux arguments, un acteur et son environnement, qui retourne une signature.

Lorsque la signature de l'acteur ne peut être déterminée, le résultat est $*$. Une telle situation se rencontre par exemple pour une alternative $c \to a, s$ dont les clauses sont de types différents. Comme le résultat d'une telle alternative est l'une ou l'autre des deux clauses, la signature de l'ensemble ne peut être déterminée.

La signature d'une application $o(a_1,\ a_2,\ \ldots,\ a_k)$ dépend de la signature de son opérateur $\vartheta(o) = t_1 \to t_2 \to \ldots \to t_n \to t_0$. Elle est t_0 si le nombre et la signature des arguments correspondent au nombre et au type des arguments attendus. Si au moins l'un des arguments a pour signature $*$ le résultat est $*$.

Un atome est soit un identificateur soit une donnée. S'il s'agit d'un identificateur, le résultat est la signature de la valeur associée dans l'environnement associé. S'il s'agit d'une donnée, le résultat est l'application de τ.

La signature d'une définition $s{:=}v$ est la signature de la valeur associée v.

La signature d'une extraction $u \cdot i$ ne peut être obtenue que si l'indexé u a la signature d'un vecteur $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$ et s'il est un identificateur ou un vecteur. Dans le cas contraire, la signature de l'extraction est $*$. Si l'index i est un entier de valeur n et que cet entier est inférieur au nombre des éléments de la signature de l'indexé, alors la signature de l'ensemble est le n^{ieme} élément. Dans les autres cas, si toutes les signatures α_i sont identiques, alors la signature de l'extraction est α_1.

La signature d'un flot $s\ f_{by}\ c$ est la signature de sa valeur initiale s. Le critère de constance vérifie que le contrat c possède la même signature.

La signature d'un identificateur i est la signature de sa valeur associée $\rho(i,E)$ dans son environnement associé $\mu(i,E)$.

La signature d'un vecteur est la composition par $\times$ des signatures de ses composantes dans l'environnement du vecteur étendu par lui-même.

La définition SCHEME de cette fonction est :

◇ **Signature/sign**

```
(define (sign actor env)
  (case (type actor)
    [alternative
     (let ([scond (sign (condition actor) env)]
           [selse (sign (clauseElse actor) env)]
           [sthen (sign (clauseThen actor) env)])
       (if (and (eq? scond 'n)
                (equal? selse sthen))
           sthen
           '*))]
    [application
     (let ([sop   (sign-op (operator actor))]
           [sargs (map (lambda (arg)
                         (sign arg env))
                       (arguments actor))])
       (if (or (eq? sop '*)
               (not (eq? (+ 2 (length sargs)) (length sop))))
           '*
           (let loop ([sargs sargs]
                      [sop   (cdr sop)])
             (if (null? sargs)
                 (car sop)
                 (if (eq? (car sargs) (car sop))
                     (loop (cdr sargs) (cdr sop))
                     '*)))))]
    [integer (sign-data actor)]
    [definition (sign (value actor) env)]
    [extraction
     (let ([sindexed (sign (indexed actor) env)]
           [sindex   (sign (index   actor) env)])
       (if (and (list? sindexed)
                (case (type (indexed actor))
                  [(identifier vector) #t]
                  [else                #f])
                (eq? (car sindexed) 'sign-vec)
                (eq? sindex 'n))
           (if (eq? (type index) 'integer)
               (if (<= index (length sindexed))
                   (list-ref sindexed index)
                   (cadr sindexed))
               (let [(ref (cadr sindexed))]
                 (let loop ([signs (cddr sindexed)])
                   (if (null? signs)
                       ref
                       (if (equal? (car signs) ref)
                           (loop (cdr signs))
                           '*)))))
           '*))]
    [flag '*]
    [identifier (sign (valueOf actor env) (assocOf actor env))]
    [operator (sign-op actor)]
    [stream (sign (state actor) env)]
    [vector
     (let* ([env    (chain: actor env)]
            [scomps (map (lambda (comp)
                           (sign comp env)) (components actor))])
       (cons 'sign-vec scomps))]))
```

C'est une fonction par cas reprenant la définition formelle. Remarquons l'utilisation conjointe de `apply` et de `map` pour obtenir un *et* logique des résultats de comparaison sur les éléments d'une liste. Notons aussi l'utilisation de `let*` dans le cas des vecteurs pour redéfinir l'environnement `env`.

5.2 Compilateur formel

Le compilateur formel des λ-matrices est un combinateur. Il agit sur les programmes vérifiant le critère de la constance (§ 5.2.1) en produisant une λ-matrice linéaire.

Un autre combinateur est utilisé pour produire le code linéaire. Ce combinateur optimise le code produit dans le sens où il évite la duplication du code.

Le code linéaire est un vecteur contenant des définitions[2]. Les identificateurs de ces définitions jouent le rôle d'adresses symboliques. Ils ont la forme $@_i$ où i est l'index de la définition dans le vecteur.

5.2.1 Constance Λ_t

La constance est un nouveau critère introduit pour permettre une compilation efficace des λ-matrices. Elle effectue un contrôle statique des types de données des expressions et signale les erreurs éventuelles. Un programme constant peut être compilé en supprimant tout contrôle dynamique des types.

Dans la section précédente, nous avons défini un opérateur capable de retourner la signature de tout acteur des λ-matrices (§ 5.1). Cet opérateur est à la base de la définition du critère de la constance. Un acteur constant est tout d'abord calculable et fermé. La définition formelle de la constance est :

$$\Lambda_t : \mathcal{X} \times \Gamma \longrightarrow \{0,1\} \quad | \tag{5.5}$$

$$\Lambda_t(x,E) = 1 \tag{atome}$$

$$\Lambda_t(\{c_1,c_2,\ldots,c_n\}_t,E) = \begin{cases} \prod_{i=1}^{n} \Lambda_t(c_i,E), \text{ si } \theta(\{c_1,c_2,\ldots,c_n\}_t,E) \neq *, \\ 0, \text{ sinon,} \end{cases} \tag{cas général}$$

$$\Lambda_t(\ i{:=}v,E) = \Lambda_t(v,E), \tag{définition}$$

$$\Lambda_t(i,E) = \Lambda_t(v,E_v), \text{ avec } v = \rho(i,E) \text{ et } E_v = \mu(i,E), \tag{identificateur}$$

$$\Lambda_t(e\, f_{by}\, c,E) = \begin{cases} \Lambda_t(e,E) \times \Lambda_t(c,E), \text{ si } e \in \overline{\mathcal{X}} \text{ et } \theta(e,E) = \theta(c,E) \neq *, \\ 0, \text{ sinon,} \end{cases} \tag{flot}$$

$$\Lambda_t(\left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix}\right],E) = \prod_{i=1}^{n} \Lambda_t(c_i,E'), \text{ avec } E' = \left[\begin{smallmatrix} c_1 \\ \vdots \\ c_n \end{smallmatrix}\right] \star E. \tag{vecteur}$$

C'est une fonction à deux arguments, un acteur et son environnement, et qui retourne soit 0 si le critère n'est pas vérifié, soit 1 s'il l'est.

Un atome est toujours constant car sa signature ne peut changer. Un identificateur i est constant dans l'environnement courant E si sa valeur associée $\rho(i,E)$ dans son environnement associé $\mu(i,E)$ est constante.

Une définition $i{:=}v$ est constante si sa valeur v est constante. D'une manière générale, tout acteur est constant si toutes ses composantes le sont.

Un flot $e\, f_{by}\, c$ est constant si sa valeur initiale e est un acteur figé et si elle a la même signature que son contrat c, elle même différente de $*$.

Enfin, un vecteur est constant si toutes ses composantes le sont.

Le critère de la stabilité impose moins de contraintes que le critère de la constance. La différence porte sur le contrôle des types des arguments des applications. Nous montrerons qu'un acteur constant est stable, et que la constance se conserve avec la régénération (§ 6).

La fonction SCHEME réalisant cette définition formelle est :

◇ **Criteria/constant**

```
(define (constant? actor env)
  (case (type actor)
    [(alternative application extraction)
     (if (not (eq? (sign actor env) '*))
         (constant? (cdr actor) env)
         0)]
    [definition
      (constant? (value actor) env)]
    [(flag integer operator)
     1]
    [identifier
     (constant? (valueOf actor env)
                (assocOf actor env))]
```

2. Rappelons qu'un vecteur est considéré comme un environnement à un seul cadre (§ 3.2).

```
[stream
 (let ([state    (state    actor)]
       [contract (contract actor)])
   (if (frozen? state)
       (let ([sstate    (sign state    env)]
             [scontract (sign contract env)])
         (if (and (not (eq? sstate '*))
                  (equal? sstate scontract))
             (* (constant? state    env)
                (constant? contract 'bottom))
             0))
       0))]
[vector
 (constant? (components actor)
            (chain: actor env))]
[else
 (apply * (map (lambda (actor)
                 (constant? actor env))
               actor))]))
```

Elle ne présente pas de difficultés particulières.

5.2.2 Producteur γ

Le producteur de code construit la λ-matrice linéaire en évitant les duplications de code. Le producteur n'évite pas les duplications «sémantique» de code où $x := 0\, f_{by} + (x,\, 1)$ et $y := 0\, f_{by} + (y,\, 1)$ seraient reconnus comme calculant la meme chose, aprés une analyse sémantique. Plus modestement, le producteur évite que des expressions rigouresement identiques ne soient produites deux fois.

Voici son expression formelle:

$$\gamma : \mathcal{X} \times \Gamma \times \mathcal{V} \longrightarrow \mathcal{X} \times \Gamma \times \mathcal{V} \quad | \qquad (5.6)$$
$$\gamma(E, A, x) = \left\{ \begin{array}{l} x \diamond E \diamond A, \text{ si } x \in \mathcal{D} + \mathcal{Z}, \\ @_i \diamond E \diamond A, \text{ si } \kappa(x, A) = @_i, \\ @_{n+1} \diamond E \diamond \eta(@_{n+1}, x, A), \text{ sinon,} \end{array} \right\}, \text{ avec } A = \left[\begin{array}{c} A_1 \\ \vdots \\ A_n \end{array} \right]$$

Par abus de langage, nous appellerons *adresse* la première valeur de retour de cet opérateur et de celui décrit dans la section suivante, qu'elle soit une donnée ou un identificateur.

C'est une fonction à trois arguments, un acteur, un environnement et un vecteur. L'acteur est à insérer dans le code linéaire et le vecteur est le code linéaire déjà produit. L'environnement est un paramètre qui ne sert que pour des raisons de compatibilité avec le compilateur décrit dans la section suivante. Cette fonction a trois valeurs de retour, un acteur, un environnement et un vecteur, noté $x \diamond E \diamond A$.

Si l'acteur à produire est une donnée de l'utilisateur ou un entier, le résultat est le couple de l'acteur et du vecteur, ce qui signifie qu'une donnée ne produit pas de code.

Si l'acteur à produire est un acteur composé, le combinateur vérifie s'il n'est pas déjà dans le code linéaire en utilisant la fonction κ (§ 3.2.7). Si c'est le cas, il retourne l'identificateur associé. Sinon, il crée une nouvelle définition dans le code linéaire et retourne le couple formé de l'identificateur de cette définition et du nouveau code linéaire.

La fonction SCHEME réalisant le producteur est :

◇ **Compiler/producer**

```
(define (produce actor env code)
  (case (type actor)
    [(integer operator flag) (list actor env code)]
    [else
     (let ([ident (identOf actor code)])
       (if (eq? ident 'bottom)
           (let ([add (address code)])
             (list add env (extend add actor code)))
           (list ident env code)))]))
```

Cette fonction ne présente pas de difficultés particulières. Elle utilise la fonction `'identOf` définie ultérieurement (§ 3.2) pour rechercher l'identificateur associé à la valeur `actor`. Si la valeur n'est pas associée, un nouvel identificateur est créé avec la fonction `address` ci-dessous. Le couple est créé avec le constructeur de paire `cons` :

◇ **Compiler/address**

```
(define (address code)
  (if (eq? code 'bottom)
      '@_1
      (string->symbol (string-append
                       "@_"
                       (number->string (length (frame code)))))))

(define (address? add)
  (if (symbol? add)
      (let ((str (symbol->string add)))
        (and (eq? (string-ref str 0) #\@)
             (eq? (string-ref str 1) #\_)))
      #f))
```

Cette fonction retourne une nouvelle adresse de la forme `@_x`, où `x` est le nombre de définitions déjà contenues dans le code linéaire, qui est considéré ici comme un environnement.

On trouve aussi le prédicat `address?` qui est vrai si l'identificateur passé en argument est une adresse[3].

5.2.3 Linéarisation ◯

La technique utilisée pour le compilateur formel permet une spécification entièrement fonctionnelle. Le compilateur compile un acteur dans son environnement courant. Il modifie cet environnement et le vecteur contenant les définitions du code linéaire. Un acteur compilé possède une adresse dans le code linéaire. Cette adresse est en fait un identificateur dont le prefix est `@_` et le postfix, le numéro de l'adresse. Le résultat de la compilation d'un acteur est donc le triplet formé par son adresse, l'environnement modifié et le nouveau code linéaire. Cette technique est réalisée plus facilement avec les outils de la sémantique dénotationnelle, et plus particulièrement avec les continuations. Mais nous conservons le formalisme algébrique utilisé jusqu'à maintenant car il est plus général.

Le combinateur du compilateur est défini par :

$$\bigcirc : \mathcal{X} \times \Gamma \times \mathcal{V} \longrightarrow \mathcal{X} \times \Gamma \times \mathcal{V} \quad | \tag{5.7}$$

$$\bigcirc_{E,A}\, x = x \diamond E \diamond A \qquad \text{(atome)}$$

$$\bigcirc_{E,A}\, \{c_1, c_2, ..., c_n\}_t = \gamma(\{@_1, @_2, ..., @_n\}_t, E_n, A_n), \qquad \text{(cas général)}$$

$$\text{avec} \begin{cases} @_i \diamond E_i \diamond A_i = \bigcirc_{E_{i-1}, A_{i-1}} c_i, \\ A_0 = A, E_0 = E \end{cases}$$

$$\bigcirc_{E,A}\, i := v = @ \diamond E'' \diamond \bar{\eta}(@_{n+1}, @, A''), \text{ avec } \begin{cases} @ \diamond E'' \diamond A'' = \bigcirc_{E',A'} v, \\ A = \begin{bmatrix} @_0 := v_0 \\ \cdots \\ @_n := v_n \end{bmatrix} \\ E' = \eta(i, @_{n+1}, E) \\ A' = \eta(@_{n+1}, \bot, A) \end{cases} \qquad \text{(définition)}$$

$$\bigcirc_{E,A}\, m \cdot n = @' \diamond E'' \diamond A'', \qquad \text{(extraction)}$$

$$\text{avec} \begin{cases} m' \diamond E' \diamond A' = \bigcirc_{E,A} m, \\ @ = \begin{bmatrix} @_1 \\ \cdots \\ @_k \end{bmatrix} = \begin{cases} \rho(m', A'), \text{ si } m' \in \mathcal{I}, \\ m', \text{ sinon}, \end{cases} \\ @' = \delta(@, n), E' = E'', A'' = A', \text{ si } n \in \mathcal{Z}, \\ @' \diamond E'' \diamond A'' = \bigcirc_{A',E'} (= (n,1) \rightarrow @_1, \\ \qquad\qquad\qquad = (n,2) \rightarrow @_2, \\ \qquad\qquad\qquad \cdots \\ \qquad\qquad\qquad = (n,k) \rightarrow @_k, \\ \qquad\qquad\qquad @_1), \text{ sinon} \end{cases}$$

3. Le niveau de sécurité de ce prédicat est suffisant si nous interdisons à l'utilisateur les identificateurs qui commencent par le préfixe des adresses.

$$\bigcirc_{E,A}\, i = \begin{cases} @\diamond E\diamond A, \text{ si } i = @ \text{ ou } v = @, \\ i, \text{ si } v = \bot, \\ \bigcirc_{E_v,A}\ i{:=}n, \text{ sinon,} \end{cases} \text{, avec } \begin{cases} v = \rho(i,E), \\ E_v = \mu(i,E) \end{cases} \qquad \text{(identificateur)}$$

$$\bigcirc_{E,A} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} @_1 \\ \vdots \\ @_n \end{bmatrix} \diamond E' \diamond A_n, \text{ avec } \begin{cases} @_i \diamond E_i \diamond A_i = \bigcirc_{E_{i-1},A_{i-1}}\, c_i, \\ A_0 = A, \\ E_0 = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \star E, \\ E_n = v \star E', \end{cases} \qquad \text{(vecteur)}$$

C'est une fonction à trois paramètres, un acteur, son environnement courant E et vecteur A représentant le code linéaire. Il retourne le triplet formé d'un acteur, de l'environnement modifié et d'un vecteur, le nouveau code linéaire.

La compilation d'un atome produit le triplet de cet atome, de l'environnement et du code linéaire inchangé, s'il n'est pas un identificateur.

Dans le cas général, le résultat de la compilation d'un acteur $\{c_1, c_2, ..., c_n\}_t$ est le résultat de la production de l'acteur $\{@_1, @_2, ..., @_n\}_t$ de même type, dont les composantes sont les adresses issues de la compilation des composantes de l'acteur.

La compilation d'une définition $i := v$ est la plus complexe de toute. En effet, une définition peut être cyclique, comme pour l'expression $x := 0\, f_{by}\, x$. L'idée est de créer une adresse associée à une valeur arbitraire ($A' = \eta(@_{n+1}, \bot, A)$) dans le code linéaire et de modifier l'environnement de la définition en associant l'identificateur à l'adresse créée ($E' = \eta(i, @_{n+1}, E)$). Nous compilons alors la valeur de la définition ($\bigcirc_{E',A'} v$). Si la valeur fait référence directement ou non à l'identificateur s, la valeur associée $@_{n+1}$ sera retournée (sa compilation retourne $@_{n+1}$). Ainsi, les cycles peuvent être traités grâce à une technique proche des définitions *en avant*, utilisées généralement dans les compilateurs pour résoudre les sauts à une adresse non encore déterminée [4, 5]. Lorsque la valeur est compilée, son adresse remplace la valeur arbitraire initialement associée dans le code linéaire ($\bar{\eta}(@_{n+1}, @, A'')$).

L'indexé m d'une extraction $m \cdot n$ est compilé en premier. Si le résultat est un identificateur, la valeur associée est recherchée. Si l'index n est un entier, la composante correspondante est extraite du le résultat de la compilation de l'indexé ($\delta(@, n)$). Si l'index n'est pas un entier, une série d'alternatives imbriquées qui extrait la composante en fonction de l'index est compilée ($= (n, k) \rightarrow \delta(@, k), ...$).

Si un identificateur i n'est pas une adresse @ ou si sa valeur associée est une adresse, le résultat est le triplet de cette adresse, de l'environnement et du code linéaire. Si aucune valeur ne lui est associée, le triplet de l'identificateur de l'environnement et du code linéaire est retourné. Ceci permet la compilation partielle où les variables libres sont conservées telles quelles. Sinon, la définition formée de cet identificateur et de sa valeur associée est compilée dans l'environnement associé ($\bigcirc_{E_v,A}\ i{:=}n$). Ainsi, la protection contre les cycles introduits par la compilation des définitions est maintenue. Si l'identificateur i ou si sa valeur associée v est une adresse @, il n'est pas compilé et le triplet formé de cette adresse @, de l'environnement modifié et du code linéaire A est retourné.

Enfin, la compilation d'un vecteur $\begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$ produit un triplet formé d'un vecteur $\begin{bmatrix} @_1 \\ \vdots \\ @_n \end{bmatrix}$ dont les composantes sont les adresses des composantes compilées, de l'environnement modifié et du nouveau code linéaire.

La définition SCHEME réalisant cette fonction est :

◇ **Compiler/compile**

```
(define (compile actor env code)
  (case (type actor)
    [(integer operator) (list actor env code)]
    [definition
      (let ((val (valueOf (identifier actor) env)))
        (if (address? val)
            (list val env code)
            (let* ([add  (address code)]
                   [env  (assign (identifier actor) add env)]
                   [code (extend add 'top code)]
```

```
                [cval (compile (value actor) env code)])
           (list (car cval)
                 (cadr cval)
                 (assign add
                         (car cval)
                         (caddr cval))))))]
[extraction
 (let* ([index    (index   actor)]
        [cindexed (compile (indexed actor) env code)]
        [vector   (if (address? (car cindexed))
                      (valueOf (car cindexed) (caddr cindexed))
                      (car cindexed))]
        [len      (length (components vector))])
   (if (eq? (type index) 'integer)
       (let ([comp (indexOf vector index)])
         (list (if (eq? comp 'bottom)
                   (indexOf vector 1)
                   comp)
               (cadr cindexed)
               (caddr cindexed)))
       (let loop ([n len]
                  [_else (indexOf vector 1)])
         (if (eq? n 0)
             (compile _else (cadr cindexed) (caddr cindexed))
             (loop (- n 1)
                   (alternative: (application: =? index n)
                                 (indexOf vector n)
                                 _else)))))))]
[identifier
 (if (address? actor)
     (list actor env code)
     (let ([val (valueOf actor env)])
       (if (eq? val 'bottom)
           (list actor env code)
           (if (address? val)
               (list val env code)
               (compile (definition: actor val)
                        (assocOf actor env)
                        code))))))]
[vector
 (let loop ([comps  (components actor)]
            [ccomps '()]
            [env    (chain: actor env)]
            [code   code])
   (if (null? comps)
       (list (apply vector: (reverse ccomps))
             (parent env)
             code)
       (let ([ccomp (compile (car comps) env code)])
         (loop (cdr comps)
               (cons (car ccomp) ccomps)
               (cadr ccomp)
               (caddr ccomp))))))]
[else
 (let loop ([comps  (cdr actor)]
            [ccomps '()]
            [env    env]
            [code   code])
   (if (null? comps)
       (produce (cons (car actor) (reverse ccomps))
                env
                code)
       (let ([ccomp (compile (car comps) env code)])
         (loop (cdr comps)
               (cons (car ccomp) ccomps)
               (cadr ccomp)
               (caddr ccomp))))))]))
```

La structure générale est la même que celle de la définition formelle. Cependant certaines techniques propres à SCHEME sont utilisées.

La compilation des acteurs dans le cas général (cas `else`) est basée sur un `let` nommé qui parcourt toutes les composantes de l'objet. Un objet de même type `(car actor)` est construit, dont les composantes sont le résultat des compilations des composantes. Nous retrouvons cette structure dans le cas des `vecteurs`.

5.3 Exemple

5.3.1 Extraction multiple

Cet exemple traite une extraction où l'index est une variable libre. Il montre comment le compilateur formel permet d'obtenir des compilations partielles. De plus, la variable libre peut à priori prendre toutes les valeurs entières. Le compilateur transforme donc la simple extraction en une succession d'alternatives testant la variable libre.

Commençons par définir le programme :

```
STk> (define a (extraction: (vector: 10 20 30 40) 'e))
#unspecified
```

Puis définissons la variable `triplet` comme le résultat de la compilation de ce programme :

```
STk> (define triplet (compile a 'bottom 'bottom))
#unspecified
```

La variable `triplet` est le triplet formé de l'adresse du programme compilé, de l'environnement modifié, qui ne nous intéresse pas ici, et du code linéaire. Examinons l'adresse de l'acteur :

```
STk> (convert (car triplet))
@_8
```

puis examinons le code linéaire :

```
STk> (convert (frame (caddr triplet)))
[
  @_8 := @_1 -> 10, @_7
  @_7 := @_2 -> 20, @_6
  @_6 := @_3 -> 30, @_5
  @_5 := @_4 -> 40, 10
  @_4 := =? (e, 4)
  @_3 := =? (e, 3)
  @_2 := =? (e, 2)
  @_1 := =? (e, 1)
]
```

L'opérateur `=?` fait partie de l'algèbre des entiers et permet leur comparaison.

Nous constatons que le code linéaire ne contient plus de vecteur ni d'extraction. L'extraction est remplacée par une succession d'alternatives testant la valeur de l'index. Notons que le nom `e` de la variable libre est conservé.

5.3.2 Filtre

Cet exemple reprend l'exemple traité tout au long de cette étude théorique et présenté auparavant (§ 2.3). Questionnons l'interprète SCHEME pour connaître la signature du système :

```
STk> (sign system 'bottom)
(sign-vec n n n)
```

La réponse indique que le système est un vecteur dont les trois composantes sont des entiers. Le système est-il constant ? Pour savoir cela, entrons la commande :

```
STk> (constant system 'bottom)
1
```

La réponse est positive, ce qui indique que le contrôle des types n'a détecté aucune erreur.

Nous pouvons donc compiler le système et conserver le résultat dans la variable `triplet` avec la commande :

```
STk> (define triplet (compile system 'bottom 'bottom)
#unspecified
```

Le résultat est le triplet formé de l'adresse du code compilé, de l'environnement modifié et du code linéaire. Que vaut l'adresse du programme compilé? Pour l'obtenir, entrons :

```
STk> (convert (car triplet))
[
  @_6
  @_2
  @_21
]
```

Nous voyons que cette adresse est en fait un vacteur d'adresses, correspondant au programme. Le code compilé est alors :

```
STk>(convert (frame (caddr triplet)))
[
  @_21 := + (@_18, @_20)
  @_20 := + (@_10, @_19)
  @_19 := + (@_12, @_11)
  @_18 := - (@_2, @_17)
  @_17 := 0 fby @_16
  @_16 := + (@_15, @_13)
  @_15 := - (@_9, @_14)
  @_14 := 0 fby @_12
  @_13 := @_14
  @_12 := @_15
  @_11 := @_16
  @_10 := @_17
  @_9  := @_18
  @_8  := @_21
  @_7  := @_21
  @_6  := @_5 -> 1, 0
  @_5  := =? (@_4, 10)
  @_4  := 1 fby @_3
  @_3  := + (@_2, 1)
  @_2  := @_4
  @_1  := @_6
]
```

On retrouve le fait que la compilation des définitions produit des identificateurs supplémentaires, qui peuvent être supprimés simplement à l'aide d'un petit analyseur.

Chapitre 6

Vérifications formelles

L'objet des vérifications formelles est de montrer que l'expression fonctionnelle de la machine abstraite de résolution permet d'établir des preuves de programmes. De plus, elles montrent que, d'une part, l'expression des fonctions de critères sont en adéquation avec la propriété recherchée, et que, dautre part, la propriété se conserve avec la régénération.

6.1 Preuve de programme

Dans cette section, nous montrons l'utilisation de la machine abstraite de résolution avec sa propriété fonctionnelle pour établir des preuves de programmes simples. Ces preuves sont simplement déduites des règles de transformations de la machine abstraite.

Cette section montre comment il serait possible d'exploiter le langage et sa formalisation fonctionnelle pour établir des preuves de programmes. Il serait intéressant de poursuivre cette ébauche de manière à construire un système de preuves complet des programmes.

6.1.1 Propriété d'un programme

Considérons le programme suivant :

$$S = [\, o{:=}0\, f_{by}\ +(o,\, i)] \tag{6.1}$$

Nous voulons montrer que si toutes les valeurs de i sont positives, alors toutes les valeurs de o sont aussi positives. Posons l'équation temporelle de S :

$$S_n = [\, o{:=}o_n\, f_{by}\ +(o,\, i_n)] \tag{6.2}$$

L'expression de la résolution est une itération qui s'arrête lorsqu'une condition d'arrêt est rencontrée. Dans cet exemple, nous supprimons toute condition d'arrêt. L'expression de la résolution devient donc une simple itération basée sur la régénération. Considérons le système S_{n+1}, le régénéré de S_n :

$$S_{n+1} = \bigtriangledown_{E_0} [\, o{:=}o_n\, f_{by}\ +(o,\, i_n)] \tag{6.3}$$

où E_0 est l'environnement qui définit l'entrée i. Il vient :

$$\begin{aligned} S_{n+1} &= [\, \bigtriangledown_{S'_n}\ o{:=}o_n\, f_{by}\ +(o,\, i_n)] \\ &= [\, o{:=}(\, \bigtriangleup_{S'_n} +(o,\, i_n))\, f_{by}\ \bigtriangledown_{S'_n} +(o,\, i_n)] \\ &= [\, o{:=}(o_n + i_n)\, f_{by}\ +(o,\, i_{n+1})] \end{aligned} \tag{6.4}$$

avec $S'_n = S_n \star E_0$. L'expression temporelle de l'état de o est $o_n + i_n$, avec $o_0 = 0$. Par conséquence, si $i_n > 0$, l'état est toujours supérieur à zéro.

6.1.2 Expression temporelle

L'exemple choisit ici utilise le filtre numérique introduit dans les sections précédentes (§ 2.3). Rappelons l'expression de ce filtre :

$$S = \left[\begin{array}{l} o{:=}+(+(+(a,\ b),\ c),\ d) \\ a{:=}-(i,\ b) \\ b{:=}0\ f_{by}\ d \\ c{:=}-(a,\ e) \\ d{:=}+(c,\ e) \\ e{:=}0\ f_{by}\ c \end{array} \right]. \tag{6.5}$$

Nous voulons obtenir formellement l'expression temporelle de ce filtre. Pour cela, nous utiliserons des indices. Nous avons constaté que la régénération ne modifie dans un programme que l'état des flots, et laisse inchangés les autres acteurs. Considérons le filtre (6.5) à l'instant n :

$$S_n = \left[\begin{array}{l} o{:=}+(+(+(a,\ b),\ c),\ d) \\ a{:=}-(i_n,\ b) \\ b{:=}b_n\ f_{by}\ d \\ c{:=}-(a,\ e) \\ d{:=}+(c,\ e) \\ e{:=}e_n\ f_{by}\ c \end{array} \right]. \tag{6.6}$$

où i_n est la valeur de l'entrée à l'instant n, b_n la valeur de l'état du flot b à l'instant n et e_n la valeur du flot e à l'instant n. A l'instant 0, i vaut i_0, b vaut 0 ainsi que e_0. Considérons le système S_{n+1}, le régénéré de S_n :

$$S_{n+1} = \bigtriangledown_{E_0} S_n \tag{6.7}$$

où l'environnement E_0 ne définit que l'entrée i en l'associant à la valeur i_{n+1}. Calculons l'expression de S_{n+1} [1] :

$$S_{n+1} = \left[\begin{array}{l} o{:=}+(+(+(a,\ b),\ c),\ d) \\ a{:=}-(i_{n+1},\ b) \\ \bigtriangledown_{S'_n}\ \ b{:=}b_n\ f_{by}\ d \\ c{:=}-(a,\ e) \\ d{:=}+(c,\ e) \\ \bigtriangledown_{S_n}\ \ e{:=}e_n\ f_{by}\ c \end{array} \right]. \tag{6.8}$$

avec $S'_n = S_n \star E_0$. Les composantes de S_n restent inchangées par la régénération, hormis celles qui dépendent de l'entrée i et celles qui sont des flots. Quelles sont les valeurs des états des flots b et e? Nous avons :

$$\begin{array}{rcl} \bigtriangledown_{S'_n}\ (\ b{:=}b_n\ f_{by}\ d) & = & b{:=}\bigtriangleup_{S'_n}\ d\ f_{by}\ \ \bigtriangledown_{S'_n}\ d \\ & = & b{:=}\bigtriangleup_{S'_n}\ +(c,\ e)\ f_{by}\ d \\ & = & b{:=}\triangleleft +(\ \bigtriangleup_{S'_n}\ c,\ \bigtriangleup_{S'_n}\ e)\ f_{by}\ d \\ & = & b{:=}\bigtriangleup_{S'_n}\ c + \bigtriangleup_{S'_n}\ e\ f_{by}\ d \\ & = & b{:=}i_n - b_n\ f_{by}\ d \end{array} \tag{6.9}$$

où les résultats de la réduction sont notés : $\triangleleft +(a,\ b) = a + b$.

1. Pour calculer le régénéré, il ne faut pas oublier l'existence de deux systèmes, le système original et le système en construction. Les expressions sont recherchées dans le système original.

De même, nous montrons que la régénération de e a pour résultat :

$$\nabla_{S'_n}\ (\ e{:=}e_n\ f_{by}\ c) = I_n - b_n - e_n\ f_{by}\ c = b_n - e_n\ f_{by}\ c \tag{6.10}$$

Donc, le résultat de la régénération de S_n est le système :

$$S_{n+1} = \left[\begin{array}{l} o{:=}+(+(+(a,\ b),\ c),\ d) \\ a{:=}-(i_n,\ b) \\ b{:=}(i_n - b_n)\ f_{by}\ d \\ c{:=}-(a,\ e) \\ d{:=}+(c,\ e) \\ e{:=}(b_n - e_n)\ f_{by}\ c \end{array}\right]. \tag{6.11}$$

avec $b_0 = e_0 = 0$. C'est une équation temporelle obtenue avec l'application de la machine fonctionnelle. Cette expression de l'évolution du programme est très intéressante car elle permet de prévoir son évolution dans le temps.

6.2 Schéma général

Dans les sections qui vont suivre, nous allons vérifier un certain nombre de propriétés des λ-matrices, quant aux fonctions de critères. Ces vérifications formelles reposent sur des inductions structurelles par hypothèse de récurrence. Voici le schéma général de ces vérifications :

Soit une hypothèse de récurrence :

$$H : P_1 \Rightarrow P_2 \tag{6.12}$$

Par inductions structurelles, nous obtenons :

$$I_1 : P_1 \Rightarrow P'_1 \tag{6.13}$$

Ce permet d'établir, avec l'hypothèse de récurrence :

$$I_2 : P'_1 \Rightarrow P'_2 \tag{6.14}$$

Si :

$$C : P'_2 \Rightarrow P_2 \tag{6.15}$$

l'hypothèse de récurrence est fondée pour cette induction.

6.3 Acteurs impliqués

Les définitions des combinateurs de la machine abstraite (§ 3) et des fonctions de critère (§ 4) sont récursives. Elle sont mathématiquement fondées car le texte définissant est «plus simple» que le texte défini. Nous laissons le soin aux spécialistes des définitions formelles de savoir s'il s'agit de définitions ou d'axiomes [40].

Dans de nombreux cas, nous aurons besoin de connaître la liste des acteurs entrant en jeu dans le calcul d'une expression. Pour cela, nous nous inspirons des notations utilisées par Von Neumann dans son théorème de la récursivité [40].

D'une manière formelle, pour tout combinateur ayant comme argument un acteur x dans son environnement E, nous pouvons écrire :

$$\begin{array}{l} f(x,E) = F((x,E), cod(f|_{x,E})), \\ \text{avec } cod(f|_{x,E}) = \{f(x_1,E_1), \ldots, f(x_n,E_n)\}, \text{ et } n \geq 0 \end{array} \tag{6.16}$$

qui signifie que $f(x,E)$ est une fonction F de x et d'un certain nombre de calculs intermédiaires, rassemblés dans l'ensemble $cod(f|_{x,E})$.

Nous appellerons $\widehat{f(x,E)}$ l'ensemble des acteurs impliqués dans l'évaluation de $f(x,E)$ tels que $\widehat{f(x,E)} = \{(x_i,E_i) : f(x_i,E_i) \in cod(f|_{x,E})\}$.

Nous avons comme propriété issue de la définition :

$$\forall i \in [1,n], \widehat{f(x_i,E_i)} \subset \widehat{f(x,E)} \tag{6.17}$$

6.4 Calculabilité

Le critère de calculabilité certifie qu'un acteur ne contient pas de cycle direct (par opposition aux cycles récurrents). Le critère de la calculabilité vérifie l'absence de cycle dans un acteur.

Dans une première étape, nous allons montrer que l'expression du critère correspond bien à la propriété recherchée, c'est à dire que tout acteur calculable ne contient pas de cycle et réciproquement et inversement.

6.4.1 Vérification du combinateur

La nature inductive de la grammaire qui définit les λ-matrices stipule qu'un acteur ne peut se contenir lui-même. Il n'est possible de créer un cycle dans l'évaluation des acteurs qu'avec des identificateurs, en créant des cycles symboliques.

La propriété d'un acteur calculable est la suivante :

$$H : \Lambda_c(x,E) \Rightarrow (x,E) \notin \widehat{\triangle_E x} \tag{6.18}$$

où $\widehat{\triangle E x}$ est l'ensemble des acteurs impliqués par l'évaluation de x dans E. Cette propriété sert d'hypothèse de récurrence. Vérifions-la par induction sur tous les acteurs du langage.

Identificateur La calculabilité d'un identificateur est :

$$\Lambda_c(i,E) = \Lambda_c(v, \eta(i,\top,E_v)), \text{avec } v = \rho(i,E) \text{ et } E_v = \mu(i,E). \tag{6.19}$$

Comme la calculabilité d'un acteur associé à la valeur $\top$ est fausse, l'identificateur i ne participe pas au calcul. Nous obtenons :

$$I_1 : \Lambda_c(i,E) \Rightarrow \Lambda_c(v,E_v) \tag{6.20}$$

On a, par hypothèse de récurrence :

$$I_2 : \Lambda_c(v,E_v) \Rightarrow (v,E_v) \notin \widehat{\triangle_{E_v} v} \tag{6.21}$$

L'évaluation nous indique que $\triangle_E i = \triangle_{E_v} v$, donc $\widehat{\triangle_E i} = \{(v,E_v)\} \cup \widehat{\triangle_{E_v} v}$. Si $(i,E) \in \widehat{\triangle_{E_v} v}$, nous aurions nécessairement $(v,E_v) \in \widehat{\triangle_{E_v} v}$, donc nous pouvons conclure :

$$C : (v,E_v) \notin \widehat{\triangle_{E_v} v} \Rightarrow (i,E) \notin \widehat{\triangle_E i} \tag{6.22}$$

Flot Posons $x = e\, f_{by}\, c$. Si un flot x est calculable, alors :

$$I_1 : \Lambda_c(x, E) \Rightarrow \begin{cases} \Lambda_c(e, E) \\ \Lambda_c(c, \bot) \end{cases} \tag{6.23}$$

Par hypothèse de récurrence, nuos avons :

$$I_2 : \Lambda_c(e, E) \Rightarrow (e, E) \notin \widehat{\triangle_{E}}\, e \tag{6.24}$$

On a $\triangle_{E}\, x = \triangle_{E}\, e$, donc $\widehat{\triangle_{E}}\, x = \{(e, E)\} \cup \widehat{\triangle_{E}}\, e$. Si $(x, E) \in \widehat{\triangle_{E}}\, e$, nous aurions nécessairement $(e, E) \in \widehat{\triangle_{E}}\, e$, donc nous pouvons conclure :

$$C : (e, E) \notin \widehat{\triangle_{E}}\, e \Rightarrow (x, E) \notin \widehat{\triangle_{E}}\, x \tag{6.25}$$

Le contrat du flot est évalué par l'opérateur de régénération dans l'environnement du flot. De $\Lambda_c(c, \bot)$ pouvons-nous conclure que $c \notin \widehat{\triangle_{E}}\, c$? La réponse est oui parce que l'évalué d'un flot est l'évalué de son état, indépendamment de son contrat : $\triangle_{E}\, e\, f_{by}\, c = \triangle_{E}\, e$. Donc l'évaluation du contrat peut faire référence au flot lui-même, sans atteindre le contrat. Cela se traduit par :

$$\Lambda_c(c, \bot) \Rightarrow c \notin \widehat{\triangle_{E}}\, c \tag{6.26}$$

Cas général Considérons l'acteur $x = \{c_1, c_2, \ldots, c_n\}_t$. Nous avons :

$$I_1 : \Lambda_c(x, E) \Rightarrow \prod \Lambda_c(c_i, E), \text{ pour } i \in [i, n] \tag{6.27}$$

Par hypothèse de récurrence, nous obtenons :

$$I_2 : (c_i, E) \notin \widehat{\triangle_{E}}\, c_i \tag{6.28}$$

On a $\triangle_{E}\, x = F(x, cod(\triangle|_{x,E}))$, avec le codomaine $cod(\triangle|_{x,E}) = \{\triangle_{E}\, c_1, \ldots, \triangle_{E}\, c_n\}$. Donc $\widehat{\triangle_{E}}\, x = \{(c_i, E)_i\} \cup \widehat{\triangle_{E}}\, c_{ii}$. Si $(x, E) \in \widehat{\triangle_{E}}\, c_i$, nous aurions nécessairement $(c_i, E) \in \widehat{\triangle_{E}}\, c_i$, donc nous pouvons conclure :

$$C : \forall i \in [1, n], ((c_i, E) \notin \widehat{\triangle_{E}}\, c_i) \Rightarrow (x, E) \notin \widehat{\triangle_{E}}\, x \tag{6.29}$$

Les autres cas sont traités de manière identique.

On admettra que l'ensemble des acteurs impliqués par l'évaluation est inclus dans l'ensemble des acteurs impliqués par la calculabilité, ce qui se démontre simplement par induction. La différence porte sur l'évaluation du flot qui le traite, pas du contrat, et sur l'évaluation de l'alternative qui ne traite que l'une des deux clauses. Ainsi, tout acteur impliqué par l'évaluation d'un acteur calculable est aussi calculable.

6.4.2 Conservation avec la régénération

L'objet de cette vérification est de montrer que le régénéré d'un acteur calculable est aussi calculable dans l'environnement régénéré. Ceci s'énonce :

$$\Lambda_c(x, E) \Rightarrow \Lambda_c(\bigtriangledown_{E}\, x, \overset{\triangledown}{E}) \tag{6.30}$$

Cette démonstration s'appuie sur le fait que l'opérateur de la régénération ne modifie que l'état des flots, en laissant inchangé le reste des acteurs du programme (§ 3.3.2). De plus, le nouvel état des flots est issu de l'évaluation du contrat, qui existe car l'acteur est calculable. Or le résultat de l'évaluation appartient à l'ensemble des acteurs figés (§ 3.3.3) (§ 2.3). Cet ensemble ne contient pas les identificateurs. Comme le critère de la calculabilité ne peut être faux que sur un identificateur, le nouvel état des flots ne peut l'annuler. Nous en concluons que le régénéré d'un acteur calculable est aussi calculable.

6.5 Fermeture

6.5.1 Vérification du combinateur

On admettra que l'ensemble des acteurs impliqués par l'évaluation est inclus dans l'ensemble des acteurs impliqués par le critère de la fermeture. Autrement dit, l'évaluation d'un acteur impliqué dans la fermeture d'un acteur est nécessairement fermé.

Une variable libre dans un programme se caractérise par le fait qu'aucune valeur ne lui est associée dans son environnement. C'est le combinateur de l'évaluation qui retourne la valeur $\perp$ lorsqu'un tel cas se produit. Cette valeur est produite par l'opérateur ρ qui recherche la valeur associée à un identificateur dans un environnement.

Le combinateur de la fermeture est annulé dès qu'un identificateur dans une expression n'est associé à aucune valeur dans son environnement. Cette annulation se produit dès que l'opérateur ρ retourne la valeur $\perp$ sur un identificateur.

Donc si un acteur annule la fermeture il sera détecté par l'évaluation. Comme la caractérisation du critère de fermeture est relatif à l'évaluation, il possible de conclure que l'expression de la fermeture détecte bien les variables libres d'un programme. De plus, on établit que si un programe contient une variable libre, il annule le critère.

6.5.2 Conservation de la propriété

Ici, nous désirons montrer que si un programme est fermé, alors son régénéré l'est aussi. Pour cette démonstration, il est possible d'utiliser une méthode d'induction structurelle. Mais, il est plus facile d'établir cette propriété à partir de certaines observations.

Le première chose à noter est que l'opérateur de la régénération ne modifie que l'état des flots. Les composantes des autres objets ne sont pas affectées par ce combinateur.

La seconde chose à observer est que le résultat de l'évaluation appartient à un ensemble d'acteurs restreint que nous avons appelés «acteurs figés». Cet ensemble ne contient que les données et les vecteurs.

Si un programme est fermé, tous ses identificateurs sont associés à une valeur dans leur environnement courant. La régénération d'un tel programme n'affecte que l'état des flots. Donc, tous les identificateurs présents hors de l'état des flots restent associés à une valeur. Quant à l'état des flots du programme régénéré, ils sont le résultat de l'évaluation de leur contrat qui appartient de ce fait à l'ensemble des acteurs figés. Comme l'ensemble des acteurs figés ne contient pas d'identificateur, ces nouveaux états ne peuvent en aucun cas annuler le critère de la fermeture.

On peut donc conclure que le critère de fermeture se conserve avec la régénération.

6.6 Stabilité

Cette section présente les vérifications formelles concernant la stabilité. Elles sont présentées sous une forme plus systématique que les vérifications précédentes car elles s'y prêtent bien.

Dans ce qui suit, H_i sont des hypothèses, R_i sont des règles ou déductions, P_i sont des références à des preuves de numéro i, et D_i sont des références à des définitions où i est le numéro de la section définissante.

Preuve 1 : $\boldsymbol{\Lambda_s(x, E) \Rightarrow |x| = |\bigtriangledown_E x|}$

Cette preuve vérifie par induction que les dimensions d'un acteur stable et de son régénéré sont identiques.

Si x est un acteur quelconque :

H_1: $x = \{c_1, c_2, \ldots, c_n\}_t$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2.D_{4.1.3} \rightarrow \prod_{i=1}^{n} \Lambda_s(c_i, E) = 1$
R_2: $P_1.R_1 \rightarrow |c_i| = |\bigtriangledown_E c_i|$
R_3: $H_1.D_{3.1.2}.D_{3.3.2} \rightarrow |\bigtriangledown_E x| = \sum_{i=1}^{n} |\bigtriangledown_E c_i|$
P_1: $R_3 \rightarrow |x| = |\bigtriangledown_E x|$

Si x est un vecteur :

H_1: $x = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2.D_{4.1.3} \rightarrow \prod_{i=1}^{n} \Lambda_s(c_i, E') = 1$
R_2: $P_1.R_1 \rightarrow |c_i| = |\bigtriangledown_{E'} c_i|$
R_3: $H_1.D_{3.1.2}.D_{3.3.2} \rightarrow |\bigtriangledown_E x| = \sum_{i=1}^{n} |\bigtriangledown_{E'} c_i|$
P_1: $R_3 \rightarrow |x| = |\bigtriangledown_E x|$

Si x est un atome :

H_1: $x \in \mathcal{A}$
H_2: $\Lambda_s(x, E) = 1$
P_1: $D_{3.1.2}.D_{3.3.2}.D_{4.1.3} \rightarrow |x| = |\bigtriangledown_E x|$

Si x est un flot :

H_1: $x = e\, f_{by}\, c$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow \Lambda_s(c, E) = 1$
R_2: $H_1.H_2 \rightarrow e \in \overline{\mathcal{X}}$
R_3: $H_1.H_2 \rightarrow \sigma(e, E) = \sigma(c, E)$
R_4: $R_1.P_1 \rightarrow |c| = |\bigtriangledown_E c|$
R_5: $P_5.P_3.R_2 \rightarrow |\bigtriangleup_E c| = \sigma(e, E)$
R_6: $P_5.R_1 \rightarrow |\bigtriangleup_E c| = \sigma(c, E)$
R_7: $P_4.R_3.R_6 \rightarrow |E| = \sigma(e, E) = \sigma(c, E) = |\bigtriangleup_E c|$
R_8: $D_{3.1.2}.D_{3.3.2} \rightarrow |\bigtriangledown_E x| = |\bigtriangleup_E c| + |\bigtriangledown_E c|$
P_1: $R_8 \rightarrow |x| = |\bigtriangledown_E x|$

Preuve 2 : $x \in \overline{\mathcal{X}} \Rightarrow \bigtriangleup_E x = x$

Cette preuve vérifie que l'évalué d'un acteur est identique à l'acteur.

Les acteurs figés sont des atomes sans les identificateurs et des vecteurs dont les composantes sont elles-mêmes des acteurs figés (§ 2.3).

L'expression de l'évaluation montre que l'évaluation d'un atome qui n'est pas un identificateur est lui-même, et que l'évaluation d'un vecteur est le vecteur des évalués (§ 3.3.3).

L'évaluation d'un acteur figé ne modifie donc pas cet acteur.

Preuve 3 : $x \in \overline{\mathcal{X}} \Rightarrow \Lambda_s(x, E) = 1$

Cette preuve montre qu'un acteur figé est stable.

Comme un atome qui n'est pas un identificateur est stable et comme un vecteur est stable si toutes ses composantes le sont (§ 4.1.3), un acteur figé (§ 2.3) est nécessairement stable.

Preuve 4 : $x \in \overline{\mathcal{X}} \Rightarrow |x| = \sigma(x, E)$

Cette preuve montre que la dimension et la norme d'un acteur figé sont identiques.

La norme (§ 4.1.4) et la dimension (§ 3.1.2) d'un atome qui n'est pas un identificateur sont 1. La norme et la dimension d'un vecteur sont respectivement la somme des normes et la somme des dimensions de ses composantes. Nous pouvons donc conclure que la norme et la dimension d'un acteur figé sont identiques.

Preuve 5 : $\Lambda_s(x, E) = 1 \Rightarrow |\triangle_E x| = \sigma(x, E)$

Cette preuve montre que la dimension de l'évalué d'un acteur stable est égale à sa norme.

Si x est une alternative :

H_1: $x = c \rightarrow a, s$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow \Lambda_s(c, E) = \Lambda_s(a, E) = \Lambda_s(s, E) = 1$
R_2: $P_5.R_1 \rightarrow |\triangle_E a| = \sigma(a, E)$
R_3: $P_5.R_1 \rightarrow |\triangle_E s| = \sigma(s, E)$
R_4: $H_2.D_{4.1.4} \rightarrow \sigma(a, E) = \sigma(s, E)$
R_4: $R_2.R_3.R_4 \rightarrow |\triangle_E a|\,|\triangle_E s|$
P_5: $R_4 \rightarrow |\triangle_e x| = \sigma(x, E)$

Si x est une extraction :

H_1: $x = m \cdot i$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow m \in \mathcal{V} \cup \mathcal{I}$, and $i \in \mathcal{Z}$
H_3: $m = \left[\begin{array}{c} c_1 \\ \vdots \\ c_n \end{array}\right] \in \mathcal{V}$
$H_{3.1}$: $i \leq n$
R_2: $\delta(i, m) = c_i$
R_3: $D_{3.3.3} \rightarrow \triangle_E x = \delta(i, \triangle_E m) = \triangle_{E'} c_i$, avec $E' = E \star m$
R_4: $D_{4.1.4} \rightarrow \sigma(x, E) = \sigma(\delta(i, m), E') = \sigma(c_i, E')$
R_5: $P_5.R_3.R_4 \rightarrow |\triangle_{E'} c_i| = \sigma(c_i, E)$
R_6: $R_5 \rightarrow |\triangle_E x| = \sigma(x, E)$
$H_{3.2}$: $i > n$
R_7: $\delta(i, m) = \perp$
R_8: $D_{3.3.3} \rightarrow \triangle_E x = \delta(i, \triangle_E m) = \perp$
R_9: $D_{4.1.4} \rightarrow \sigma(x, E) = \sigma(\delta(i, \triangle_E m), E') = \perp$
R_{10}: $R_8.R_9 \rightarrow |\triangle_E x| = \sigma(x, E)$
H_4: $m = s_1 \in \mathcal{I}$

$$R_{11}: D_{4.1.3}.H_2 \rightarrow \left\{\begin{array}{l} \Lambda_s(\ \rho(s_1, E_1 = E) \cdot i, \mu(s_1, E_1)) = 1 \\ \cdots \\ \Lambda_s(\ \rho(s_k, E_k) \cdot i, \mu(s_k, E_k)) = 1 \\ \Lambda_s(\ m' \cdot i, E') = 1, \text{ avec } m' \in \mathcal{V} \end{array}\right.$$

$$R_{12}: D_{4.1.4}.H_2 \rightarrow \sigma(x, E) = \left\{\begin{array}{l} \sigma(\ \rho(s_1, E_1 = E) \cdot i, \mu(s_1, E_1)) \\ \cdots \\ \sigma(\ \rho(s_k, E_k) \cdot i, \mu(s_k, E_k)) \\ \sigma(\ m' \cdot i, E') \end{array}\right.$$

$$R_{13}: D_{3.3.3}.H_1 \rightarrow \triangle_x E = \left\{\begin{array}{l} \delta(i, \triangle_E s_i) \\ \delta(i, \triangle_{\mu(s_1, E_1 = E)} \rho(s_i, E_1)) \\ \cdots \\ \delta(i, \triangle_{\mu(s_k, E_k)} \rho(s_k, E_k)) \\ \delta(i, \triangle_{E'} m') \end{array}\right.$$

R_{14}: $R_{10} \rightarrow |\delta(i, \triangle_{E'} m')| = \sigma(\ m' \cdot i, E')$
P_5: $R_{14} \rightarrow |\triangle_E x| = \sigma(x, E)$

Si x est un flot :

H_1: $x = e\, f_{by}\, c$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow s \in \overline{\mathcal{X}}$
R_2: $P_2.P_4 \rightarrow |e| = \sigma(e, E) = |\triangle_{E}\, e|$
P_5: $R_2 \rightarrow |\triangle_{E}\, x| = \sigma(x, E)$

Les preuves concernant les autres acteurs sont construites sur le même modèle.

Preuve 6 : $\Lambda_s(x, E) = 1 \Rightarrow \sigma(x, E) = \sigma(\bigtriangledown_{E} x, \overset{\triangledown}{E})$

Cette preuve montre que si un acteur est stable, sa norme et la norme de son régénéré dans l'environnement régénéré sont identiques.

Si x est une alternative :

H_1: $x = c \rightarrow a, s$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow \Lambda_s(a, E) = \Lambda_s(s, E) = 1$
R_2: $P_6.R_1 \rightarrow \sigma(\bigtriangledown_{E} a, \overset{\triangledown}{E}) = \sigma(\bigtriangledown_{E} s, \overset{\triangledown}{E})$
P_6: $R_2 \rightarrow \sigma(x, E) = \sigma(\bigtriangledown_{E} x, \overset{\triangledown}{E})$

Si x est une extraction :

H_1: $x = m \cdot i$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow m \in \mathcal{V} \cup \mathcal{I}$, and $i \in \mathcal{Z}$
H_3: $m = \left[\begin{array}{c} c_1 \\ \cdots \\ c_n \end{array}\right] \in \mathcal{V}$
$H_{3.1}$: $i \leq n$
R_2: $D_{4.1.4} \rightarrow \sigma(x, E) = \sigma(c_i, E')$, avec $E' = E \star m$
R_3: $H_2 \rightarrow \Lambda_s(c_i, E \star m)$

$$R_4\text{: } P_6.R_3 \rightarrow \sigma(c_i, E') = \left\{\begin{array}{l} \sigma(\bigtriangledown_{E'} c_i, \overset{\triangledown}{E'}) \\ \sigma(\delta(i, \bigtriangledown_{E} m), \overset{\triangledown}{E}) \\ \sigma(\bigtriangledown_{E} x, \overset{\triangledown}{E}) \end{array}\right.$$

$H_{3.2}$: $i > n$
R_5: $D_{4.1.4}.H_5 \rightarrow \sigma(x, E) = \sigma(\delta(i, m), E) = \perp$
R_6: $D_{4.1.4} \rightarrow \sigma(\bigtriangledown_{E} x, \overset{\triangledown}{E}) = \sigma(\delta(i, \bigtriangledown_{E'} m), \overset{\triangledown}{E'}) = \perp$
R_7: $R_6 \rightarrow \sigma(x, E) = \sigma(\bigtriangledown_{E} x, \overset{\triangledown}{E})$
H_4: $m = s_1 \in \mathcal{I}$

$$R_8\text{: } D_{4.1.3}H_2 \rightarrow \left\{\begin{array}{l} \Lambda_s(\, \rho(s_1, E_1 = E) \cdot i, \mu(s_1, E_1)) = 1 \\ \cdots \\ \Lambda_s(\, \rho(s_k, E_k) \cdot i, \mu(s_k, E_k)) = 1 \\ \Lambda_s(\, m' \cdot i, E') = 1, \text{ avec } m' \in \mathcal{V} \end{array}\right.$$

$$R_9\text{: } D_{4.1.4}.H_2 \rightarrow \sigma(x, E) = \left\{\begin{array}{l} \sigma(\, \rho(s_1, E_1 = E) \cdot i, \mu(s_1, E_1)) \\ \cdots \\ \sigma(\, \rho(s_k, E_k) \cdot i, \mu(s_k, E_k)) \\ \sigma(\, m' \cdot i, E') \end{array}\right.$$

$$R_{10}\text{: } P_6.R_9 \rightarrow \sigma(\, m' \cdot i, E') = \left\{\begin{array}{l} \sigma(\bigtriangledown_{E'}\ \ m' \cdot i, \overset{\triangledown}{E'}) \\ \sigma(\bigtriangledown_{E_k}\ \ s_k \cdot i, \overset{\triangledown}{E_k}) \\ \cdots \\ \sigma(\bigtriangledown_{E_1}\ \ s_1 \cdot i, \overset{\triangledown}{E_1}) \\ \sigma(\bigtriangledown_{E} x, \overset{\triangledown}{E}) \end{array}\right.$$

$\underline{\underline{P_6\text{: } R_7 R_{10} \rightarrow \sigma(x,E) = \sigma(\bigtriangledown_E x, \overset{\bigtriangledown}{E})}}$

Si x est un flot :

H_1: $x = e\, f_{by}\, c$
H_2: $\Lambda_s(x,E) = 1$

R_1: $D_{4.1.4}.H_1 \rightarrow \sigma(x,E) = \sigma(e,E)$
R_2: $D_{4.1.4}.D_{3.3.2}.H_1 \rightarrow \sigma(\bigtriangledown_E x, \overset{\bigtriangledown}{E}) = \sigma(\bigtriangledown_E c, \overset{\bigtriangledown}{E})$
R_3: $D_{4.1.3} \rightarrow \Lambda_s(x,E) = 1$
R_4: $P_5 \rightarrow |\bigtriangleup_E c| = \sigma(c,E)$
R_5: $P_4 \rightarrow |\bigtriangleup_E c| = \sigma(\bigtriangleup_E c, \overset{\bigtriangledown}{E})$
R_6: $D_{4.1.4} \rightarrow \sigma(c,E) = \sigma(s,E)$
R_7: $R_4.R_5.R_6 \rightarrow \sigma(s,E) = \sigma(\bigtriangleup_E c, \overset{\bigtriangledown}{E})$

$\underline{\underline{P_6\text{: } R_2 \rightarrow \sigma(x,E) = \sigma(\bigtriangledown_x E, \overset{\bigtriangledown}{E})}}$

Preuve 7 : $\Lambda_s(x,E) = 1 \Rightarrow \Lambda_s(\bigtriangledown_E x, \overset{\bigtriangledown}{E}) = 1$

Cette preuve montre que la stabilité se conserve avec la régénération.

Si x est un flot :

H_1: $x = e\, f_{by}\, c$
H_2: $\Lambda_s(x,E) = 1$

R_1: $D_{4.1.3}.H_1.H_2 \rightarrow \Lambda_s(c,E) = 1$
R_2: $P_7.R_1 \rightarrow \Lambda_s(\bigtriangledown_E c, \overset{\bigtriangledown}{E}) = 1$
R_3: $P_2 \rightarrow \bigtriangleup_E x \in \mathcal{X}$
R_4: $R_3 \rightarrow \sigma(\bigtriangleup_E c, E) = |\bigtriangleup_E c|$
R_5: $P_5.R_1 \rightarrow |\bigtriangleup_E c| = \sigma(c,E)$
R_6: $P_6.R_1 \rightarrow \sigma(c,E) = \sigma(\bigtriangledown_E c, \overset{\bigtriangledown}{E})$
R_7: $R_6 \rightarrow \sigma(\bigtriangleup_E c, E) = \sigma(\bigtriangledown_E c, \overset{\bigtriangledown}{E})$

$\underline{\underline{P_7\text{: } R_6.R_3.R_2 \rightarrow \Lambda_s(\bigtriangledown_E x, \overset{\bigtriangledown}{E}) = 1}}$

Si x est une extraction :

H_1: $x = m \cdot i$
H_2: $\Lambda_s(x,E) = 1$

R_1: $H_1.H_2 \rightarrow m \in \mathcal{V} \cup \mathcal{I}$, and $i \in \mathcal{Z}$
H_3: $m = \left[\begin{smallmatrix} c_1 \\ \dot{c}_n \end{smallmatrix}\right] \in \mathcal{V}$
R_2: $H_1.H_2.H_3 \rightarrow \bigtriangledown_E m = m' \in \mathcal{V}$
R_3: $P_7 \rightarrow \Lambda_s(m', \overset{\bigtriangledown}{E}) = 1$
R_4: $\rightarrow \Lambda_s(\bigtriangledown_E x, \overset{\bigtriangledown}{E}) = 1$
H_4: $m = s_1 \in \mathcal{I}$
R_5: $D_{4.1.3} \rightarrow \bigtriangledown_E m = s_1$
R_6: $D_{4.1.3}.H_2 \rightarrow \begin{cases} \Lambda_s(\rho(s_1, E_1 = E) \cdot i, \mu(s_1, E_1)) = 1 \\ \dots \\ \Lambda_s(\rho(s_k, E_k) \cdot i, \mu(s_k, E_k)) = 1 \\ \Lambda_s(m' \cdot i, E') = 1, \text{ avec } m' \in \mathcal{V} \end{cases}$
R_7: $R_6 \rightarrow \Lambda_s(\bigtriangledown_{E'} m' \cdot i, \overset{\bigtriangledown}{E'}) = 1$
R_8: $R_7 \rightarrow \Lambda_s(\bigtriangledown_E m \cdot i, \overset{\bigtriangledown}{E}) = 1$

$\underline{\underline{P_7\text{: } R_4.R_8 \rightarrow \Lambda_s(\bigtriangledown_E x, \overset{\bigtriangledown}{E}) = 1}}$

Si x est une alternative :

H_1: $x = c \rightarrow a, s$
H_2: $\Lambda_s(x, E) = 1$

R_1: $H_1.H_2 \rightarrow \Lambda_s(c, E) = \Lambda_s(a, E) = \Lambda_s(s, E) = 1$

R_2: $P_5.R_1 \rightarrow \Lambda_s(\bigtriangledown_E c, \overset{\bigtriangledown}{E}) = \begin{cases} \Lambda_s(\bigtriangledown_E a, \overset{\bigtriangledown}{E}) \\ \Lambda_s(\bigtriangledown_E s, \overset{\bigtriangledown}{E}) = 1 \end{cases}$

R_3: $D_{4.1.4}.H_1.H_2 \rightarrow \sigma(a, E) = \sigma(s, E)$

R_4: $P_6.R_3 \rightarrow \sigma(\bigtriangledown_E a, \overset{\bigtriangledown}{E}) = \sigma(\bigtriangledown_E s, \overset{\bigtriangledown}{E})$

P_7: $R_2.R_4 \rightarrow \Lambda_s(\bigtriangledown_E x, \overset{\bigtriangledown}{E}) = 1$

Les preuves concernant les autres acteurs sont construites selon le meme modèle.

Chapitre 7

Vers le parallélisme

Le langage des λ-matrices permet une description modulaire des applications. C'est un langage primitif, dans le sens où il ne contient pas d'expressions superflues. Sa définition sémantique est obtenue à l'aide d'une machine fonctionnelle abstraite.

Les déterminismes en temps et en ressources des applications sont vérifiés à l'aide de trois fonctions de critère, elles mêmes décrites sous forme fonctionnelle. Les propriétés sont vérifiées formellement.

Le langage comporte les primitives qui permettent de modulariser les applications. Ces aspects modulaires compliquent la définition de la machine abstraite. Il sont supprimés par un compilateur formel après que la λ-matrice a subi un contrôle strict des types.

Le résultat de la compilation est une λ-matrice linéaire d'où les dépendances fonctionnelles peuvent être aisément extraites. En effet, le code cible ne possède que quatre types de constructions, la définition, l'application, l'alternative et le flot.

Le fait que le langage des λ-matrices ne soit pas complètement défini, quant aux données manipulées et à leurs opérateurs, permet de les décrire comme des «ordonnanceurs de tâches». Les tâches élémentaires concernent les données et sont définies dans des algèbres.

L'utilisation intensive de définitions fonctionnelles pures permet de donner au formalisme des propriétés algébriques. L'objet de cette thèse n'est pas l'étude approfondie des propriétés mathématiques. Cependant, nous montrons comment tirer parti de cette propriété avec des vérifications formelles.

Dans la dernière partie de ce rapport de thèse, nous verrons d'une manière non formalisée comment sont construits les outils relatifs aux λ-matrices. Notamment, le langage concret λ-FLOW et le langage graphique λ-GRAPH seront décrits. Ils permettent une programmation modulaire en étendant les λ-matrices avec des capacités d'abstractions. Un programme permettant d'ordonnancer les tâches élémentaires issues de la compilation des λ-matrices sera décrit. Le code résultant pourra être exécuté sur le simulateur de l'architecture parallèle statique.

Troisième partie

Réalisations pratiques : les λ-outils

Chapitre 1

Introduction

Cette troisième partie de la thèse traite des outils logiciels réalisés [27]. Ces outils reposent sur l'étude de l'existant de la première partie, et sur l'étude théorique de la seconde partie.

Le premier chapitre décrit l'interface objet de l'interprète SCHEME utilisé (§ 2). Cette interface définit le langage STKLOS. Nous ne décrivons que les aspects qui seront utilisés plus loin pour spécifier les outils développés.

Le premier outil est l'interface de programmation graphique λ-GRAPH (§ 3). Il propose un langage graphique permettant une programmation modulaire et hiérarchique des applications. Le langage est indépendant des algèbres[1] de l'utilisateur qui sont lues dynamiquement. Dans ce chapitre, l'interface est spécifiée à l'aide d'une analyse objet et réalisée à l'aide de STKLOS, une couche objet de SCHEME. Un traducteur permet de traduire un programme λ-GRAPH en λ-FLOW.

Le second outil est λ-FLOW (§ 4). C'est un langage concret construit sur les λ-matrices. La plupart des expressions sont celles des λ-matrices. Cependant, il définit deux nouvelles expressions qui sont l'abstraction et l'exportation. Elles améliorent les possibilités de programmation modulaire des applications, offertes de manière primitive par les λ-matrices.

Le compilateur λ-FLOW est un compilateur optimisé écrit en C. Il permet la prise en compte dynamique des algèbres, ce qui signifie que les données manipulées par les programmes ne sont pas connues du compilateur. De plus, le code produit par λ-FLOW est lui aussi défini de manière dynamique dans le compilateur. Actuellement, quatre cibles sont définies, produisant du C, du SCHEME, de l'assembleur INTEL-386 et du SPA, code pouvant être manipulé par le paralléliseur. Une cible est définie au moyen d'un fichier contenant une vingtaine de définitions.

Le troisième outil est le simulateur graphique de l'architecture parallèle statique (§ 5). Il est analysé avec un formalisme objet, puis la partie non-graphique est spécifiée à l'aide STKLOS. Cette spécification permet de construire directement le simulateur de l'architecture. L'interface graphique du simulateur permet de visualiser l'évolution des programmes dans chaque contrôleur en mode pas à pas ou en mode animation. Il permet également de modifier dynamiquement le contenu des mémoires et registres et de positionner des points d'arrêt.

Enfin, le dernière outil présenté est le paralléliseur de code pour l'architecture parallèle statique (§ 6). Cet outil est décrit à l'aide d'un algorithme en SCHEME. Il permet de répartir les tâches de la cible `SPA` parmi les contrôleurs de l'architecture parallèle. Nous montrerons à l'aide d'un exemple comment le paralléliseur fonctionne. Les rendements obtenus sont présentés sous la forme d'un graphique. Dans certains cas, le gain de performances atteint près de 60%.

Dans la quatrième partie, ces outils seront utilisés dans le cadre d'une application concrète issue de l'industrie.

1. Rappelons que nous appelons une algèbre un type de donnée et les opérateurs qui s'y rapportent.

Chapitre 2

Introduction (succincte) à stklos

Dans ce chapitre, nous présentons brièvement le formalisme objet de STK nommé STKLOS. Il est utilisé pour spécifier à la fois l'interface graphique λ-GRAPH (§ 3) et le simulateur de l'architecture parallèle (§ 5). Sa puissance expressive permet de l'utiliser comme langage de spécification d'une application. L'avantage est alors que la spécification devient exécutable.

Ici, nous introduisons succinctement cette interface objet, en ne décrivant que les caractéristiques utilisées. Beaucoup d'aspects de STKLOS ne sont pas abordés.

STKLOS est la couche objet de STK, de ERIC GALLESIO [37]. Sa réalisation est issue de la version `1.3` du logiciel `tiny clos` de GREGOR KICKZALES [53]. Il a été étendu pour être compatible avec CLOS, *The Common Lisp Object System* [49].

2.1 Définition des termes

Dans un formalisme objet, un objet appartient à une classe. Une classe définit la structure de l'objet, c'est à dire ses composantes. Une classe peut hériter d'une autre classe alors appelée super-classe, ou classe-parent. Avec un langage autorisant l'héritage multiple, une classe peut hériter directement de plusieurs super-classes. C'est le cas avec STKLOS, mais pas avec SMALLTALK. En général, il existe une classe racine, dont toutes classes héritent par défaut.

Une classe définit la structure de l'objet. La structure d'un objet est l'ensemble de ses attributs, qui peuvent être considérés comme des variables appartenant à chaque objet créé. Lorsqu'un objet est créé, on dit que c'est une instance de la classe à laquelle il appartient, ou que l'on a instancié la classe. Les attributs réels d'un objet sont l'union de tous les attributs définis par la classe de l'objet et toutes les classes dont il hérite.

Associées aux classes, les méthodes peuvent être appliquées aux objets de la classe. Une méthode est une sorte de fonction. Comme elles sont définies par rapport aux classes, plusieurs méthodes de même nom peuvent être définies dans la même application. Par exemple, nous pourrons définir la méthode `afficher` pour toutes les classes d'un programme. En général, une méthode possède un argument implicite appelé `self`, et qui représente l'objet lui-même. Pour accéder aux attributs d'un objet, les langages définissent en général des méthodes dont le nom est le nom de l'attribut.

Si une classe hérite d'une autre classe, et si les deux classes définissent une méthode de même nom, nous dirons que la première classe surcharge la méthode. Lorsque la méthode est invoquée sur un objet de la première classe, c'est effectivement celle définie par la première classe qui est considérée. En général, le langage permet d'accéder aux méthodes surchargées. Pour cela, STKLOS définit la fonction `next-method`.

2.2 Définition des classes

Dans STKLOS, Une classe est définie avec la fonction `define-classe`. Sa syntaxe est la suivante :

```
(define-class classe (superClasse-1 superClasse-2 ...)
   (attribut-1 attribut-2 ...))
```

La `classe` définit les `attributs` et hérite des `superClasses`. Les `attributs` peuvent être soit simplement un identificateur, soit des listes donnant les propriétés de l'attribut, comme nous allons le voir plus loin. Par exemple, considérons la classe des nombres complexes :

```
(define-class <complex> (<number>)
   (real imag))
```

Par convention, les noms des classes sont écrits `nom`. Cette classe `<complexe>` contient deux attributs appelés `real` et `imag` qui prendront respectivement la valeur réelle et la valeur imaginaire du nombre complexe. Cette classe hérite de la classe prédéfinie `<number>`.

2.3 Hiérarchie des classes et héritage des attributs

La définition d'une classe permet de spécifier les classes dont elle hérite. STKLOS permet l'héritage multiple. Considérons les définitions suivantes :

```
(define-class A () (a))
(define-class B () (b))
(define-class C () (c))
(define-class D (A B) (d a))
(define-class E (A C) (e c))
(define-class F (D E) (f))
```

Les classes `A`, `B`, `C` n'ont pas de super-classes. D'une manière implicite, toute classe hérite au moins de la classe `<object>`, même si cela n'est pas indiqué. Les classes `D`, `E`, `F` utilisent l'héritage multiple : chacune d'elle hérite des deux classes précédentes. Ces définitions créent une hiérarchie de classes, comme nous pouvons le voir dans la figure 2.1.

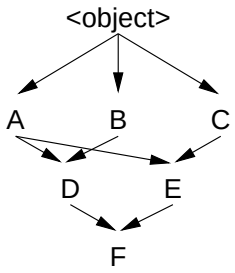


FIG. 2.1 - *Hiérarchie de classes. La classe* `object` *est la classe racine.*

L'ensemble des attributs d'une classe particulière est obtenu en effectuant l'union des attributs des super-classes. Par exemple, chaque instance de la classe `D` possède les trois attributs `a`, `b` et `d`.

2.4 Création d'instances et accès aux attributs

La fonction `make` permet la création d'une instance d'une classe définie. Le seul argument indispensable est le nom de la classe. Elle peut être invoquée avec d'autres arguments généralement utilisés pour initialiser les objets. Par exemple, nous avons :

```
(define c (make <complex>))
```

qui créé un objet de classe `<complex>` associé à `c` dans l'environnement global de STK.

Pour connaître la classe d'un objet, STKLOS définit la fonction `classOf` qui retourne la classe de l'objet. Par exemple :

```
STk> (classOf c)
<complex>
```

Pour accéder, respectivement en lecture et en écriture, aux attributs d'un objet, STKLOS a défini les fonctions standards `slot-ref` et `slot-set!`. Par exemple, nous aurons le dialogue suivant avec STK :

```
STk> (slot-set! c 'real 10)
#unspecified

STk> (slot-set! c 'imag 3)
#unspecified

STk> (slot-ref c 'real)
10

STk> (slot-ref c 'imag)
3
```

2.5 Description des attributs

Un attribut peut être défini avec plus d'options que son simple nom. Chaque option est indiquée à l'aide d'un mot clef, dont la liste est donnée ci-dessous :

- `:initform` permet de spécifier la valeur initiale d'un attribut. Cette valeur initiale est obtenue par l'évaluation de l'expression donnée après `:initform`.
- `:init-keyword` permet de spécifier le mot clef qui permettra d'initialiser un attribut. Ce mot clef est utilisé pendant la création de l'instance de l'objet, comme argument optionnel de la fonction `make`. Indiquer la valeur initiale d'un attribut de cette manière annule l'effet de l'option `:initform`.
- `:accessor` permet de définir une fonction autorisant l'accès en lecture/écriture aux attributs d'un objet. Ce nouveau moyen d'accès supplée aux fonctions `slot-ref` et `slot-set!`.

Pour illustrer l'usage des options dans la définition des attributs, redéfinissons la classe `<complex>` :

```
(define-class <complex> (<number>)
   ((real :initform 0 :accessor real-part :init-keyword :r)
    (imag :initform 0 :accessor imag-part :init-keyword :i)))
```

Avec cette nouvelle définition, les attributs `real` et `imag` sont initialisés à `0`. La valeur initiale de ces attributs peut aussi être spécifiée comme argument de la fonction `make`, à l'aide des mots clefs `:r` et `:i`. Les fonctions `real-part` et `imag-part` permettent un accès direct aux attributs, en plus des moyens donnés par `slot-ref` et `slot-set!`. Par exemple, nous aurions le dialogue :

```
STk> (define c1 (make <complex> :r 1 :i 2))
#unspecified

STk> (real-part c1)
1

STk> (set! (real-part c1) 12)
#unspecified

STk> (real-part c1)
12
```

```
STk> (define c2 (make <complex> :r 2))
#unspecified

STk> (real-part c2)
2

STk> (imag-part c2)
0
```

2.6 Définition des méthodes

STKLOS n'utilise pas le mécanisme des messages pour définir les méthodes des classes, comme c'est le cas dans la plupart des langages objets. Au lieu de cela, STKLOS utilise le concept de fonctions génériques. Lorsqu'une méthode est définie avec la fonction `define-method`, le système maintient une liste des méthodes de même nom qui diffèrent par le nombre et le type de leurs paramètres.

Lorsqu'une méthode est appliquée à un objet, le système recherche dans la liste celle qui est la plus spécifique. Nous dirons qu'une méthode `M` est plus spécifique qu'une méthode `M'` si les classes de ses paramètres sont plus spécifiques que celles de `M'`.

Par exemple, considérons les définitions des méthodes suivantes :

```
(define-method M ((a <integer>) b) 'integer)
(define-method M ((a <real>) b) 'real)
(define-method M (a b) 'object)
```

Ces écritures créent dans le système une liste des méthodes de nom `M` à trois éléments. Les méthodes sont différenciées par le type des paramètres. La première définition indique que le premier argument `a` doit être de type `<integer>`, et n'indique pas le type du second, `b`. La deuxième définition indique que le premier argument doit être de type `<real>`, sans indiquer le type du second. Enfin, la troisième méthode n'indique pas le type des arguments. En fait, lorsque le type d'un argument n'est pas indiqué, cela revient à lui donner le type `<object>`. La première définition est donc équivalente à :

```
(define-method M ((a <integer>) (b <object>)) 'integer)
```

Considérons maintenant le dialogue suivant avec l'interprète :

```
STk> (M 2 3)
integer

STk> (M 2 #t)
integer

STk> (M 1.2 'a)
real

STk> (M #3 'a)
real

STk> (M #t #f)
object

STk> (M 1 2 3)
error (since no method exists for 3 parameters)
```

Les définitions précédentes diffèrent seulement par le type du premier paramètre. Il est possible de les différencier par tous les paramètres. Dans ce cas, la liste des paramètres est parcourue de gauche à droite pour déterminer quelle est la méthode à appliquer. Par exemple, considérons :

```
(define-method M ((a <integer>) (b <number>)) 'integer-number)
(define-method M ((a <integer>) (b <real>))   'integer-real)
(define-method M (a (b <number>)) 'object-number)
```

Dans ce cas, nous aurons le dialogue :

```
STk> (M 1 2)
```

```
integer-integer

STk> (M 1 1.0)
integer-real

STk> (M 1 #t)
integer

STk> (M 'a 1)
'object-number
```

2.7 La «prochaine» méthode

Lorsqu'une méthode est définie, elle est toujours ajoutée à la liste des méthodes de même nom, qui peut être créée à cette occasion. La méthode la plus spécifique est toujours choisie dans cette liste, en fonction des arguments. Une méthode particulière peut être définie pour invoquer la méthode immédiatement avant elle, dans la liste, c'est à dire la méthode immédiatement moins spécifique. Cette invocation est réalisée avec la fonction standard **next-method**. Considérons :

```
(define-method Test ((a <integer>)) (cons 'integer (next-method)))
(define-method Test ((a <number>))  (cons 'number  (next-method)))
(define-method Test (a)     (list 'top))
```

Si l'on considère que la classe **<integer>** hérite directement de la classe **<number>**, il vient :

```
STk> (Test 1)
(integer number top)

STk> (Test 1.0)
(number top)

STk> (Test #t)
(top)
```

2.8 Conclusion

Cette très brève présentation de STKLOS sera tout de même suffisante pour spécifier les applications modélisées avec un formalisme à objets de la thèse.

STKLOS est beaucoup plus riche que cette présentation ne le laisse paraître. Notamment, il définit une hiérarchie de classes qui abstraient l'interface graphique TK [64]. Il devient alors très aisé de programmer une interface graphique avec STK, qui combine à la fois la puissance expressive du langage SCHEME et la souplesse de TK.

Chapitre 3

Langage graphique : λ-graph

Ce chapitre décrit l'interface graphique λ-GRAPH et son langage [24]. Ce langage graphique est un sous-ensemble du langage λ-FLOW. Il permet une programmation hiérarchique et la définition dynamique des algèbres. Son mode de programmation est à rapprocher de la programmation des graphes dataflow, et plus précisément des réseaux d'opérateurs (§ I-2).

Dans un premier temps, λ-GRAPH sera spécifiée, et plus particulièrement les éléments graphiques de son langage (§ 3.1). Nous conduirons une analyse orientée objet de λ-GRAPH, qui aboutira au modèle objet et au graphe d'héritage (§ 3.2). Cette analyse sera conduite de manière à être indépendante de toute interface graphique. Puis nous décrirons le modèle de programmation (§ 3.3) et enfin, nous présenterons une copie d'écran représentant l'interface graphique (§ 3.4).

3.1 Spécification de λ-graph

Dans cette section, l'application λ-GRAPH est spécifiée. Les éléments graphiques du langage sont décrits, puis la maquette de l'interface est donnée.

3.1.1 Langage graphique

λ-GRAPH permet une représentation graphique et hiérarchique d'une partie des acteurs de λ-FLOW. Voici les acteurs représentés :

Les éléments graphiques proviennent de la grammaire du langage λ-FLOW. Cette grammaire graphique peut être apparentée aux réseaux d'opérateurs [11] (§ I-2).

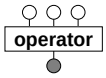
On remarque que l'extraction de λ-FLOW n'a pas de représentation graphique explicite. En fait elle apparaît comme un lien sur une sortie dans un module.

3.1.2 Maquette de l'interface

Un module possède donc deux représentations graphiques : une forme compacte, lorsque le module est utilisé comme un composant, et une forme dilatée qui permet d'éditer son contenu. Cette forme dilatée est l'interface de λ-GRAPH, dont la maquette est montrée dans la figure 3.1.

La fenêtre de l'interface est organisée en trois parties :

- un menu qui donne accès à un certain nombre de commandes usuelles, comme la lecture, la sauvegarde et l'impression des programmes, les opérations d'édition couper/coller/copier ;
- une zone de commande qui permet de créer les objets graphiques ;
- une zone de dessin qui permet de dessiner le contenu du module.

	une **alternative** possède trois entrées : une condition, une clause-alors et une clause-sinon. Elle a une seule sortie.
	une **application** possède plusieurs entrées, dépendantes de l'opérateur représenté, et une seule sortie.
	un **atome** n'a qu'une sortie qui fournit une constante. Son contenu dépend de l'algèbre à laquelle il rattaché.
	un **flot** possède deux entrées, l'état et le contrat, et une seule sortie.
	un **port d'entrée** correspond à un paramètre d'une abstraction de λ-FLOW.
	un **port de sortie** correspond à une exportation dans un vecteur de λ-FLOW.
	un **module** est une abstraction nommée d'un vecteur de λ-FLOW. Son apparence est identique à celle d'une application, mais pouvant posséder plusieurs sorties.

TAB. 3.1 - *Expressions graphiques du langage de* λ-GRAPH.

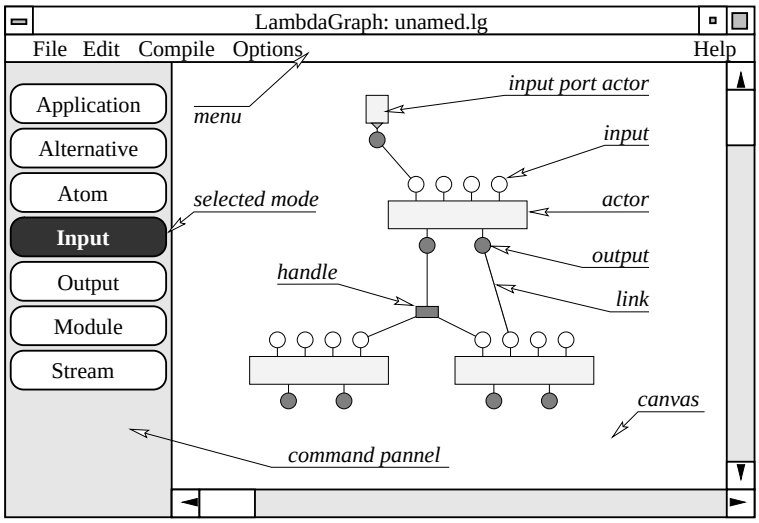


FIG. 3.1 - *Maquette de l'interface de* λ-GRAPH.

Les entrées / sorties des acteurs dans la zone d'édition sont colorées de manière différente, pour les différencier. Une entrée ne peut être liée qu'à une seule sortie, et une sortie peut être liée à plusieurs entrées. Une poignée n'est liée en amont qu'à une seule sortie ou une seule poignée, et en aval à plusieurs entrées ou poignées.

Tout acteur de la zone de dessin peut être déplacé, les liens suivant automatiquement les déplacements. Si un objet est détruit, les liens qui lui sont attachés sont aussi détruits.

La zone de dessin permet les opérations de sélection et de suppression. Les opérations de copie et de collage sont effectuées par rapport au presse-papier.

Un programme peut être sauvé dans un fichier ou lu d'un fichier. De plus, un module peut être édité dans une fenêtre séparée : dans ce cas, toute modification de son interface (entrées/sorties) est répercutée sur tous les programmes utilisant ce module.

Les atomes et les applications sont configurés par une boîte de dialogue en fonction des algèbres disponibles. Les algèbres sont celles définies pour le compilateur λ-FLOW (§ 4.2).

3.2 Analyse objet

3.2.1 Modèle objet de λ-graph

Cette section introduit le modèle objet de l'interface. Chaque objet possède des attributs et des relations avec les autres objets de l'application. La méthode utilisée pour cette modélisation s'inspire de celle utilisée pour modéliser les bases de données [62]. Le graphe du modèle peut être examiné dans la figure 3.2.

Un acteur possède zéro, un ou plusieurs (ZUP) entrées et ZUP sorties. Une entrée est connectée à zéro ou un (ZU) lien, et une sortie est connectée à ZUP liens. Donc un lien est connecté à ZU entrée et ZUP sorties. Une poignée est toujours connectée à un seul lien en amont, et un lien à ZUP poignées. Remarquons qu'un lien possède en amont soit une sortie soit une poignée, et en aval soit une entrée soit une poignée.

A partir de ce modèle de l'application, le graphe d'héritage peut être directement déduit.

3.2.2 Graphe d'héritage

Cette section décrit les classes des objets de λ-GRAPH, déduites du modèle objet. Alors que le modèle objet traite des relations entre les instances des objets, le graphe d'héritage décrit

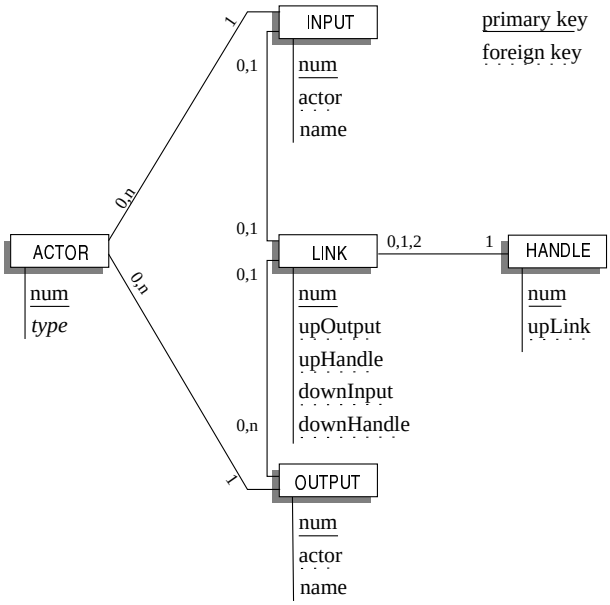


FIG. 3.2 - *Modèle objet de λ-GRAPH qui montre les relations entre les instances des objets de l'application.*

la construction des objets par spécialisation d'objets plus généraux, comme nous pouvons le voir dans la figure 3.3.

La classe `<TK>` représente la boîte à outils graphique comme une interface. Cette boîte à outils permet la création d'objets graphiques dans une zone de dessin, comme les lignes, les carrés ou les cercles. Chaque objet créé est identifié par une clef unique. Les objets graphiques peuvent être manipulés à l'aide de leur clef.

La boîte à outils graphiques est simulée par la classe `<TK>`, ce qui rend la modélisation indépendante de toute réalisation particulière. De manière à simuler aussi la gestion des clefs, cette classe définit la variable de classe[1] `tkKey` qui est le nombre d'objets graphiques crées. La classe `TK` définit aussi la variable d'instance `tkNum` qui représente la clef des objets graphiques. Une variable d'instance est créée pour chaque instance d'objet de la classe, ou pour chaque instance d'objet qui hérite de la classe.

La classe `<DB>` est une base de données qui contient tous les objets créés dans l'application. Elle permet d'effectuer des sélections sur ces objets. Pour cela, elle définit la variable de classe `objects` qui est un ensemble.

Nous avons donc trois groupes d'objet instanciables : les acteurs, les reliables et les liens. La classe `<Actor>` est toujours spécialisée par une classe instanciable.

Nous pouvons maintenant entamer l'étude de la réalisation de l'interface λ-GRAPH, en commençant par la définition des classes.

3.3 Modèle de programmation

Cette section donne le modèle de programmation de λ-GRAPH. Cette modélisation utilise le langage STKLOS. Dans la première partie, les classes de l'application sont décrites, puis dans les parties suivantes, les méthodes des objets sont définies.

1. Une variable de classe peut être vue comme une variable globale accessible seulement aux objets de la classe.

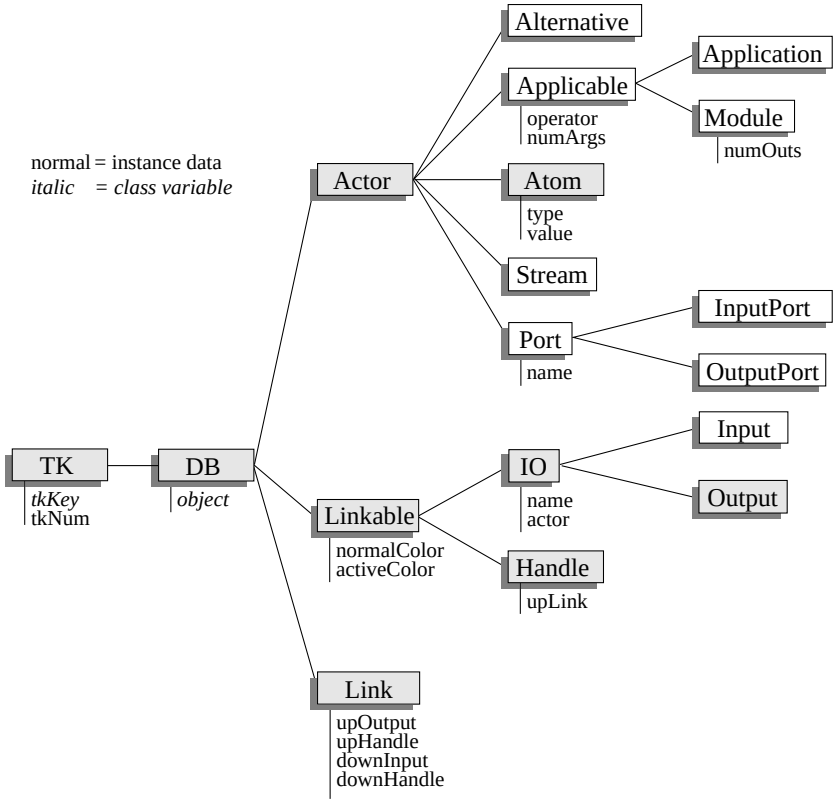


FIG. 3.3 - *Graphe d'héritage des objets de λ-GRAPH. Les objets grisés sont décrits dans l'analyse.*

3.3.1 Définition des classes

Classes primitives

Toutes les classes de λ-GRAPH héritent des classes primitives. La première classe primitive est la classe `<TK>` qui simule l'interface avec la boîte à outils graphique. Cette classe définit une variable d'instance `tkNum` pour chaque objet graphique créé : cette variable est la clef unique de l'objet. Nous avons :

```
(define-class <TK> ()
  ((tkNum :initform (new-tkKey) :accessor tkNum)))
```

où la fonction `new-tkKey` est un générateur de symboles, définie comme suit :

```
(define *tkKey* 0)

(define (new-tkKey)
  (let ((tkKey *tkKey*))
    (set! *tkKey* (+ 1 *tkKey))
    tkKey))
```

La gestion des clefs est normalement assurée par la boîte à outils. La boîte à outils fournit aussi les méthodes de création graphique `tkDraw` pour chaque objet, et d'effacement `tkClean`. Ces deux méthodes permettent de définir la méthode `tkMove` qui déplace un objet graphique.

Comme STKLOS ne permet pas de définir de variables de classe, nous remarquons que la variable de classe `tkKey` du modèle est transformée en variable globale que nous notons `*tkKey*`.

La réalisation de la classe `<DB>` est plus délicate en STKLOS, du fait de l'impossibilité de

définir des variables de classes. Là encore, nous utiliserons les variables globales :

```
(define *objects* '())

(define-class <DB> (<TK>) '())
```

La variable globale `*object*` contient tous les objets de l'application. La classe `<DB>` sera utilisée pour définir les méthodes correspondantes. Elles seront basées sur les opérateurs ensemblistes introduits dans la première partie (§ I-4.7)

Acteur

Un acteur est un objet graphique représentant une expression du langage. La classe `<Actor>` est la classe primitive de tous les acteurs. Elle ne définit aucune variable d'instance, et elle est seulement utilisée pour partager des méthodes entre tous les acteurs.

```
(define-class <Actor> (<Graph>) '())
```

La classe `<Atom>` hérite de la classe `<Actor>`. Elle définit les variables d'instance `type` et `value` dont le contenu dépend des algèbres utilisées dans l'application. Par exemple, la valeur d'un entier peut être la chaîne `123`.

```
(define-class <Atom> (<Actor>)
  ((type  :initform "" :accessor type)
   (value :initform "" :accessor value)))
```

Les classes des autres acteurs reposent sur le même modèle que la classe `<Atom>`.

Objets reliables

Les objets de classe `<Linkable>` sont les entrées / sorties des acteurs, et les poignées. La classe primitive `<Linkable>` définit deux variables d'instances `normalColor` et `activeColor`, utilisées respectivement lorsque la souris est ou n'est pas sur l'objet.

```
(define-class <Linkable> (<Graph>)
  ((normalColor :initform "blue" :accessor normalColor)
   (activeColor :initform "red"  :accessor activeColor)))
```

La classe `<IO>` est la classe primitive des entrées/sorties. La classe déclare la variable d'instance `actor` car une entrée ou une sortie ne peut appartenir qu'à un seul acteur. De plus, ces objets sont associés à un nom.

```
(define-classe <IO> (<Linkable>)
  ((name  :initform "" :accessor name)
   (actor              :accessor actor)))
```

La classe `<Input>` est la classe des entrées. Elle hérite directement de la classe `<IO>`. La classe `<Output>` est bâtie selon le même modèle.

```
(define-class <Input> (<IO>) '())
```

Une poignée de classe `<Handle>` est attachée à un seul lien en amont, donc la classe déclare la variable d'instance `upLink`.

```
(define-class <Handle> (<Linkable>)
  ((upLink :accessor upLink)))
```

Un lien appartient à la classe `<Link>`. Il est rattaché soit à une sortie amont `upOutput` soit à une poignée amont `upHandle`, et il est aussi rattaché soit à une entrée avale `downInput` soit à une poignée avale `downHandle`.

```
(define-class <Link> (<Graph>)
  ((upOutput   :accessor upOutput   :init-keyword :upOutput)
   (upHandle   :accessor upHandle   :init-keyword :upHandle)
   (downInput  :accessor downInput  :init-keyword :downInput)
   (downHandle :accessor downHandle :init-keyword :downHandle)))
```

3.3.2 Méthodes

Dans cette section, nous présentons les méthodes associées aux classes définies ci-dessus. Ces méthodes permettent de créer, déplacer, supprimer, relier les différents objets graphiques de l'interface.

Gestion de la base de donnée

La base de données permet de maintenir l'ensemble de tous les objets créés dans l'application. Cette base de données est gérée pour effectuer des sélections sur ces objets. Le fait que STKLOS n'autorise pas la définition de variables de classe complique quelque peu la réalisation de cette base de données en obligeant l'utilisation de variables globales.

La réalisation de la base de données s'appuie sur les opérateurs ensemblistes introduits dans la première partie (§ I-4.7).

L'initialisation de la base de données est faite une fois pour toute, à l'initialisation de l'application, avec l'instruction `(define *objects* '())`.

Les opérations d'ajout, de sélection et de suppression utilisent directement les opérateurs ensemblistes correspondants :

```
(define-method add ((self <DB>) object)
  (addset object *objects*))
```

et :

```
(define-method remove ((self <DB>) object)
  (subset object *objects*))
```

puis :

```
(define-method select ((self <DB>) proc)
  (select proc *objects*))
```

L'initialisation de chaque objet de la classe `<DB>` ajoute l'objet instancié dans la base de données, avec :

```
(define-method initialize ((self <DB>) arguments)
  (next-method)
  (add self))
```

3.3.3 Initialisation des objets

La méthode suivante est utilisée pour créer des objets de classe `<Actor>`. Elle a comme paramètre l'endroit où l'acteur doit être dessiné à l'écran, spécifié avec le mot clef `:location`. L'acteur est ensuite dessiné avec la méthode `tkCreate`. L'acteur est enfin retourné :

```
(define-method initialize ((self <Actor>) arguments)
  ; invokes the <DB> and <TK> initialize methods
  (next-method)

  ; creates the graphical object
  (tkCreate self (get-keyword :location arguments))
  self)
```

Les liens sont initialisés soit avec une poignée ou une sortie en amont, soit avec une poignée ou une entrée en aval. Ces arguments sont introduits avec les mots clefs `upHandle` ou `upOutput`, et `downHandle` ou `downHandle`. Les liens sont dessinés entre leurs deux extrémités, donc l'invocation à `tkCreate` n'a pas de paramètres. Il vient :

```
(define-method initialize ((self <Link>) arguments)
  (next-method)
  (tkCreate self)
  self)
```

La création d'une poignée entraîne la création d'un lien supplémentaire, car une poignée est toujours créée en coupant un lien en deux :

```
(define-method initialize ((self <Handle>) arguments)
  (next-method)
  (let* (; gets the options
         (upHandle (get-keyword :upHandle arguments))
         (upOutput (get-keyword :upOutput arguments))

        ; crerates the new link
         (upLink (make <Link> :upHandle (downLink upHandle)
                              :upOutput (downLink upOutput)
                              :downHandle self
                              :downInput #f)))

    ; changes the conexion
    (set! (downLink upHandle) self)
    (set! (dowmLink upOutput) #f)
    self))
```

Méthodes de sélection et de modification

Les méthodes suivantes sont des sélecteurs des objets instanciés. Elles permettent de sélectionner des objets à l'aide d'un critère. Certaines de ces méthodes consultent simplement les variables d'instances, d'autres construisent un résultat sous la forme d'un ensemble en effectuant une sélection sur tous les objets d'une classe. Cette sélection est effectuée avec la méthode standard de classe `select:` qui a comme paramètre une fonction qui retourne un booléen. Cette fonction est alors appliquée à tous les éléments de la classe. Lorsqu'elle retourne vrai, l'objet est sélectionné.

La méthode de sélection pour les objets `Actor` retourne l'ensemble de ses entrées (`inputs`) ou l'ensemble de ses sorties (`outputs`).

```
(define-method inputs ((self <Actor>))
  (select self (lambda (elem)
                 (and (eq? (classOf elem) <Input>)
                      (eq? (actor   elem) self)))))


(define-method outputs ((self <Actor>))
  (select self (lambda (elem)
                 (and (eq? (classOf elem) <Output>)
                      (eq? (actor   elem) self)))))
```

Le sélecteur `links` de la classe `<IO>` des entrées/sorties retourne l'ensemble des liens qui y sont attachés :

```
(define-method links ((self <IO>))
  (if (eq? (classOf self) <Input>)
    (select self (lambda (elem)
                   (and (eq? (classOf elem) <Link>)
                        (eq? (downInput elem) self))))
    (select self (lambda (elem)
                   (and (eq? (classOf elem) <Link>)
                        (eq? (upOutput elem) self))))))
```

Si `self` est une entrée, nous recherchons tous les liens dont l'entrée en aval vaut `self`, et si `self` est une sortie, nous recherchons tous les liens dont la sortie en amont vaut `self`.

Le sélecteur `downLinks` des instances de `Handle` retourne l'ensemble des liens avals de cette poignée :

```
(define-method downLinks ((self <Handle>))
  (select self (lambda (elem)
                 (and (eq? (classOf elem) <Link>)
                      (eq? (upHandle elem) self)))))
```

La prochaine méthode permet de savoir si un lien est valide ou pas. Un lien est invalide lorsqu'il n'est plus dans la base de données des objets. Cette méthode est utilisée lors de la

destruction d'une poignée qui peut engendrer un processus récursif, si cette méthode n'est pas utilisée :

```
(define-method isValid? ((self <Link>))
  (eq? 1 (length (select self (lambda (elem)
                                (and (eq? (classOf elem) <Link>)
                                     (eq? elem self)))))))
```

Méthodes de création graphique

Les méthodes de création graphique dessinent effectivement les objets sur l'écran, par des invocations à la boîte à outils graphique. Ces méthodes ont comme argument la position graphique de classe `<Point>` où l'objet doit être dessiné.

Au moment de sa création graphique, chaque acteur crée ses entrées et ses sorties. Prenons le cas d'un atome :

```
(define-method tkCreate ((self <Atom>) (location <Point>))
  (tkDraw self)
  (make <Output> :location (make <Point>
                                 :x (+ (x location) 50)
                                 :y (+ (y location) 100))
                 :actor self))
```

La création graphique des autres acteurs est basée sur le même modèle.

Méthode de déplacement

Les méthodes suivantes permettent de déplacer physiquement un acteur dans l'interface λ-GRAPH, vers une nouvelle position `loc`. Avant tout déplacement de l'acteur, la méthode déplace préalablement ses entrées/sorties. La boîte à outils est alors invoquée pour effectuer le déplacement.

```
(define-method move ((self <Actor>) (location <Point>))
  (for-each (lambda (input)
              (move input location))
            (inputs self))
  (for-each (lambda (output)
              (move output location))
            (outputs self))
  (tkMove self location))
```

Cette méthode utilise les sélecteurs `inputs` et `outputs` préalablement définis.

Lorsqu'un objet `<IO>` est déplacé, tous les liens qui lui sont attachés sont préalablement effacés, l'objet est déplacé, et les liens sont redessinés :

```
(define-method move ((self <IO>) (location <Point>))
  (let ((links (links self)))
    (for-each (lambda (link)
                (tkClean link)) links)
    (tkMove self location)
    (for-each (lambda (link)
                (tkDraw link)) links)))
```

Lorsqu'une poignée est déplacée, son lien amont `upLink` est déplacé préalablement, puis tous ses liens avals `downLinks` :

```
(define-method move ((self <Handle>) (location <Point>))
  (let ((downLinks (downLinks self)))
    (for-each (lambda (link)
                (tkClean link)) downLinks)
    (tkClean (upLink self))
    (tkMove self location)
    (for-each (lambda (link)
                (tkDraw link)) downlinks)
    (tkDraw (upLink self))))
```

Méthode de liaison

Les méthodes décrites ici permettent de lier les entrées aux sorties, ou les entrées aux poignées. Ces méthodes sont intimement liées à la prise en charge de la souris dans la boîte à outils. Elles utilisent la variable de classe `startLink` de la classe `<Graph>`, réalisée sous la forme d'une variable globale.

Les conditions pour qu'un lien puisse être créé sont les suivantes : une entrée ne peut être liée qu'à une seule sortie ou une seule poignée, une sortie ou une poignée peuvent être liées à plusieurs entrées ou poignées.

La méthode `pair` d'un objet `<Linkable>` détermine si le paramètre `pair` peut être relié à `self` :

```
(define-method pair? ((self <Linkable>) (pair <Linkable>))
  (if (eq? (classOf self) <Input>)
    (or (eq? (classOf pair) <Output>)
        (eq? (classOf pair) <Handle>))
    (eq? (classOf pair) <Input>)))
```

La méthode `attach` attend en second paramètre une entrée. Donc si le premier paramètre est une entrée, elle invoque la méthode `attach` en inversant l'ordre des arguments, de manière à ce que l'entrée soit en seconde position :

```
(define-method attach ((self <Input>) (pair <Linkable>))
  (if (pair? self pair)
    (attach pair self)))
```

Lorsqu'une sortie est reliée à une entrée, et que cette entrée n'est pas reliée, un lien est créé :

```
(define-method attach ((self <Output>) (in <Input>))
  (if (pair? self pair)
    (if (zero? (length (links in)))
      (make <Link> :upHandle   self
                   :upOutput   #f
                   :downHandle #f
                   :downInput  in))))
```

De même, lorsqu'une poignée est reliée à une entrée et que cette entrée n'est pas déjà reliée, un lien est créé :

```
(define-method attach ((self <Handle>) (in <Input>))
  (if (pair? self pair)
    (if (zero? (length (links in)))
      (make <Link> :upHandle   #f
                   :upOutput   self
                   :downHandle #f
                   :downInput  in))))
```

Méthodes de suppression

Lorsqu'un acteur est détruit, toutes ses entrées/sorties sont préalablement détruites. L'objet graphique est alors supprimé :

```
(define-method destroy ((self <Actor>))
  (for-each destroy (union (inputs self) (outputs self)))
  (tkClean self)
  (remove self))
```

Lorsqu'une entrée est supprimée, tous les liens qui lui sont attachés sont préalablement supprimés. L'objet graphique est alors effacé :

```
(define-method destroy ((self <Input>))
  (for-each destroy (links self))
  (tkClean self)
  (remove self))
```

La destruction d'une poignée est légèrement plus difficile. Si la poignée coupe simplement un lien en deux, elle est supprimée et les deux parties du lien sont réunies. Si la poignée est reliée à plusieurs liens en aval, chacun d'eux est alors supprimé.

```
(define-method destroy ((self <Handle>))
  (let ((upLinkValid (isValid? (upLink self))) ; is the upLink valid ?
        (downLinks (downLinks self)))          ; set of downLinks

    (if (eq? 1 (length downLinks))
      ; if only one downLink
      (let ((theDownLink (car downLinks)))
        (if upLinkValid
          (begin
            ; updates the upLinks
            (set! (downInput  (upLink self))
                  (downInput  theDownLinks))
            (set! (downHandle (upLink self))
                  (downHandle theDownLinks))
            ; redowns the upLink
            (tkClean (upLink self))
            (tkDraw  (upLink self))))
        (destroy theDownLinks))

      ; if several downLinks
      (if upLinkValid
        (destroy (upLink self))
        (for-each destroy downLinks)))

    ; supresses the graphical objet
    (tkClean self)

    ; removes the self from the database
    (remove self)))
```

Lorsqu'un lien est supprimé, il est détruit et enlevé de la base de données en premier, ce qui a pour effet de le rendre inaccessible de toute sélection, évitant ainsi les récursions. Les poignées en amont et en aval sont supprimées :

```
(define-method destroy ((self <Link>))

  ; supresses the graphical objet
  (tkClean self)

  ; removes the self from the database
  (remove self)

  ; suppresses the upHandle, if it exists
  (if (upHandle self)
    ; so, suppresses all the downLink of the upHandle
    (for-each destroy (downLinks (upHandle self))))

  ; suppresses the downHandle, if it exists
  (if (downHandle self)
    (destroy (downHandle self))))
```

3.4 λ-graph

L'éditeur graphique λ-GRAPH est construit en suivant l'analyse ci-dessus [23]. Il est programmé dans le langage STK [37], qui est un SCHEME possédant une interface avec la boîte à outils graphique TK [64]. Son aspect est montré dans la figure 3.4.

Sur la partie gauche de la fenêtre, les boutons dans le panneau de commandes permettent de créer les acteurs en cliquant dans la zone de dessin centrale. Les acteurs peuvent être déplacés, configurés, sélectionnés, reliés et détruits. Le presse-papier autorise les opérations classiques couper/copier/coller.

Dans cette fenêtre, le filtre que nous avons utilisé tout au long de la thèse est programmé. Il contient des acteurs (`-`, `i`, `fby`,...), des poignées (`A'`) et des liens (`A→A'`).

Un acteur possède plusieurs entrées (en haut) et plusieurs sorties (en bas). Une sortie ne peut être liée qu'à une sortie ou une poignée. Une poignée est la continuation d'une sortie.

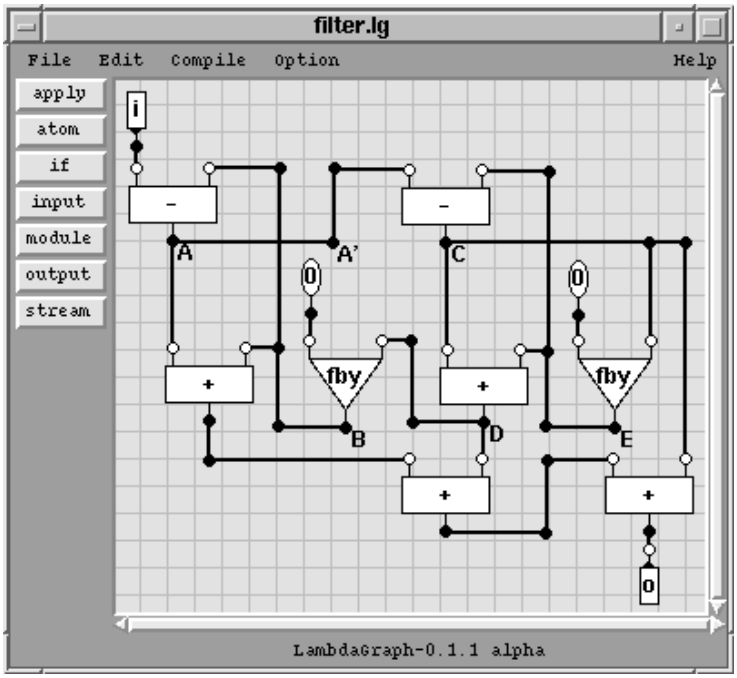


FIG. 3.4 - *Éditeur graphique* λ-GRAPH.

Elle peut être manipulée séparément. Les entrées, les sorties et les poignées peuvent avoir des noms (`A`, `A'`). Les liens peuvent être détruits en cliquant sur eux.

Les atomes et les applications sont configurés en fonction de l'algèbre courante. Plusieurs algèbres peuvent être utilisées dans la même application. Une algèbre définit un type de données caractérisé par son nom et sa fonction de vérification. La fonction de vérification vérifie la syntaxe des valeurs qu'entre l'utilisateur dans la boîte de dialogue de configuration. Les algèbres définissent aussi des opérateurs caractérisés par un nom et une signature. Les opérateurs sont des fonctions et ne peuvent avoir qu'une seule sortie. Par exemple, l'opérateur `+` est fourni par l'algèbre des entiers. La chaîne de caractères `0` est vérifiée et validée par la fonction de vérification de cette algèbre.

Un programme (aussi appelé un module) est défini par son corps qui contient plusieurs acteurs reliés les uns aux autres. Un programme peut être sauvé dans un fichier, relu ou imprimé. Les modules sont organisés en librairies.

Dans la figure 3.4, le programme représenté est le filtre numérique traité tout au long de la thèse (§ II-2.3).

3.5 Conclusion

Dans ce chapitre, nous avons conduit l'analyse de l'interface graphique λ-GRAPH. Cet outil permet une programmation graphique des applications destinées à l'architecture parallèle statique, qui est le plus haut niveau d'abstraction de la chaîne d'outils développés durant la thèse.

Les programmes λ-GRAPH sont modulaires et hiérarchiques. Une application est un module qui peut utiliser d'autres modules. Les opérateurs primitifs et les données sont définis par des algèbres chargées dynamiquement. L'interface contrôle les connections entre les acteurs, en évitant par exemple qu'une sortie soit reliée à une sortie.

L'interface possède les principales caractéristiques que l'on peut attendre d'un outil gra-

phique : les opérations de sélections, les opérations copier/coller, etc. Les objets peuvent être déplacés, ou détruits. La zone d'édition peut être imprimée ou sauvée dans le format POST-SCRIPT.

Les fichiers de λ-GRAPH peuvent être traduits dans le langage λ-FLOW, mais l'inverse n'est pas vrai. Cela signifie que l'interface n'effectue pas de contrôles sémantiques. Ces contrôles sont effectués par le compilateur λ-FLOW.

La maquette du projet λ-GRAPH est actuellement terminée et fonctionnelle. L'un des objectifs est de réécrire cette interface avec des outils plus efficaces et plus portables, avec le langage C et la boîte à outils graphique MOTIF.

Chapitre 4

Langage concret : λ-flow

Le langage λ-FLOW et son compilateur [28, 30] (par la suite ce nom désignera soit le langage soit le compilateur, en fonction du contexte) sont basés sur les λ-matrices. Ils leur ajoutent une grammaire concrète et deux expressions nouvelles destinées à faciliter la modularisation des applications.

λ-FLOW étend les possibilités de modularisation des applications primitives des λ-matrices, basées sur les vecteurs, en définissant de nouveaux acteurs, comme l'abstraction du première ordre, l'exportation et l'extraction symbolique. Ces nouvelles expressions donnent au langage toute la souplesse des langages modernes.

λ-FLOW reprend les concepts déjà mis en œuvre dans la définition des λ-matrices, notamment, l'indépendance du langage avec les données manipulées. Les types de données sont décrits dans des fichiers chargés dynamiquement dans le compilateur. La reconnaissance des données repose sur l'utilisation d'expressions régulières. Ceci donne au langage un très grand pouvoir expressif.

De plus, le code produit n'est pas figé dans le compilateur: lui aussi est défini dans des fichiers textes chargés dynamiquement. Nous avons défini quatre cibles reposant sur des langages de nature totalement différentes, comme C, SCHEME, l'assembleur du INTEL-386 et le format `SPA` destiné au compilateur parallèle. Pour permettre un telle souplesse, les fichiers de définition des cibles utilisent un véritable langage interprété.

La structure générale du compilateur est montré dans la figure 4.1.

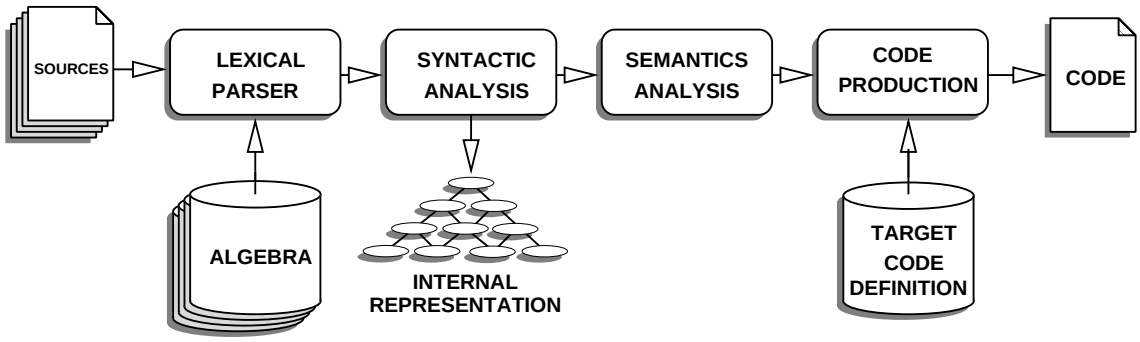


FIG. 4.1 - *Structure générale du compilateur* λ-FLOW. *Les algèbres et les descriptions de cibles sont des fichiers accessibles à l'utilisateur.*

Le compilateur reprend une structure classique [5], mais possède un certain nombre de particularités. L'analyseur lexical traite les fichiers source. Les lexèmes sont enrichis à l'aide des types de données introduits par les algèbres, à l'aide d'expressions régulières [5, 44, 54]. Il est réalisé à l'aide du programme `lex` [58].

L'analyseur syntaxique, réalisé avec `yacc` [46] détecte les erreurs de syntaxe, et construit

la représentation interne du programme à compiler. Cette représentation interne n'est pas accessible à l'utilisateur directement, mais λ-FLOW permet de retranscrire l'arbre sous une forme utilisable, ce qui donne à l'utilisateur la possibilité de contrôler la manière avec laquelle le compilateur a regroupé les expressions [29].

A partir de la représentation interne, le compilateur entreprend l'analyse sémantique du programme. Cette analyse effectue un certain nombre de contrôles, introduits dans la formalisation du compilateur (§ II-5). Il utilise un algorithme complexe de propagation des types dans l'arbre abstrait, ce qui le rend particulièrement efficace.

Une fois que le programme a été vérifié correct, la production de code peut commencer. Elle dépend d'un fichier de description de la cible, chargé dynamiquement dans le compilateur. Le résultat de la compilation est un fichier dont le langage est décrit par la description de la cible.

Dans la section suivante, nous examinons les différentes expressions du langage λ-FLOW (§ 4.1), puis la manière de construire les algèbres (§ 4.2). Puis nous verrons succinctement comment le compilateur effectue son analyse sémantique (§ 4.3) et la production de code, avec les fichiers de définitions de cibles (§ 4.4). Alors nous traiterons l'exemple du filtre repris tout au long de cette thèse (§ 4.5).

4.1 Expression du langage

4.1.1 Commentaire

λ-FLOW utilise les deux formes de commentaires du langage C++. Un commentaire commence par `/*` et finit avec `*/` ou il commence par `//` pour finir à la fin de la ligne.

4.1.2 Identificateurs

Toute chaîne de caractères qui ne contient ni blancs ni caractères spéciaux sont des identificateurs. Un blanc est un espace, un retour à la ligne ou une tabulation. Les caractères spéciaux sont `[ ] ( ) ~ ? : .` ; , ! Ils sont utilisés comme mots clefs dans le langage. Par exemple :

```
toto fooéé #er 123
```

sont des identificateurs. Remarquons que la chaîne de caractères `123` n'est pas directement reconnue comme un entier, comme dans les langages traditionnels, mais comme un identificateur. Cet identificateur ne sera converti en une donnée de type entier qu'après vérification auprès des algèbres chargées dans le compilateur.

4.1.3 Donnée et opérateur

λ-FLOW ne spécifie pas les données et les opérateurs qui y sont associés. Ces spécifications sont rassemblées dans des fichiers de description des algèbres qui sont chargés dynamiquement dans le compilateur. Une algèbre définit un type de données identifié par une expression régulière [5, 44, 54]. Cette expression régulière peut suivre la norme des principaux programmes UNIX utilisant des expressions régulières, comme **`egrep`**, **`sed`**, **`awk`**, etc. λ-FLOW propose un mode qui est proche de celui utilisé par le programme **`lex`**. Par exemple, l'expression régulière des entiers est :

```
check = flow: "[+-]?[0-9]+"
```

λ-FLOW propose par défaut l'algèbre des entiers, qui définit le type **`int`**, car elle est nécessaire dans le langage. Les définitions de cette algèbre sont rassemblées dans le fichier **`integer.alg`** de la distribution standard.

Les opérateurs sur données de λ-FLOW sont aussi définis dans les algèbres, avec un nom, une signature et un commentaire. Par exemple, l'opérateur d'addition sur les entiers est défini par la ligne :

```
+ = int->int->int, integer addition
```

où `+` est le nom de l'opérateur, `int->int->int` sa signature et le reste la ligne est son commentaire. La signature est donnée sous la forme utilisée dans la sémantique dénotationnelle, où le type le plus à droite est le type de la valeur de retour de l'opérateur, et les autres types sont les types des arguments.

Le compilateur λ-FLOW sait gérer des opérateurs avec des noms identiques. Par exemple, l'addition sur des réels sera définie par :

```
+ = real->real->real, real addition
```

Lorsque l'opérateur `+` est rencontré, λ-FLOW vérifie la signature des arguments et choisit l'opérateur d'addition correspondant. Remarquons que λ-FLOW ne définit pas de tour des types, comme c'est le cas dans la plupart des langages. Les conversions de types devront être effectuées manuellement.

4.1.4 Définition

Une définition associe un identificateur à une valeur. Remarquons que nous avons utilisé le verbe associer et non affecter : λ-FLOW n'est pas un langage impératif, mais un langage fonctionnel. Par exemple :

```
baz is foo;
foo is 3;
```

définit deux identificateurs, `baz` et `foo`. L'identificateur `baz` est associé à la valeur de `foo` et `foo` est associé à la valeur `3`.

Remarquons que l'ordre des définitions n'a aucune importance : ici, `foo` est défini *après* que la définition de `baz` ne l'utilise, car dans λ-FLOW, le mot *après* n'a pas de sens.

La valeur d'une définition peut être n'importe quelle expression du langage. Le nom doit être un identificateur. De plus, une définition n'est prise en compte que si elle est une composante directe d'un vecteur. Par exemple, la définition de `baz` n'a aucun effet dans l'écriture `foo := baz := 3`.

Si un identificateur est défini plus d'une fois dans le même environnement, seule la première définition est prise en compte, et le compilateur émet un avertissement.

λ-FLOW utilise une syntaxe à deux niveaux : une syntaxe longue, plus explicite, et une syntaxe courte. Par exemple, la forme courte de `is` est `:=`[1]. L'exemple donnée ci-dessus devient en syntaxe courte :

```
baz := foo;
foo := 3;
```

4.1.5 Alternative

Une alternative est une expression conditionnelle qui permet d'effectuer un choix entre deux expressions appelées *clause-alors* et *clause-sinon*, en fonction d'une expression, la *condition*. Par exemple :

```
if x then 1 else 2;
```

est une alternative. Son résultat dépend de la valeur associée à l'identificateur `x`. La condition et les deux clauses peuvent être n'importe quelle expression du langage. La condition

1. Par la suite, le qualificatif *court* sera plus évident !

doit avoir la signature d'un entier, où `0` dénote la valeur fausse et les autres entiers, la valeur vraie.

Ici, il important de noter que λ-FLOW est un langage sans effets de bord : la question de savoir si l'une seulement des deux clauses sera évaluée, ou si les deux le seront, n'a aucune importance.

La syntaxe courte de l'alternative est :

```
x ? 1 : 2;
```

4.1.6 Application

Une application applique des arguments à un opérateur. Par exemple :

```
+ (1, 2);
```

est l'application de l'opérateur `+` aux arguments `1` et `2`. Pour vérifier une application, λ-FLOW établit en premier lieu la signature des arguments. Dans cet exemple, leur signature est `int`. Il cherche alors dans les algèbres chargées si un opérateur nommé `+` dont la signature des deux arguments est `int` est défini.

Lorsqu'une application n'a que deux arguments, la syntaxe infixée peut être utilisée. L'exemple ci-dessus est alors écrit :

```
1 + 2;
```

Dans cette écriture,les deux espaces de part et d'autre de l'opérateurs sont nécessaires. S'ils sont absents, λ-FLOW interprète l'expression comme l'identificateur `1+2`.

4.1.7 Flot

Le flot de données de λ-FLOW permet de manipuler le temps dans une application. Dans un langage impératif, les données sont rangées dans des cellules mémoires accessibles de deux manières, en lecture ou en écriture. Une cellule est associée à un type de données, et seules des valeurs de même type peuvent être affectées à la cellule.

Avec ces deux moyens d'accès, un langage impératif définit naturellement la séquence, connue comme étant le flot de contrôle du programme. Lorsqu'une valeur est affectée à une variable, deux mondes sont créés : le monde avant l'affectation et le monde après. Le problème lié aux langages impératifs est de connaître l'état d'un programme à un moment de son exécution.

Le flot des langages dataflow est la transposition fonctionnelle de la variable : il définit sa valeur initiale et son futur dans la même expression. Il devient donc possible d'exprimer le futur d'un programme simplement, et donc de connaître son état à un instant donné. Le programme peut alors être formalisé algébriquement, et les vérifications formelles sont possibles.

Dans λ-FLOW, la définition d'un flot est très simple : un flot est composé d'une valeur initiale appelée son état, et de son futur appelé son contrat. Par exemple :

```
0 followed-by 1;
```

définit le flot dont la première valeur est l'entier `0` et toutes les autres valeurs sont l'entier `1`. La sémantique des flots de λ-FLOW est la même que celle utilisée dans le langage LUCID (§ I-5.1). Le flot infini des entiers naturels est écrit :

```
x := 0 followed-by x + 1;
```

Avec la syntaxe courte, ce flot est écrit :

```
x := 1 ~ x + 1;
```

4.1.8 Vecteur

Un vecteur rassemble plusieurs expressions dans une seule structure. Un vecteur commence avec le mot réservé `begin` et se termine avec `end`. Toutes les expressions qu'il contient sont séparées par un point-virgule. Le point-virgule de la dernière expression peut être omis. Par exemple :

```
begin
  1;
  f := 2;
  x := f + 1;
end;
```

est un vecteur.

Les vecteurs jouent le rôle de cadres dans les environnements : toutes les définitions qu'ils contiennent créent des associations accessibles dans tout l'environnement et tous les sous-environnements. Dans le vecteur ci-dessus, les identificateurs `f` et `x` sont visibles dans toutes les expressions du vecteur. Dans l'exemple suivant :

```
begin
  a := 1;
  b := begin
    a := 2;
    c := 3;
    d := a + c;
  end;
end;
```

l'évaluation de `d` est `5`. L'identificateur `a` est appelé variable libre du vecteur `b`. Les variables libres d'un vecteur peuvent être entendues comme des entrées du vecteur. Cela définit une possibilité très primitive de programmation modulaire, qui est grandement améliorée par l'abstraction, définie plus loin.

La syntaxe courte des vecteurs est :

```
[
  f := 2;
  x := f + 1;
];
```

4.1.9 Exportation

Une exportation est une expression qui n'a de sens que comme composante directe des vecteurs. Elle est utilisée pour exporter une composante d'un vecteur avec un nom. Elle s'écrit :

```
foo output 1 + 3;
```

qui exporte l'identificateur `foo` à l'extérieur du vecteur. Une exportation n'est pas une définition, et dans ce cas, l'identificateur `foo` n'est pas défini dans cet environnement. Une valeur exportée peut être lue avec une extraction, expliquée plus loin. Si deux valeurs sont exportées avec le même nom, seule la première est considérée.

La syntaxe courte des exportations est :

```
foo #= 1 + 3;
```

4.1.10 Extraction

Une extraction permet de lire d'une manière indexée une valeur à l'intérieur d'un vecteur. Elle est composée d'une expression indexée et d'un index, et elle s'écrit :

```
indexed extract index
```

λ-FLOW n'essaie pas de lier la valeur de l'index à une définition de l'environnement courant. Il est nécessaire qu'une exportation avec l'index comme symbole soit présente dans le vecteur indexé. La valeur indexée doit être un identificateur faisant référence à un identificateur ou à un vecteur. Par exemple :

```
begin
  x #= 1;
  y #= 2;
  z #= 3;
end extract y;
```

sera évalué comme 2.

λ-FLOW propose aussi les extractions numériques. Dans ce cas, l'index est un entier (ou une expression s'évaluant comme un entier) indiquant le numéro de l'expression à extraire du vecteur, en commençant par 1. Si l'index n'est pas une valeur valable, la première composante du vecteur est extraite, ce qui donne un caractère fonctionnel à l'extraction.

Si l'index est un entier variable, toutes les signatures du vecteur indexé doivent avoir les mêmes signatures. Par exemple :

```
table := [123; 432; 234; 556;];
coef  := table extract x + 1;
```

Dans ce cas, λ-FLOW génère le code correspondant à l'expression :

```
table := [123; 432; 234; 556;];
coef  :=      if (x + 1) = 1 then table extract 1
         else if (x + 1) = 2 then table extract 2
         else if (x + 1) = 3 then table extract 3
         else if (x + 1) = 4 then table extract 4
         else                     table extract 1;
```

La syntaxe courte des extractions est simplement :

```
indexed . index
```

4.1.11 Abstraction

L'abstraction permet de modulariser les programmes λ-FLOW d'une manière simple et puissante. Une abstraction agit un peu à la manière d'une macro-définition du langage C, mais avec une gestion réelle des variables. Elle s'écrit :

```
lambda a, b, c. expr
```

où les identificateurs `a`, `b` et `c` sont les paramètres de l'abstraction, et `expr` son corps.

Si `expr` a comme variables libres l'un ou plusieurs des paramètres, celles-ci sont liées à la valeur que prendront ces paramètres lors de l'instanciation de l'abstraction.

Les paramètres peuvent être typés ou pas. Pour typer un paramètre, il suffit d'écrire :

```
lambda a:type1, b:type2, c:type3. expr
```

où `types` sont types d'algèbres. Il est possible qu'une même abstraction ait des paramètres typés et d'autres sans types.

L'usage des abstractions non typées permet de définir des modules polymorphes qui encouragent la réutilisation du code. Par exemple, l'abstraction suivante est non-typée ; elle est valide si l'opérateur `*` peut être trouvé dans une algèbre :

```
sqr := lambda a. a * a;
```

La syntaxe courte des abstractions est :

```
 a, b, c. expr
```

4.1.12 Déclaration des abstractions

λ-FLOW permet la compilation séparée des programmes. Il est donc nécessaire de pouvoir déclarer les abstractions, sans pour autant les définir, un peu à la manière de déclarer des prototypes de fonctions en C. Pour cela, l'écriture suivante est autorisée :

```
lambda a, b, c
```

qui est l'entête d'une abstraction sans corps.

4.1.13 Instanciation des abstractions

La syntaxe utilisée pour une instanciation est la même que celle utilisée pour les applications. Pour instancier l'abstraction `sqr` définie dans la section précédente, il faut écrire, par exemple :

```
foo := sqr (3);
```

Le mécanisme utilisé dans l'instanciation ressemble plus à un macro-remplacement qu'à un appel de fonction. En fait, le corps de l'abstraction dans lequel les paramètres sont remplacés par leur valeur remplace l'instanciation. Ceci est effectué en conservant la liaison statique des variables. Ce point est crucial pour conserver son homogénéité au langage.

4.1.14 Parenthèses

Toutes les expressions de λ-FLOW peuvent être placées entre parenthèses, en cas d'ambiguïté. Par exemple, les deux expressions suivantes sont équivalentes :

```
1 ~ 2 ~ 3
```

et :

```
1 ~ (2 ~ 3)
```

L'usage des parenthèses est important dans ce langage, car la priorité des opérateurs n'existe pas parcequ'ils sont définis dans des algèbres. Le compilateur offre un mode de fonctionnement qui permet de comprendre comment il a compris une expression [29].

4.1.15 Programme

Un programme λ-FLOW est un ensemble d'expressions qui peuvent être définies dans plusieurs fichiers sources. Parmi ces expressions, il doit y avoir une seule définition de l'abstraction `main` définissant un vecteur qui est le module principal du programme. Si le module `main` possède des paramètres, ils doivent impérativement être typés et ils sont les entrées principales du programme. Si le vecteur possède des exportations, elles sont les sorties principales du programme.

4.2 Algèbre

Une algèbre est un fichier de description qui permet d'ajouter des types de données et des opérateurs sur ces données dans λ-FLOW. Ce sont des fichiers textes que l'utilisateur peut aisément modifier ou créer. Une algèbre est chargée dans le compilateur à l'aide d'une option sur la ligne de commande, ou dans le fichier de configuration du compilateur [29].

La description dynamique des algèbres est une caractéristique importante du langage et de son compilateur qui devient totalement indépendant des applications traitées. Le nombre des algèbres utilisées par un programme n'est pas limité.

Dans cette section, nous présentons l'algèbre des entiers, disponible par défaut dans la distribution standard de λ-FLOW.

4.2.1 Forme générale

Une algèbre est un fichier texte dont la forme générale est la suivante :

```
[type]
name    = int
comment = Basic algebra for integer arithmetic
check   = flow: "[+-]?[0-9]+"

[operators]
+/-     = int->int,      change sign
+       = int->int->int, addition
...
```

où `int` est le nom du type de données défini dans cette algèbre. Le champs `check` est une expression régulière [5, 44, 54] qui permet de reconnaître une chaîne de caractères comme étant un entier ou pas. λ-FLOW propose plusieurs modes d'interprétation des expressions régulières[2]. Le mode le plus simple est celui qui est utilisé dans l'analyseur lexical `lex`, dont le nom est `flow:`

Le nombre des opérateurs définis dans la section `[operator]` n'est pas limité. Un opérateur est défini par un nom, une signature et un commentaire. La signature des opérateurs peut contenir des types provenant d'autres algèbres. Par exemple, nous pouvons définir le convertisseur entier vers un réel en ajoutant la ligne :

```
[operators]
...
int2real = int->real, integer to real convert
```

Le nom des opérateurs définis dans une algèbre peut être utilisé dans d'autres algèbres. Par exemple, l'addition réelle est définie dans l'algèbre des réels par :

```
[operators]
...
+ = real->real->real,  real addition
...
```

Les deux opérateurs sont alors distingués par leur signature.

Il faut remarquer que les définitions des algèbres sont indépendantes de toute réalisation. Elles ne précisent pas le sens de ces opérateurs, mais seulement la manière de les utiliser. La correspondance entre les définitions d'une algèbre et une réalisation particulière se situe dans un fichier texte, lui aussi chargé dynamiquement dans le compilateur, qui définit la cible de la compilation (§ 4.4).

4.2.2 Algèbre des entiers

L'algèbre des entiers de λ-FLOW est définie dans le fichier `integer.alg` de la distribution standard. Ce fichier contient les définitions suivantes :

```
[type]
name    = int
comment = Basic algebra for integer arithmetic
check   = flow: "[+-]?[0-9]+"

[operators]
+/-     = int->int,      change sign
+       = int->int->int, add
-       = int->int->int, subtract
*       = int->int->int, multiply
/       = int->int->int, divide
%       = int->int->int, modulo
=       = int->int->int, equality
>       = int->int->int, greater
<       = int->int->int, smaller
>=      = int->int->int, greaterEq
<=      = int->int->int, smallerEq
<>      = int->int->int, different
```

2. Ce sont les modes utilisés dans différents programmes de l'environnement UNIX.

```
<<      = int->int->int, shift left
>>      = int->int->int, shift left
&       = int->int->int, shift right
|       = int->int->int, shift right
&&      = int->int->int, shift right
||      = int->int->int, shift right
zero    = int->int,      is zero
```

Les opérateurs définis rappellent ceux du langage C. Le nombre des opérateurs peut être augmenté sans limite.

4.3 Analyse sémantique

Le compilateur λ-FLOW effectue une analyse sémantique basée sur les fonctions de critères vues dans la partie théorique de cette thèse (§ II-4). Notamment, le compilateur vérifie qu'un programme est calculable, fermé et constant. Cependant, le compilateur tient compte des abstractions et de la liaison statique des variables, ce qui complique quelques peu l'algorithme.

4.3.1 Calculabilité

La calculabilité des programmes garantit leur déterminisme en temps nécessité par l'architecture parallèle statique. Ce critère interdit les équations de point-fixe dans les programmes. Une telle équation intervient lorsque la valeur qui définit un identificateur utilise directement ou non cet identificateur sans passer par un flot, comme dans :

```
x := x + 1;
```

qui définit une équation de point-fixe sur `x`. Une équation de point-fixe peut ou non avoir une solution ; le problème est que l'obtention de cette solution n'est pas déterministe en temps. Une équation récurrente crée un cycle au travers d'un flot, ce qui fait que l'identificateur est considéré à des instants différents. Par exemple :

```
x := 0 followed-by x + 1;
```

définit le flot des entiers naturels. Les deux `x` sont considérés à des instants différents. λ-FLOW doit aussi contrôler le contrat des flots, qui peuvent eux aussi créer des cycles, comme dans :

```
x := [0; 0] followed-by [a:=a + 1; 2];
```

où le contrat `[a:=a + 1; 2]` ne peut être évalué du fait du cycle sur `a`. L'algorithme utilisé par λ-FLOW pour vérifier l'existence des cycles est rendu complexe par la prise en compte des abstractions. Considérons le programme de la figure 4.2.

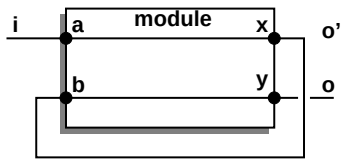


FIG. 4.2 - *Instanciation d'un module.*

Ce module est programmé avec :

```
module := lambda a, b. begin
  x #= a;
  y #= b;
end;
```

Une instanciation de ce module pourrait être :

```
instance := module (i, o');
o'       := instance . x;
o        := instance . y;
```

Il est clair que la définition de `o'` semble créer un cycle. Le compilateur doit donc suivre les liens symboliques pour s'apercevoir de l'absence de cycles.

4.3.2 Fermeture

Un programme fermé ne contient pas de variables libres. λ-FLOW utilise ce critère pour savoir s'il peut passer à la production du code à partir d'un programme source, ou s'il doit s'arrêter à la vérification sémantique.

λ-FLOW utilise une liaison statique des identificateurs, qui est plus naturelle pour le programmeur, mais surtout, qui permet des définitions de modules dont les comportements sont connus et déterministes, ne dépendant pas du contexte d'utilisation. Considérons l'exemple suivant :

```
a := 3;
f := lambda x. x + a;
begin
  a := 4;
  f (3);
end;
```

Dans ce programme, la variable libre `a` du module `f` est liée à la première définition de `a`, même lorsqu'elle est instancié dans un environnement définissant `a`. Nous avons indiqué plus haut que le compilateur agissait à la manière d'un macro-processeur ; on se rend compte ici qu'il effectue un travail de liaison des variables beaucoup plus complexe qu'un simple processeur de macro-définitions.

4.3.3 Constance

Le critère de la constance effectue un contrôle des types des expressions. λ-FLOW utilise un algorithme progressif complexe pour vérifier les types des données. Ce contrôle approfondi libère le programmeur des nombreuses déclarations de types que l'on rencontre dans les langages typés traditionnels. Un programme λ-FLOW peut ne contenir aucune déclaration de types, sauf pour les entrées principales : le compilateur se charge de déduire les types de toutes les expressions. Cette stratégie libère le programmeur et permet aussi une programmation polymorphe autorisant une réutilisation maximale du code.

Les entrées principales du programme doivent être typées, comme par exemple :

```
main := lambda in1 : type1, in2 : type2, ...
```

La signature des expressions est soit déduite de leur syntaxe, lorsqu'elles sont des données, soit déduite de la signature de l'opérateur, soit forcée par le programmeur avec des abstractions totalement ou partiellement typées.

Considérons le programme suivant :

```
sum := lambda init, unit. [
  state := init followed-by state + unit;
  out #= state;
];
```

qui est indépendant de tout type de données. Cette définition récursive peut être instanciée avec :

```
integers := sum(0, 1) . out;
```

ou avec :

```
complexes := sum(0+i0, 1+i0) . out;
```

où `a+ib` serait reconnu par l'algèbre des nombres complexes. Dans ces deux instanciations, l'opérateur `+` doit être défini dans une algèbre pour les données utilisées. Bien évidemment, il aurait été possible de spécialiser l'abstraction `sum` aux nombres complexes avec la définition :

```
sum   := lambda init:complex, unit:complex. [
  state := init followed-by state + unit;
  out   #= state;
];
```

Avec cette nouvelle définition, l'instanciation `sum(0, 1)` provoque une erreur du compilateur, parceque `0` et `1` sont de type entier, et que `sum` attend deux nombres complexes.

4.4 Production de code

Le code produit par le compilateur λ-FLOW est indépendant du compilateur. Chaque cible est définie dans un fichier de configuration appelé TCDF pour *Target Code Definition File*. Ce fichier est chargé dans le compilateur par une option sur sa ligne de commande, ou par une option dans son fichier de configuration.

Par la suite, nous appellerons λ-FLOW-TARGET le compilateur λ-FLOW pour la cible TARGET.

4.4.1 Organisation du code produit

Le code produit par λ-FLOW est basé sur une itération principale[3]. Cette itération contient plusieurs parties distinctes clairement identifiées. La figure 4.3 montre cette structure.

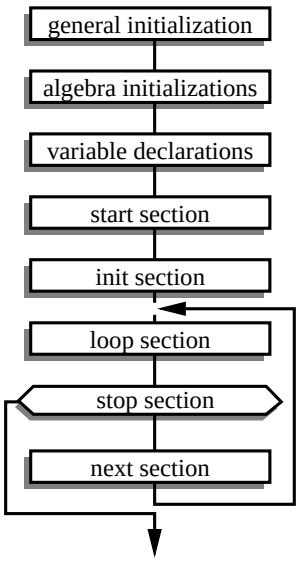


FIG. 4.3 - *Structure de code produit par* λ-FLOW.

Le code est organisé en huit sections. Les trois premières sont déclaratives. Les cinq suivantes réalisent effectivement l'itération. Après une initialisation générale, chaque algèbre utilisée est initialisée, puis toutes les variables du programme sont déclarées. La section `start` calcule toutes les variables nécessaires à l'initialisation des flots. La section `init` initialise les flots du programme ainsi que les entrées principales. La section `loop` est le corps de l'itération. Elle calcule toutes les variables nécessaires pour les sorties principales du programme, la

3. Rappelons que l'objectif est de produire un code pouvant être implanté sur l'architecture parallèle statique décrite dans l'introduction (§ 1-1.3).

condition d'arrêt et pour les nouvelles valeurs des flots. La section `stop` évalue la condition d'arrêt : si elle est vraie, le programme effectue un saut après la section `next`. Si la condition d'arrêt est fausse, le programme recycle tous les flots et le processus reprend alors à la section `loop`.

4.4.2 Chaînes de formats des TCDF

Cette structure très générale du code produit peut être adaptée pour plusieurs cibles. Une cible est un fichier texte chargé dynamiquement dans le compilateur. Il a la même structure que les fichiers d'algèbres, contenant des sections qui regroupent des définitions.

Les définitions sont données sous la forme de chaînes de formats, qui peuvent revêtir plusieurs formes. Une chaîne de formats est, d'une manière générale, une chaîne de caractères contenant des «trous». Ces trous vont être remplacés par des valeurs fournies par le compilateur. Les fonctions de la famille de `printf` du langage C utilisent de telles chaînes de formats.

λ-FLOW propose trois formes de chaînes de formats :

- Simple chaîne : Le format est une simple chaîne de caractères, qui sera placée sans modifications dans le fichier cible, comme :

```
item = une chaîne
```

- Format `printf` : ce format permet des remplacements simples. Il est basé sur les formats des fonctions `printf` du langage C. Nous avons par exemple :

```
command = "gcc %s %s -o %s"
```

  qui définit le format `command`. Les `%s` seront remplacés par des valeurs. Dans ce cas, il s'agira des options à donner à la commande externe, du fichier source et du fichier cible, dans cet ordre.

- Format `slang` : ce format permet de placer des commandes dans le langage SLANG. il s'agit de fonctions SLANG comme :

```
command = slang (opt, in, out) {\
            return Sprintf ("gcc %s %s -o", opt, in, out, 3);\
          }
```

  qui se comporte de la même manière que le format `printf` précédent. Nous remarquons tout d'abord qu'un format peut contenir plusieurs lignes, si celles-ci se terminent par le caractère \. Le format `slang` est une fonction dont les arguments sont déterminés par la définition. Ici, dans le cas de `command`, ce sont les options de la commande externe, le nom du fichier source et le nom du fichier cible.

  L'utilisation du langage SLANG permet une extrême souplesse dans la définition des TCDFs.

4.4.3 Définitions générales dans un TCDF

Dans cette section, nous allons prendre comme exemple le TCDF pour le langage C, qui est le fichier `c.trg` de la distribution standard. Lorsque ce fichier est chargé dans le compilateur, le code produit est du C. Nous avons aussi défini les TCDF pour le langage SCHEME et pour l'assembleur du `Intel-386`.

Les premières définitions sont regroupées dans la section `target` du TCDF. Elles concernent des définitions générales, et des définitions permettant de réaliser la boucle principale. Voici ces premières définitions :

```
[target]
extension    = c
```

```
command      = "gcc %s %s -o %s"
linear       = yes
comment      = "/* %s */"
width        = 50
init         = #include <stdio.h>\n

identifier   = "%s"
alternative  = "(%s) ? (%s) : (%s)"

pre-start    = \n main() {\n  start:
pre-init     = "\n  init:"
pre-loop     = "\n  loop:"
exit         = "    if (%s) exit(0);"
pre-next     = "\n  next:"
post-next    = "    goto loop;\n}"
```

L'`extension` sera extension du fichier produit ; ainsi, si nous compilons le fichier source `exemple.lf`, le compilateur produira le fichier `a.c`. La `command` est la commande externe que λ-FLOW exécutera après avoir produit le fichier cible. Elle attend trois arguments, les options de la commande qui peuvent être indiquées sur la ligne de commande de λ-FLOW, le nom du fichier source de la commande, ici `exemple.c` et le nom du fichier cible de la commande, ici `a.c`. L'option `linear` permet d'indiquer au compilateur s'il autorise les expressions imbriquées ou pas. La définition `comment` donne la forme des commentaires du langage cible. Elle attend un argument, la chaîne de commentaire. L'option `width` permet d'indiquer la largeur maximale en caractères du fichier cible.

La définition de `init` sera placée comme entête du fichier produit. `Identifier` est la manière d'écrire les identificateurs dans le langage cible. Par exemple, certains langages n'acceptent pas de caractères autres que des lettres comme identificateurs. Cette définition permettrait de traiter ces identificateurs avant de les écrire dans le fichier cible. La définition de `alternative` donne le format à utiliser pour écrire les alternatives dans le langage cible. Lorsque la cible est un langage de haut niveau, cette définition ne pose pas de problèmes particuliers. Mais avec un langage d'assemblage, cette définition est réalisée avec des expressions en SLANG qui permettent de gérer les index des labels.

Les définitions suivantes réalisent la structure du fichier cible. Par exemple, `pre-start` est placée avant la section `start` du fichier cible.

4.4.4 Transcription des algèbres dans un TCDF

Les sections suivantes des TCDF traduisent les opérateurs des algèbres dans le langage cible. Pour chaque type de données utilisé dans le programme source, il est nécessaire de définir une section portant le nom du type dans le TCDF. Pour réaliser l'algèbre des entiers, le TCDF de C possède une section `[int]`.

Cette section commence par définir un certain nombre de définitions à usage général, puis définit tous les opérateurs de l'algèbre. Nous avons les définitions générales suivantes :

```
[int]
declare      = "static int %s;"
assign       = "    %s = %s;"
init         = "/* get/putint functions */\
                int getint(int port) {\
                  int tmp;\
                  if (scanf ("%%d", &tmp) == EOF) exit(0);\
                  return tmp;\
                }\
                #define putint(port,value) printf("%%d ", value)\n"

data         = "%s"
```

La première définition permet de déclarer les variables dans cette cible. Elle attend un argument, le nom de la variable à déclarer. La définition suivante définit l'affectation d'une valeur à une variable. Elle a deux arguments, le nom de la variable et la valeur à affecter. La définition suivante, `init`, sera placée lors de l'initialisation des algèbres, comme nous l'avons vu plus haut. Ici, cette initialisation définit en C les fonctions d'entrée / sortie. Enfin, `data` donne le moyen d'écrire les entiers.

Les définitions suivantes traduisent tous les opérateurs de l'algèbre des entiers en C :

```
[int]
+/-          = int->int,        "- (%s)"
+            = int->int->int,   "(%s) + (%s)"
-            = int->int->int,   "(%s) - (%s)"
*            = int->int->int,   "(%s) * (%s)"
/            = int->int->int,   "(%s) / (%s)"
%            = int->int->int,   "(%s) % (%s)"
=            = int->int->int,   "(%s) == (%s)"
>            = int->int->int,   "(%s) > (%s)"
<            = int->int->int,   "(%s) < (%s)"
>=           = int->int->int,   "(%s) >= (%s)"
<=           = int->int->int,   "(%s) <= (%s)"
<>           = int->int->int,   "(%s) != (%s)"
<<           = int->int->int,   "(%s) << (%s)"
>>           = int->int->int,   "(%s) >> (%s)"
&            = int->int->int,   "(%s) & (%s)"
|            = int->int->int,   "(%s) | (%s)"
&&           = int->int->int,   "(%s) && (%s)"
||           = int->int->int,   "(%s) || (%s)"
zero         = int->int,        "(%s) == 0"
@in          = int->int,        "getint (%s)"
@out         = int->int->int,   "putint (%s, %s)"
```

Les opérateurs `@in` et `@out` ne font pas partie des algèbres et sont définis pour effectuer les entrées / sorties principales du programme. Pour tous les types de données définis, il est nécessaire de définir ces deux fonctions.

4.5 Exemple

Ici, nous reprenons l'exemple donné tout au long de cette thèse (§ II-2.3). En λ-FLOW, le filtre est réalisé avec :

```
filter := lambda i. begin
  o #= a + b + c + d;
  a := i - b;
  b := 0 followed-by d;
  c := a - e;
  d := c + e;
  e := 0 followed-by c;
end;
```

Le programme principal est défini par :

```
main := lambda in:int . [
  out #= filter (in) . o;
];
```

Ce programme possède une entrée principale, `in` et une sortie `out`. λ-FLOW impose que les entrées principales du programme soient typées, car il ne peut déduire ces types automatiquement.

Ce programme est écrit dans deux fichiers, `filter.lf` et `main.lf`, car λ-FLOW permet la compilation séparée des programmes.

Pour compiler ce fichier et produire du code C, nous entrons la commande :

```
> flow --target c --algebra integer main.lf filter.lf
message: main parameter 'i' is assigned to port '1'
message: main output 'out' is assigned to port '1' and name 'out_1'

Ah, we have... 0 error and 1 warning.
time=2s, heapsize=196608, gc calls=1
```

Cette commande produit un fichier exécutable `a.out`. Ce fichier utilise la commande externe définie dans le fichier TCDF du C. Pour conserver le fichier C intermédiaire, il faut utiliser la commande suivante :

```
> flow -k --target c --algebra integer main.lf filter.lf
message: main parameter 'i' is assigned to port '1'
```

```
message: main output 'out' is assigned to port '1' and name 'out_1'
warning: no specified post-compiler in target 'c'

Hum... So, we have... 0 error and 1 warning.
time=2s, heapsize=196608, gc calls=1
```

qui produit le fichier `a.c` suivant :

```
#include <stdio.h>

int getint(int port) {
  int tmp;
  if (scanf ("%d", &tmp) == EOF) exit(0);
  return tmp;
}
#define putint(port,value) printf("%d ", value)

static int d;
static int e;
static int i;
static int out_1;
static int a;
static int b;
static int c;

main() {
  start:

  init:
    i = getint (1);
    b = 0;
    e = 0;

  loop:
    a = (i) - (b);
    c = (a) - (e);
    d = (c) + (e);

    out_1 = putint (1, (a) + ((b) + ((c) + (d))));

  next:
    i = getint (1);
    b = d;
    e = c;

    goto loop;
}
```

Pour produire du SCHEME plutôt que du C, il suffit d'entrer la commande :

```
> flow -k --target scheme --algebra integer main.lf filter.lf
message: main parameter 'i' is assigned to port '1'
message: main output 'out' is assigned to port '1' and name 'out_1'
warning: no specified post-compiler in target 'scheme'

Hum... So, we have... 0 error and 1 warning.
time=2s, heapsize=196600, gc calls=1
```

qui produit le fichier `a.scm` suivant :

```
(define (getint port) (read))
(define (putint port value) (display value))
(define (bool->int bool) (if bool 1 0))
(define (int->bool  int) (if (zero? int) #f #t))

(let* (
      )
(let loop (
  (i (getint 1))
  (b 0)
  (e 0)
         )
(let* (
  (a (- i b))
  (c (- a e))
  (d (+ c e))
  (out_1 (putint 1 (+ a (+ b (+ c d)))))
       )
(loop
```

```
  (i (getint 1))
  (b d)
  (e c)
))))
```

Chapitre 5

Simulateur graphique : λ-spaS

Dans ce chapitre, nous présentons le simulateur de l'architecture parallèle statique. Pour cela, nous utilisons le langage SCHEME, et plus particulièrement, la couche objet STKLOS de l'interprète STK définie dans la section (§ 2).

L'utilisation conjointe d'un formalisme objet et du langage SCHEME permet de spécifier dans ces lignes la totalité du simulateur de l'architecture parallèle statique. Les aspects graphiques du simulateur ne sont cependant pas abordés.

Ce simulateur permet l'exécution des programmes en mode pas à pas ou en mode animation. Il est possible de changer dynamiquement le contenu des registres de chaque contrôleur, de modifier le contenu de la mémoire globale ou de changer chacun des programmes des contrôleurs. De plus, il permet le positionnement de points d'arrêt dans le code.

Cette spécification montre la faisabilité de l'architecture parallèle statique. Ce simulateur pourra servir de point de départ à un simulateur plus précis de l'architecture et à sa réalisation matérielle.

Dans la première section, l'architecture sera spécifiée, notamment à l'aide du langage d'assemblage des contrôleurs qui la compose (§ 5.1). Puis nous effectuerons une analyse objet de l'application, en présentant le modèle objet puis le graphe d'héritage (§ 5.2). Par la suite, nous conduirons la définition des méthodes du simulateur (§ 5.3) puis nous montrerons l'aspect de son interface graphique (§ 5.4).

5.1 Spécifications du simulateur de l'architecture parallèle statique

L'architecture parallèle est une simple interconnection entre des contrôleurs classiques, la mémoire commune et les dispositifs d'entrée/sortie. Le modèle de l'architecture est montré dans la figure 5.1.

Les contrôleurs sont directement connectés au bus passif, qui donne son nom à l'architecture. La mémoire globale, accessible à tous les contrôleurs, est aussi connectée au bus, ainsi que les dispositifs d'entrée/sortie.

Chaque contrôleur possède sa mémoire vive RAM, sa mémoire morte ROM et un accès au bus commun. La ROM de chaque contrôleur contient le programme. Tous les programmes sont indépendants. La mémoire vive n'est pas utilisée actuellement dans le modèle de compilation que nous proposons. Les contrôleurs ont deux registres `A` et `B` à usage général, un pointeur de programme `CP`, trois pointeurs de pile `SP`, `SB` et `BP` inutilisés. Les indicateurs `N`, pour la négation, `Z` pour la nullité et `W` pour les dépassements de capacité sont positionnés par les différentes instructions. Tous les registres sont localisés dans la mémoire vide des contrôleurs

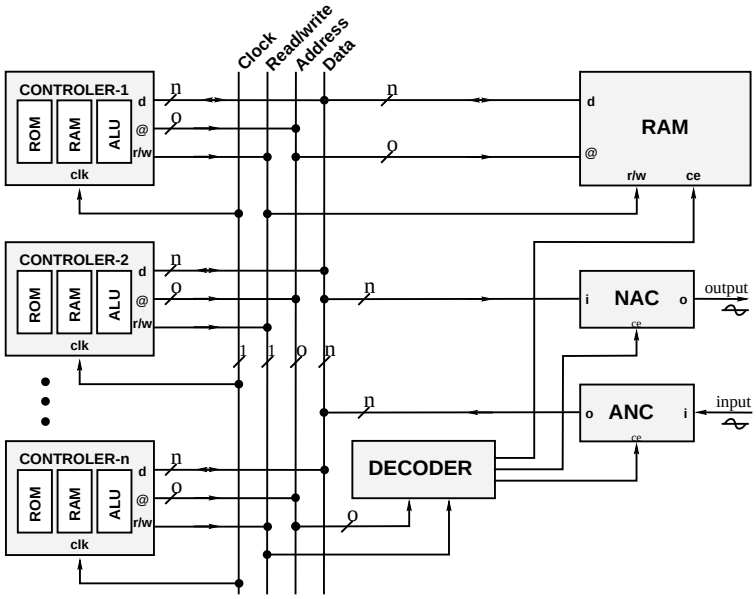


FIG. 5.1 - *Modèle de l'architecture parallèle statique.*

aux adresses indiquées par la table suivante :

register	address	name
A	0	first general
B	1	second general
CP	2	code pointer
SB	3	stack base
SP	4	stack pointer
BP	5	base pointer
N	6	negative flag
W	7	overflow flag
Z	8	zero flag

Le langage d'assemblage des contrôleurs est très simple. Les instructions sont données dans la table suivante :

code	cycle	NWZ	action
			jump
jmpa c	1		$cp \leftarrow c$
jmp c	1		$cp \leftarrow cp + c$
jn c	1		$n = 0\ ?\ cp \leftarrow cp + c\ :\ cp \leftarrow cp + 2$
jnn c	1		$n \neq 0\ ?\ cp \leftarrow cp + c\ :\ cp \leftarrow cp + 2$
jw c	1		$w = 0\ ?\ cp \leftarrow cp + c\ :\ cp \leftarrow cp + 2$
jnw c	1		$w \neq 0\ ?\ cp \leftarrow cp + c\ :\ cp \leftarrow cp + 2$
jz c	1		$z = 0\ ?\ cp \leftarrow cp + c\ :\ cp \leftarrow cp + 2$
jnz c	1		$z \neq 0\ ?\ cp \leftarrow cp + c\ :\ cp \leftarrow cp + 2$
			move
mov @	1		$[@] \leftarrow A, cp \leftarrow cp + 2$
get @	1		$A \leftarrow [@], cp \leftarrow cp + 2$
ldc #	1		$A \leftarrow \#, cp \leftarrow cp + 2$
lda &	1		$A \leftarrow [\&], cp \leftarrow cp + 2$
sta &	1		$[\&] \leftarrow A, cp \leftarrow cp + 2$
exc &	1		$A \leftrightarrow [\&], cp \leftarrow cp + 2$
			stack
ldsb #	1		$sb \leftarrow \#, sp \leftarrow sb, cp \leftarrow cp + 2$
ldbp &	1		$bp \leftarrow [\&], cp \leftarrow cp + 2$

...

...

code	cycle	NWZ	action
push &	2		$[sp] \leftarrow [\&], sb \leftarrow sb + 1, cp \leftarrow cp + 1$
pop &	2		$[\&] \leftarrow [sp], sb \leftarrow sb - 1, cp \leftarrow cp + 1$
top #	1		$A \leftarrow [bp - \#], cp \leftarrow cp + 2$
procedure			
call c	4		$push(a), push(bp), push(cp), cp \leftarrow c$
ret	3		$pop(cp), pop(bp), pop(a)$
math			
neg &	1	•	$A \leftarrow [\&], A \leftarrow -A, cp \leftarrow cp + 2$
inc &	1	•	$A \leftarrow [\&], A \leftarrow A + 1, cp \leftarrow cp + 2$
dec &	1	•	$A \leftarrow [\&], A \leftarrow A - 1, cp \leftarrow cp + 2$
add &	1	•	$B \leftarrow [\&], A \leftarrow A + B, cp \leftarrow cp + 2$
sub &	1	•	$B \leftarrow [\&], A \leftarrow A - B, cp \leftarrow cp + 2$
mul &	3	•	$B \leftarrow [\&], A \leftarrow A * B, cp \leftarrow cp + 2$
div &	3	•	$B \leftarrow [\&], A \leftarrow A/B, cp \leftarrow cp + 2$
mod &	3	•	$B \leftarrow [\&], A \leftarrow A\%B, cp \leftarrow cp + 2$
logic			
not &	1	•	$A \leftarrow [, A \leftarrow not\ A, cp \leftarrow cp + 2$
shr &	1	•	$B \leftarrow [\&], A \leftarrow A >> B, cp \leftarrow cp + 2$
shl &	1	•	$B \leftarrow [\&], A \leftarrow A << B, cp \leftarrow cp + 2$
and &	1	•	$B \leftarrow [\&], A \leftarrow A\&B, cp \leftarrow cp + 2$
or &	1	•	$B \leftarrow [\&], A \leftarrow A\|B, cp \leftarrow cp + 2$
xor &	1	•	$B \leftarrow [\&], A \leftarrow A \wedge B, cp \leftarrow cp + 2$
nor &	1	•	$B \leftarrow [\&], A \leftarrow !(A\|B), cp \leftarrow cp + 2$
nand &	1	•	$B \leftarrow [\&], A \leftarrow !(A\&B), cp \leftarrow cp + 2$
other			
nop	1		$cp \leftarrow cp + 1$
cmp &	1	•	$B \leftarrow [\&], A - B, cp \leftarrow cp + 2$
trap #	1		interrupt #

• = les indicateurs N, W et Z sont modifiés
= constante
& = adresse RAM locale - [&] valeur adressée
@ = adresse RAM externe - [@] valeur adressée
c = adresse ROM locale

Le jeu d'instructions est voulu aussi réaliste que possible mais il est limité. Les instructions sur plusieurs cycles sont modélisées, comme la multiplication. Les instructions d'accès à la mémoire permettent soit un accès à la RAM locale avec `ldc`, `lda` et `sta`, soit un accès à la RAM globale avec `mov` et `get`. Comme les registres sont situés en mémoire vive, toutes les d'accès à la mémoire locale peuvent être utilisés pour les registres. Par exemple `add 0` est identique à `add A`. Un langage d'assemblage effectue cette conversion.

La mémoire globale est aussi utilisée pour les entrées/sorties. Les accès à l'adresse `0` de la mémoire globale sont en fait des accès aux convertisseurs analogiques/numériques et numériques/analogiques.

Le bus commun n'effectue aucun contrôle d'accès. La seule protection des différents composants réside en des tampons d'entrée/sortie ayant une logique à trois états. En cas de conflit d'accès, le comportement de l'architecture ne peut être prévu. La répartition des accès à la mémoire entre les contrôleurs est effectuée pendant la phase de compilation de l'application.

5.2 Analyse objet

Cette section mène une analyse objet du simulateur de cette architecture. Le but de cette analyse est de dégager les différents objets logiciels du simulateur et leurs interactions, afin de faciliter sa programmation.

5.2.1 Modèle objet

Un des objectifs de cette section est de dégager les objets logiciels nécessaires à la modélisation de l'application, ainsi que leurs attributs et leurs relations. La méthode utilisée s'inspire des méthodes de modélisation des bases de données [62].

Le schéma relationnel des objets instanciés de l'application est montré en figure 5.2.

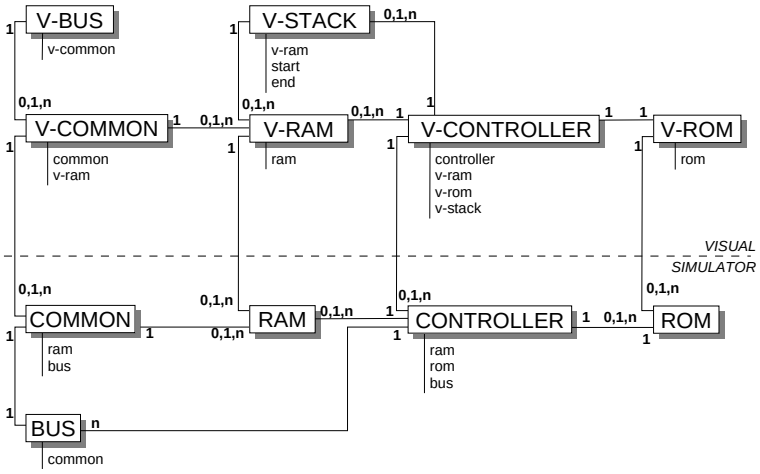


FIG. 5.2 - *Modèle objet du simulateur. Il montre les relations entre les instances des objets. Les noms commençant par* `V-` *sont des objets qui peuvent être affichés via l'interface graphique.*

Les contrôleurs ne sont reliés qu'à une seule RAM et une seule ROM, et ils sont connectés à un seul bus. L'objet `COMMON` représente la RAM globale de l'architecture et les lignes d'entrées/sorties. Il a une seule RAM et il est relié à un seul bus.

Les objets visuels sont reliés logiquement aux objets du simulateur au lieu d'être des spécialisations de ces objets. Ainsi, l'indépendance des deux parties du logiciel, le simulateur et son interface graphique, est préservée.

A partir de ce modèle objet, étudions le graphe d'héritage des objets.

5.2.2 Graphe d'héritage

Cette section modélise les classes d'objets de l'application, déduites du modèle objet. Alors que le modèle objet s'intéresse aux instances des objets et à leurs relations, le graphe d'héritage décrit la construction des objets par une spécialisation d'objets plus primitifs.

Les objets graphiques du simulateur reposent de manière intensive sur les objets proposés par le langage hôte, STK, et son interface objet, STKLOS.

Le graphe d'héritage est donné en figure 5.3.

Les objets de la classe `<Connectable>` disposent de méthodes leur permettant d'être connectés, alors que les objets héritant de la classe `<Castable>` permettent d'effectuer des conversions limitant la taille des données, en rapport du nombre de bits du dispositif. Remarquons que la `<Ram>` hérite des deux classes, indiquant que cet objet convertit les données qu'il contient, par rapport au nombre de bits des cellules mémoire, et qu'il peut être connecté. La classe `<Common>` est une spécialisation de la mémoire, qui effectue en plus un décodage d'adresse pour effectuer les entrées/sorties de l'architecture.

Ici, nous ne décrirons que les objets du simulateur, sans traiter les objets visuels.

5.3 Définition des méthodes

Cette section spécifie le simulateur de l'architecture parallèle statique [26]. Les aspects graphiques du simulateur ne sont pas abordés.

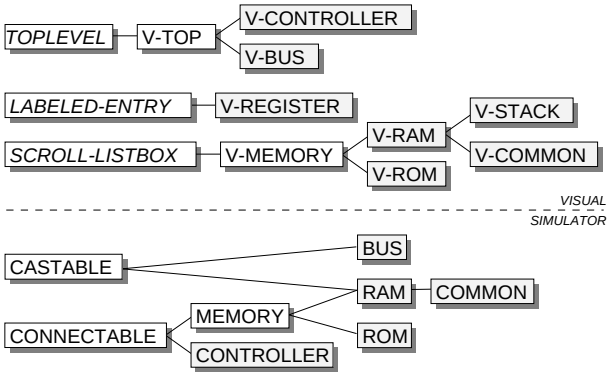


FIG. 5.3 - *Graphe d'héritage des objets du simulateur. Les boîtes grisées sont les feuilles du graphe, qui représentent des objets pouvant être instanciés. Les noms en italique sont les classes fournies par le langage* STKLOS..

5.3.1 Préliminaires

Les principes de la programmation objet utilisés ici sont l'héritage (partage statique de caractéristiques) et la spécialisation des méthodes (partage dynamique des caractéristiques). Ces deux formes de partage des ressources réduisent de manière significative la taille du programme. Voici brièvement les objets du simulateur.

L'architecture consiste en un bus principal sur lequel sont connectés les différentes composantes, plusieurs processeurs et une mémoire globale.

La classe `<Bus>` possède deux fonctionnalités : recevoir l'horloge globale et la propager aux objets connectés, et gérer les accès à la mémoire globale en signalant les conflits. Sa principale méthode est `connect` qui permet la connexion des objets à l'initialisation, `reset` qui initialise tous les objets connectés, `clock` qui reçoit un signal d'horloge et le propage, enfin, `read` et `write!` pour les accès à la mémoire globale.

La classe `<Common>` représente les ressources partagées de l'architecture qui comprennent la mémoire globale et les lignes d'entrées/sorties de l'architecture (on la nommera *mémoire globale* par la suite). Dans le but de simuler les entrées/sorties, deux fichiers sont ouverts, l'un en lecture pour les entrées, et l'autre en écriture, pour les sorties. Les entrées/sorties sont effectuées en accédant à l'adresse `0` des ressources partagées. Les principales méthodes de cette classe sont `read` et `write!`.

La classe `<Ram>` et `<Rom>` sont deux spécialisations de la classe `<Memory>`. Une mémoire est un tableau de cellules contenant des valeurs. Nous accèdons aux cellules avec un index numérique. Les méthodes de la classe `<Memory>` sont `read` et `write!`. La classe `<Rom>` possède aussi la méthode `write!`, peu naturelle pour ce type de mémoire : elle sera utilisée par la méthode `dump` à l'initialisation, pour entrer le programme de chaque contrôleur.

La classe `<Controller>` possède comme attributs une `<Ram>`, une `<Rom>`. Les registres des contrôleurs sont une partie de la mémoire vive. La plupart des méthodes sont relatives au décodage des instructions. De plus, nous trouverons les méthodes `reset` pour ré-initialiser un contrôleur, `clock` pour exécuter la micro instruction courante, `macroInstruction` qui lit et décode la macro-instruction de la ROM et `microInstruction` qui exécute la micro-instruction courante.

Enfin, la classe primitive `<Connectable>` donne à un objet qui en hérite la possibilité de se connecter au bus. La classe `<Castable>` qui fournit à un objet la capacité de forcer des entiers en fonction d'un nombre de bits.

5.3.2 Programme principal : création de l'architecture

Le programme principal du simulateur consiste en la création des objets et en leur connexion logique. Puis le programme entre dans une itération où il active l'horloge du bus à chaque pas. Dans cet exemple, une architecture à deux contrôleurs est créée :

```
; Bus creation
(define bs (make <Bus>))

; Common creation
(define cm (make <Common>
                 :bus bs                  ; bus
                 :size 1000               ; global RAM size
                 :in ''input.dat''        ; input file for IOs
                 :out ''output.dat''))    ; output file for IOs

; First controller creation
(define c1 (make <Controller>
                 :bus bs                  ; bus
                 :ram 200                 ; local RAM size
                 :rom 200                 ; local ROM size
                 :code ''example-2-1.S'')) ; program file to dump in local ROM

; Second controller creation
(define c2 (make <Controller>
                 :bus bs
                 :ram 200
                 :rom 200
                 :code ''example-2-2.S''))

; Main loop
(let mainLoop ()
  (clock bs)
  (mainLoop))
```

Le bus est crée en premier. Il est utilisé dans la création des ressources partagées (mémoire globale et dispositifs d'entrée/sortie). Les entrées/sorties seront effectuées par rapport à deux fichiers donnés en arguments. Les deux contrôleurs sont alors créés de manière à être reliés au bus. Le nom du fichier qui contient le programme en assembleur est donné en argument.

Le programme entre alors dans une itération sans fin à l'aide d'un `let` nommé. La seule action de cette itération est d'invoquer la méthode `clock` du bus. Cette méthode active alors les différents objets qui sont reliés au bus.

5.3.3 Modélisation objet du bus

Le bus est l'interface entre l'architecture et l'utilisateur : il offre plusieurs commandes comme `clock` ou `reset`, et un accès aux données qui y sont transférées. Le bus doit aussi détecter les différents conflits d'accès aux ressources communes. Cela est réalisé avec la méthode `clock`. Lorsque l'horloge du bus est activée, l'horloge de chaque contrôleur est activée, ce qui a pour effet d'exécuter son instruction courante. Cette instruction peut être une opération de lecture ou d'écriture sur le bus ; elle est alors placée dans une liste du bus. Lorsque tous les contrôleurs ont été activés, le bus fait le bilan des accès dans le but de détecter les conflits.

La classe `<Bus>` définit un certain nombre d'attributs : les ressources partagées, la liste des contrôleurs rattachés, la liste des accès en lecture au bus et la liste des accès en écriture au bus :

```
(define-class <Bus> (<Castable>)
  ((common                :accessor common)
   (connected :initform '() :accessor connected)
   (lread     :initform '() :accessor lread)
   (lwrite    :initform '() :accessor lwrite)))
```

La méthode `reset` vide les listes des accès et ré-initialise tous les dispositifs qui sont rattachés au bus :

```
(define-method reset ((self <Bus>))
 ; flushes the lists
  (set! (lread  self) '())
```

```
  (set! (lwrite self) '())
  ; resets connected chips
  (reset (common self))
  (map (lambda (controller)
         (reset controller))
       (connected self)))
```

La méthode `connect` permet de rattacher un objet au bus. Deux sortes d'objets peuvent être connectés : l'objet des ressources communes et les autres objets. Le premier modifie l'attribut `common` de l'instance alors que les autres objets sont simplement placés dans la liste `connected` des objets connectés. Il y a donc deux méthodes `connect`, une par type d'objet connecté :

```
(define-method connect ((self <Bus>)
                        (chip <Common>))
  (set! (common self) chip))
```

et :

```
(define-method connect ((self <Bus>) chip)
  (set! (connected self)
        (cons chip (connected self))))
```

La méthode suivante est invoquée lorsqu'un dispositif accède au bus en lecture. Elle ajoute la liste formée par le dispositif, l'adresse à lire et la procédure à invoquer lorsque la lecture sera effectivement effectuée dans la liste `lread` de la classe `<Bus>` :

```
(define-method read ((self <Bus>)
                     (controller <Controller>)
                     address
                     procedure)
  (set! (lread self)
        (cons (list controller address procedure)
              (lread self))))
```

La méthode `write!` procède de la même manière, en ajoutant la liste formée du dispositif de l'adresse et de la valeur à écrire dans la liste `lwrite` :

```
(define-method write! ((self <Bus>)
                       (controller <Controller>)
                       address
                       value)
  (set! (lwrite self)
        (cons (list controller address (cast self value))
              (lwrite self))))
```

La méthode `clock` est la méthode principale du simulateur. L'horloge de tous les dispositifs connectés est activée. Certains d'entre eux accèdent au bus en lecture ou en écriture : ces accès sont enregistrés dans les listes `lread` et `lwrite!`. Le contrôle des accès peut alors être effectué :

```
(define-method clock ((self <Bus>))
  ; checks if each address of the lread list matches
  (let ((callProcRead
          (lambda (address value)
            (map (lambda (ContAdProc)
                   (if (eq? address (cadr ContAdProc))
                     (apply (caddr ContAdProc)
                            (list value))))
                 (lread self)))))

    ; flushes bus access lists
    (set! (lread  self) '())
    (set! (lwrite self) '())

    ; clocks each controller
    (for-each (lambda (controller)
                (clock controller))
              (connected self))

    ; bus accesses controls
    (case (length (lwrite self))
```

```
; no write access
(0 (if (not (zero? (lread self)))
       (let* ((address (cadad (lread self)))
              (value (cast self (read (common self) address))))
         (callProcRead address value))))

; one write access
(1 (let ((address (cadar (lwrite self)))
         (value (cast self (caddar (lwrite self)))))
      (callProcRead address value)
      (write! (common self) address value)))

; too many write accesses
(else
   (error 'Too many writes accesses on the bus'))))))
```

5.3.4 Modélisation objet des contrôleurs

Cette description porte sur l'aspect externe des contrôleurs et non sur leur structure interne. Nous nous bornerons à donner une interface réaliste des contrôleurs qui se manifeste par un langage d'assemblage. La classe est :

```
(define-class <Controller> (<Connectable>)
  ((ram   :accessor ram)
   (rom   :accessor rom)
   (micro :accessor micro :initform '())

   ; the following register value is its address into the ram
   (A     :accessor A     :initform 0)
   (B     :accessor B     :initform 1)
   (CP    :accessor CP    :initform 2)
   (SB    :accessor SB    :initform 3)
   (SP    :accessor SP    :initform 4)
   (BP    :accessor BP    :initform 5)
   (W     :accessor W     :initform 6)
   (Z     :accessor Z     :initform 7)))
```

Un contrôleur est constitué d'une `ram`, d'une `rom` et de registres `A` à `Z` localisés dans la mémoire vive locale. Les dix premières adresses de la RAM local sont réservées. Chaque contrôleur possède une liste `micro` des micro-instruction à exécuter. La gestion au niveau de la micro-instruction autorise les instructions s'exécutant sur plusieurs cycles.

La fonction d'initialisation des contrôleurs crée la `ram` et la `rom` avec les tailles spécifiées lors de l'invocation de la méthode. Le programme d'assemblage est alors chargé dans la `rom` à partir du fichier indiqué :

```
(define-method initialize ((self <Controller>) arguments)
  (next-method)
  (set! (ram self)
        (make <Ram> :size (get-keyword :ram arguments 100)))
  (set! (rom self)
        (make <Rom> :size (get-keyword :rom arguments 100)))
  (dump self (get-keyword :code arguments '())))
```

La méthode `dump` invoquée lors de l'initialisation d'un contrôleur charge un programme à partir d'un fichier dans la `rom` en utilisant la méthode `dump` de la classe `<Rom>`. Cette dernière possède comme argument une liste de code opératoires. Pour obtenir cette liste à partir d'un fichier, la méthode standard `port->list` est utilisée :

```
(define-method dump ((self <Controller>) fileName)
  (dump (rom self)
        (port->list read (open-input-file fileName))))
```

La méthode `reset` vide la liste `micro` des micro instructions et force les dix premières cellules de la mémoire vive à zéro :

```
(define-method reset ((self <Controller>))
  (set! (micro self) '())
  (let loop ((ad 0))
    (if (not (eq? ad 10))
      (begin
```

```
        (write! (ram self) ad 0)
        (loop (+ 1 ad))))))
```

Les deux méthodes suivantes permettent l'accès symbolique aux registres. La méthode `register` permet les deux accès, en lecture si elle est invoquée sans argument, en écriture avec un argument :

```
; read access to the registers
(define-method register ((self <Controller>) reg)
  (read (ram self) (reg self)))



; write access to the registers
(define-method register ((self <Controller>) reg value)
  (write! (ram self) (reg self) value))
```

La méthode `goto` modifie la valeur du pointeur de programme `CP` pour la positionner à l'argument `address`. Elle est utilisée par la méthode de sauts :

```
(define-method goto ((self <Controller>) address)
  (register self CP address))
```

Les méthodes suivantes positionnent les indicateurs du contrôleur en fonction de l'argument `value` :

```
(define-method setFlags ((self <Controller>) value)
  (register self N (if (< value 0) 1 0))
  (let ((casted (cast (ram self) value)))
    (register self W (if (eq? value casted) 0 1)))
    (register self Z (if (zero? value) 1 0)))
```

La fonction `read` retourne la prochaine macro-instruction lue à partir de la `rom` du contrôleur à l'adresse pointée par le contenu du registre `CP` et incrémente ce pointeur :

```
(define-method read ((self <Controller>))
  (let ((opcode (read (rom self) (register self CP))))
    (goto self (+ (register self CP) 1))
    opcode))
```

La fonction `clock` traite la prochaine instruction à exécuter. Cette instruction est soit une macro-instruction lue à partir de la `rom` soit une micro-instruction issue de la liste `micro`, si elle n'est pas vide :

```
(define-class clock ((self <Controller>))
  (if (null? (micro self))
     (macroInstruction self)
     (microInstruction self)))
```

Si la liste des micro-instructions n'est pas vide, la prochaine instruction à exécuter par le contrôleur est située à sa tête. La tête de la liste est alors déplacée :

```
(define-method microInstruction ((self <Controller>))
  (eval (car (micro self)))
  (set! (micro self) (cdr (micro self))))
```

Les macro-instructions sont lues directement dans la `rom` du contrôleur. Les actions entreprises dépendent de l'instruction :

```
(define-method macroInstruction ((self <Controller>))
  (let ((opcode (read self)))
    (case opcode
      ((nop)
        'nothing)
      ((cmp
        (cmpInstruction self (read self))))
      ((jmp jn jnn jw jnw jz jnz)    ; jump instructions
        (jumpInstructions self opcode (read self)))
      ((mov get ldc lda sta exc)     ; move instructions
         (moveInstruction self opcode (read self)))
      ((ldsb ldbp push pop top)      ; stack instructions
```

```
        (stackInstruction self opcode (read self)))
      ((neg inc dec not)            ; unary instructions
        (math1Instructions self opcode (read self)))
      ((add sub mul div mod         ; binary instructions
and or xor nor nand shr shl)
        (math2Instructions self opcode (read self)))
      (else
        (error ''Unknown instruction: ~a~%'' opcode)))

   ; the Z flag is altered with some instructions
    (case opcode
        ((neg inc dec add sub and or xor nor nand)
          (setFlags self (register self A))))))
```

Maintenant, chaque instruction peut être exécutée par une méthode spécifique. La méthode suivante traite l'instruction de comparaison `cmp`. Les actions à entreprendre sont déduites de la table page 5.1 :

```
(define-method cmpInstruction ((self <Controller>) operand)
 ; set B with the value at the address operand
  (register self B (read (ram self) operand))

 ; set Z according to the A-B result
  (setFlags self (- (register self A) (register self B))))
```

Les instructions de saut modifient la valeur du pointeur d'instruction `CP`, en l'incrémentant avec la valeur de l'opérande, ce qui réalise un saut relatif :

```
(define-method jumpInstructions ((self <Controller) opcode operand)
  (if (eq? opcode) 'jmpa)
    ; absolut jump
    (goto self operand)

    ; relative jump
    (let ((new-cp (+ (register self CP) operand)))
      (case opcode
        ; unconditional jump
        (jmp (goto self new-cp))

        ; jump if N == 1
        (jn  (if (zero (register self N))
                 (goto self new-cp)))

        ; jump if N != 1
        (jnn (if (not (zero (register self N)))
                 (goto self new-cp)))

        ; jump if W == 1
        (jw  (if (zero (register self W))
                 (goto self new-cp)))

        ; jump if W != 1
        (jnw (if (not (zero (register self W)))
                 (goto self new-cp)))

        ; jump if Z == 1
        (jz  (if (zero (register self Z))
                 (goto self new-cp)))

        ; jump if Z != 1
        (jnz (if (not (zero (register self Z)))
                 (goto self new-cp)))))))
```

Les instructions suivantes traitent les transferts des données entre les registres et les ressources partagées, et les transferts entre les registres et la mémoire locale :

```
(define-method moveInstructions ((self <Controller>) opcode operand)
  (case opcode
    (mov
     ; write on the bus the value in A
      (write! (bus self) self operand (register self A)))

    (get
     ; put in A the value read from the bus
      (read (bus self) self operand
            (lambda (value)
                (register self A value))))
```

```
(ldc
 ; load in A a constant
 (register self A operand))

(lda
 ; load in A the value at the address operand
 (register self A (read (ram self) operand)))

(sta
 ; stores at the address operand the value in A
 (write! (ram self) operand (register self A)))

(exc
 ; exchange A and the value at the address operand
 (let ((tmp (register self A)))
     (register self A (read (ram self) operand))
     (write! (ram self) operand tmp)))))
```

Les prochaines instructions manipulent la pile des contrôleurs. Certaines de ces instructions nécessitent plus d'un cycle d'instruction, ce qui met en œuvre la liste des micro-instructions des contrôleurs. Dans ce cas, la liste est formée des actions à entreprendre à chaque cycle. Les actions seront évaluées à l'aide de `eval` par la suite, dans la fonction `microInstruction`:

```
(define-method stackInstructions ((self <Controller>) opcode operand)
  (case opcode
    (ldsb
     ; put in SB and SP the constant operand
     (register self SB operand)
     (register self SP (register self SB)))

    (ldbp
     ; put in BP the value at the address operand
     (register self BP (read (ram self) operand)))

    (push
     ; write at the address in SP ...
     ; ... the value at the address operand
     (write! (ram ,self) (register ,self SP)
             (read (ram ,self) ,operand))
     ; micro-coded
     (set! (micro self)
          `(; decrement SP
            (register ,self SP (dec (register ,self SP))))))

    (pop
     ; write at the address operand ...
     ; ... the value at the address SP
     (write! (ram ,self) ,operand
             (read (ram ,self) (register , self SP)))
     ; micro-coded
     (set! (micro self)
          `(; decrement SP
            (register ,self SP (inc (register ,self SP))))))

    (top
     ; put in A the value at the address BP-operand
     (register ,self A
               (read (ram self) (sub (register self BP) operand))))))
```

Les opérateurs unaires de l'assembleur sont maintenant traités. L'argument `opcode` est un opérateur mathématique de SCHEME, ce qui facilite la réalisation :

```
(define-method math1Instructions ((self <Controller>) opcode operand)
  (register self A (eval (list opcode (read (ram self) operand)))))
```

Les opérateurs binaires sont exécutés en un ou plusieurs cycles :

```
(define-method math2Instructions ((self <Controller>) opcode operand)
  (case opcode
    ((mul div mod)
       (register ,self B (read (ram ,self) ,operand))
      ; micro-coded
       (set! (micro self)
             `((register ,self A (eval (list ,opcode
```

```
                                        (register ,self A)
                                        (register ,self B))))
            (setFlags ,self (register ,self A)))))

   (else
      (register self B (read (ram self) operand))
      (register self A (eval (list opcode
                                   (register self A)
                                   (register self B)))))))
```

Tous les opérateurs binaires et unaires doivent être définis dans l'environnement d'exécution du simulateur, comme `define add +`, ce qui permet de les évaluer simplement avec `eval`.

5.3.5 Modélisation objet des ressources partagées

Les ressources partagées sont constituées d'une mémoire vide globale et des lignes d'entrées/sorties décodées à l'adresse zéro. Pour simuler le monde extérieur, deux fichiers sont utilisés, l'un en lecture pour les entrées, et l'autre en écriture, pour les sorties.

La classe `<Common>` qui définit les ressources communes hérite de la classe `<Ram>` et possède deux attributs, `in` pour le fichier d'entrée et `out` pour le fichier de sortie. De plus, les attributs `cin` et `cout` contiennent les dernières valeurs transférées, nécessaires pour l'interface graphique :

```
(define-classe <Common> (<Ram>)
  ((in  :accessor in)
   (out :accessor out))
   (cin :accessor currentIn)
   (cout:accessor currentOut)))
```

La méthode `initialize` lit dans les arguments les noms du fichier d'entrée et de sortie pour affecter les fichiers correspondant aux attributs ;

```
(define-method initialize ((self <Common>) arguments)
  (let ((file (get-keyword :in arguments '())))
    (set! (in self)
          (if (null? file) '()
               (oen-input-file file))))
  (let ((file (get-keyword :out arguments '())))
    (set! (out self)
          (if (null? file) '()
               (open-output-file file))))
  (next-method))
```

Lorsque la méthode `read` est invoquée, l'adresse est décodée. S'il s'agit de l'adresse `0`, le fichier d'entrée `in` est lu, sinon la méthode `next-method` est invoquée, ce qui a pour effet d'invoquer la méthode `read` de la classe `<Ram>` :

```
(define-method read ((self <Common>) address)
  (if (zero? address)
    (let ((value 0))
       (if (not (null? (in self)))
         (begin
          ; uses the SCHEME read function
          (set! value (read (in self)))
          (if (eof? value)
           (begin
             (close-input-port (in self))
             (set! value 0)
             (set! (in self) '())))))
       (set! (currentIn self) (cast self val))
       (currentIn self))
    (next-method)))
```

La méthode `write!` possède la même structure que la méthode `read`, en transformant les accès en lecture en accès en écriture :

```
(define-method write! ((self <Common>) address value)
  (if (zero? address)
    (if (not (null? (out self)))
```

```
      (begin
        (set! (currentOut self) (cast self value))
        (write (currentOut self) (out self))))
    (next-method)))
```

5.3.6 Modélisation objet de la mémoire

Une mémoire est un tableau de cellules réalisé sous la forme d'un vecteur de SCHEME. La classe `<Memory>` hérite de la classe `<Connectable>` pour autoriser la connexion au bus, et de la classe `<Castable>` car les valeurs sont codées en un nombre limité de bits :

```
(define-class <Memory> (<Connectable> <Castable>)
  ((array :accessor array)))
```

La classe `<Ram>` et la classe `Rom` héritent simplement de la classe `<Memory>` :

```
(define-class <Ram> (<Memory>) ())
(define-class <Rom> (<Memory>) ())
```

La méthode `initialize` de la classe `<Memory>` crée le vecteur de cellule, dont la taille est lue dans les arguments de la fonction. Le nombre de bits de la mémoire est par défaut de seize bits :

```
(define-method initialize ((self <Memory>) arguments)
  (next-method)
  (set! (array self)
        (make-vector (get-keyword :size arguments 0) 0)))
```

La taille de la mémoire est la taille du vecteur `array` :

```
(define-method size ((self <Memory>))
  (vector-length (array self)))
```

La fonction `read` accède au vecteur `array` à l'aide de l'argument `address` en index. Si l'adresse est trop grande, la valeur `0` est retournée :

```
(define-method read ((self <Memory>) address)
  (if (< address (size self))
      (vector-ref (array self) address)
      0))
```

La méthode `write!` écrit une valeur dans le vecteur, si l'adresse est correcte. La valeur est codée en un nombre limité de bits, à l'aide de la fonction `cast` :

```
(define-methode write! ((self <Memory>) address value)
  (if (< address (size self))
    (vector-set! (array self) address (cast self value))))
```

Bien que l'écriture dans une mémoire morte semble peu naturelle, elle est utile, notamment avec l'interface graphique, en permettent de modifier un programme de manière dynamique :

```
(define-method write! ((self <Rom>) address value)
  (if (< address (size self))
    (vector-set! (array self) address value)))
```

La méthode `dump` permet de copier une liste d'instructions à partir du début de la `rom` d'un contrôleur :

```
(define-method dump ((self <Rom>) list)
  (let loop ((ad 0)(l list))
    (if (not (null? l))
      (begin
        (write! self ad (car l))
        (loop (+ ad 1) (cdr l))))))
```

5.3.7 Modélisation des objets connectables

Un objet de la classe `<Connectable>` peut être connecté au bus de l'architecture. Il supporte donc les méthodes `reset` et `clock` en plus de la méthode standard `initialize`. La classe est :

```
(define-class <Connectable> ()
  ((bus :accessor bus)))
```

La méthode d'initialisation consulte les arguments à la recherche de l'option `:bus`. Sans cette option, une instance de la classe `<NulBus>` est connectée :

```
(define-method initialize ((self <Connectable>) arguments)
  (next-method)
  (let ((bs (get-keyword :bus arguments (make <NullBus>))))
    (set! (bus self) bs)
    (connect bs self)))
```

La fonction `reset` est invoquée par l'utilisateur. Elle n'a aucun effet direct sur les instances de cette classe et elle est définie pour des raisons de compatibilités :

```
(define-method reset ((self <Connectable>))
  'nothing)
```

La méthode `clock` est aussi définie pour des raisons de compatibilités :

```
(define-method clock ((self <Connectable>))
  'nothing)
```

5.3.8 Modélisation des objets castables

Les objets de la classe `<Castable>` sont associés à un nombre de bits pour coder les entiers des objets qui en héritent. Pour simplifier le programme, la classe contient les limites supérieures et inférieures. Ces limites seront utilisées pour transformer, le cas échéant, les valeurs écrites. La classe est définie par :

```
(define-class <Castable> ()
  ((width :accessor width :initform 16 init-keyword :width)
   (max   :accessor maximum)
   (min   :accessor minimum)))
```

Les limites supérieures et inférieures sont calculées à la création des objets :

```
(define-method initialize ((self <Castable>) arguments)
  (next-method)
  (set! (maximum self)    (expt 2 (width self)))
  (set! (minimum self) (- (expt 2 (width self)))))
```

Enfin, la méthode `cast` transforme un entier en fonction du nombre de bits maximal pour le coder :

```
(define-method cast ((cast <Castable>) value)
  (if   (< value (minimum self)) (minimum self)
    (if (> value (maximum self)) (maximum self)
       value))
```

5.4 Interface graphique

Dans cette section, nous présentons l'interface graphique du simulateur qui est réalisée par les objets dont les noms commencent par `V-` dans le modèle objet. Pour rester simple, le code de cette interface n'est pas présenté. Seuls sont présentés les différents composants de cette interface.

Comme nous pouvons le voir dans la figure 5.4, `cont.3` est une instance de la classe `<VisualController>`. C'est une représentation graphique des registres, de la pile (classe

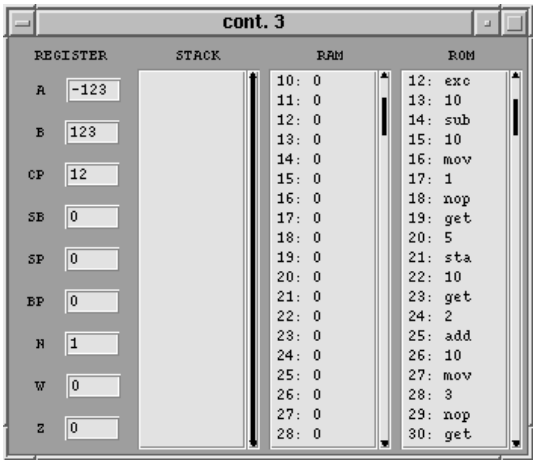


FIG. 5.4 - *Représentation graphique d'un contrôleur.*

`VisualStack`), de la RAM (classe `<VisualRam>`) et de la ROM (classe `<VisualRom>`). Les contenus des différentes composantes peuvent être modifiés dynamiquement.

Les ressources partagées sont aussi représentées graphiquement, comme le montre la figure 5.5. Nous remarquons que la mémoire globale commence à l'adresse `1`, car l'adresse `0` est réservée pour les opérations d'entrée/sortie. L'objet graphique affiche aussi les valeurs précédemment lues ou écrites à l'adresse `0`.

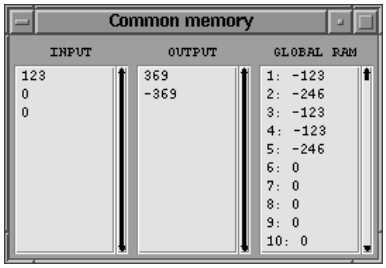


FIG. 5.5 - *Représentation graphique des ressources partagées.*

Le panneau de contrôle `Control panel` est la représentation graphique du bus principal, en figure 5.6. Pour chaque contrôleur, les accès aux ressources partagées sont affichés. L'interface permet l'exécution pas à pas ou en animation. Les points d'arrêts peuvent être placés dans le code de chaque contrôleur.

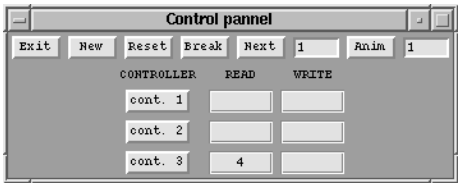


FIG. 5.6 - *Interface graphique du bus principal.*

5.5 Conclusion

Dans cette section, le simulateur de l'architecture parallèle statique est spécifié à l'aide d'un formalisme objet. L'efficacité d'une modélisation objet permet de décrire totalement une application réputée complexe, le simulateur, en quelques pages.

Cette spécification montre la faisabilité de l'architecture et pourra servir de point de départ à la réalisation d'un simulateur plus complexe, et à sa réalisation matérielle.

Cette spécification commence par la description de l'architecture, et notamment l'assembleur des contrôleurs. A partir de cette spécification, nous avons établi le modèle objet qui montre les relations entre les différents objets de l'application. Ce modèle sépare clairement les aspects graphiques de l'interface du simulateur proprement dit. De ce modèle, nous en déduisons le graphe d'héritage des classes d'objets qui décrivent la structure de l'application.

Nous abordons ensuite la spécification des méthodes de l'application en décrivant les classes et les méthodes de chaque objet. Cette étude aboutit naturellement à la réalisation concrète d'un simulateur en état de marche. Cependant, l'interface graphique n'est pas décrite : on en montre seulement les écrans.

Chapitre 6

Compilateur parallèle : λ-spaC

Ce chapitre décrit le compilateur parallèle λ-SPAC, qui produit du code parallèle pour l'architecture parallèle statique. Il traite les fichiers fourni par le compilateur λ-FLOW, en mode `spa`. Dans ce mode, λ-FLOW produit un code intermédiaire qui organise une application en itération, composée de tâches élémentaires. λ-SPAC répartit ces tâches élémentaires sur les contrôleurs de l'architecture.

Dans les premières sections, nous examinerons la structure des tâches produites par λ-FLOW-SPA (§ 6.1), puis les différents types de tâches possibles (§ 6.2), et enfin la manière d'exprimer leurs dépendances fonctionnelles (§ 6.3). Le générateur d'instructions sera spécifié (§ 6.4), et utilisé dans le répartiteur d'instructions (§ 6.5). Nous décrirons ensuite les différentes optimisations possibles du compilateur (§ 6.6). L'exemple du filtre numérique sera alors traité par le compilateur (§ 6.7), ce qui nous conduira à étudier les performances de l'algorithme de répartition (§ 6.8).

6.1 Structure d'une tâche

Une tâche est une liste constituée d'une sortie, de la liste des entrées, et de l'opération. Les entrées / sorties de tâches sont des variables, ou des constantes entières. Les variables seront remplacées par la suite par des adresses dans la mémoire commune. Par exemple, la tâche ;

```
@x := @a + @b
```

sera codée :

```
'(@x (@a @b) add)
```

La structure et la syntaxe des listes sont celles du langage SCHEME[1]. Quels sont les types de tâches produites par λ-FLOW-SPA ?

6.2 Types de tâches

Le code produit par λ-FLOW peut être décrit à l'aide d'un petit nombre de types de tâches. Nous avons l'alternative `@x := if @c then @t else @e` sera codée `'(@x (@c @t @e) alt)`.

Nous avons aussi l'évaluation de la condition d'arrêt `if c then exit`. Cette tâche ne possède pas de sortie, et elle sera codée `'(#f (@c) exit)`.

Les deux formats suivants sont liés aux entrées/sorties principales du programme. Elles sont codées `'(@x (@a) input)` et `'(@x (@a @b) output)`, où `@a` est le numéro du port et `@b` la variable à écrire.

1. Le compilateur λ-SPAC etant écrit en SCHEME, l'usage de ce langage dans le fichier produit facilite sa relecture.

Nous avons aussi l'affectation d'une constante à une variable `@x := cte` qui se traduit `'(@x () cte)`.

Les autres tâches sont directement liées à l'algèbre des entiers de λ-FLOW. Par exemple la tâche `@x := @a + b` est codée `'(@x (@a @b) add)`.

6.3 Dépendance fonctionnelle des tâches

Une application compilée par λ-FLOW-SPA est structurée en sections décrites dans la section (§ 4). Nous avons les sections `start`, `init`, `loop` et `next`. La section `stop` qui évalue la condition d'arrêt est jointe à la section `loop`, pour des raisons d'efficacité. De plus, ces sections sont précédées d'une section de déclaration des variables.

Dans le code produit, ces sections sont définies comme des listes SCHEME tâches. Par exemple, si la section `init` est constituée des tâches `@x := @a + @b` et `@y = @a * @c`, nous aurions :

```
(set! *init* '((@x (@a @b) add)
               (@y (@a @c) mul)))
```

Ce code est lu par un interprète SCHEME, utilisé par le compilateur. L'usage de SCHEME permet une programmation extrêmement compacte.

Le code produit par λ-FLOW-SPA a donc la forme :

```
(set! *vars*  '(...))
(set! *start* '(...))
(set! *init*  '(...))
(set! *loop*  '(...))
(set! *next*  '(...))
```

où ... sont des listes de tâches, ou une liste de variables, dans le premier cas.

La première sections `*vars*` contient la liste de variables de l'application. Les sections sont ordonnées, c'est à dire que la section `start` précède toutes les autres sections. De plus, la liste des tâches de chaque section est aussi ordonnée en fonction des dépendances fonctionnelles. Cela signifie que la première tâche de la liste doit être exécutée avant toutes les tâches suivantes de la liste.

L'apport de λ-FLOW est de permettre cet ordonnancement complet des tâches qui composent une application.

6.4 Générateur d'instructions

Nous avons vu qu'une tâche est une liste formée de sa sortie, de ses entrées et du type de la tâche. Chaque tâche produit donc un certain nombre d'instructions. Par exemple, la tâche `'(@x (@a @b) add)` produit la liste d'instructions :

```
'((get @a)       ; A  <-  @a
  (exc B)        ; A  <-> B
  (get @b)       ; A  <-  @b
  (exc B)        ; A  <-> B
  (add B)        ; A  <-  A + B
  (mov @x))      ; @x <-  A
```

Il faut noter que le compilateur doit tenir compte du fait que les entrées peuvent être des constantes entières. Par exemple, la tâche `'(@x (1 @b) add)` produit la liste d'instructions :

```
'((lda 1)        ; A  <-  1
  (exc B)        ; A  <-> B
  (get @b)       ; A  <-  @b
  (exc B)        ; A  <-> B
  (add B)        ; A  <-  A + B
  (mov @x))      ; @x <-  A
```

Voici la fonction qui retourne soit `get @a` si l'entrée est un symbole, soit `lda 1`, si c'est une constante entière :

```
(define (get-input input)
  (if (number? input)
     `(lda ,input)
     `(get ,input)))
```

La quasi-quote est utilisée ici comme constructeur des expressions (§ I-4.5.2). Elle simplifie beaucoup les expressions.

La fonction suivante génère les instructions pour les commandes simples à une seule entrée :

```
(define (generate-1 input output opcode)
  (cond
    ((eq? opcode 'change-sign)
     `(,(get-input input)
       (neg A)
       (mov ,output)))
    ((eq? opcode 'zero)
     `(,(get-input inputs)
       (lda Z)
       (mov ,output)))))
```

La fonction retourne une liste de listes de symboles quotés. Chacune des listes de symboles est une instruction. La fonction suivante génère les instructions d'une tâche à deux entrées :

```
(define (generate-2 input-1 input2 output opcode)
   `(,(get-input (cadr inputs)
     (exc B)
     ,(get-input (car inputs))
     (exc B)
     (,opcode B)
     (mov ,output))))
```

La fonction suivante génère les instructions pour une alternative. La particularité du codage est qu'il doit être déterministe en temps, ce qui signifie que les deux branches de l'alternative sont de durées identiques. Rappelons qu'une alternative a la forme `if @c then @t else @e`, où `@c`, `@t` et `@e` sont soit des variables soit des constantes. La fonction est :

```
(define (generate-alt condition then else output)
  `(,(get-input condition)
    (jz 5)
    ,(get-input then)
    (jmp 5)
    ,(get-input else)
    (nop)
    (mov ,output)))
```

L'instruction `nop` permet d'équilibrer les durées des deux branches. Le compilateur parallèle définit d'autres fonctions de générations, pour les entrées/sorties et pour les comparateurs. La fonction principale de la génération des instructions est :

```
(define (generate task)
  (let ((output (car task))
        (inputs (cadr task))
        (opcode (caddr task)))
  (case type
    ((change-sign zero)              ; unary operators
     (generate-1 (car inputs)
                 output
                 opcode))

    ((add sub and or xor mod shr shl) ; binary operators
     (generate-2 (car inputs)
                 (cadr inputs)
                 output
                 opcode))

    ((alt)                           ; alternative
     (generate-alt (car inputs)
                   (cadr inputs)
                   (caddr inputs)
```

```
              output))

((input)                          ; input
 (generate-in (car inputs)
              output))

((output)                         ; output
 (generate-out (car inputs)
               (cadr inputs)
               output))


((equal different upper lower     ; Comparators
  upper-equal lower-equal)
 (generate-cmp (car inputs)
               (cadr inputs)
               output
               opcode))

(else                             ; simple assignement
  (list (get-input type)
        `(mov ,output))))
```

6.5 Algorithme de répartition

Dans l'algorithme décrit dans cette section, un contrôleur est une paire formée de la liste des instructions à exécuter et du fichier de sortie.

L'algorithme de répartition est basé sur la notion d'instruction. Chaque processeur se voit affecter une liste d'instructions issue de la liste des tâches restantes. Cette liste est initialement la liste des tâches d'une section.

La liste `todo` des instructions à exécuter à chaque instant est remplie avec les instructions de chaque processeur, en prenant en compte les accès au bus. A chaque étape, un seul accès au bus est autorisé (`access`). Si une instruction accède au bus en écriture, la variable est placée dans la liste des variables disponibles (`available`). Si une instruction accède au bus en lecture, la variable accédée doit être dans la liste des variables disponibles.

Si toutes les instructions de la liste des instructions à traiter sont `#f`, la compilation est terminée, et la liste des variables disponibles est retournée. Sinon, chaque instruction est écrite dans le fichier associé à chaque processeur. Si l'instruction est `#f`, alors `nop` est écrit.

La fonction SCHEME réalisant cette répartition est :

```
(define (_compile conts tasks available)
  (let ((todo '()))                       ; opcode to do

    ; for each controller, adds its current opcode to the todo list
    ; if there is no access conflict, or adds false
    (for-each
     (lambda (cont)
       (let ((opcodes (car cont))         ; opcode list of the controller
             (access  #f))                ; true is the bus is accessed

         ; Assign a task to each free controller
         (if (and (not (null? tasks))
                  (null? opcodes))
             (begin
              ; the task to assign is the head of the tasks list
              (set! opcodes (generate (car tasks)))
              (set-car! cont opcodes)
              (set! tasks (cdr tasks))))

         (if (not (null? opcodes))
             ; share the bus accesses
             (let* (; current opcode : '(opcode arg)
                    (opcode (car opcodes))
                    ; opcode name
                    (name   (car opcode))
                    ; accessed variable, if it exists
                    (var    (cadr opcode)))

               (cond ((eq? name 'mov)     ; write access
                      (if (not access)
```

```
                        (begin
                         ; disable other access, adds the opcode in the todo list
                         ; and adds the write variable in the available list
                         (set! access #t)
                         (set! todo (cons opcode todo))
                         (set! available (cons var available)))
                       ; Nothing to do
                        (set! todo (cons #f todo))))

                     ((eq? name 'get)    ; read access
                      (if (and (not access)
                               ; is the read variable in the available lists ?
                               (member var available))
                          (begin
                           (set! access #t)
                           (set! todo (cons opcode todo)))
                        (set! todo (cons #f todo))))

                     (else               ; other opcode
                      ; adds the opcode in the todo list
                      (set! todo (cons opcode todo))))))))
  conts)
 ; is the todo list empty ?
 (if (apply or todo)
     (begin
      ; for each cont and opcode, write the opcode or nop
      (for-each
       (lambda (opcode cont)
         (display "        " (cdr cont))
         (if (opcode)
             (begin
              (for-each (lambda (inst)
                          (display inst) (cdr cont)
                          (display " ")  (cdr cont))
                        opcode)
              (set-car! cont (cddr cont)))
           (display "nop" (cdr cont)))
         (newline (cdr cont)))
       (reverse todo) conts)

      ; compile the next step
      (_compile cont task available))

    ; else, returns the available variables
    available)))
```

Cet algorithme est un modèle qui souffre de certaines limitations. La plus importante est que le premier processeur de la liste des processeurs accède toujours au bus en premier.

L'algorithme réel apporte un certain nombre d'optimisations, décrites dans la section suivante.

La fonction qui réalise la compilation de toutes les sections est la suivante :

```
(define (compile number-of-controllers spa-file)
  (let (; list of controllers
        (conts (let loop ((cnt 0) (conts '()))
                 (if (eq? cnt number-of-controllers)
                     (reverse conts)
                   (loop (+ 1 cnt)
                         ; A controller is a pair of an opcode list and an output file
                         (cons (cons '() (open-output-port
                                          (string-append "a."
                                                         (number->string cnt)
                                                         ".S")))))))))

        ; list of available variables
        (available '())


        ; Print a message in all the controllers output file
        (print (lambda (message)
                 (for-each (lambda (cont)
                             (display message (cdr cont))
                             (newline (cdr cont)))
                           cont)))

        ; declares the variables for all the controllers
        (declare (lambda (lst)
                   (let loop ((lst lst)(cnt 10))
```

```
                    (if (not (null? lst))
                        (begin
                         (for-each
                          (lambda (cont)
                            (display (string-append "        equ "
                                                    (symbol->string (car lst))
                                                    " "
                                                    (number-string cnt))
                                     (cdr cont)))
                          conts)
                         (loop (cdr lst) (+ 1 cnt)))))))

      ; compiles a labeled list of tasks and sets the avilable inputs list
       (do-compile (lambda (tasks label)
     (print label)
     (if (not (null? tasks))
 (set! available (_compile conts lst available))))))

  ; loads the SPA file
  (load spa-file)

  ; declares the variables
  (print "vars:")
  (declare  *vars*)

  ; compiles all the sections
  (do-compile *start* "start:")
  (do-compile *init*  "init:")
  (do-compile *loop*  "loop:")
  (do-compile *next*  "next:")

  ; Printd the last jump
  (print "       jmp loop")

  ; Closes all the controller output file
  (for-each (lambda (cont) (close-output-port (cdr cont))) cont)))
```

La fonction construit la liste des contrôleurs. Un contrôleur est une paire formée d'une liste d'instructions initialement vide et d'un fichier de sortie.

Puis la fonction charge le fichier des sections produit par λ-FLOW-SPA décrit dans une section ultérieure. Cette lecture par `load` a pour effet de définir les variables globales `*vars*`, `*start*`, `*init*`, `*loop*` et `*next*`. Ces variables contiennent la liste des tâches de chaque section.

Le compilateur déclare alors les variables de l'application. L'adresse de début est `10`, les dix premières adresses étant réservées pour les entrées/ sorties.

Puis le compilateur compile toutes les sections, et termine en écrivant l'instruction de saut à la section `loop`. Il peut alors fermer les fichiers de sortie des contrôleurs.

6.6 Optimisation du compilateur

- **Répartition de la priorité d'accès :** la répartition des accès est soumise à un niveau de priorité qui répartit le droit d'accès à tous les processeurs.

- **Accès simultané en lecture sur la même variable :** l'accès au bus en lecture sur la même variable est autorisé.

- **Permutation des entrées :** lorsqu'une tâche possède plusieurs entrées, leur lecture peut être permutée en fonction de leur disponibilité. Ainsi, si les entrées `a` et `b` de la tâche `'(x (a b) add` sont respectivement indisponible et disponible, la tâche est transformée en `'(x (b a) add`. L'algorithme tient compte de la commutativité des opérateurs.

- **Prise en compte des registres :** le contenu des registres `A` et `B` des contrôleurs est pris en compte pour limiter, lorsque cela est possible, les accès au bus. Cette optimisation pourrait être étendue à l'ensemble de la mémoire locale des contrôleurs. Cette optimisation permettrait une diminution très importante des accès au bus.

– **instruction multi-cycles :** le générateur d'instruction permet de prendre en compte les instructions prenant plusieurs cycles d'horloge pour être totalement exécutées.

L'ensemble de ces optimisations rendent l'algorithme plus complexe. Le gain en performance des programmes se situe aux alentours de 7%.

6.7 Exemple

Dans cet exemple, nous complétons l'exemple traité tout au long de la thèse et introduit dans la première partie (§ II-2.3). Le module λ-FLOW correspondant situé dans le fichier `filter.lf` est :

```
filter := lambda i. begin
  o #= a + b + c + d;
  a := i - b;
  b := 0 followed-by d;
  c := a - e;
  d := c + e;
  e := 0 followed-by c;
end;
```

et le programme principal `main.lf` :

```
main := lambda i:int. begin
 out #= filter(i) extract o;
end;
```

Pour compiler en mode `SPA` ce programme, nous entrons la commande suivante :

```
> flow --target spa -k filter.lf main.lf
message: main parameter 'i' is assigned to port '1'
message: main output 'out' is assigned to port '1' and name 'out_1'
warning: no specified post-compiler in target 'spa'

Hum... So, we have... 0 error and 1 warning.
time=2s, heapsize=196608, gc calls=1
```

Cette commande produit le fichier `a.spa` suivant :

```
(set! *vars * '(@d @e @i @out1 @out2 @out3 @out @a @b @c))
(set! *start* '())
(set! *init*  '((@i    (1)        input)
                (@b    ()         0)
                (@e    ()         0)))
(set! *loop*  '((@a    (@i @b)    sub)
                (@c    (@a @e)    sub)
                (@d    (@c @e)    add)
                (@out1 (@c @d)    add)
                (@out2 (@b @out1) add)
                (@out3 (@a @out2) add)
                (@out  (@1 @out3) output)))
(set! *next*  '((@i (1)           input)
                (@b ()            d)
                (@e ()            c)))
```

Ce fichier est la liste des tâches de l'application. Pour les répartir sur une architecture à trois contrôleurs, par exemple, nous utilisons la commande :

```
> spac -r -p -d -n 3 a.spa
```

où l'option `-r` autorise l'optimisation par registre, `-p` l'inversion des entrées et `-d` la lecture simultanée des mêmes variables d'entrées, `-n 3` indique le nombre de contrôleurs et `a.spa` est le fichier des tâches. Cette commande produit la sortie :

```
compilation of       : a.spa
register optimization: yes
inverter optimization: yes
read  optimization   : yes
tasks           start: 0
```

```
                init: 3
                loop: 7
                next: 3
               --------------
                 sum: 13
variables            : 10
instructions         : 37
sum of instructions  : 37x3=111
1-controller instruc.: 45
register optimization: 1
inverter optimization: 3
read optimization    : 0
optimization ratio   : 18%
 |  cont |  task |  read | write |  nop  |
 |-------|-------|-------|-------|-------|
 |     1 |     4 |     6 |     4 |    22 |
 |     2 |     5 |     6 |     5 |    19 |
 |     3 |     4 |     4 |     4 |    23 |
 |-------|-------|-------|-------|-------|
 |   sum |    13 |    16 |    13 |    64 |
-done
Output file are from spa-1.S to spa-3.S
```

et trois fichiers sources `spa-1.S`, `spa-2.S` et `spa-3.S`. Pour visualiser plus commodément, nous utilisons la commande ;

```
> mergespa -l spa-1.S spa-2.S spa-3.S -w 25 -s "| "
```

où `-l` autorise l'affichage des numéros de lignes, `-w 25` donne la largeur en caractères de chaque fichier, et `-s "| "` indique que `|` sera le séparateur placé entre chaque fichier. La sortie de cette commande est :

```
 1|vars:              |vars:              |vars:
 2|       equ @d 10   |       equ @d 10   |       equ @d 10
 3|       equ @e 11   |       equ @e 11   |       equ @e 11
 4|       equ @i 12   |       equ @i 12   |       equ @i 12
 5|       equ @out1 13|       equ @out1 13|       equ @out1 13
 6|       equ @out2 14|       equ @out2 14|       equ @out2 14
 7|       equ @out3 15|       equ @out3 15|       equ @out3 15
 8|       equ @out 16 |       equ @out 16 |       equ @out 16
 9|       equ @a 17   |       equ @a 17   |       equ @a 17
10|       equ @b 18   |       equ @b 18   |       equ @b 18
11|       equ @c 19   |       equ @c 19   |       equ @c 19
12|start:             |start:             |start:
13|init:              |init:              |init:
14|@i:    get 1       |@b:    lda 0       |@e:    lda 0
15|       nop         |       mov @b      |       nop
16|       nop         |       nop         |       mov @e
17|       mov @i      |       nop         |       nop
18|loop:              |loop:              |loop:
19|@d:    nop         |@a:    get @i      |@c:    exc b
20|       get @e      |       exc b       |       nop
21|       exc b       |       get @b      |       nop
22|       nop         |       exc b       |       nop
23|       nop         |       sub b       |       nop
24|       nop         |       mov @a      |       nop
25|       nop         |@out1: nop         |       get @a
26|       nop         |       nop         |       sub b
27|       nop         |       nop         |       mov @c
28|       nop         |       nop         |@out2: get @b
29|       get @c      |       nop         |       exc b
30|       add b       |       nop         |       nop
31|       mov @d      |       nop         |       nop
32|@out3: nop         |       get @d      |       nop
33|       get @a      |       exc b       |       nop
34|       exc b       |       get @c      |       nop
35|       nop         |       add b       |       nop
36|       nop         |       mov @out1   |       nop
37|       nop         |@out:  nop         |       get @out1
38|       nop         |       nop         |       add b
39|       nop         |       nop         |       mov @out2
40|       get @out2   |       nop         |       nop
41|       add b       |       nop         |       nop
42|       mov @out3   |       nop         |       nop
43|       nop         |       get @out3   |       nop
44|       nop         |       mov 1       |       nop
45|next:              |next:              |next:
46|@b:    nop         |@e:    nop         |@i:    get 1
```

```
47|       get @d       |       nop          |       nop
48|       nop          |       get @c       |       nop
49|       nop          |       nop          |       mov @i
50|       mov @b       |       nop          |       nop
51|       nop          |       mov @e       |       nop
52|       jmp loop     |       jmp loop     |       jmp loop
53|                    |                    |
```

6.8 Performances

L'efficacité du code produit par λ-SPAC est très simple à évaluer. En effet, un programme est une itération dont le corps est déterministe en temps. L'efficacité d'un programme est dans ce cas inversement proportionnelle au nombre d'instructions de ce corps.

Dans l'exemple de la section précédente, nous constatons que le code produit pour l'exemple du filtre est plus efficace de 18% que le code produit pour un seul processeur.

Ce résultat est décevant.

Mais il faut le tempérer par le fait que l'application traitée est particulièrement défavorable. Tout d'abord, les tâches sont peu nombreuses et fortement séquentielles. De plus, elles sont principalement constituées d'entrées / sorties, pour une seule opération utile.

Le gain réel du modèle de compilation basé sur l'architecture parallèle statique est donné par une étude dont les résultats sont résumés dans la figure 6.1.

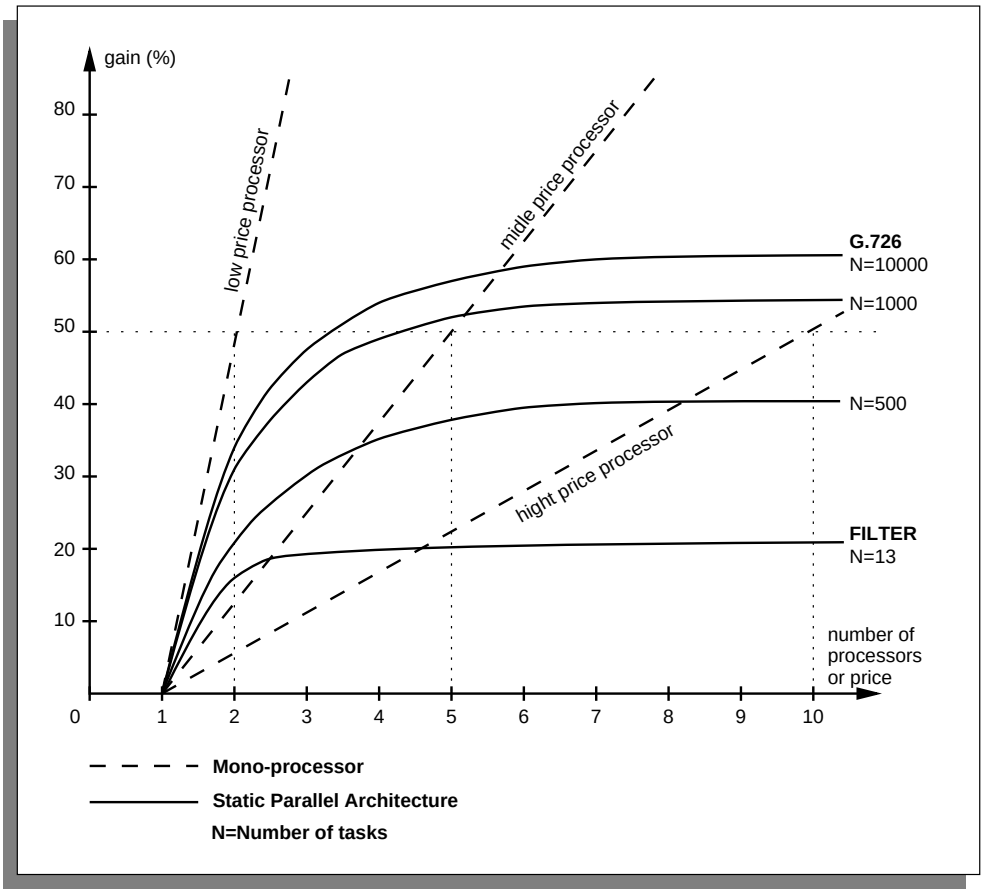


FIG. 6.1 - *Performances du modèle de compilation de l'architecture parallèle statique, en fonction du nombre de tâches de l'application. Les performances sont comparées avec celles d'une architecture mono-processeur classique, à coût équivalent.*

Ces courbes permettent de comparer les performances et les coûts d'architectures parallèles statiques variant par leur nombre de processeurs et d'architectures classique mono-processeurs.

Comment les lire ?

Dans un premier temps, nous nous attacherons aux architectures multi-processeurs, matérialisées par les courbes en traits pleins.

Prenons le cas du filtre étudié tout au long de ce mémoire (courbe FILTER). Ce filtre est caractérisé par le petit nombre d'instructions qu'il comporte (13).

Nous constatons que le passage de un à deux processeurs permet d'obtenir un gain de près de 20% : si nous avions 100 instructions, nous en obtenons maintenant 80. Par contre, nous constatons que le passage de deux à trois processeurs ne fait rien gagner. Le passage de un à deux processeurs occasionne un doublement du coût de l'architecture. En effet, nous pouvons ramener le prix d'une architecture au prix des processeurs qu'elle utilise.

Considérons maintenant une application ayant plus d'instructions, la norme G.726 qui en comporte près de 10000. Là, le passage à deux processeurs permet un gain de 35%, et le passage à trois processeurs ajoute 10% de plus. Nous constatons que le gain n'est pas du tout linéaire et le gain maximum est obtenu pour 7 processeurs.

Oublions pour un instant les architectures parallèles (matérialisées par les courbes en traits pleins) et considérons des architectures classiques mono-processeur (matérialisées pas les courbes en pointillés). Avec ce type d'architectures, l'augmentation des performances est obtenu par un changement de processeur. Nous pouvons classer les processeurs en trois gammes : le bas de gamme, comme le 8051 de INTEL, le milieu de gamme, comme le 8086 du même constructeur et le haut de gamme avec le RISC *i*860.

La question est : pour une gamme de processeur donnée, que va coûter le doublement des performances ? On peut estimer que, dans le bas de gamme, le doublement des performances occasionne un doublement du prix affecté au processeur. Ainsi, dans le bas de gamme, obtenir un gain de 50% coûte deux fois plus cher (2 sur l'axe des ordonnées). Par contre, dans le haut de gamme, le doublement des performances entraîne une multiplication par 10 du prix du processeur. De plus, si on utilise le processeur le plus performant du marché, il n'existe aucun moyen d'augmenter encore les performances.

Maintenant, rapprochons les deux types d'architectures, les architectures mono-processeurs (courbes en pointillés) et les architectures parallèles statiques (courbes en traits pleins). Pour cela, nous allons considérer le filtre simple (FILTER) et des processeurs bas de gamme. Avec l'architecture parallèle statique, le passage à deux processeurs occasionne un gain de 20% et un doublement du coût. Avec l'architecture mono-processeur, prendre un processeur deux fois plus puissant permet un gain de 50%. Il est clair que dans cette configuration, il est préférable d'utiliser une architecture mono-processeur.

Considérons maintenant le même filtre, mais avec des processeurs haut de gamme. Nous avons vu qu'avec l'architecture parallèle statique, le passage à deux processeurs occasionne un gain de 20% et un doublement du prix. Avec l'architecture mono-processeur et avec des processeurs haut de gamme, qu'avons-nous pour deux fois plus cher ? En regardant la courbe, nous constatons que nous avons un processeur qui procure un gain de 5%. Dans ce cas de figure, nous choisirons donc l'architecture parallèle statique avec deux processeurs.

Ce raisonnement peut être tenu pour tout type d'applications différenciées par le nombre d'instructions, et pour toute gamme de processeur.

Ainsi, pour une gamme de processeurs donnée et un type d'application (nombre d'instructions), nous constatons qu'il est préférable d'utiliser l'architecture parallèle statique chaque fois que la courbe en trait plein passe au dessus de la courbe en pointillés.

La figure 6.2 montre l'impact des optimisations du compilateur sur l'évolution des performances.

La courbe ALL de cette figure montre que l'ensemble des optimisations apporte un gain constant de près de 7%, quelque soit le nombre de contrôleurs. Il est intéressant de noter que ce chiffre moyen est obtenu alors que chacune des optimisations n'évolue pas de manière constante.

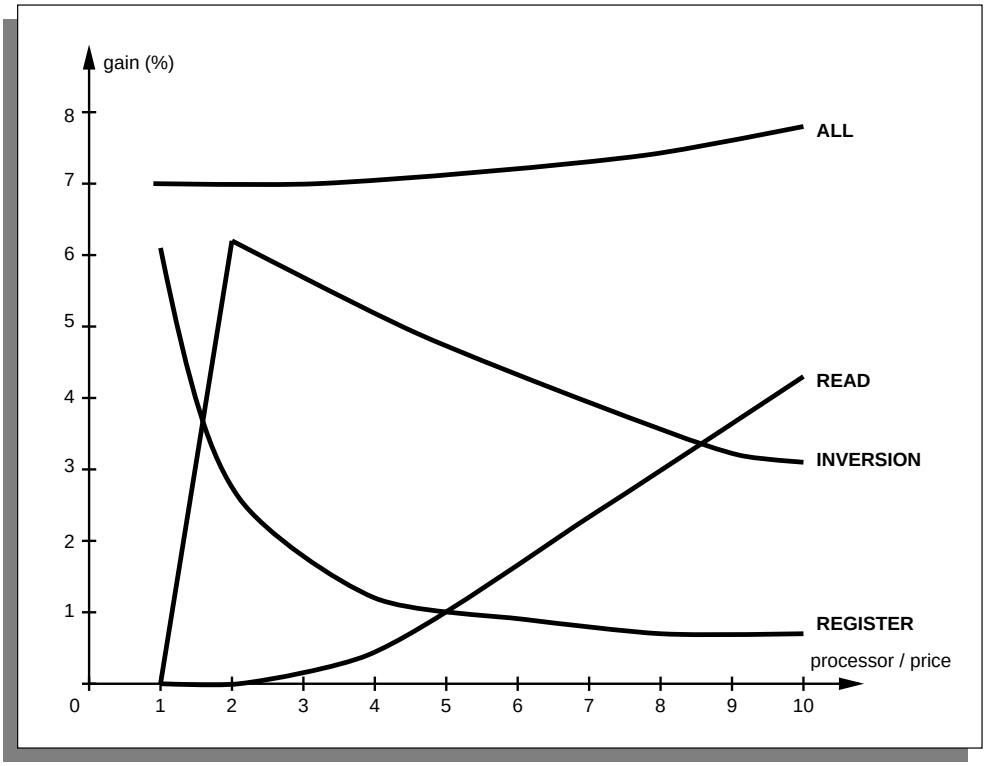


FIG. 6.2 - *Influence des optimisations de λ-SPAC sur les performances moyennes des applications.*

L'optimisation READ autorise plusieurs accès au bus, si l'accès se fait sur la même adresse de la mémoire globale. Cette optimisation ne peut qu'évoluer favorablement avec le nombre de processeurs, en multipliant le nombre d'accès simultanés potentiels.

L'optimisation INVERSION permute l'ordre de l'acquisition des entrées d'une tâche. Avec un seul processeur, cette optimisation ne produit aucune effet. Elle est par contre plus efficace avec un petit nombre de contrôleurs, car peu de contrôleurs exécutent les tâches antérieures à une tâche.

Enfin, l'optimisation REGISTER gère le contenu des registres des processeurs pour éviter les accès au bus. L'efficacité de cette optimisation décroît fortement avec le nombre de contrôleurs. En effet, les tâches sont réparties en fonction de leurs dépendances fonctionnelles. Plus le nombre de contrôleurs augmente, plus la probabilité qu'un même processeur exécute des tâches jointives décroît. Nous nous rendons cependant bien compte qu'en augmentant le nombre de registres, ou en utilisant la mémoire locale des contrôleurs comme un cache de la mémoire globale, nous pourrions accroître les performances de manière importante.

Cette étude montre bien l'intérêt de l'architecture parallèle statique. Le nombre de processeur optimal se situe entre 3 et 5, et les processeurs sont dans le haut de la gamme. Là, les gains en performances peuvent atteindre 60% pour une multiplication entre 3 et 5 du coût.

6.9 Perspectives

Le compilateur λ-SPAC est un prototype montrant qu'il est possible d'exploiter de manière efficace le code produit par λ-FLOW pour l'architecture parallèle statique.

Ce prototype peut être considérablement amélioré, notamment en permettant :

les itérations imbriquées : cette possibilité est facile à réaliser. Elle permettrait de réaliser

des équations de point-fixe dans les programmes, ce qui, bien sur, supprimerait du même coup le déterminisme en temps des applications ;

les appels de fonctions : là, la réalisation est plus complexe. Elle permettrait d'implanter des applications complexes sur l'architecture ;

des processeurs hétérogènes : cette extension semble répondre a une attente de l'industrie. Elle pose des problèmes complexe dans sa réalisation.

Quatrième partie

Application concrète : la norme de compression G726

Chapitre 1

Introduction

Dans cette quatrième partie, nous allons traiter une application réelle issue de l'industrie avec les outils développés durant la thèse.

Dans un premier chapitre, nous décrirons la norme utilisée de manière succincte (§ 2), puis nous donnerons des extraits de la spécifications de certaine parties de cette norme (§ 3). Enfin, nous programmerons la norme à l'aide de λ-GRAPH et de λ-FLOW (§ 4). Puis, nous évaluerons les résultats obtenus (§ 4.4).

Chapitre 2

La norme CCITT G.726

2.1 Préambule

La norme `G.726` permet la compression des signaux sonore, et plus particulièrement de la voix. Typiquement, elle est utilisée dans les téléphones portables. La compression est un moyen simple d'augmenter la capacité de transport des média de communication. Les données peuvent être compressées avec ou sans dégradation. La seconde possibilité demande une puissance de calcul plus importante que la première. Il existe plusieurs algorithmes de compression sans dégradation, comme les algorithmes LEMPEL-ZIP ou UFFMAN. La compression avec dégradation peut souvent être réalisée de manière efficace.

Un signal sonore peut être transmis de manière analogique ou numérique. Les avantages du codage numérique du signal sont multiples. Il permet un traitement informatique, une meilleure immunité au bruit et une meilleure régénération, un encryptage facile et efficace ainsi que l'uniformité dans la transmission des signaux.

Les signaux sonores sont transmis dans un canal de communication par un flot de bits PCM (*Pulse Code Modulation*) à $64\,kb/s$ (kilo bits par seconde). Bien évidemment, il existe un compromis entre la capacité de transport et la dégradation de la qualité.

Une des solutions recommandées pour accomplir une transmission efficace et de bonne qualité est la Modulation Codée par Impulsion Différentielle Adaptative (ADPCM). A $32\,kb/s$, la solution ADPCM permet de doubler la capacité du canal. Cette compression ADPCM est réalisée à l'aide d'un filtre numérique dont l'entrée est le signal sonore analogique et la sortie, le signal compressé numérique.

Le premier traitement que subit le signal analogique est la numérisation. Mais ce premier traitement introduit une erreur de quantification, due à la résolution utilisée, comme le montre la figure 2.1.

Dans la réalité, il convient d'ajouter au signal d'erreur obtenu, le signal d'erreur propre au canal de transmission. Pour un système idéal, la qualité du signal transmis dépend uniquement du processus de numérisation.

Puis le signal subit d'autres traitements avant d'être transmis sur le canal de communication. La figure 2.2 montre une représentation générale d'un canal de communication numérique.

Le signal est d'abord filtré par un filtre anti-repliement, puis il est échantillonné pour obtenir un signal numérique. Enfin, il est codé pour être transmis sur le canal numérique. A la réception, le signal est décodé et converti en signal analogique. Ce signal est interpolé et lissé par un filtre avant d'être restitué.

2.2 Introduction

La transmission des données nécessite toujours des débits plus élevés de par la nature des transmissions. La norme G.726 étudiée succinctement ici se propose de compresser les

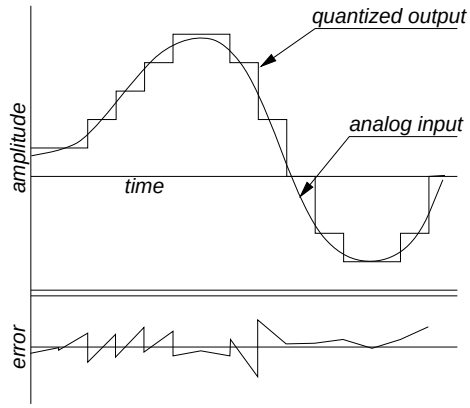


FIG. 2.1 - *Erreur de quantification du signal.*

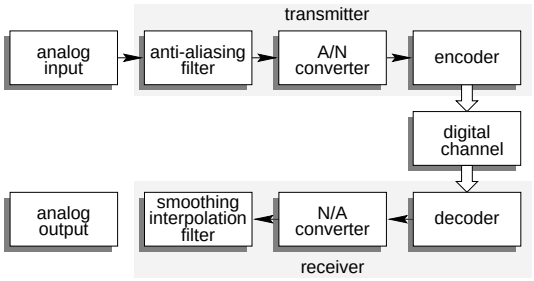


FIG. 2.2 - *Canal de communication numérique.*

données transmises. Cette compression s'accompagne d'une dégradation de qualité dont le niveau est paramétrable. La norme exploite les caractéristiques du type de signal transmis, c'est à dire la parole.

La norme G.726 convertit un signal numérique à $64\,kb/s$ en un signal à 16, 24, 32 ou $40\,kb/s$ et inversement (encodeur et décodeur). Dans le cas de l'encodeur, l'entrée est un signal modulé par impulsion codée (*MIC ou* PCM*: Pulse Code Modulation*) suivant la loi A ou μ. Ces deux lois permettent de tenir compte des caractéristiques des pays utilisateurs. La figure 2.3 montre le principe d'un codage PCM.

La transmission directe d'un signal PCM de débit $64\,kb/s$ demande un canal à haute largeur de bande. De plus il est montré qu'il est possible de coder la parole à des débits considérablement inférieurs à $64\,kb/s$ et peut atteindre $2\,kb/s$, tout en maintenant une fidélité de reproduction acceptable. Le prix à payer est l'augmentation de la complexité du système.

Le codage à faible débit exploite les caractéristiques statistiques du signal. Il élimine autant que possible la redondance du signal et n'utilise que le nombre de bits nécessaires pour coder les parties non redondantes. Par exemple, le signal parole présente une grande corrélation entre deux échantillons adjacents.

En tirant avantage de ces propriétés pour les signaux de parole, des techniques de codage plus efficaces ont été conçues pour réduire le nombre de bits transmis, tout en préservant une qualité globale du signal.

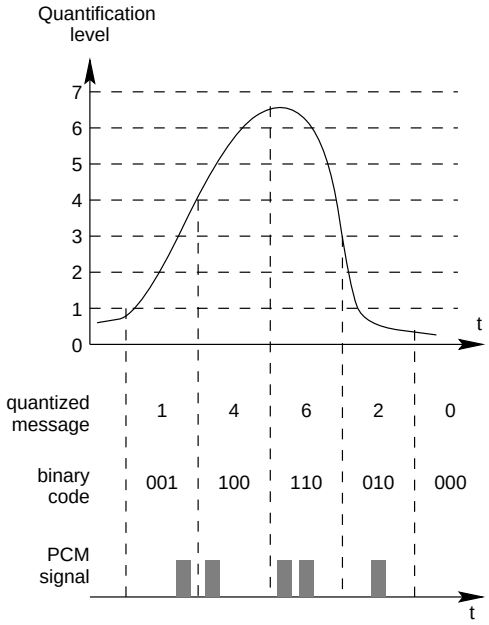


FIG. 2.3 - *Modulation* PCM.

2.3 Présentation générale de la norme G.726

La norme G.726 réduit de 8 à 5, 4, 3 ou 2 le nombre de bits par échantillon. Elle utilise un transcodage ADPCM (*Adaptative Differential Pulse Code Modulation*) qui combine les caractéristiques des systèmes APCM (*Adaptative* PCM) et DPCM (*Differential* PCM).

2.3.1 APCM

L'APCM est une méthode qui peut s'appliquer à des quantificateurs uniformes ou non uniformes et qui adapte la quantification du codeur avec les variations du signal. Les variations d'amplitude d'un signal de parole sont accomodées pour différents interlocuteurs. Par exemple, l'adaptation peut être instantanée ou adaptative. Dans le second cas, le codeur s'adapte aux évolutions lentes du signal.

Le principe de base pour un système adaptatif rétroactif APCM est montré par la figure 2.4. Dans les deux cas (*transmitter*, *receiver*), le signal $I(k)$ est traité par le module *stepsize adapter* pour créer un signal $q(k)$, qui adapte la taille du pas du quantificateur ou du quantificateur inverse.

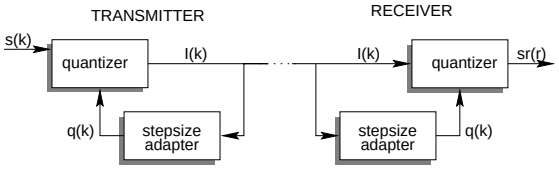


FIG. 2.4 - *Schéma bloc d'un module* APCM.

2.3.2 DPCM

La méthode utilisant la redondance des échantillons d'un signal est appelée *Diffential* PCM. La différence entre deux échantillons adjacents, ou signal d'erreur, produit un signal avec une dynamique beaucoup plus faible. De même, une variance plus faible peut être attendue entre des échantillons de signal d'erreur. Ainsi, un signal avec une plus petite dynamique peut être quantifié avec moins de bits pour un même rapport signal sur bruit. La figure 2.5 montre le système DPCM.

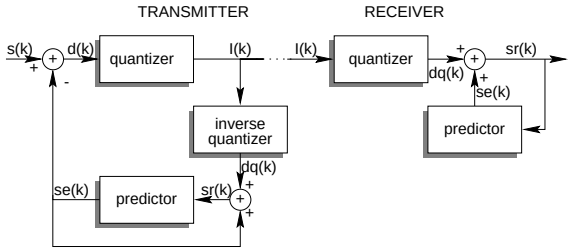


FIG. 2.5 - *Schéma bloc d'un module* DPCM.

Le signal d'erreur $d(k)$ est déterminé en utilisant un signal estimé $se(k)$ plutôt que le signal précédent. En utilisant un signal estimé $se(k)$, le transmetteur utilise la même information disponible par le receveur. Un quantificateur inverse est utilisé à la fois par le transmetteur et le receveur, pour déterminer le signal d'erreur quantifié $dq(k)$, à partir du signal transmis $I(k)$.

2.3.3 ADPCM

La figure 2.6 montre les blocs de base combinant les caractéristiques adaptatives et différentielles dans un système ADPCM.

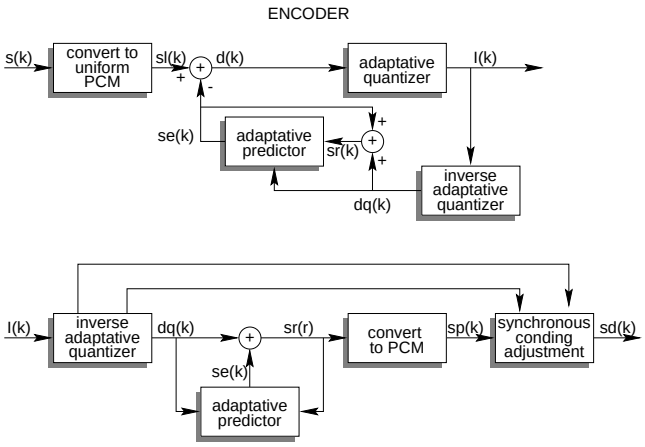


FIG. 2.6 - *Schéma bloc d'un module* ADPCM.

La quantification adaptative et le calcul du signal d'erreur demande le stockage en mémoire de plusieurs échantillons à la fois pour l'encodeur et le récepteur. De plus, l'encodeur doit s'assurer que le décodeur fonctionne de manière synchrone. Ceci est accompli en utilisant uniquement le signal transmis $I(k)$ pour déterminer l'adaptation de la taille du pas pour le quantificateur et le quantificateur inverse, et pour prédire le prochain signal estimé. De cette manière, les blocs du décodeur peuvent être identiques à ceux de l'encodeur.

Le système ADPCM utilisé dans la norme G.726 n'est pas un système de codage du signal d'origine, mais un transcodeur convertissant les codes PCM (en fait log_2 PCM) et ADPCM. Le système ADPCM incorpore à la fois un quantificateur adaptatif et un prédicteur adaptatif. Le capacité d'adaptation du quantificateur signifie qu'il s'adapte aux changements de niveau ou de spectre du signal d'entrée.

Les méthodes PCM et ADPCM travaillent toutes deux dans le domaine temporel. Les normes G.721 et G.723 sont antérieures à G.726. Elles ont un fonctionnement identique à G.726 mais ne présentent pas les mêmes caractéristiques et proposent respectivement une compression de 64 à $32\,kb/s$ et de 64 à 16 ou $24\,kb/s$. En fonction du débit sélectionné, le code ADPCM est représenté par 5, 4, 3 ou 2 bits, le bit de poids le plus fort représentant le signe, et le reste l'amplitude.

2.4 L'encodeur de la norme G.726

L'encodeur ADPCM de la figure 2.7 reçoit un signal en *log* PCM à $64\,kb/s$ et le transcode en un signal ADPCM à 16, 24, 32 ou $40\,kb/s$, ce qui représente un codage sûr, respectivement de 2, 3, 4 ou 5 bits.

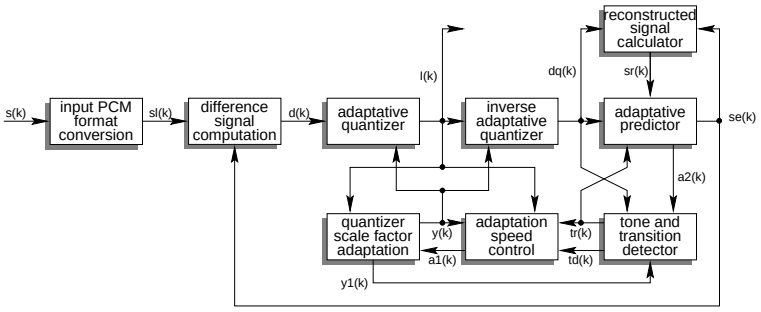


FIG. 2.7 - *Schéma bloc du codeur* ADPCM.

Le transcodage est accompli tout d'abord en convertissant le signal $s(k)$ en *log* PCM en un signal linéaire $sl(k)$, auquel un signal estimé $se(k)$ est soustrait pour obtenir un signal d'erreur $d(k)$.

L'étape suivante est la quantification adaptative de ce signal d'erreur $d(k)$. Le log_2 de $sl(k)$ est normalisé par le facteur d'échelle de quantification (*scale factor*) $y(k)$, et finalement le résultat $I(k)$ est codé.

Le quantificateur adaptatif inverse utilise le même signal $I(k)$, qui a été transmis pour reconstruire la version quantifiée de l'erreur $dq(k)$. Ce dernier est l'entrée du prédicteur adaptatif qui l'utilise pour calculer un signal estimé $se(k)$. Ce signal $se(k)$ est combiné avec le signal d'erreur $dq(k)$ pour déterminer un signal reconstruit $sr(k)$, qui est la sortie du décodeur. Cette sortie est alors soustraite à l'échantillon suivant pour compléter la boucle rétroactive.

2.4.1 Conversion en format PCM

Ce bloc convertit le signal d'entrée $s(k)$ en loi-A ou loi-μ en un signal PCM uniforme $sl(k)$ comme le montre la figure 2.8.

Le signal $s(k)$ en *log* PCM est étendu pour créer une valeur PCM linéaire $sl(k)$. Nous passons en effet d'une représentation 8 bits, en complément à deux signé sur 13 bits si $s(k)$ est en loi A, ou sur 14 bits si $s(k)$ est en loi μ. Le bit de signe est égal à 0 si la valeur est positive ou nulle, et 1 si la valeur est négative. Le décodeur compresse le signal reconstruit $sr(k)$ pour créer le signal en *log* PCM, qui est $sp(k)$.

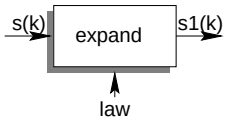


FIG. 2.8 - *Schéma bloc de l'expanseur.*

2.4.2 Différentiel d'erreur

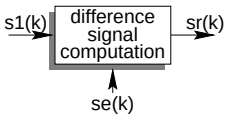


FIG. 2.9 - *Schéma bloc du différentiel d'erreur.*

Comme le montre la figure 2.9, ce bloc calcule le signal d'erreur $d(k)$ en soustrayant au signal $sl(k)$ le signal estimé $se(k)$. Ceci est exprimé par :

$$d(k) = sl(k) - se(k) \tag{2.1}$$

2.4.3 Quantification adaptative

On utilise un quantificateur non uniforme pour mieux s'adapter aux statistiques non stationnaires caractéristiques d'un signal de parole. Deux cas sont distingués.

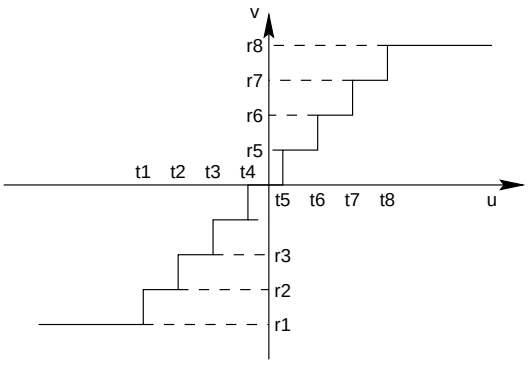


FIG. 2.10 - *Quantificateur uniforme.*

Le cas uniforme où u est l'axe du signal réel, v l'axe du signal quantifié, r_i les niveaux de sorties. Le choix entre deux niveaux possibles se fait par un seuil de décision $t(k)$: si $t(k) < u \leq t(k+1)$ alors $v = r(k+1)$.

La largeur du pas de quantification Δ ou *stepsize* est définie par:

$$\Delta = t(k) - t(k-1) \tag{2.2}$$

Dans le cas d'un quantificateur uniforme, Δ est une constante.

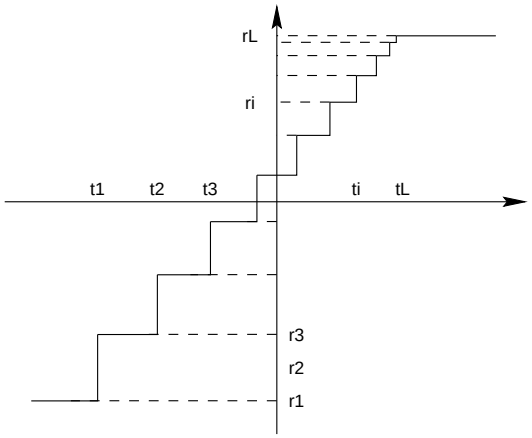


FIG. 2.11 - *Quantificateur non-uniforme.*

Le cas non uniforme La largeur du pas de quantification Δ s'adapte au signal d'entrée u. Ainsi Δ est petit si u a des petites valeurs, et Δ est grand si u a de grandes valeurs.

La figure 2.12 montre le schéma de principe du quantificateur.

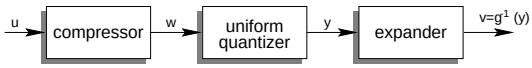


FIG. 2.12 - *Schéma de principe du quantificateur.*

En téléphonie, les lois A et μ contrôlent la fonction g, c'est-à-dire le degré de compression des données. les États Unis et le Japon utilisent la loi μ, l'Europe la loi A.

En adaptant le pas de quantification à l'amplitude du signal, c'est à dire en utilisant des pas de quantification plus larges pour des signaux d'amplitude plus grande, et des pas plus petits pour des signaux d'amplitude plus petite, les bits de données sont utilisés plus efficacement, tout en maintenant une qualité satisfaisante du signal.

Un quantificateur supprime les informations non pertinentes d'un signal. La quantification adaptative est utilisée pour déterminer le facteur d'échelle de l'adaptation (*scale factor*), et le contrôle de la vitesse (*speed control* qui contrôle à quelle vitesse le facteur d'échelle de l'adaptation est adapté.

La norme G.726 propose une quantification à 5, 4, 3 ou 2 bits, contenant le bit de signe, qui est le format du signal $I(k)$ transmis. La figure 2.13 montre le schéma bloc de base du quantificateur adaptatif.

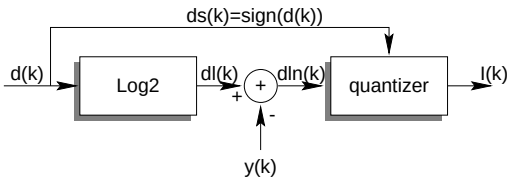


FIG. 2.13 - *Schéma de principe du quantificateur adaptatif.*

Le signal $d(k)$ est normalisé en prenant son logarithme en base 2 et en lui soustrayant le

Entrées normalisées	$\|I(k)\|$	Sorties normalisées
[3.12, $+\infty$]	7	3.32
[2.72, 3.12]	6	2.91
[2.34, 2.72]	5	2.52
[1.91, 2.34]	4	2.13
[1.38, 1.91]	3	1.66
[0.62, 1.38]	2	1.05
[-0.98, 0.62]	1	0.031
[$-\infty$, -0.98]	0	$-\infty$

TAB. 2.1 - *Quantification normalisée pour un débit de* $32\,kb/s$.

facteur d'échelle quantifié $y(k)$.

$$|I(k)| = log_2|d(k)| - y(k) \tag{2.3}$$

Le signe de $I(k)$ est le signe du signal d'erreur $d(k)$.

La table 2.4.3 est utilisée pour fournir l'amplitude du résultat de la quantification $I(k)$ à partir de son entrée normalisée, pour une transmission à $32\,kb/s$.

2.4.4 Adaptation du facteur d'échelle

Ce bloc calcule le facteur d'échelle $y(k)$ nécessaire au quantificateur et au quantificateur inverse. Son rôle est d'adapter le pas de quantification Δ au signal d'entrée $d(k)$. Ainsi, avec un signal de faible amplitude, le facteur d'échelle $y(k)$ adapte le signal à un pas de quantification plus faible (figure 2.14).

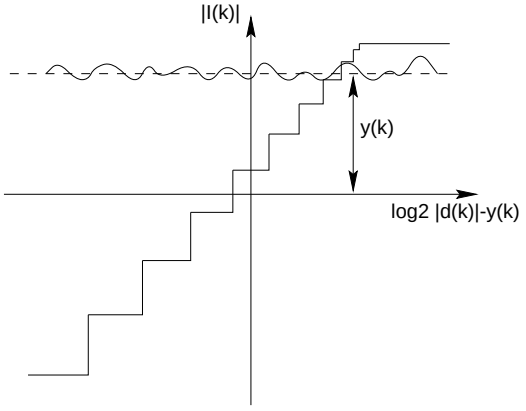


FIG. 2.14 - *Adaptation du pas de quantification à un signal de faible amplitude.*

Il en est de même pour un signal de grande amplitude comme le montre la figure 2.15.

Le principe de base utilisé pour l'échelonnage du quantificateur est une adaptation bimodale qui est soit rapide, pour les signaux qui produisent un signal d'erreur $d(k)$ avec de grandes fluctuations comme la parole, soit lente, pour les signaux qui produisent un signal d'erreur $d(k)$ avec de petites fluctuations.

La vitesse de l'adaptation est contrôlée par une combinaison des facteurs d'échelles rapides, $yu(k)$, et lentes, $yl(k)$.

$$y(k) = a1(k).yu(k-1) + (1 - a1(k)).yl(k-1) \tag{2.4}$$

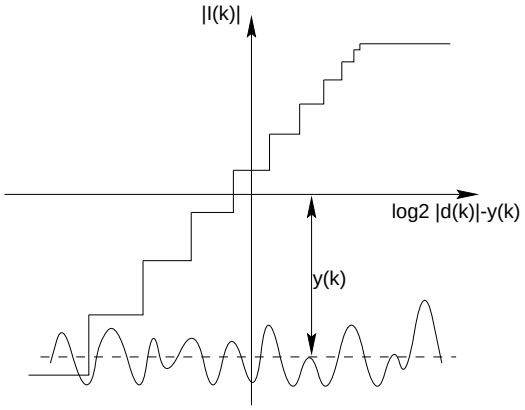


FIG. 2.15 - *Adaptation du pas de quantification à un signal de grande amplitude.*

$\|I(k)\|$	7	6	5	4	3	2	1	0
$W[I(k)]$	70.13	22.19	12.38	7.00	4.00	2.56	1.13	-0.75

TAB. 2.2 - *Fonction discrète* $wi(k)$ *pour un débit de* $32\,kb/s$.

Les deux facteurs $yu(k)$ et $yl(k)$ sont pondérés par le facteur de contrôle de la vitesse $a1(k)$ (*speed control*). Ce dernier tend vers 1 pour un signal de parole, et vers 0 dans les autres cas.

Le facteur rapide $yu(k)$

Le facteur $yu(k)$ est considéré comme non bloqué car il peut s'adapter rapidement à des signaux variant rapidement. Il n'est pas nécessaire de conserver beaucoup de valeurs. Ce facteur est récursivement déterminé par $y(k)$ et la fonction discrète $W(I)$.

$$yu(k) = (1 - 2^{-5}).y(k) + 2^{-5}.W(I(k)) \tag{2.5}$$

où $1.06 \leq yu(k) \leq 10.00$

La fonction discrète $W(I)$ est définie par la table 2.4.4 pour un codage ADPCM à $32\,kb/s$.

Cette fonction $W(I)$ demande à $yu(k)$ d'adapter par de plus grands pas les valeurs plus grandes de I. Ceci donne à $yu(k)$ la liberté de suivre un signal presque instantanément.

Le facteur lent $yl(k)$

Le facteur $yl(k)$, ou facteur bloqué, s'adapte plus lentement et suit les signaux qui changent lentement. Ce signal est dérivé de $yu(k)$ avec un filtre passe bas.

$$yl(k) = (1 - 2^{-6}).yl(k-1) + 2^{-6}.yu(k) \tag{2.6}$$

En incluant $yu(k)$, $yl(k)$ est implicitement limité au même intervalle de valeur que les limites explicites placées pour $yu(k)$.

2.4.5 Adaptation de la vitesse

Ce bloc calcule le facteur de contrôle de la vitesse $al(k)$ qui est dérivé d'une mesure des variations des valeurs du signal d'erreur. Il ajuste le poids relatif des deux facteurs $yu(k)$

$\|I(k)\|$	7	6	5	4	3	2	1	0
$W[I(k)]$	7	3	1	1	1	0	03	0

TAB. 2.3 - *Fonction discrète $fi(k)$ pour un débit de $32\,kb/s$.*

et $yl(k)$. En utilisant des moyennes à court ou long terme de la sortie $I(k)$, respectivement $dms(k)$ et $dml(k)$, il détermine à quelle vitesse le signal change. Comme le facteur $y(k)$ ne peut être plus grand que $yu(k)$ et $yl(k)$, $al(k)$ est limité à 1 même si le facteur *speed control* $ap(k)$ prédit est plus grand que 1.

$$a1(k) = \begin{cases} 1, \text{ si } ap(k-1) > 1 \\ ap(k-1), \text{ sinon} \end{cases} \tag{2.7}$$

avec

$$ap(k) = \begin{cases} (1-2^{-4}).ap(k-1) + 2^3, \text{ si } |dms(k) - dml(k)| \leq 2^{-3}.dml(k) \\ (1-2^{-4}).ap(k-1) + 2^3, \text{ si } y(k) < 3 \\ (1-2^{-4}).ap(k-1) + 2^3, \text{ si } td(k) = 1 \\ 1, \text{ si } tr(k) = 1 \\ (1-2^{-4}).ap(k-1), \text{ sinon} \end{cases} \tag{2.8}$$

Ainsi, $ap(k)$ tend vers 2 si la différence entre $dms(k)$ et $dml(k)$ est grande, et vers 0 si la différence est petite. $Ap(k)$ tend également vers 2 pour un canal non occupé, ce qui est indiqué par : $y(k) < 3$, ou des signaux à bandes partielles, indiqué par $td(k) = 1$. Enfin, $ap(k)$ tend vers 1 s'il y a détection d'une transition pour un signal à bande partielle, indiqué par $tr(k) = 1$. Quand aucune de ces conditions n'existe, nous supposons que nous avons un signal uniforme variant lentement, comme cela arrive dans la transmission de données.

Le terme court $dms(k)$, et le terme long $dml(k)$ sont des moyennes du signal transmis ADPCM $I(k)$. Ils sont déterminés en moyennant la fonction pondérée $F(I)$ du signal transmis I, comme le montre la table 2.4.5.

$$\begin{aligned} dms(k) &= (1-2^{-5}).dms(k-1) + 2^{-5}.f(I(k)) \\ dml(k) &= (1-2^{-7}).dms(k-1) + 2^{-7}.f(I(k)) \end{aligned} \tag{2.9}$$

En conclusion, le paramètre $al(k)$ de contrôle de vitesse prend ses valeurs dans l'intervalle continu $[0, 1]$. Il tend vers 1 pour des signaux de parole et vers 0 pour d'autre types de signaux. La mesure de la fréquence des changements des valeurs du signal d'entrée détermine ce paramètre. Avec ce facteur de contrôle de la vitesse, le système peut tirer profit d'une adaptation instantanée ou "syllabic", qui s'adapte efficacement à la nature du signal transmis.

2.4.6 Quantificateur adaptatif inverse

Ce bloc calcule une version quantifiée $dq(k)$ du signal d'erreur $d(k)$ comme le montre la figure 2.16.

Le signal ADPCM $I(k)$ est utilisé pour déterminer le logarithme normalisé du signal d'erreur à partir de la table correspondant au débit utilisé. Le résultat $dq(k)$ est calculé en ajoutant le facteur d'échelle $y(k)$ à la valeur spécifiée par la table 2.4.3) et en calculant le logarithme inverse en base 2 de cette somme. Nous avons :

$$dq(k) = log_2^{-1}[log_2|dq(k)| - y(k) + y(k)] \tag{2.10}$$

Pour l'encodeur et le décodeur, le signal d'erreur quantifié $dq(k)$ est l'entrée des modules de calcul du signal reconstruit et du prédicteur adaptatif.

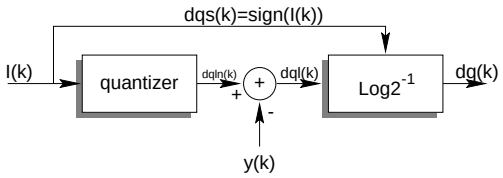


FIG. 2.16 - *Schéma bloc du quantificateur adaptatif inverse.*

2.4.7 Prédicteur adaptatif

Le prédicteur calcule le signal estimé $se(k)$ en minimisant la variance du signal d'erreur de manière à maximiser le rapport signal sur bruit. Les signaux d'entrées de notre système sont non stationnaires : la fonction d'auto-corrélation et la densité spectrale de puissance varient dans le temps. C'est ce qui justifie l'emploi d'un prédicteur dont les coefficients varient dans le temps puisqu'ils permettent d'améliorer les performances du prédicteur.

La norme G.726 utilise deux structures de prédicteurs adaptatifs. Il s'agit d'un filtre d'ordre 6 modélisant les zéros du signal d'entrée, et un filtre d'ordre 2 modélisant les pôles. La figure 2.17 montre le schéma de principe du filtre prédicteur.

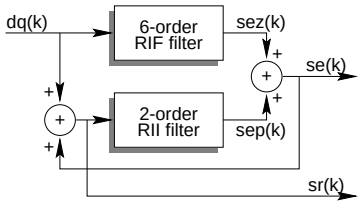


FIG. 2.17 - *Filtre prédictif.*

Le filtre d'ordre 2 avec les valeurs des coefficients adaptatifs contraints est conçu de manière à suivre les variations lentes d'un signal de parole. Cependant, ce type de filtre est particulièrement sensible aux erreurs. Nous utilisons alors un filtre tout zéro d'ordre 6 pour donner au signal une stabilité, même en cas d'erreur de transmission.

Pour les pôles où les zéros respectivement, les coefficients $ai(k)$ et $bi(k)$, sont adaptés en utilisant un algorithme simplifié de gradient qui ajuste le modèle du filtre au signal d'entrée. Les figures 2.18 et 2.19 décrivent les filtres.

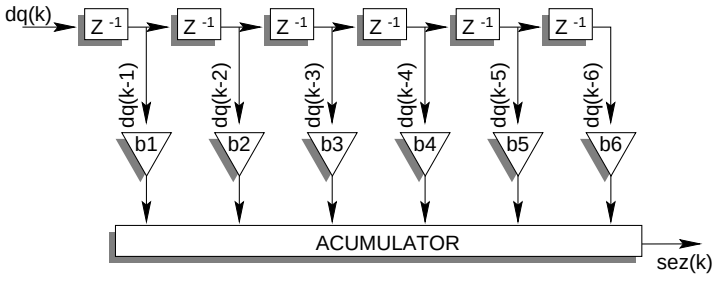


FIG. 2.18 - *Filtre* RIF *tout zéro d'ordre* 6.

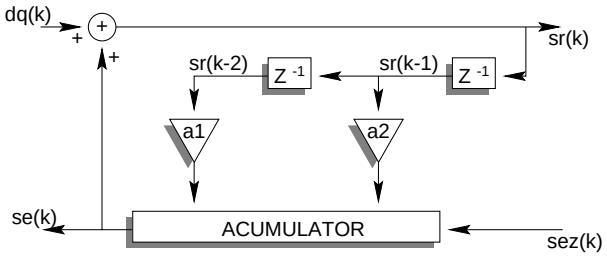


FIG. 2.19 - *Filtre* RII *d'ordre* 2.

Le signal estimé $se(k)$ représente la somme du filtre tout pôle et du filtre tout zéro.

$$\begin{array}{rcl} se(k) & = & \sum_{i=1}^{2} ai'k-1) \times sr'k-i) + sez(k) \\ sez(k) & = & \sum_{i=1}^{6} bi(k-1) \times dq(k-i) \\ sr(k-i) & = & se(k-i) + dq(k-i) \end{array} \tag{2.11}$$

Adaptation des coefficients $ai(k)$ (pôles) du filtre d'ordre 2

Les étapes suivantes montrent l'adaptation des $ai(k)$.

$$\begin{array}{rcl} a1(k) & = & (1-2^{-8}).a1(k-1) + 3.2^{-8}.sgn\ p(k).sgn\ p(k-1) \\ a2(k) & = & (1-2^{-7}).a2(k-1) + 2^{-7}. \\ & & \{sgn\ p(k).sgn\ p(k-2) - f(a1(k-1)).sgn\ p(k).sgn\ p(k-1)\} \end{array} \tag{2.12}$$

où

$$p(k) = dq(k) + sez(k) \tag{2.13}$$

et

$$f(a1) = \begin{cases} 4.a1, \text{ si } |a1| \leq 1/2 \\ 2.sgn\ a1, \text{ sinon} \end{cases} \tag{2.14}$$

La fonction de gradient est déterminée par le signal $p(k)$ qui est équivalent au signal reconstruit moins la contribution de la sortie du filtre en pôle.

Les stabilités du filtre sont apportées en limitant explicitement les coefficients :

$$\begin{array}{l} |a1(k)| \leq 1 - 2^{-4} - a2(k) \\ |a2(k)| \leq 0.75 \end{array} \tag{2.15}$$

Adaptation des coefficients $bi(k)$ (zéros) du filtre d'ordre 6

La procédure d'adaptation est similaire à celle des pôles, mais la limite est implicite dans les équations à un maximum de 2. La fonction de gradient, dans ce cas, est déterminée par le signal d'erreur courant $dq(k)$, et le signal d'erreur correspondant $dq(k-i)$.

$$bi(k) = (1-2^{-8}).bi(k-1) + 2^{-7}.sgn\ dq(k).sgn\ dq(k-i) \tag{2.16}$$

pour $i = 1, 2, ..., 6$ et avec $-2 \leq bi(k) \leq +2$.

Pour un codage à $40\,kb/s$, le prédicteur adaptatif est changé afin de diminuer le facteur de fuite utilisé pour le calcul des coefficients. Dans ce cas l'équation précédente devient:

$$bi(k) = [1-2^{-9}].bi(k-1) + 2^{-7}.sgn\ dq(k).sgn\ dq(k-i) \tag{2.17}$$

Conclusion

Le prédicteur utilise un modèle ARMA (*AutoRegressive Moving Average*) de filtre, la partie *Moving Average* est d'ordre 6 alors que la partie AutoRegressive est d'ordre 2.

2.4.8 *Tone and transition detector*

De manière à améliorer les performances pour des signaux modems à modulation de fréquence, un mécanisme de détection à deux pas est défini.

Tone detection

Dans un premier temps, le quantificateur peut être conduit dans le mode rapide d'adaptation s'il y a détection d'un signal à bande partielle, comme pour le son.

$$td(k) = \begin{cases} A, \text{ si } a2(k) < 0.75 \\ 0, \text{ sinon} \end{cases} \tag{2.18}$$

Transition detector

Le quantificateur peut être forcé dans le mode rapide d'adaptation et les coefficients du prédicteur mis à 0 s'il y a une transition d'un signal à bande partielle.

$$tr(k) = \begin{cases} 1, \text{ si } a2(k) < 0,71875 \text{ et } |dq(k)| > 24,2^{y1(k)} \\ 0, \text{ sinon} \end{cases} \tag{2.19}$$

On remarque que si $tr(k) = 1$ alors $b1(k) = b2(k) = ... = b6(k) = 0$ et $a1(k) = a2(k) = 0$.

Chapitre 3

Spécification

Ce chapitre décrit la manière avec laquelle la norme G.726 est décrite dans le document fourni par l'organisme de normalisation `CCITT`. La première partie de la norme décrit le fonctionnement du compresseur et du décompresseur ADPCM, comme cela a été fait dans le chapitre précédent.

Puis chacune des composantes du dispositif est décrite à l'aide d'un langage de boîtes, pour la partie structurelle, et d'un pseudo langage C pour les expressions. Nous avons extrait des parties de la norme et donné la copie de la spécification.

3.1 Module ASC

La spécification du module qui adapte le facteur de vitesses est donnée sous la forme d'un graphe :

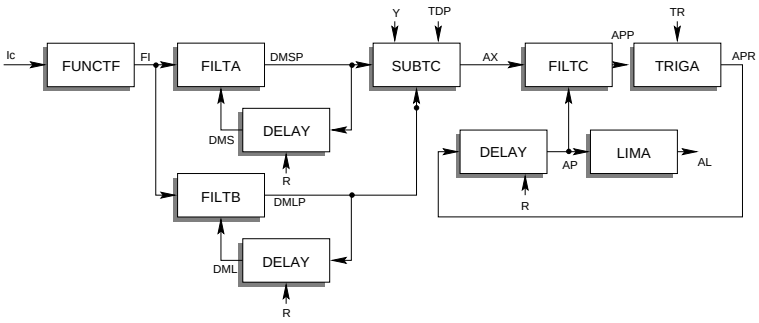


FIG. 3.1 - *Adaptation speed control.*

Les éléments de ce graphe sont spécifiés soit à l'aide de graphes, soit à l'aide d'une spécification par pseudo-équations, comme dans la section suivante.

3.2 Module FILTA

FILTA
Inputs: FI, DMS
Outputs: DMSP
Function: Update of short-term average of $F(I)$.

$$\begin{array}{lcl} DIF & = & ((FI << 9) + 8192 - DMS) \,\&\, 8191 \\ DIFS & = & DIF >> 12 \\ DIFSX & = & \begin{cases} DIF >> 5, & DIFS = 0 \\ (DIF >> 5) + 3840, & DIFS = 1 \end{cases} \\ DMSP & = & (DIFSX + DMS) \,\&\, 4095 \end{array} \tag{3.1}$$

L'opérateur $<<$ représente un décalage binaire à gauche et $>>$ un décalage à droite. L'opérateur & est le *et* binaire.

3.3 Retard unitaire delay

La spécification du bloc *delay* est :

DELAY
Inputs: x, r, R
Outputs: y
Function: Memory block. For the input x, the output is given by:

$$y(k) = \begin{cases} x(k-1), & R = 0 \\ r, & R = 1 \end{cases} \tag{3.2}$$

Ce module définit une variable à état qui introduit un retard unitaire.

3.4 Module FUNCTF

FUNCTF
Inputs: I
Outputs: FI
Function: Map quantizer output into the $F(I)$ function.

$$\begin{array}{lcl} IS & = & I >> 3 \\ IM & = & \begin{cases} I\&7, & IS = 0 \\ (15 - I) \,\&\, 7, & IS = 1 \end{cases} \\ FI & = & \begin{cases} O, & O \le IM \le 2 \\ 1, & 3 \le IM \le 5 \\ 3, & IM = 6 \\ 7, & IM = 7 \end{cases} \end{array} \tag{3.3}$$

Chapitre 4

Réalisation

Dans ce chapitre, nous présentons la modélisation de la norme G.726 à l'aide des outils présentés dans la troisième partie de la thèse. Nous nous préoccuperons essentiellement des spécifications de la section précédente.

4.1 Programmation de l'application

La programmation de l'application G.726 utilise deux outils, l'éditeur λ-GRAPH et le langage λ-FLOW. Les programmes λ-GRAPH doivent être traduits en λ-FLOW avant d'être compilés.

4.1.1 Module ASC

La spécification de ce module dans la norme est graphique, à l'aide d'un langage de boîtes. Nous utilisons donc le langage λ-GRAPH pour programmer ce module.

Ce module est composé avec d'autres modules définis ci-dessous, à l'aide de λ-FLOW. Cette représentation graphique est très proche de sa spécification. L'outil permet une saisie graphique confortable.

4.2 Module FILTA

La description de FILTA utilise le langage λ-FLOW :

```
FILTA is lambda FI:int, DMS:int. begin
  DIF   is (((FI << 9) + 8192) - DMS) & 8191;
  DIFS  is DIF >> 12;
  DIFSX is if   (DIFS = 0)
           then (DIF >> 5)
           else ((DIF >> 5) + 3840);
  DMSP  output (DIFSX + DMS) & 4095;
end;
```

Dans ce module, nous avons utilisé une syntaxe longue, (`is`, `lambda`, etc). Les opérateurs arithmétiques sont définis dans l'algèbre des entiers de λ-FLOW (§ III-4.2). Là encore, nous remarquons que la programmation est très proche de la spécification.

4.2.1 Retard unitaire delay

La norme indique que le retard unitaire est une variable. En λ-FLOW, cette variable est programmée à l'aide d'un flot. Les paramètres de ce module sont la valeur initiale de la variable, l'expression de ses valeurs suivantes et la commande de réinitialisation :

```
\\ Polymorphic abstraction
DELAY :=  init, next, reset. [
```

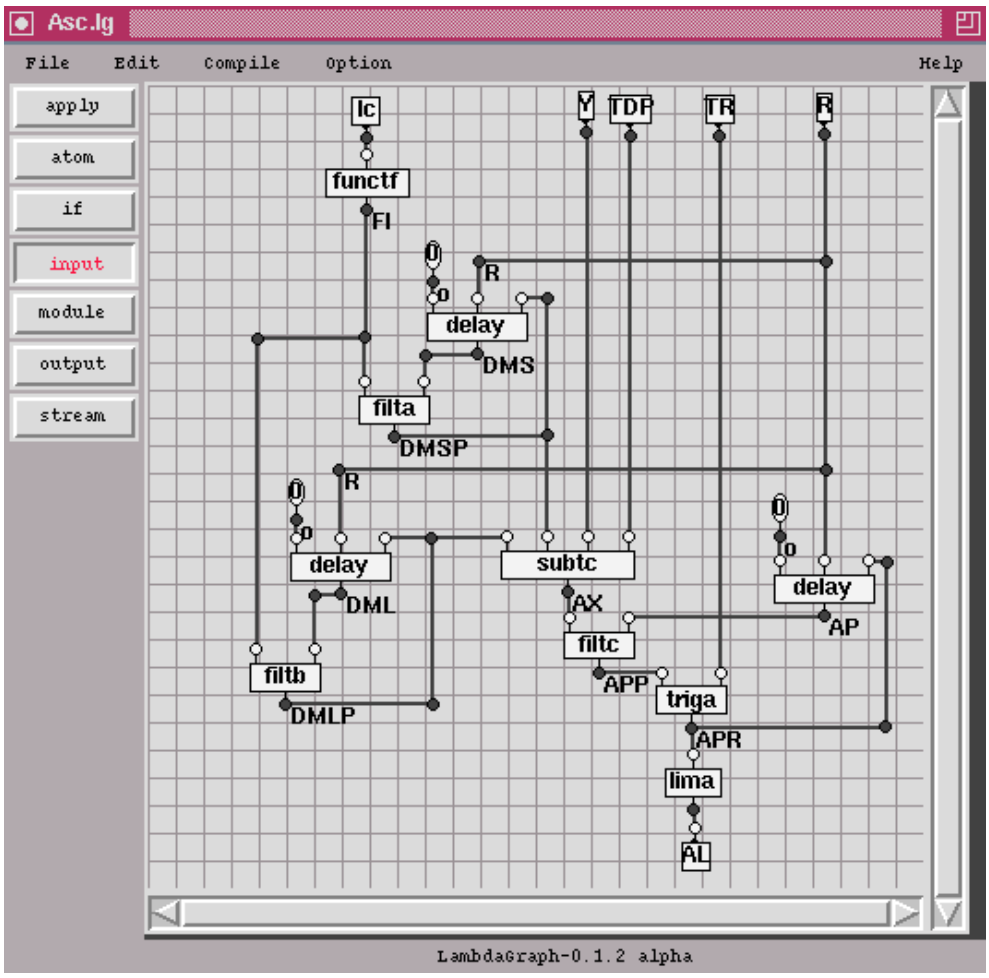


FIG. 4.1 - *Module* ASC *modélisé à l'aide de* λ-GRAPH.

```
  out #= init ~ (reset ? init : next);
]
```

Dans ce module, nous avons utilisé la syntaxe courte de λ-FLOW. De plus, les paramètres du module sont polymorphes, ce qui signifie qu'il peut être utilisé avec tout type de données.

Les possibilités d'abstraction polymorphes de λ-FLOW permettent une réutilisation maximale du code.

4.2.2 Module FUNCTF

Enfin, le module FUNCTF est programmé comme suit :

```
FUNCTF := lambda I:int. begin
  /* for 32 kb/s */
  in := lambda min, value, max.
          ((min <= value) && (value <= max));
  IS := I >> 3;
  IM := if (IS = 0) then (I & 7) else ((15 - I) & 7;
  FI #=       if in (0, IM, 2) then 0
        else if in (3, IM, 5) then 1
        else if (IM = 6)      then 3
        else                       7;
end;
```

Là, nous voyons comment réaliser les choix multiples avec des alternatives imbriquées.

4.2.3 Autres modules

Les autres modules sont décrits de la même manière. Le module `main` possède une entrée et une sortie, qui seront l'entrée et la sortie principale du filtre. L'ensemble du programme est codé en moitié moins d'octets que l'équivalent en C.

4.3 Compilation

Le compilateur λ-FLOW permet de produire différentes cibles. Par exemple, la distribution standard permet de produire du SCHEME, du C, de l'assembleur INTEL-386 et du SPA pour le compilateur parallèle. Les programmes λ-GRAPH doivent être préalablement traduits en λ-FLOW.

4.3.1 Conversion des fichiers λ-graph en λ-flow

Le programme λ-GRAPH sont dans des fichiers dont l'extension est `.lg`. Pour les convertir en λ-FLOW, il suffit d'entrer :

```
> g2f *.lg
```

ce qui produit des fichiers de noms identiques et d'extension `.lf`.

4.3.2 Production de C

La compilation de l'ensemble des modules λ-FLOW réalisant la norme G.726 en C s'effectue simplement en entrant :

```
> flow --target c *.lf
message: main parameter 'i' is assigned to port '1'
message: main output 'out' is assigned to port '1' and name 'out_1'

Hum... So, we have... 0 error and 1 warning.
time=2s, heapsize=853008, gc calls=589
```

Ceci produit le fichier exécutable `a.out`[1]

Le programme généré déclare 550 variables de type *int*. Le code compte 200 lignes de code non linéaire, construit en une itération simple.

Pour exécuter le programme résultant, il suffit de rediriger les entrées / sorties standards du programme vers le fichier d'entrée / sortie avec :[2]

```
> a.out < test.in > test.out
```

4.3.3 Production de SPA

Pour obtenir le programme de l'architecture parallèle statique, nous entrons simplement la commande :

```
> flow --target spa *.lf
message: main parameter 'i' is assigned to port '1'
message: main output 'out' is assigned to port '1' and name 'out_1'

Hum... So, we have... 0 error and 1 warning.
time=2s, heapsize=853045, gc calls=594
```

1. Le fichier `a.out` est fourni par la version UNIX de λ-FLOW, et `a.exe` par la version DOS.
2. Cette utilisation provient des entrées/sorties réalisées par l'algèbre des entiers. L'utilisateur peut spécifier d'autres moyens d'entrée/sortie.

qui produit le fichier `a.spa`. Ce fichier contient la liste des tâches de l'application. Ces taches élémentaires sont réparties sur une architecture parallèle statique à 5 contrôleurs, par exemple, avec la commande :

```
> spac -n 5 -r -d -p a.spa
LambdaSPAC : Static Parallel Architecture Compiler- (c) 96-97 gdw

compilation of       : a.spa
register optimization: yes
inverter optimization: yes
read  optimization   : yes
tasks           start: 0
                 init: 43
                 loop: 2290
                 next: 43
               --------------
                  sum: 2376
variables            : 2358
instructions         : 3877
sum of instructions  : 3877x5=19385
1-controller instruc.: 8824
register optimization: 63
inverter optimization: 594
read optimization    : 118
optimization ratio   : 56%
 |  cont |  task |  read | write |  nop  |
 |-------|-------|-------|-------|-------|
 |     1 |   302 |   401 |   301 |  2075 |
 |     2 |   286 |   409 |   285 |  2206 |
 |     3 |   302 |   419 |   301 |  2089 |
 |     4 |   297 |   424 |   296 |  2146 |
 |     5 |   305 |   414 |   304 |  2041 |
 |-------|-------|-------|-------|-------|
 |   sum |  1492 |  2067 |  1487 | 10557 |
-done
Output file are from spa-1.S to spa-5.S
```

qui produit 5 fichiers de `spa.1` à `spa.5` contenant le programme de chacun des contrôleurs.

4.4 Évaluation

La courbe du gain de performances obtenu en fonction du nombre de processeurs est donnée dans la figure 4.2.

Cette courbe montre qu'un asymptotique de 60% peut être atteint. Le doublement de performances intervient avec trois ou quatre processeurs, ce qui correspond à une augmentation du coût d'un même facteur.

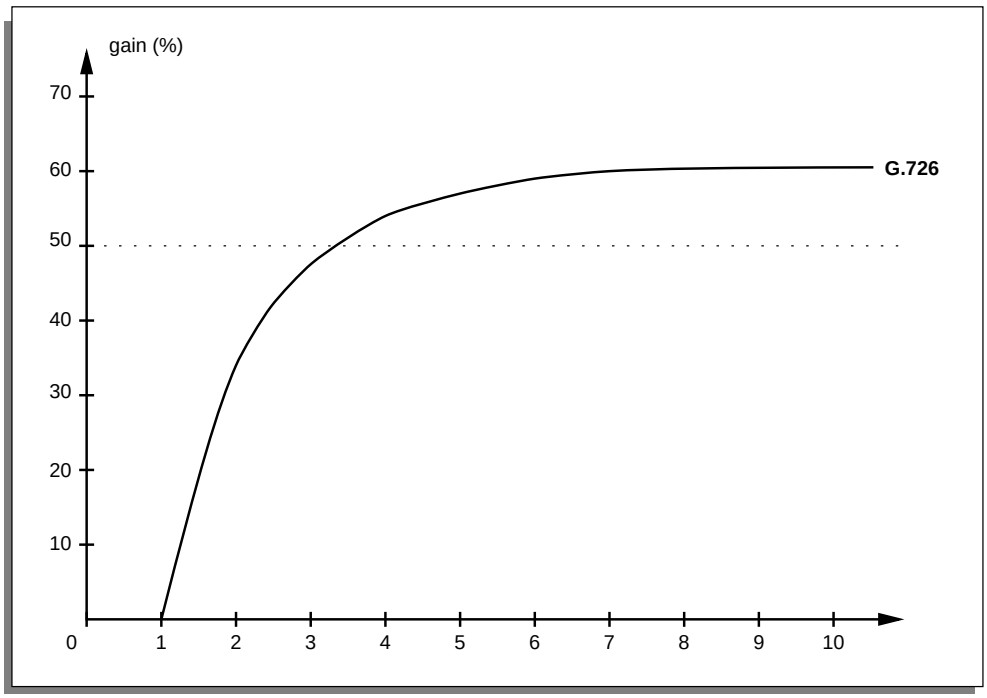


FIG. 4.2 - *Gain en fonction du nombre de processeurs.*

Chapitre 5

Conclusion

La quatrième partie de la thèse nous a permis de confronter les méthodes et les outils que nous avons développés à un exemple réel issu de l'industrie. Cet exemple est suffisamment général pour être acceptable.

Le bénéfice de l'utilisation des λ-outils est mesurable dans la phase de programmation de l'application, comme dans la phase de réalisation, au niveau des performances.

Dans la phase de programmation de l'application, l'interface graphique λ-GRAPH et son langage sont particulièrement efficaces, car ils permettent une programmation à un très haut niveau d'abstraction. La programmation modulaire et polymorphe qu'offre ce langage permet une réutilisation maximale du code.

Le langage λ-FLOW et son compilateur permettent une programmation à haut niveau sans souffrir des limitations de λ-GRAPH, notamment pour des applications peu modulaires ou de grande taille. La déconnection totale du langage avec les algèbres est une originalité importante. Elle permet une très grande adaptabilité du langage à toutes sortes de problème. Le compilateur utilise un algorithme inductif complexe de contrôle des types qui libère le programmeur des nombreuses déclarations de types que l'on rencontre habituellement dans les autre langages. De plus, il encourage la définition polymorphe des modules pour une réutilisation maximale du code.

Le fait que λ-FLOW soit complètement indépendant de la cible qu'il produit lui donne, là encore, une très grande généralité. Nous avons défini quatre cibles dont les formats sont très différents : le langage C, SCHEME, l'assembleur INTEL-386 et le format SPA pour le paralléliseur de code. La définition d'une cible nécessite une vingtaine de définitions, ce qui la rend simple et accessible.

Le code C produit par λ-FLOW est de bonne qualité. En effet, il est relativement simple à lire, car il est construit autour d'une itération. De plus, il est bien documenté et permet, en particulier, de retrouver l'instruction λ-FLOW correspondante au code produit. Du point de vue de l'efficacité, le code C produit par λ-FLOW n'est pas compact, car il ne définit aucune fonction. Cependant, les performances à l'exécution sont comparables aux performances obtenues par une programmation C manuelle.

Du point de vue des performances, l'intérêt de λ-FLOW est de produire du code au format SPA, utilisable par le compilateur parallèle. Là, pour une augmentation du coût de l'architecture d'un facteur 3 ou 4, les performances augmentent de 50%, et jusqu'à 60%. L'usage de l'architecture parallèle statique n'est rentable qu'avec des processeurs haut de gamme.

L'usage des λ-outils se montre donc complètement satisfaisant, tant du point de vue de la programmation des applications que du point de vue des performances de l'architecture parallèle proposée dans cette thèse.

Conclusion

Cette thèse avait comme objectif initial l'étude des langages fonctionnels par rapport à l'exploitation du parallélisme des applications, et plus particulièrement des applications de traitement du signal. Le sujet de cette thèse fut choisi car il semble que l'exploitation du parallélisme à grain fin a donné des résultats peu satisfaisants. De plus, les langages fonctionnels semblent avoir un regain d'intérêt du fait de leur propriétés sémantiques.

Nous avons donc commencé cette étude en analysant les différents formalismes se rapprochant à la fois de l'exploitation du parallélisme, comme les graphes dataflow, et des langages fonctionnels.

Nous avons constaté que les graphes dataflow avaient été très étudiés par le passé, mais que les résultats quant à l'exploitation du parallélisme semblaient décevant. Cependant ces formalismes sont souvent représentés d'une manière graphique, ce qui les rend très accessibles aux utilisateurs.

Cette étude nous a très vite conduits à consulter des ouvrages traitant du parallélisme et des techniques d'exploitation. Nous avions alors la conviction qu'une architecture parallèle à grain fin devait avoir des temps d'inter-connections les plus faibles possibles. Nous avons alors eu l'idée du modèle de l'architecture parallèle statique dont la simplicité nous semblait être un atout majeur. Du fait du caractère statique de l'architecture, il était évident que les programmes de chacun des contrôleurs devait être déterministe, c'est à dire que l'ordonnancement des instructions devait être immuable. Cette propriété pouvait être obtenue simplement si les programmes de chacun des contrôleurs était construit en une itération.

Laissant de coté notre modèle d'architecture, nous nous sommes ensuite intéressés aux formalismes fonctionnels, en commençant par leur fondement, le λ-calcul. Nous nous sommes aperçus qu'ils possèdent des propriétés sémantiques évidentes qui leur confèrent à la fois une grande simplicité et une puissance expressive hors du commun. Dans le cadre des applications du signal, qui sont souvent amenées à être embarquées sur des dispositifs mobiles, ces caractéristiques sont particulièrement appréciées.

La famille des langages fonctionnels comprend un certain nombre de langages à la fois fonctionnels et basés sur les flots de données. Ces langages sont souvent utilisé dans la conception de circuits ou de filtres numériques.

Les critiques apportées à ces formalismes nous ont permis de concevoir un formalisme fonctionnel et à flots de données, appelé λ-matrices. L'objectif était d'obtenir un langage aussi simple que possible et dont la résolution était absolument fonctionnelle. Ainsi, la sémantique du langage serait-elle donnée par l'expression de la machine de résolution.

L'expression de la résolution des λ-matrices est absolument fonctionnelle, ce qui signifie qu'il s'agit d'une fonction. Le caractère itératif de la résolution d'une application se traduit dans un formalisme fonctionnel par une récursion terminale. Dès lors que nous avions obtenu une expression de la résolution des λ-matrices sous la forme d'une expression mathématique à récursion terminale, nous savions que ce formalisme allait être utilisé pour programmer l'architecture parallèle statique.

Nous avons alors utilisé les propriétés sémantiques des λ-matrices pour établir les conditions pour lesquelles nous obtenions les déterminismes en temps et en ressources. Ce fut l'établissement des fonctions de critères et la vérification de leurs propriétés.

Comme nous étions convaincus que la réutilisation du code passe nécessairement par une

programmation modulaire, nous avions doté les λ-matrices de modules primitifs. L'existence des modules dans un programme empêchait d'utiliser directement les λ-matrices pour programmer l'architecture parallèle statique. Il fallait concevoir une phase de compilation. Nous avons donc conçu un compilateur formel qui supprime les modules dans une λ-matrice. La λ-matrice résultante était suffisamment simple pour permettre une programmation efficace de l'architecture parallèle statique.

Nous avions donc un langage abstrait, déterministe en temps et en ressources (ou plutôt qui permettait d'établir ces propriétés), et un compilateur formel. Il nous restait à concevoir des logiciels mettant en œuvre ce langage abstrait et la chaîne de traitement associée.

Nous avons donc conçu le compilateur λ-FLOW et son langage, directement inspirés des λ-matrices. L'indépendance des algèbres dans le formalisme abstrait devait se retrouver dans le langage, car il s'agissait d'une caractéristique intéressante et novatrice. De plus, du fait de la structure relativement simple du code produit, nous avons aussi permis la description dynamique des cibles du compilateur. Le compilateur λ-FLOW est le logiciel le plus abouti. Puis, nous nous sommes intéressés à l'interface graphique de λ-FLOW qui a donné naissance à l'interface graphique λ-GRAPH. Cette interface fut programmée rapidement à l'aide d'outils de développement efficaces.

Nous avons alors conçu le paralléliseur de code λ-SPAC. Sa programmation fut assez rapide du fait de la structure relativement simple et pré-traitée du code produit par λ-FLOW. Pour valider le code produit, nous avons programmé un simulateur graphique de l'architecture parallèle statique.

Il était donc maintenant possible de programmer graphiquement ou textuellement une application, de la compiler et de l'exécuter sur le simulateur graphique. Il nous restait à étudier un cas concret. Les relations du laboratoire I3S avec l'Industrie ont permis d'obtenir un problème concret : il s'agissait de la norme de compression utilisée dans les téléphones portables.

L'étude de cette application réelle, et sa programmation avec λ-GRAPH et λ-FLOW ont montré l'intérêt de ces langages. Ils libèrent complètement le concepteur de toute interprétation des spécifications, en diminuant d'autant les risques d'erreurs. De plus les temps de programmation sont considérablement réduits. La puissance expressive de ces langages permettrait de les utiliser directement comme spécifications.

La compilation du programme obtenu permet d'obtenir indifféremment du code dans toutes les cibles définies pour le compilateur λ-FLOW. En particulier, le code C produit se révèle être de bonne qualité, comparativement au code produit manuellement. L'utilisation du paralléliseur λ-SPAC permet d'obtenir le doublement des performances pour une architecture parallèle statique à quatre contrôleurs.

L'utilisation des formalismes et outils développés durant cette thèse possède au moins deux avantages. D'une part, ils proposent des langages de haut niveau, qui permettent une abstraction totale de la réalisation finale. D'autre part, les performances obtenues sont très encourageantes.

Cependant, un certain nombre d'améliorations n'ont pu être apportées durant la thèse. Notamment, le langage λ-FLOW serait plus puissant s'il permettait la programmation de boucles intérieures. Un pré-étude nous a montré que cette possibilité permettrait toujours d'exploiter l'architecture parallèle statique. Cependant, nous perdrions le déterminisme en temps des programmes.

De plus, le compilateur λ-FLOW ne gère pas les variables temporaires dans le code produit en permettant de réutiliser celles qui ne sont plus utiles. Une telle gestion permettrait de réduire de manière importante les besoins en mémoires des réalisations.

Il serait intéressant de définir une cible VHDL pour λ-FLOW. De cette manière, le langage pourrait être directement utilisé dans la conception des circuits électroniques numériques.

L'interface graphique λ-GRAPH est actuellement programmée avec un langage interprété fonctionnant sous UNIX Il serait intéressant de la reprogrammer en C avec une boite à outils graphique adaptable à différents systèmes d'exploitation.

Le modèle du compilateur parallèle λ-SPAC n'exploite qu'une architecture parallèle sta-

tique à processeurs homogènes. Il serait intéressant d'étendre ce modèle à des architectures à processeurs hétérogènes.

De plus, le compilateur λ-SPAC possède une optimisation qui permet de tenir compte du contenu des registres des contrôleurs. Cette optimisation permet de réduire les accès à la mémoire globale. Mais avec deux registres, elle se révèle peu efficace. Il serait intéressant d'étendre cette optimisation à toute la mémoire locale des contrôleurs, qui jouerait alors le rôle d'un cache de la mémoire globale. Cette optimisation laisse espérer en moyenne, un doublement des performances, par rapport aux performances déjà obtenues, ce qui porterait le gain en performances à 75%, ce qui est très important. Par contre, l'algorithme deviendrait particulièrement complexe, car il serait nécessaire de gérer la durée de vie des variables.

Ces trois années de thèse nous ont été particulièrement profitables, du point de vue de la connaissances de l'informatique. Nous avons pu approfondir des thèmes comme le calcul formel et les langages fonctionnels, la programmation orientée objets, les techniques de compilation, les techniques utilisées dans les simulateurs, le parallélisme.

Je tiens à remercier très chaleureusement tous ceux qui, directement ou indirectement, m'ont aidé dans cette étape importance de ma vie

Bibliographie

[1] H. Abelson, G.J. Susman, and J. Susman. *Structure and Interpretation of Computer Programs*. MIT Press, 1987.

[2] W.B. Ackerman and J.B. Dennis. VAL – a value oriented algorithmic language. Technical Report LCS/TR-218, MIT, Cambrige, MA, June 1979.

[3] D.A. Adams. A computation model with dataflow sequencing. Technical Report CS-117, Computer Science Department, Standfors University, 1968.

[4] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Inc, 1986.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers*. Addison-Wesley Publishing Company, Inc, 1986.

[6] M. Amamiya and R. Hasegawa. Dataflow computing and eager and lazy evaluations. *New Generation Computing*, 2(2):105–129, 1984.

[7] Arvind and David E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–53, 1986.

[8] Arvind and D.E. Culler. Managing resources in a parallel machine. *Fifth Generation Computer Architectures*, pages 103–121, 1987.

[9] Arvind and K.P. Gostelow. Some relationships between asynchronous interpreters of dataflow languages. *Formal Description of Programming Languages, New-York*, 1977.

[10] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *j-CACM*, 20(7):519–526, July 1977.

[11] E.A. Ashcroft and R. Jagannathan. Operator nets. *Fifth Generation Computer Architectures*, pages 177–202, 1986.

[12] E.A. Ashcroft and W.W. Wadge. LUCID, a formal system for writing and proving programs. *SIAM j. Comput*, 5(3):336–354, September 1976.

[13] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions*, 35:535–546, 1990.

[14] G. Berry and G. Gonthier. The synchronous programming language ESTEREL, design, semantics, implementations. Technical Report 842, INRIA, 1988.

[15] L. Bic. A process-oriented model for efficient execution of dataflow programs. *Journal of Parallel and Distributed Computing*, pages 42–51, 1990.

[16] A. P. W. Bohm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft. SISAL 2.0 reference manual. Report CS-91-118, Computer Science Department, Colorado State University, Fort Collins, CO, 1991.

[17] P. Caspi. Lucid synchrone. *OPOPAC, HERMES Paris*, pages 79–93, 1993.

[18] W. Clinger and J. Rees. Revised[4] report on the algorithmic language scheme. Technical report, MIT Artificial Intelligence Laboratory, CSDTR 174, october 1990.

[19] D.E. Culler and Arvind. Resources requirements of dataflow programs. In *Proceedings 15th Annual International Symposium on Computer Architectures*, pages 141–150. ACM SIGARCH, 1988.

[20] D.E. Culler, K.E. Schauser, and Thorsten von Eicken. Two fundamental limits on dataflow multiprocessing. *Architecture and Compilation Techniques for Fine and Medium Grain Parallelism (A-23)*, pages 153–164, 1993.

[21] L. Damas and R. Milner. Principal type-schemes for functional programs. pages 207–212, 1982.

[22] G. de Wailly. Implémentation des λ-matrices à l'aide du λ-calcul. Technical Report 95-34, I3S, july 1995.

[23] G. de Wailly. User manual of lambda graph, the graphical interface of the functional synchronous data flow language lambda flow. Technical Report 95-33, I3S, July 1995.

[24] G. de Wailly. A graphical interface for the functional synchronous data-flow language λ-flow. In *Dynamic Object Workshop (DOW'96)-USA*. Object World East, may 1996.

[25] G. de Wailly and F. Boéri. Lambda matrices: A formal approach to functional modeling of synchronous data-flow systems. Technical Report 95-38, I3S, July 1995.

[26] G. de Wailly and F. Boéri. A parallel architecture simulator for the lambda matrices. In *Association of Lisp Users Meeting and Workshop Proceedings (LUV'95)-USA*, august 1995.

[27] G. de Wailly and F. Boéri. A cad tool chain for signal processing applications, with parallel implementation issues. In *Groningen Information Technology Conference-Holand*, February 1996.

[28] G. de Wailly and F. Boéri. Dataflow language for signal processing modeling with parallel implementations issues. In *VIII European Signal Processing Conference (EUSIPCO'96)-Italy*. EURASIP, september 1996.

[29] G. de Wailly and F. Boéri. λ-flow: User manual. Technical Report 95-39, I3S, July 1996.

[30] G. de Wailly and F. Boéri. Specification of a functional synchronous dataflow language for parallel implementation with the denotational semantics. In *Symposium on Applied Computing (SAC'96)-USA*. ACM, February 1996.

[31] J. B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Colloque sur la Programmation*. Springer-Verlag, Berlin, DE, 1974.

[32] J. B. Dennis. *A Highly Parallel Processor Using a Data Flow Machine Language*. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1979.

[33] Jack B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, November 1980.

[34] J.B. Dennis. *Evolution of "static" data-flow architecture*. Advanced Topics in Dataflow computing, Prentice Hall, 1991.

[35] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data flow processor. In *Proceedings of the Second Annual Symposium on Computer Architecture*, pages 126–132. ACM, 1975.

[36] A. Filinski. Representing monads. *Proceedings of the 21th Annal ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–22, 1994.

[37] E. Gallesio. Stk reference manual, version 2.2. Technical report, Laboratoire I3S-CNRS URA 1376 - ESSI, web:`kaolin.unice.fr:/pub/`, October 1995.

[38] T. Gautier, P. le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 257–277. Springer-Verlag, Berlin, DE, 1987.

[39] G.M.Papapopoulos and K.R. Traub. Multitheading: A revisionist view of dataflow architecture. *Proceedings 18th Annual International Symposium on Computer Architecture*, pages 342–351, 1991.

[40] P. Gochet and P. Gribomont. *Logique: méthodes formelles pour l'étude des programmes.* Hermes, 1994.

[41] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Pulishers, 1993.

[42] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs : le langage LUSTRE. *Techniques et Sciences Informatiques*, 10(2):139–158, 1991.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1319, 1991. Published as Proceedings of the IEEE, volume 79, number 9.

[44] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Adison Wesley, Reading, MA, 1979.

[45] R.A. Iannucci. Toward a dataflow/VON NEUMANN hybrid architecture. *Proceedings 19th Annual International Symposium on Computer Architecture - ACM Press*, pages 131–140, 1988.

[46] S.C. Johnson. YACC – yet another compiler compiler. Technical Report 32, Comp. Sci – AT&T Bell Labs, Muray Hill, NJ, 1975.

[47] S.L. Peyton Jones. *The implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[48] S.L. Peyton Jones and P. Walder. Imperative functional programming. *ACM-0-89791-561-5/93/0001/0071*, pages 71–84, 1993.

[49] GL. Steele JR. *Common Lisp: The Language, 2nd Edition.* Digital Press (Bedford, MA), 1990.

[50] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress.* North Holland, Amsterdam, 1974.

[51] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, 1966.

[52] Takuya Katayama. Type inference and type checking for functional programming languages: A reduced computation approach. In *Lisp and Functional Programming*, pages 263–272, 1984.

[53] G. Kickzales. Tiny-clos. Source available on ftp.xerox.com in directory /pub/mops, December 1992.

[54] S.C. Kleene. Representation of eventss in nerve sets. *Shannon and McCarthy*, pages 3–40, 1956.

[55] J.L. Krivine. *Lambda-Calcul, types et modèles.* Masson, 1990.

[56] P.J. Landin. The next 700 programming languages. *Communication of ACM*, 9:157–166, march 1966.

[57] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[58] M.E. Lesk. LEX – a lexical analyser generator. Technical Report 39, Comp. Sci – AT&T Bell Labs, Muray Hill, NJ, 1975.

[59] A. Lloyd. *A practical introduction to denotational semantics.* Cambridge Computer Science Texts 23, 1986.

[60] B. Meyer. *Introduction à la théorie des langages de programmation.* Inter Edition, 1992.

[61] T. C. Miller. Type checking in an imperfect world. pages 237–243, 1979.

[62] S. Miranda. *L'art des bases de données - Tome I: Introduction aux bases de données.* Eyrolles, 1988.

[63] E. Moggi. Computational lambda-calculus and monads. *Proceedings of the Fourth Annual Symposium on Logic in Computer Science - California*, pages 14–23, 1990.

[64] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[65] C. Queinnec. *Les langages* LISP. Inter Édition, 1994.

[66] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming.* Cambrige University Press, 1988.

[67] J.E. Rodriguez. A graph model for parallel computation. Technical Report TR-64, Project MAC, Massachusetts Institute of Technology, Cambridge, 1969.

[68] C.A. Ruggerio and J. Sargeant. Control of parallelism in the manchester dataflow machine. In *Proceedings of the 1987 Conference on Functional Programming and Computer Architecture.* ACM, 1987.

[69] Mark A. Sheldon and David K. Gifford. Static dependent types for first class modules. In *Lisp and Functional Programming - 6th Conference*, pages 20–29, 1990.

[70] J.E. Stoy. *Denotational Semantics: The Scott — Strachey Approach to Programming Language Semantics.* MIT Press Series in Computer Science, 1977.

[71] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architecture. *Computings Surveys*, 14(1):93–143, March 1982.

[72] W.W. Wadge and E.A. Ashcroft. *Lucid, the DataFlow Programming Language.* Academic Press, 1985.

[73] Y. Yamaguchi, S. Sakai, K. Hiraki, Y. Kodama, and T. Yuba. An architectural design of a highly parallel machine. In G.X. Ritter, editor, *Proceedings Information Processing 89 (IFIP'89).* Elsevier Science Publishers, 1989.

Index