N° d'ordre : 107

Année 1993

THÈSE

présentée en vue de
l'obtention du titre de

DOCTEUR

de

L'ECOLE NATIONALE SUPERIEURE
DE L'AERONAUTIQUE ET DE L'ESPACE

SPECIALITE : INFORMATIQUE

par

Christophe COUSTET

PARALLELISATION EXPLICITE DE PROGRAMMES FONCTIONNELS ET SYNCHRONISATION DE PROCESSUS : APPLICATION AU MODELE MaRS.

Soutenue le 25 janvier 1993 devant la Commission d'Examen :

MM.	L. BOUGÉ	Président - Rapporteur
	J.L. DURIEUX	
	L. FERAUD	
	B. LECUSSAN	Directeur de thèse
	G. RUGGIU	
	P. SALLE	Rapporteur
	J.P. SANSONNET	Rapporteur

Résumé : Cette thèse traite de la parallélisation dans le cadre fonctionnel et des interactions entre granularité, taux d'activité et rendement des processus. Le modèle servant de toile de fond à notre étude est MaRS (Machine à Réduction Symbolique) développé au CERT-ONERA qui se caractérise entre autres choses par des processus à coût particulièrement faible mais en nombre limité.

L'objectif de cette thèse est de dériver par une critique constructive du modèle MaRS un nouveau modèle au champ d'application étendu. Pour cela une première partie est consacrée à la présentation du cadre fonctionnel et du modèle MaRS dont nous exhibons les caractéristiques les plus marquantes. La deuxième partie montre l'influence du coût d'accès aux données sur le taux de parallélisme effectif et expose l'implémentation performante sous forme combinatoire de diverses structures de données. La troisième partie traite de l'influence du taux d'activité des processus sur le taux de parallélisme effectif et étudie deux approches qui visent à augmenter le taux de parallélisme effectif en diminuant le taux d'attente des différents processus. Enfin, la quatrième partie analyse le paradigme pipe-line qui, bien qu'étant probablement le plus expressif et aisé à utiliser dont dispose le programmeur fonctionnel, s'avère un bien piètre outil de parallélisation de par la faible granularité des processus généralement engendrés. Une nouvelle approche de la synchronisation entre producteur et consommateur est proposée qui permet, tout en conservant intact l'aspect algorithmique du paradigme, de franchir ce cap de la granularité. Des exemples illustrent les différents mécanismes de synchronisation proposés et en particulier l'algorithme de recherche IDA*.

Mots-clés : Parallélisation explicite, algorithmique fonctionnelle paresseuse, structures de données, synchronisation inter-processus, paradigme pipe-line.

Abstract : This thesis deals with parallelizing in a functional framework, and analyzes interaction between granularity, activity rate and process efficiency. The underlying model is MaRS (Machine à Réduction Symbolique), which has been developed at CERT-ONERA and is featured with, among other things, processes at a particularly low cost but available in a limited number.

The purpose of this thesis is to induce an extended application field model through a MaRS constructive review. The first part is dedicated to the functional framework and the MaRS model, giving its main features. The second part shows the influence of data access cost on the actual parallelism rate and presents an efficient way of implementing various data structures, using a combinatory form. The third part is about the influence of processes activity rate on the actual parallelism rate and studies two methods to raise the later by lowering the waiting rate of the various processes. The fourth part at last, analyzes the pipe-line paradigm, which, in spite of probably being the most expressive and easy-to-use parallelizing tool available to the functional programmer, appears to be quite poor, by mainly generating low granularity processes. A new approach for synchronization between producer and consumer, which leads to a better granularity while remaining the algorithmic aspect of the paradigm unchanged, is provided. Some examples suggesting various synchronization mechanisms, including the 'IDA*' searching algorithm, are shown.

Keywords : Explicit parallelization, lazy functional programming, data structures, inter-process synchronisation, pipe-line paradigm.

Je remercie Monsieur *Luc BOUGE*, professeur à l'Ecole Normale Supérieure de Lyon, pour m'avoir fait l'honneur de présider ce jury.

Je tiens à remercier Monsieur *Luc BOUGE*, professeur à l'Ecole Normale Supérieure de Lyon, Monsieur *Jean-Paul SANSONNET*, directeur de recherches CNRS à Orsay et Monsieur *Patrick SALLE*, professeur à l'Institut National Polytechnique de Toulouse, de s'être intéressé à mon travail et d'avoir accepté de rapporter cette thèse. Leurs remarques constructives et convergentes m'ont été très profitables.

Je remercie également Monsieur *Jean-Louis DURIEUX*, Professeur à l'Université Paul Sabatier Toulouse III, Monsieur *Louis FERAUD*, Professeur à l'Université Paul Sabatier Toulouse III, Monsieur *Bernard LECUSSAN*, Professeur à l'Ecole Nationale Supérieure de l'Aéronautique et de l'Espace et Monsieur *Gilles RUGGIU*, directeur de l'informatique de BERTIN & Cie à Plaisir, d'avoir participé à ce jury.

C'est à *Bernard LECUSSAN*, qui a encadré ce travail au mieux de ses disponibilités, que je dois d'avoir travaillé sur le projet MaRS. Je ne peux que l'en remercier vivement.

Je remercie également *Michel LEMAITRE*, ingénieur au CERT-ONERA pour son fastidieux travail de relecture et pour ses conseils bénéfiques et constants, mais aussi bien sur tous ceux qui font ou ont fait partie de l'équipe MaRS : *Manuel CARCENAC, Alessandro CONTESSA, Eric COUSIN, Michel CUBERO-CASTAN, Marie Hélène DURAND, Guy DURRIEU, Bernard LECUSSAN, Michel LEMAITRE, Michel MAGNIN, Daniel MARRE, Paulino NG, Roland PAINDAVEINE, Richard PORCHE* et *Dominique VIZCAINO* qui, chacun dans son domaine, ont su créer l'environnement technique et convivial propice au travail de chacun.

Je suis également très reconnaissant à *René JACQUART* de m'avoir reçu au sein du Département d'Etudes et de Recherches en Informatique du CERT.

Je remercie enfin tous ceux que j'ai eu la chance de rencontrer au CERT au cours de ces années, les secrétaires du DERI *Noëlle DESBLANCS Florence GAIARIN* et *Christiane PAYRAU*, le personnel administratif, l'ensemble des ingénieurs et stagiaires, *Jean-Paul NIGOUL* du service d'édition, et tous les autres, en commençant par ceux avec qui j'ai partagé mes bureaux successifs, qui font de ce lieu un cadre de travail privilégié.

Introduction 9

Chapitre 1 Le cadre 13

1.1 La programmation fonctionnelle 13
1.1.1 Présentation 13
1.1.2 Les différentes sortes de glu 14
1.1.3 le parallélisme 17
1.2 Première présentation de MaRS et de MaRS_Lisp 18
1.2.1 Bref historique 18
1.2.2 Le langage MaRS_Lisp 18
1.2.3 Le modèle d'exécution 19
1.3 Le langage MaRS_Lisp 21
1.3.1 Partition des objets de base 23
1.3.2 Syntaxe 23
1.3.3 Quelques primitives utiles 24
1.3.4 Règles d'évaluation 26
1.4 Les combinateurs 27
1.4.1 Les principales familles de combinateurs MaRS 27
1.4.2 Le micro-codage par champs 29
1.5 La compilation 29
1.5.1 Le bon parallélisme 31
1.5.2 Le mode 'parallélisation automatique' du compilateur MaRS_Lisp 33
1.5.3 Le mode 'parallélisation manuelle' du compilateur MaRS_Lisp 34
1.5.4 Le passage de paramètres 34
1.5.5 Compilation et stricticité 35
1.5.6 Compilation de l'évaluation paresseuse 35
1.6 La réduction de graphe 37
1.6.1 La synchronisation 37
1.6.2 Implications de la synchronisation au niveau des processus 37
1.7 L'architecture MaRS 39
1.7.1 Généralités 39
1.7.2 Les caractéristiques du processus 41
1.7.3 La récupération mémoire 42
1.8 Le simulateur 43
1.8.1 Généralités 43
1.8.2 Les statistique d'exécution 43
1.8.3 Le profil parallèle 43
1.8.4 Le mode trace 43
1.8.5 Différences principales par rapport au modèle 44
1.8.6 Conséquences 45
1.9 Acquis et limites de la parallélisation MaRS_Lisp 45
1.9.1 Les moyens de la parallélisation 45
1.9.2 Quelques objectifs poursuivis lors de parallélisations 46
1.9.3 Conclusions relatives à la parallélisation 46
1.10 Objectifs de cette thèse 47

Chapitre 2 Des structures de données en logique combinatoire 49

2.1 Nécessité 49

2.2 Les valeurs multiples 50

2.2.1 Description 50
2.2.2 Une solution simple 54
2.2.3 Une première approche combinatoire 54
2.2.4 Une meilleure approche combinatoire 56
2.2.5 Conclusion 59

2.3 Les structures 59

2.3.1 Description 59
2.3.2 Une solution simple 60
2.3.3 Une solution combinatoire 60

2.4 Les vecteurs 62

2.4.1 Le vecteur 62
2.4.2 L'opérateur vref 62
2.4.3 L'opérateur vset 63

2.5 Conclusion 64

Chapitre 3 Augmenter le parallélisme effectif par diminution des phases de synchronisation 65

3.1 Le coût des processus en programmation fonctionnelle 65

3.1.1 Le problème général 65
3.1.2 Les particularités du modèle MaRS 66
3.1.3 Eléments de solutions 67

3.2 Contrôler le déclenchement des processus 69

3.2.1 Les moyens du contrôle 69
3.2.2 Conséquences du contrôle 70
3.2.3 Le coût de la parallélisation 71
3.2.4 Une approche optimisante 72
3.2.5 Conséquences sur la parallélisation 74

3.3 Exemple 74

3.3.1 Le programme 75
3.3.2 Expérimentations 76
3.3.3 Conclusion 78

3.4 Une source d'inactivité : la recomposition de résultats 79

3.4.1 Un cadre pour résoudre ce problème 81
3.4.2 Sémantique de p-program 90
3.4.3 Un nouveau typage 91
3.4.4 Déterminisme 96
3.4.5 Discussion 98
3.4.6 Conclusion 99

3.5 Un exemple significatif : IDA* 99

3.5.1 Présentation du problème 99
3.5.2 Présentation de l'algorithme 101
3.5.3 La solution fonctionnelle 102
3.5.4 La solution avec p-program 103
3.5.5 Conclusion 106

3.6 Conclusion 108

Chapitre 4 Le paradigme pipe-line revu et corrigé 109

4.1 Le paradigme pipe-line 109
4.1.1 Généralités 109
4.1.2 Description du paradigme 111
4.1.3 Plusieurs flots en sortie 116
4.1.4 Mise en œuvre 120
4.2 Critique du paradigme 120
4.2.1 Flots paresseux 120
4.2.2 Flots industrieux 120
4.3 Un multi pipe-line à coût minimum 121
4.3.1 Origine du coût 121
4.3.2 Gain de performances par augmentation de la granularité : Exemple 122
4.4 Concilier évaluation stricte et effet pipe-line 124
4.4.1 Le principe 124
4.4.2 Les opérateurs de construction de streams 125
4.4.3 Interprétation en terme de listes 126
4.4.4 Premier exemple d'utilisation d'un stream 126
4.4.5 Le système de types 127
4.4.6 Typage des streams 128
4.4.7 Déterminisme 129
4.4.8 Persistance des streams 130
4.4.9 Propriétés 133
4.4.10 Réalisation 134
4.4.11 Exemple d'utilisation des streams : le crible d'Eratosthène 138
4.4.12 Mise en œuvre d'un multi pipe-line 140
4.5 Conclusion 145

Conclusion 147

Annexe Compilation des structures 151

Bibliographie 161

Liste des figures

Chapitre 1
Exemple de graphe 2
Exemple de graphe binaire 20
Exemple de séquence de réduction 21
Une autre étape de réduction 21
Les objets MaRS_Lisp 23
Syntaxe MaRS_Lisp 23
Combinateurs de base de MaRS 27
Quelques exemples d'utilisation de la primitive cons 29
Cycles des états du graphe 37
Architecture simplifiée de MaRS 39
Le réseau de communication 39
Exemple de profil parallèle 43

Chapitre 2
Mauvaise parallélisation d'un multiple-value-bind 53

Chapitre 3
Fibonacci en QLISP avec génération contrôlée du parallélisme 70
Le programme cecilia 75
Composition synchronisée 79
Composition commutative non synchronisée 80
La fonction de fibonacci en style MaRS_Lisp impératif 82
Une session SML imaginaire 91
L'espace de recherche envisagé par IDA* 102
Version fonctionnelle parallèle classique de IDA* 102
Résultats d'exécution de la solution fonctionnelle 102
Le programme avec des variables modifiables 103
Types des principales définitions 106
Résultats d'exécution de la solution avec variables affectables 106

Chapitre 4
Le traitement pipe-line 109
Consommateur bloqué en attente du producteur 110
Producteur et consommateur actifs 110
Parallélisme industrieux 113
Parallélisme ET 114
Consommateur bloqué en attente du producteur 135
Producteur et consommateur actifs 135
Les trois étapes du producteur 137
Le crible d'Eratosthène à base de listes 139
Résultats d'exécution du crible d'Eratosthène à base de listes 139
Le crible d'Eratosthène à base de streams 139
Résultats d'exécution du crible d'Eratosthène à base de streams 139
Le quicksort à base de listes 141
Le quicksort à base de streams 142

Introduction

Les premiers langages de programmation évolués (tels que FORTRAN ou COBOL encore utilisés de nos jours), directement descendants de langages assembleur, étaient calqués sur l'architecture von Neumann qu'ils commandaient. Cette filiation était due d'une part à une continuité logique avec les assembleurs et d'autre part à des raisons de facilité de traduction et d'efficacité.

Ainsi est apparue la première famille de langages de programmation : les langages impératifs basés sur la modification d'une mémoire.

Plus tard, avec l'écriture de programmes de plus en plus imposants, il s'est avéré que le style impératif comportait de gros défauts aussi bien dans la capacité à écrire des programmes lisibles et maintenables que dans la réutilisation de code précédemment développé *[BAKUS 78]*.

Certains de ces défauts n'étant pas partagés par les paradigmes de la programmation fonctionnelle, l'intérêt des chercheurs s'est alors porté dans cette direction *[HENDERSON 80]*, *[ABELSON 85]*, *[PEYTON-JONES 87b]*, *[BIRD 88]* et parallèlement sont apparus de nouveaux langages tel que ML ou plus récemment MIRANDA et HASKELL.

Ensuite le besoin de puissance et les capacités de la technologie ont poussé les chercheurs à explorer le parallélisme; les limitations des langages impératifs dans le domaine de la productivité et de la qualité n'en sont alors apparues que plus fortes.

En réaction à ces limitations, les années 80 ont vu logiquement naître de nombreux projets de recherche en programmation fonctionnelle parallèle, tels que *[DARLINGTON 81]*, *[HALSTEAD 85]*, *[HUDAK 86]* ou *[MOHR 90]*.

Parmi ces projets, le projet MaRS (Machine à Réduction Symbolique) a vu le jour en 1985 et a d'abord été centré sur l'étude des langages purement fonctionnels *[DURAND 86]*. Ensuite, le projet s'est orienté vers la réduction de graphe comme modèle d'exécution pour sa continuité avec le lambda-calcul, base théorique des langages fonctionnels *[MaRS 86b]*, *[MaRS 87b]*. Enfin et grâce aux progrès récents en CAO VLSI, les architectes ont pu définir une architecture multiprocesseur originale supportant le modèle d'exécution et réalisée à l'aide de processeurs spécifiques *[MaRS 86c]*, *[CASTAN 88]*.

Les particularités du modèle MaRS qui intéressent particulièrement le programmeur sont d'une part que les espaces d'adressage sont uniques (les processus ne voient qu'une mémoire, la mémoire ne voit qu'un ensemble de processus), et d'autre part que les processus ont la caractéristique peu banale d'être à coût de gestion minimal mais en nombre (fortement) limité, ce qui à de grandes répercussions sur la façon de *bien* paralléliser.

C'est dans ce cadre qu'ont déjà été réalisées de nombreuses études sur la parallélisation, parmi lesquelles nous pouvons mettre en avant la réalisation d'un compilateur-paralléliseur automatique (parallélisation implicite) *[COUSIN 91]* et l'étude d'un certain nombre de méthodes de parallélisation *à la main* (parallélisation explicite) basée sur l'utilisation de paradigmes tels que diviser pour régner ou le pipe-line *[MARRE 92]*.

Si les résultats apportés par ces études sont indéniables, il reste néanmoins des classes d'algorithmes qui ne peuvent en bénéficier que très partiellement : il s'agit des algorithmes nécessitant des structures de données plus complexes que de simples listes, d'algorithmes mettant en jeu des synchronisations inter-processus implicites contraires à une bonne utilisation des ressources processus, ou même d'une grande partie des algorithmes admettant une expression satisfaisante à l'aide du paradigme pipe-line mais qui souffrent alors d'un manque flagrant d'efficacité lié à la granularité excessivement faible des processus générés.

L'objectif de cette thèse est donc de montrer que notre modèle de réduction et ses choix architecturaux originaux, loin de devoir en limiter la portée à un sous-ensemble de problèmes, constitue un cadre dans lequel la majorité des problèmes parallélisables trouvent une expression satisfaisante.

Ainsi le Chapitre 2 est consacré à l'introduction de nouvelles structures de données (valeurs multiples, structures - *record*- et vecteurs), basée sur l'interprétation de ces structures en tant qu'objet actif, c'est à dire en tant que fonction, et sur leur expression en terme de lambda-expressions.

Le Chapitre 3 traite de deux façons de diminuer le coût associé aux processus en attente : introduire des primitives de création dynamique de processus, aptes à adapter leur comportement à l'évolution des calculs, et introduire de nouvelles constructions parallèles, syntaxiquement semblables à de la programmation impérative mais conservant la tranparence du référentiel, qui permettent de s'affranchir de certaines synchronisations de processus. Dans les deux cas, le but poursuivi sera l'augmentation de l'acti-

vité parallèle du programme par diminution de la part relative des processus en attente.

Le Chapitre 4 traite du paradigme pipe-line, qui est très expressif mais dont le rendement en terme d'augmentation du parallélisme effectif est compromis par le très grand nombre de processus qu'il met en œuvre, et présente une nouvelle approche visant à conserver l'effet pipe-line tout en ne mettant en œuvre qu'un seul processus par traitement.

Enfin, dans la Conclusion nous essaierons de faire la synthèse de ce travail et de présenter les perspectives qu'il peut offrir.

Chapitre 1

Le cadre

Objectif : Présentation du cadre, en particulier des mécanismes essentiels de la programmation fonctionnelle parallèle et des grandes lignes de la machine MaRS et de MaRS_Lisp son langage source. Présentation de la parallélisation en MaRS_Lisp.

1.1 La programmation fonctionnelle

1.1.1 Présentation

La programmation fonctionnelle doit son nom à ce qu'elle est basée sur les fonctions (au sens mathématique) : les parties atomiques d'un programme sont des fonctions ou des constantes (qui ne sont rien d'autre que des fonctions sans argument), le reste c'est à dire la façon dont est organisé le programme faisant partie des mécanismes mathématiques classiques en matière de fonctions (application, composition, abstraction, ...).

Pour décrire l'ensemble de ces mécanismes, qui assurent la cohésion entre les diverses parties d'un programme, nous reprendrons la terminologie utilisée dans *[HUGHES 90]* et parlerons de glu. La puissance d'un paradigme de programmation, tel que la programmation fonctionnelle, peut s'appréhender à travers la puissance de sa glu. C'est en effet sur celle-ci qu'est basé le raisonnement du programmeur (ou du moins qu'il devrait l'être), et non sur les caractéristiques de telle ou telle primitive particulière.

Pour ne plus laisser planer de doute précisons que la partie du paradigme fonctionnel qui nous intéressera dans toute la suite de cette thèse est la programmation purement fonctionnelle également appelée applicative *[BACKUS 78]*.

Celle-ci se caractérise par une totale absence d'effet de bord ce qui a pour conséquence immédiate de garantir que tout appel de fonction retourne le

même résultat à chaque appel. Cette propriété connue sous le nom de transparence du référentiel est à la base même de la puissance de la programmation fonctionnelle : grâce à elle nous pouvons garantir qu'une fonction testée indépendamment du reste du programme conservera un même fonctionnement une fois intégrée, ainsi que dans toute utilisation ultérieure dans un autre programme (c'est pour cela que la programmation fonctionnelle est louée pour sa réutilisabilité).

En plus de cela, la mise au point elle-même des fonctions est facilitée par cette absence de référence à une mémoire dont le programmeur ne sait jamais trop ce qu'elle pourra effectivement contenir lors d'une exécution réelle.

1.1.2 Les différentes sortes de glu

Les deux sortes de glu les plus importantes de la programmation fonctionnelle sont la paresse et les fonctions d'ordre supérieur (voir par exemple *[HUGUES 90]*). Une excellente introduction à ces deux notions se trouve dans *[BIRD 88]*.

Avant de présenter l'évaluation paresseuse, nous commencerons par présenter les notions de stricticité, d'évaluation par valeur et d'évaluation par nom.

1.1.2.a Stricticité

Le qualificatif *strict* s'emploie pour un paramètre d'une application : une sous-expression e_i d'une expression $(f\ e_1 \ldots e_{i-1}\ e_i\ e_{i+1} \ldots e_n)$ est dite stricte si sa définition est nécessaire à la définition de toute l'expression. Plus formellement e_i est stricte dans $(f\ e_1 \ldots e_{i-1}\ e_i\ e_{i+1} \ldots e_n)$ si et seulement si $\forall\ x_1 \ldots x_{i-1}\ x_{i+1} \ldots x_n, (f\ x_1 \ldots x_{i-1}\ \perp\ x_{i+1} \ldots x_n) = \perp$ où $\perp$ représente l'indéfini, c'est à dire un calcul qui ne termine pas.

La stricticité de e_i dans $(f\ e_1 \ldots e_{i-1}\ e_i\ e_{i+1} \ldots e_n)$ ne dépend en aucune manière de sa valeur mais uniquement de la définition de la partie fonction de cette application, c'est à dire de la définition de f. Une approximation sûre de la stricticité peut être inférée à partir d'informations sur les opérateurs primitifs du langage, voir par exemple *[PEYTON-JONES 87]*, *[WADLER 87b]* et *[WADLER 87c]*.

Comme exemple d'arguments stricts et non stricts nous prendrons la classique fonction `IF` : le prédicat qui détermine quelle branche constituera la suite des calculs est indispensable à l'évaluation du `IF`, et est donc strict, alors qu'aucune des deux branches du `IF` n'est a priori indispensable puisqu'une des deux ne sera pas utilisée. La primitive `IF` est donc stricte en son premier

argument et non stricte en ses deux autres, ce qui peut s'établir formellement en exhibant :

$$\forall\ e_1\ e_2,\ (\mathrm{IF}\ \bot\ e_1\ e_2) = \bot$$
$$\forall\ e,\ (\mathrm{IF\ True}\ e\ \bot) = e$$
$$\forall\ e,\ (\mathrm{IF\ False}\ \bot\ e) = e$$

1.1.2.b L'évaluation par valeur

La conséquence première de la stricticité d'un argument est qu'il peut être évalué n'importe quand lors de l'évaluation de l'expression qui l'englobe puisqu'il sera de toute façon nécessaire à un moment du calcul. Par exemple l'évaluation de l'argument peut être faite avant même d'appliquer la fonction à ses arguments, ce qui correspond à un appel par valeur tel qu'il est pratiqué dans la majorité des langages conventionnels.

1.1.2.c L'évaluation par nom

A l'opposé de l'évaluation par valeur, l'évaluation par nom concerne les arguments non stricts d'une application : comme il n'est pas certain que leur valeur soit nécessaire pendant le calcul, leur évaluation préalable n'est pas du tout justifiée. La stratégie d'évaluation convenable pour de tels arguments est alors l'évaluation par nom qui consiste à n'évaluer l'argument que lorsque et chaque fois que sa valeur devient nécessaire à la poursuite des calculs.

La limitation principale de cette technique est de conduire à évaluer plusieurs fois les arguments qui sont effectivement utilisés plusieurs fois.

1.1.2.d L'évaluation paresseuse

Il s'agit d'une optimisation de l'appel par nom qui consiste à utiliser un partage des expressions passées en paramètre de manière à garantir qu'elles ne sont évaluées qu'au plus une fois.

L'avantage d'une telle stratégie par rapport à l'évaluation par valeur est d'économiser l'évaluation de tous les arguments non-stricts qui s'avèrent inutiles à l'obtention du résultat final, ceux qui s'avèrent nécessaires étant simplement évalués à un instant différent mais une seule fois.

1.1.2.e L'évaluation par nécessité

L'évaluation par nécessité, dont une apologie peut être trouvée dans *[HUGHES 90]*, consiste à utiliser l'évaluation paresseuse pour le passage des paramètres et des constructeurs de structures de données (essentiellement la primitive `cons`) paresseux.

Cette dernière caractéristique est simplement la possibilité de construire une structure dont les constituants ne sont évalués qu'à l'instant où ils deviennent nécessaires à la poursuite des calculs.

Pour en faire une des glus majeures du paradigme fonctionnel, il faut que l'évaluation paresseuse ait de grandes implications dans l'écriture des algorithmes. Voyons donc quelles en sont les conséquences :

Grâce au fait que les éléments d'une structure de données virtuellement infinie (la liste des entiers par exemple) ne seront pas évalués avant leur nécessité, il est possible de définir des objets mathématiquement, c'est à dire sans se préoccuper de leur finitude.

Cette caractéristique fort inhabituelle pour qui n'est familier que de langages à sémantique stricte passe couramment pour un gadget sans application pratique. Pourtant diverses études, par exemple *[CARCENAC 90]* et *[CARCENAC 91]* pour ce qui nous concerne, montrent que la paresse permet d'envisager la programmation de nombreux algorithmes de façon totalement nouvelle : un domaine à parcourir (par exemple un champ de potentiel à échantillonner) sera décrit de manière équationnelle sans préoccupation de quantification, de limitation d'étendue ou de précision et seul le déroulement des calculs déterminera les parties du modèle qui seront réellement évaluées.

Plus important encore, le principe d'évaluation paresseuse permet par définition récursive du modèle (un peu comme un objet fractal) d'écrire un modèle qui se construira au cours des calculs juste au niveau de précision nécessaire à l'obtention d'une certaine précision du résultat global.

Enfin, le fait de décrire un objet tel qu'il est, à l'aide de sa définition mathématique, est une immense simplification par rapport à l'écriture d'un maillage ou de tout autre description spatiale discrétisée le représentant.

1.1.2.f Les applications partielles

Tout opérateur ou fonction est caractérisé par son arité, c'est à dire son nombre d'arguments. Une application d'un opérateur est dite partielle lorsque le nombre d'arguments effectivement appliqués est inférieur à l'arité de l'opérateur.

1.1.2.g Les fonctions d'ordre supérieur

Le terme de fonction d'ordre supérieur recouvre les fonctions qui prennent un argument de type fonction ou retournent une fonction comme résultat. Malgré la terminologie imposante, cette notion est d'une très grande simplicité. Par exemple un opérateur de dérivation qui prend une fonction en argument et calcule sa fonction dérivée est une fonction d'ordre supérieur.

L'utilisation de fonctions d'ordre supérieur sans restriction ou complication syntaxique est un outil très puissant qui permet l'écriture de véritable schémas généraux dont il est ensuite possible de tirer une instance particulière par simple application partielle.

Par exemple, si nous écrivons une fonction d'ordre supérieur `trier` qui prends une relation d'ordre (par exemple < ou ≥) et une liste de valeurs comme arguments et qui retourne la liste triée en résultat, nous pouvons définir une fonction `trier>` qui trie par ordre croissant simplement par application partielle de `trier : trier> = (trier >)`.

Un tel mécanisme permet une factorisation de code maximum ainsi que la définition d'opérateurs aussi généraux et réutilisables que possible.

De plus les fonctions d'ordre supérieur en conjonction avec les propriétés formelles du lambda-calcul permettent d'établir de nombreuses propriétés générales utilisables dans des preuves ou dérivations de programmes en tant que simples théorèmes (voir par exemple *[BIRD 88]* ou *[LEMAITRE 90]*).

1.1.3 le parallélisme

Le choix du parallélisme dans de nombreux projets de recherche en programmation applicative est motivé par un besoin croissant de puissance que ne peuvent satisfaire ni les machines monoprocesseur ni même les machines parallèles conventionnelles, telles que les machines vectorielles, qui sont consacrées au calcul numérique intensif et ne sont d'aucune utilité dans les domaines d'applications habituels de la programmation fonctionnelle.

La programmation applicative constitue un cadre privilégié dans la mise en œuvre du parallélisme. Par exemple une simple analyse de stricticité permet de détecter des opportunités sûres d'appel par valeur, qui sont également de bonnes opportunités de déclenchement de parallélisme à partir d'un programme séquentiel, en ce sens que les processus ainsi lancés participent effectivement au calcul du résultat (parallélisation implicite). Une autre approche non exclusive consiste à laisser le programmeur désigner les expressions qu'il désire voir évaluer par valeur ou en parallèle (parallélisation explicite).

Dans tous les cas la parallélisation sera immensément facilitée par l'assurance que deux expressions indépendantes ne peuvent se perturber quelque soit leur ordre d'évaluation relatif, y compris parallèle.

Cette facilité se retrouve dans la majorité des langages fonctionnels dans lesquels la désignation d'une expression à évaluer en parallèle se fait sans préciser sur quel processeur doit être lancée la tâche correspondante, ni les moyens de synchronisation avec le processus père.

Par exemple lancer l'évaluation de la sous-expression e_i en parallèle avec le reste de l'expression `(f` e_1 ... e_i ... e_n`)` se fera par :

`(f` e_1 ... `(future` e_i`)` ... e_n`)` en Multilisp *[HALSTEAD 85]* et en PaiLisp *[ITO 89]* et par :

`(f` e_1 ... `(processus` e_i`)` ... e_n`)` en MaRS_Lisp *[MaRS 91b]*.

1.2 Première présentation de MaRS et de MaRS_Lisp

1.2.1 Bref historique

Le projet MaRS (Machine à Réduction Symbolique) a vu le jour en 1985 et a d'abord été centré sur la famille des langages purement fonctionnels dont l'expressivité, l'économie de concepts et l'aptitude au parallélisme en font les candidats idéaux dans le cas d'une machine parallèle *[DURAND 86]*.

Ensuite, et tout en développant MaRS_Lisp dialecte Lisp purement fonctionnel, nous avons retenu la réduction de graphe comme modèle d'exécution pour sa continuité avec le lambda-calcul, base théorique des langages fonctionnels *[MaRS 86b]*, *[MaRS 87b]*.

Enfin et grâce aux progrès récents en CAO VLSI, les architectes ont pu définir une architecture multiprocesseur originale supportant notre modèle d'exécution et réalisée à l'aide de processeurs spécifiques *[MaRS 86c]*, *[CASTAN 88]*.

1.2.2 Le langage MaRS_Lisp

Les bases de MaRS_Lisp reposent sur le lambda-calcul et les concepts fondamentaux du langage sont donc semblables à ceux de bien des langages purement fonctionnels. Les points forts du langage seront la modularité, la réutilisabilité et l'existence de bases théoriques fortes permettant de prouver une transformation ou l'équivalence entre un programme et sa spécification. Toutefois, deux choix fondamentaux particularisent MaRS_Lisp : l'évaluation par nécessité et le parallélisme.

L'adoption de l'évaluation par nécessité comme mode d'évaluation par défaut, loin de n'être qu'une sorte de gadget théorique nous permettant d'affirmer que notre langage possède la plus grande définition possible ou est capable de manipuler des listes infinies ("- Pour quoi faire?", pensent certains), a de grandes répercussions sur les performances. Diverses études algorithmiques ont permis de mettre en évidence cette caractéristique souvent insoupçonnée y compris à l'intérieur de la communauté fonctionnelle. Par exemple *[CARCENAC 90]* en montre l'intérêt dans le cas du lancer de rayon, d'autres études étant en cours sur des domaines faisant actuellement appel aux calculateurs vectoriels, tels que les simulations d'écoulements. Nous reviendrons sur la compilation de l'évaluation paresseuse au 1.5.6.

Un autre choix qui caractérise le langage est le parallélisme : les propriétés du lambda-calcul nous garantissant la confluence, toute sous-expression stricte peut être évaluée par valeur séquentielle mais aussi par valeur parallèle sans diminuer la définition du langage. Cette propriété permet une introduction particulièrement simple du parallélisme puisqu'elle peut se faire sans aucune autre précaution que la limiter aux expressions strictes (sous peine d'utiliser des ressources à des calculs inutiles).

Précisons en outre, que bien que la mémoire soit physiquement répartie sur l'ensemble des processeurs mémoire de la machine, elle n'en constitue pas moins une mémoire logiquement partagée par tous les processeurs de réduction (l'espace d'adressage est unique). De ce fait le langage ne comporte aucune construction particulière liée à l'implantation du graphe en mémoire (telles que les annotations présentées dans *[BURTON 84]* et *[HUDAK 86]*), sa localisation étant invisible par le programmeur.

1.2.3 Le modèle d'exécution

Le langage ayant des propriétés théoriques fortes, les initiateurs du projet ont choisi un modèle d'exécution en continuité avec ces propriétés : la réduction de graphe de combinateurs. Les combinateurs qui sont les instructions de base mises en œuvre dans ce modèle sont équivalents à des lambda-expressions[1], cette équivalence permet de prouver formellement la compilation des programmes MaRS_Lisp en code combinatoire ainsi que les différentes passes d'optimisation pratiquées sur le code compilé[2].

La réduction se caractérise par le fait que programme et données ne font qu'un : toute expression subit au cours de son exécution des récritures successives suivant un ensemble de règles (telles que `(first (a . d) --> a)`. Lorsque plus aucune règle n'est applicable, l'expression est dite *irréductible* ou *en forme normale* et cette forme normale est le résultat de l'exécution.

Si l'exécution termine, elle consiste en une suite de récritures à valeur égale qui mènent de l'expression initiale à des expressions équivalentes, non pas forcément de plus en plus simples mais de plus en plus proche du résultat final, et dont la dernière est le résultat du programme. Sinon la suite de récritures se poursuit indéfiniment.

Notons que certaines stratégies d'évaluation peuvent ne pas terminer dans l'évaluation d'une expression, même si celle-ci admet une forme normale. C'est en particulier le cas de l'évaluation par valeur (cf. 1.1.2.b) si elle est appliquée à des arguments non-stricts.

Comme pour le lambda-calcul, nous bénéficions de la confluence et donc toute séquence de réduction qui termine appliquée à l'expression initiale ne peut que donner un seul et même résultat. Cette propriété permet de choisir toute séquence de réduction sûre (c'est à dire qui termine s'il existe un résultat) et en particulier d'effectuer en parallèle plusieurs applications de règle de réduction sur autant de sous-expressions strictes indépendantes.

Parmi les divers modèles de réduction, la réduction de graphe se caractérise par le fait qu'elle permet un partage des expressions ce qui assure qu'une

1. En fait, toute lambda-expression ne contenant pas de variable libre (on parle de lambda-expression fermée) est appelée combinateur.

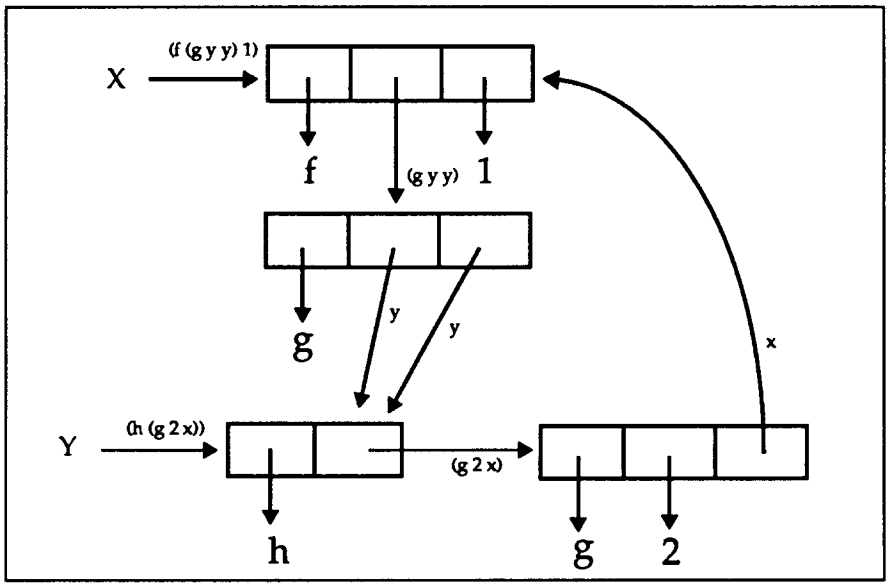
2. Exemple de propriété prouvable formellement : le code MaRS_Lisp compilé a une longueur inférieure à deux fois la longueur du code source *[MaRS 87b]*.

sous-expression ne peut être réduite qu'au plus une fois (si la stratégie de réduction retenue est sûre, une sous-expression n'est évaluée que si elle est nécessaire au résultat).

Du fait de la possibilité de partages, la forme générale de l'expression à réduire est un graphe quelconque dont les feuilles sont les constantes et les noeuds sont les applications des fonctions aux arguments.

Par exemple, nous donnons Figure 1 la représentation sous forme d'un graphe du système d'équations x = (f (g y y) 1) y = (h (g 2 x)).

Figure 1 Exemple de graphe

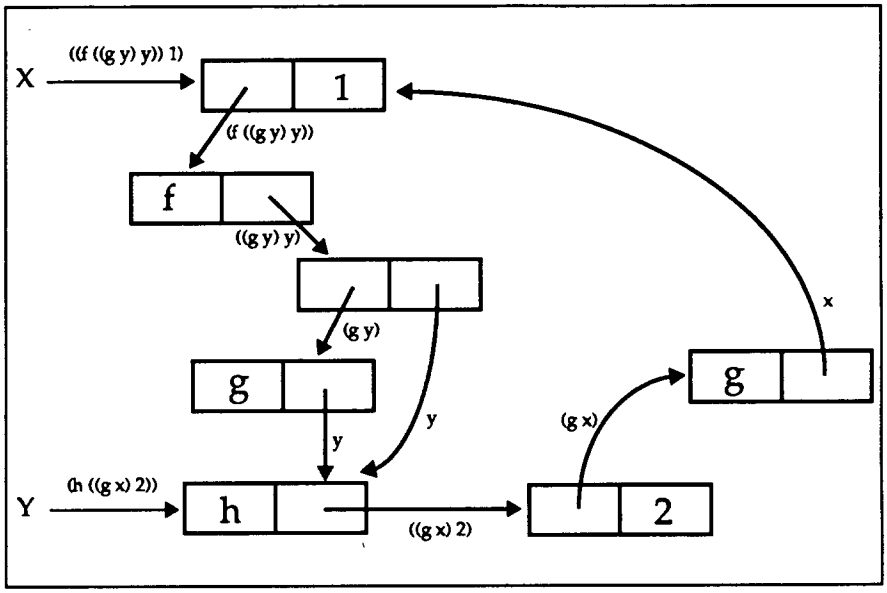


Une telle représentation a le défaut de nécessiter différentes sortes de noeuds (une sorte pour chaque arité d'opérateur), ce qui nous amène à utiliser l'associativité à gauche de l'application pour représenter les expressions uniquement à l'aide de noeuds binaires. Cette convention d'écriture s'énonce $\forall f\ x\ y$, $(f\ x\ y) = ((f\ x)\ y)$, ce qui permet de ne plus utiliser que des noeuds binaires : toute application naire (f e_1 e_2 ... e_n) peut s'écrire à l'aide d'applications binaires en (((f e_1) e_2) ... e_n).

Ainsi notre système d'équations précédent est équivalent à x = ((f ((g y) y)) 1) y = ((h ((g x) 2)) 1) dont le graphe binaire correspondant est représenté Figure 2.

A partir d'un graphe binaire simple nous allons voir les différentes étapes d'une séquence de réduction mettant en œuvre les opérateurs -, +, * et le combinateur d'identité !I.

Figure 2 Exemple de graphe binaire



Nous avons représenté Figure 3 la séquence de réduction (réduction de l'expression `E = (* (- 5 3) (+ 2 1))` en son résultat `6` après trois étapes de réduction faisant appel aux règles de récritures des différents opérateurs) ①⟶②⟶③⟶④, mais la séquence ①⟶②'⟶③⟶④ (cf. Figure 4) est équivalente (les deux séquences ne diffèrent que par l'ordre d'évaluation de leurs sous-expressions) alors que le séquence ①⟶③⟶④ est équivalente par réduction simultanée (en parallèle) des deux sous-expressions `(- 5 3)` et `(+ 2 1)`.

De plus, l'expression `E'` qui a le noeud `@3` en commun avec `E` n'aura jamais à effectuer sa réduction car celui-ci a été récrit en son résultat `(!I 2)` lors du calcul de `E`. C'est grâce à cette conjonction du partage des sous-expressions et de la récriture que toute expression sera réduite au plus une fois.

1.3 Le langage MaRS_Lisp

Nous ne donnerons ici qu'une présentation synthétique de MaRS_Lisp, une présentation plus détaillée (bien qu'imparfaite) est disponible dans *[MaRS 91b]*. Les caractéristiques essentielles du langage sont :

- purement fonctionnel
- évaluation paresseuse
- mise en œuvre simple du parallélisme
- fonctions d'ordre supérieur
- applications partielles implicites

Figure 3 Exemple de séquence de réduction

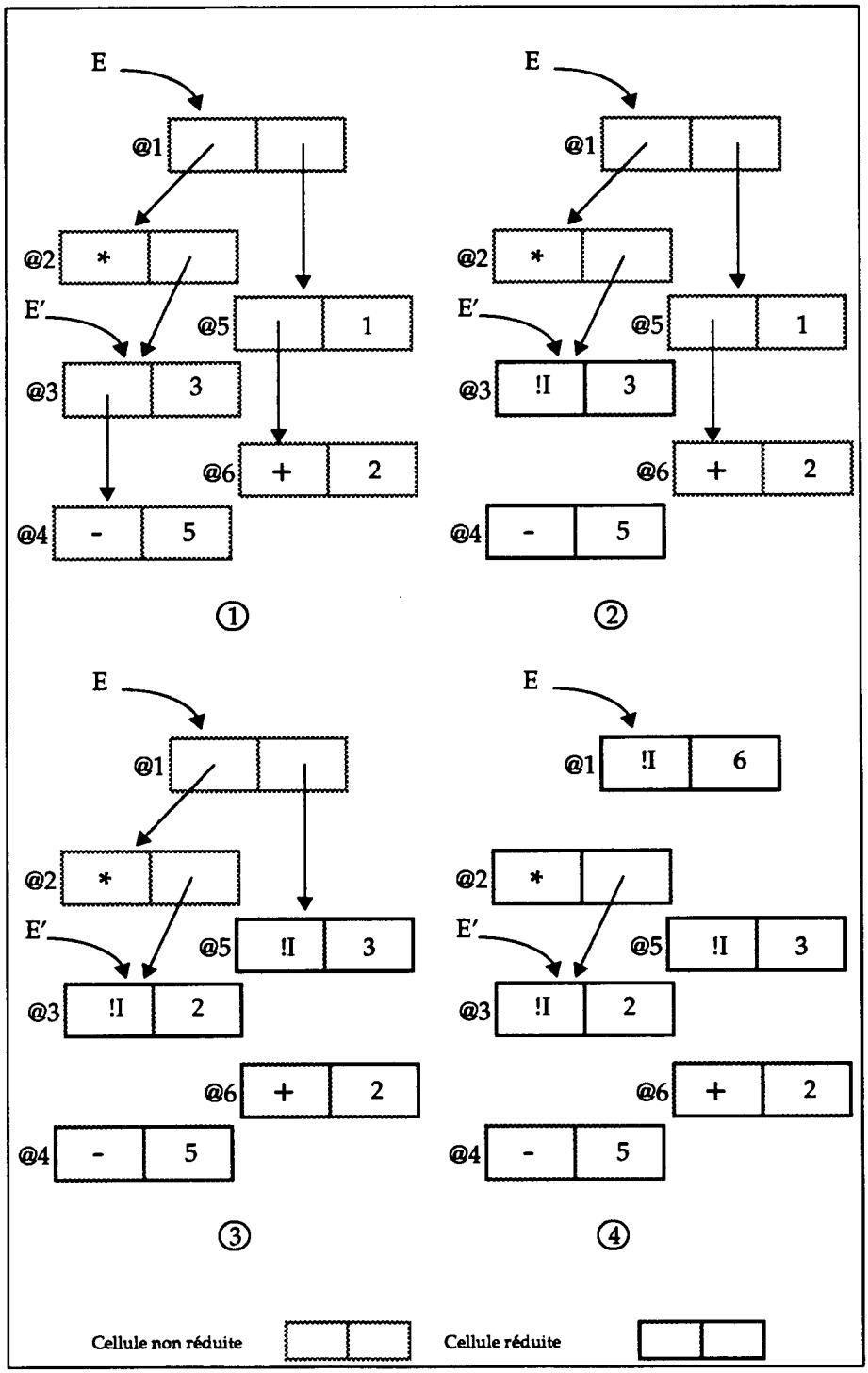
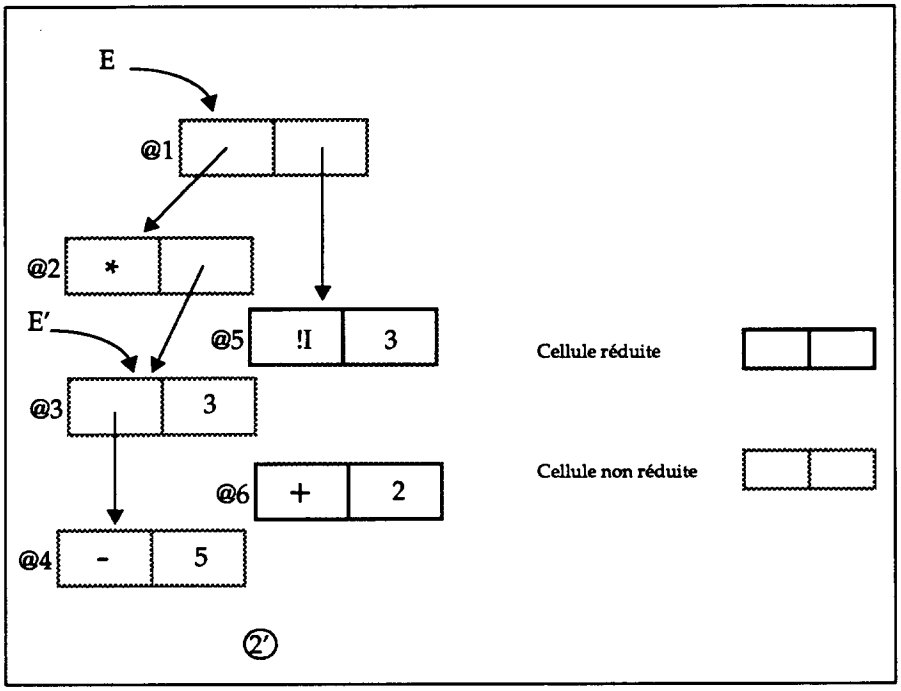


Figure 4 Une autre étape de réduction



- non typé
- syntaxe lispienne

1.3.1 Partition des objets de base

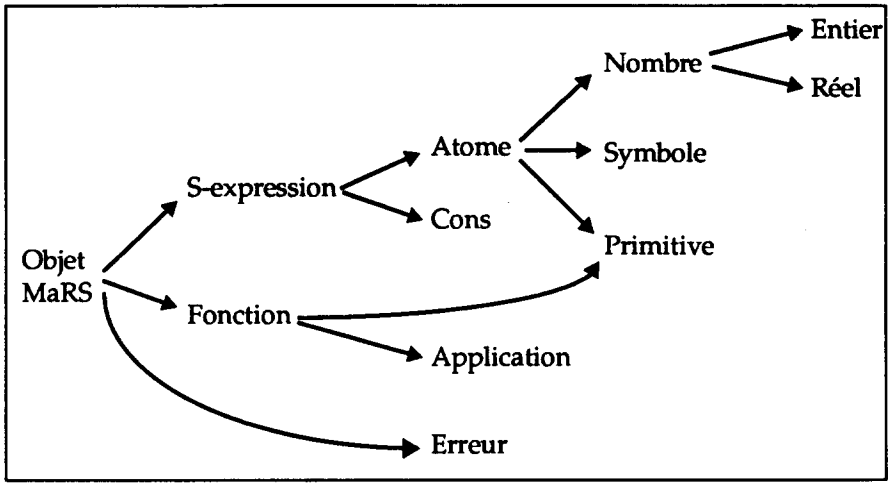
La hiérarchie des objets MaRS_Lisp dont une partition est {atome, cons, application, erreur} est présentée Figure 5.

1.3.2 Syntaxe

La syntaxe qui est présentée Figure 6 est une syntaxe Lisp classique, les seules particularités notables sont :

- Des annotations de stricticité afin de palier à l'absence d'analyse automatique.
- La construction `eager` qui force le compilateur à utiliser localement le mode d'évaluation par valeur séquentielle. Cette construction est utilisée pour optimiser l'exécution en modifiant localement l'ordre d'évaluation (cf. 1.5.6).
- La construction `processus` qui force le compilateur à utiliser localement le mode d'évaluation par valeur parallèle. Cette construction est utilisée pour déclencher explicitement du parallélisme.

Figure 5 Les objets MaRS_Lisp



Nous donnons ci-dessous le programme correspondant à la fonction de Fibonacci avec sa déclaration de stricticité et un déclenchement de processus.

```
(defun fib (n) (declare (strict n))
  (if (<= n 2)
      1
      (* (fib (- n 1)) (processus (fib (- n 2))))))
```

1.3.3 Quelques primitives utiles

Les primitives DEFUN et DEF sont utilisées pour définir un objet global avec ou sans paramètre :

```
(defun identite (x) x)

(def pi 3.14)
```

Les définitions locales sont introduites par LET (liaisons en parallèle), LET* (liaisons en séquence) ou LETREC (définitions mutuellement récursives) :

```
(def pi 3.14)

(let ((a pi) (pi 3) (b pi)) (list a pi b))
= (3.14 3 3.14)

(let* ((a pi) (pi 3) (b pi)) (list a pi b))
= (3.14 3 3)

(letrec ((a pi) (pi 3) (b pi)) (list a pi b))
= (3 3 3)
```

Les opérateurs sur les entiers sont ADD (addition), ADD1 (addition de 1), SUB (soustraction), SUB1 (soustraction de 1), MUL (multiplication), DIV (division

Figure 6 Syntaxe MaRS_Lisp

PROGRAMME	::	({ DEF-GLOBALE | FORME } *)
DEF-GLOBALE	::	(defun VARIABLE LAMBDA-LIST [ANNOTATION] FORME) | (def {VARIABLE FORME}+)
FORME	::	VARIABLE | CONSTANTE | APPLIC-SOURCE | DEF-LOCALE | (processus FORME) | (eager FORME) | (quote S-EXPRESSION)
APPLIC-SOURCE	::	(FORME+)
DEF-LOCALE	::	({let | let* | letrec} ({ (VARIABLE FORME) }*) [ANNOTATION] FORME) | (lambda LAMBDA-LIST [ANNOTATION] FORME)
LAMBDA-LIST	::	({VARIABLE}*)
ANNOTATION	::	(declare (strict {VARIABLE | (VARIABLE {0 | 1}*) }*))
S-EXPRESSION	::	ATOME | ({S-EXPRESSION}*) | ({S-EXPRESSION}+ . S-EXPRESSION)
ATOME	::	NOMBRE | NOM
CONSTANTE	::	t | nil | () | NOMBRE | PRIMITIVE
PRIMITIVE	::	OPERATEUR | COMBINATEUR
VARIABLE	::	NOM, sauf t, nil et les MOT-CLEs (noms de primitives ou constructions)

entière), REM (reste de la division entière), FLOAT (conversion en un flottant), EQN (égalité ?), EQ0 (égalité avec 0 ?), NEQN (non égalité ?), GT (supérieur strict ?), LT (inférieur strict ?), GE (supérieur ou égal ?), LE (inférieur ou égal ?) et leurs équivalents pour les flottants FADD, FSUB, FMUL, FDIV, TRUNCATE (conversion en l'entier dont la valeur absolue est inférieure ou égale), FEQN, FNEQN, FGT, FLT, FGE, FLE.

Les constructeurs de liste sont le classique CONS (paresseux sur ses deux arguments) et les moins classiques XY-CONS (sur lesquels nous reviendrons avec la Figure 8, où X désigne le mode d'évaluation du premier argument et Y celui du deuxième, parmi L 'paresseux', E 'par valeur', et P 'parallèle' : LL-CONS est donc synonyme de CONS). Les opérateurs d'accès sont FIRST pour le premier argument et REST pour la fin de liste, NULL étant le prédicat de liste vide.

La conditionnelle est IF, mais attention, en MaRS_Lisp les applications partielles sont des fonctions : `(if x y)` = `(lambda (z) (if x y z))` et non pas `(if x y nil)` comme c'est le cas par exemple en Common Lisp.

Le prédicat de comparaison est EQ qui détecte l'égalité entre atomes (sauf les nombres, qui se comparent par EQN ou FEQN) ainsi que l'égalité physique entre S-expressions.

Enfin l'opérateur REDUCE, et sa version parallèle P-REDUCE, permettent de nombreuses formes d'itérations sur les listes, ce qui leur a valu d'être directement intégrés au jeu de combinateurs. Leurs définitions sont :

```
reduce f l e
= if (null l) e (f (first l) (reduce f (rest l) e))

p-reduce f l e
= if (null l) e (f (first l) (processus (p-reduce f (rest l) e)))
```

L'opérateur `reduce` correspond à l'opérateur `foldr` de MIRANDA *[BIRD 88]* selon la relation : `reduce f l e = foldr (f) e l`.

Enfin, et pour tenter d'être complets, signalons la possibilité de définir des macros (par DEFMACRO) ou des constantes textuelles (par DEFCONSTANT), ce qui nous permettra par exemple d'utiliser de nouvelles appellations pour les opérateurs arithmétiques (+, −, *, /, =, ≠, >, <, ≥, ≤, ...) ou de définir les fonctions LISP classiques qui nous feraient défaut :

```
(defconstant append '(reduce cons)) ;; append paresseux

(defconstant map ;; map paresseux
  '(lambda (f l)
    (if (null l) nil (cons (f (first l)) (map f (rest l))))))

(defmacro + (a b) '(ADD ,a ,b))
```

1.3.4 Règles d'évaluation

Nous terminerons notre présentation succincte de MaRS_Lisp en donnant les règles d'évaluation d'une expression découlant d'une compilation en mode parallélisation manuelle (cf. 1.5).

Rappelons que le mode d'évaluation par défaut de MaRS_Lisp est l'évaluation paresseuse, mais celui-ci peut être localement remplacé par une évaluation par valeur séquentielle (grâce aux déclarations de stricticité ou à `eager`) ou par une évaluation par valeur parallèle (grâce à `processus`).

Une expression est dite *touchée* lorsque le processus courant demande sa valeur, c'est à dire lorsqu'il va lire un pointeur vers le graphe représentant l'expression.

Lorsqu'une expression `(f` e_1 ... e_n`)` est touchée :

- Un nouveau processus est lancé pour évaluer chaque e_i de la forme `(processus ...)`.

Par exemple dans `(* (fib (- n 1)) (processus (fib (- n 2))))`, le calcul de `(fib (- n 2))` démarre en parallèle dès que l'expression complète est touchée.

- Les expressions `(eager ...)` et tous les e_i stricts (c'est à dire dont le paramètre formel correspondant a été déclaré strict dans la définition de `f`) ne rentrant pas dans le cadre de la première règle sont évalués séquentiellement par le processus courant.

  Par exemple dans `(* (fib (- n 1)) (processus (fib (- n 2))))`, le calcul de `(fib (- n 1))` qui est un argument strict de `*` (la multiplication a besoin de ses deux arguments évalués) est effectué juste après le démarrage du processus sur `(fib (- n 2))`.
- Le processus courant continue par l'évaluation de `f` et par l'application de `f` à ses arguments. Les arguments ne rentrant pas dans le cadre des deux premières règles seront éventuellement évalués lorsqu'ils seront touchés (évaluation paresseuse).

1.4 Les combinateurs

Les combinateurs indicés de MaRS sont initialement inspirés des combinateurs S K I de Shönfinkel *[CURRY 58]* et des combinateurs indexés de Abdali *[ABDALI 76]*. Ils sont conçus comme des opérateurs agissant sur une pile dont le sommet est l'instruction à exécuter.

Les combinateurs font partie du jeu d'instructions du processeur de réduction (l'autre partie étant les primitives classiques telles que +, cons, ...). A chaque famille de combinateurs est associée une règle de réduction décrivant son exécution.

Les combinateurs de MaRS diffèrent d'autres types de combinateurs, tels que les combinateurs de la Categorical Abstract Machine *[COUSINEAU 85]* de par le fait qu'ils conduisent à une exécution par réduction de graphe et non à une exécution sur une architecture von Neumann classique. En ce sens ils sont à rapprocher des super-combinateurs *[HUGHES 82]*, mis à part qu'ils constituent un jeu d'instruction fixe.

1.4.1 Les principales familles de combinateurs MaRS

Nous reproduisons Figure 7 les familles de base de combinateurs telles qu'elles sont présentées dans *[MaRS 86a]*. De nombreuses autres familles existent, que nous utiliserons par la suite en précisant la règle de réduction qui leur est associée, mais celles-ci ne sont pas essentielles à la compréhension du principe.

Cette présentation suggère un indice non borné pour les combinateurs. Toutefois l'indice, qui indique le nombre maximum d'élément de la pile que peut manipuler un combinateur, ne peut raisonnablement être supérieur à la taille physique de cette pile et ce pour des raisons d'efficacité[1]. Dans la machine l'indice est ainsi limité à $Lmax$=32, ce dont tiennent compte les paramètres courants du compilateur.

Figure 7 Combinateurs de base de MaRS

Identité :	$!I\ x \rightarrow x$
Application :	$!B_n\ e_1 \ldots e_n \rightarrow e_1\ (e_2 \ldots e_n)$, pour $n \geq 3$
Déplacement :	$!M_n\ e_1\ e_2 \ldots e_n \rightarrow e_1\ e_n\ e_2 \ldots e_{n-1}$, pour $n \geq 3$
Copie et déplacement :	$!W_n\ e_1\ e_2 \ldots e_n \rightarrow e_1\ e_n\ e_2 \ldots e_{n-1}\ e_n$, pour $n \geq 2$
Déplacement et application :	$!N_n\ e_1\ e_2 \ldots e_n \rightarrow e_1\ (e_2\ e_n)\ e_3 \ldots e_{n-1}$, pour $n \geq 3$
Copie et application :	$!Q_n\ e_1\ e_2 \ldots e_n \rightarrow e_1\ (e_2\ e_n)\ e_3 \ldots e_n$, pour $n \geq 2$
Destruction :	$!K_n\ e_1\ e_2 \ldots e_n \rightarrow e_1\ e_2 \ldots e_{n-1}$, pour $n \geq 2$
Point fixe :	$!Y\ f \rightarrow f\ (!Y\ f)$

Cette limitation peut laisser croire à une limitation de la puissance du modèle. Il est pourtant aisé de montrer le contraire : tout combinateur hors normes produit à la compilation peut être remplacé par une séquence équivalente ne contenant que des combinateurs d'indices inférieurs. Cette propriété a initialement été exhibée lors d'une analyse critique des codes compilés pour des structures de données présentées dans *[COUSTET 90]*. Dans cet article le choix c'est finalement porté sur une compilation ne produisant aucun combinateur hors normes mais le remède subsiste et a été présenté en détails dans *[COUSIN 91]*.

Par exemple, nous pouvons remplacer tout combinateur $!M_n$ où $n > Lmax$ par l'expression combinatoire $(!B_{Lmax}\ !M_{n-Lmax+3}\ !M_{Lmax-1})$, comme le prouve la séquence de réduction qui suit.

$(!B_{Lmax}\ !M_{n-Lmax+3}\ !M_{Lmax-1})\ e_1 \ldots e_n$

$= !B_{Lmax}\ !M_{n-Lmax+3}\ !M_{Lmax-1}\ e_1 \ldots e_n$

$= !M_{n-Lmax+3}\ (!M_{Lmax-1}\ e_1 \ldots e_{Lmax-2})\ e_{Lmax-1} \ldots e_n$;; exécution de !B

$= (!M_{Lmax-1}\ e_1 \ldots e_{Lmax-2})\ e_n\ e_{Lmax-1} \ldots e_{n-1}$;; exécution de !M

$= !M_{Lmax-1}\ e_1 \ldots e_{Lmax-2}\ e_n\ e_{Lmax-1} \ldots e_{n-1}$

$= e_1\ e_n\ e_2 \ldots e_{Lmax-2}\ e_{Lmax-1} \ldots e_{n-1}$;; exécution de !M

$= e_1\ e_n\ e_2 \ldots e_{n-1}$

1. La pile physique du processeur de réduction (de taille bornée) constitue le sommet de la pile d'exécution, le reste de cette pile étant stockée dans la mémoire commune.

Nous venons donc de voir une séquence de réduction qui réalise le même fonctionnement que celui d'un combinateur $!M_n$ où n>*Lmax*. Les combinateurs mis en œuvre sont soit d'indice inférieur à *Lmax*, soit d'indice inférieur à l'indice initial n, et nous pouvons alors appliquer la même substitution autant de fois que nécessaire à l'obtention d' indices tous inférieurs à *Lmax*.

Un mécanisme similaire est possible pour chaque famille de combinateur.

1.4.2 Le micro-codage par champs

Nous disposons en plus des combinateurs de base de la possibilité de définir des combinateurs dérivés *[CASTAN 88]*. Pour cela, il nous suffira d'affecter une valeur (à l'intérieur du domaine autorisé) à chacun des champs qui composent un combinateur. Les principaux champs permettent de désigner l'indice de(s) argument(s) du combinateur, de choisir le mode d'évaluation des arguments (non évaluation, évaluation séquentielle, évaluation parallèle) d'effectuer certains mouvements ou duplications d'arguments dans la pile, et finalement de fixer le micro opérateur exécuté après cette phase de préparation.

Un cas particulier d'application de ce micro-codage est la primitive `cons` si importante dans un langage où la structure de données privilégiée est la cellule cons : un combinateur dont le micro-opérateur serait *micro-cons* et dont tous les autres champs auraient les valeurs par défaut effectuerait un cons doublement paresseux (paresseux sur le `first` et sur le `rest`). Cette primitive `cons` existe et est tout simplement le `cons` de base de MaRS_Lisp. Son nom est soit `cons` soit `ll-cons` (pour *Lazy-Lazy-*`cons` où le premier *Lazy* caractérise le mode d'évaluation du `first` et le second celui du `rest`).

Toutefois en faisant varier les modes d'évaluation du `first` et du `rest` parmi {*Lazy, Eager, Parallel*} nous pouvons construire neuf primitives différentes qui font appel au micro-opérateur *micro-cons* (`ll-cons`, `le-cons`, `lp-cons`, `el-cons`, `ee-cons`, `ep-cons`, `pl-cons`, `pe-cons` et `pp-cons`). L'existence de ces variantes de cons offre de grandes possibilités de parallélisation d'algorithmes *[MARRE 92]*, comme le montre la Figure 8.

Le même mécanisme peut être mis à l'œuvre pour toute combinaison des champs définissant un combinateur, nous utiliserons cette possibilités dans la suite pour coder sous forme de combinateurs *ad-hoc* certaines lambda-expressions de manière efficace.

1.5 La compilation

Avant d'envisager le compilateur MaRS_Lisp *[COUSIN 91]*, qui peut fonctionner selon deux modes (parallélisation automatique ou parallélisation manuelle), nous allons dégager les conditions de déclenchement de bon parallélisme sur la machine MaRS.

Figure 8 Quelques exemples d'utilisation de la primitive cons

```
(defun somme (l) (if (null l) 0 (+ (first l) (somme (rest l)))))
```

```
(defun liste-1 (i n)
  (if (= 0 n) nil (ll-cons i (liste-1 (+ i 1) (- n 1)))))
```

la fonction `liste-1` ne produit pas des nombres évalués mais une liste dont les éléments sont `i`, `(+ i 1)`, `(+ (+ i 1) 1)`, ...

```
(somme (liste-1 0 10)) = 45
```

Construction de la liste doublement paresseuse : les nombres ne sont évalués que lorsqu'ils sont touchés par le + de la fonction somme et la fin de la liste n'est évaluée que lorsqu'elle est touchée par le `rest` de somme.

```
(defun liste-2 (i n)
  (if (= 0 n) nil (el-cons i (liste-2 (+ i 1) (- n 1)))))
```

La fonction `liste-2` produit des nombres évalués, mais la production est paresseuse donc conditionnée au besoin d'un consommateur.

```
(somme (liste-2 0 10)) = 45
```

Construction paresseuse de la fin de liste : les nombres sont évalués par le `el-cons` de `liste-2` et la fin de la liste n'est évaluée que lorsqu'elle est touchée par le `rest` de la fonction somme.

```
(defun liste-3 (i n)
  (if (= 0 n) nil (ee-cons i (liste-3 (+ i 1) (- n 1)))))
```

La fonction `liste-3` produit une liste de nombres de façon stricte, comme dans un LISP standard.

```
(somme (liste-3 0 10)) = 45
```

Fonctionnement doublement industrieux : nombres et fin de liste sont évalués par le `ee-cons` de `liste-3` qui est un cons par valeur classique.

```
(defun liste-4 (i n)
  (if (= 0 n) nil (ep-cons i (liste-4 (+ i 1) (- n 1)))))
```

La fonction `liste-4` produit une liste de nombres évalués au moyen d'un processus par nombre produit.

```
(somme (liste-4 0 10)) = 45
```

Construction parallèle de la liste : chaque nombre est évalué par un `ep-cons` de `liste-4` et la fin de la liste est lancée en parallèle par ce même cons. Ici il y a effet pipe-line car la liste est produite par une succession de processus indépendants du processus qui calcule la somme.

Précisons également une des caractéristiques intéressantes du code compilé : il ne contient aucune référence à la configuration sur laquelle il doit s'exécuter, de ce fait le code est indépendant de la configuration et peut être exécuté sur toute version de la machine comportant au moins un processeur de chaque type.

1.5.1 Le bon parallélisme

1.5.1.a Influence de l'architecture

La *bonne* parallélisation d'un programme dépend intimement des caractéristiques de la machine cible : mémoire commune ou répartie, coût de gestion des processus, débit des communications ...

Aussi allons nous aborder ici les principaux points architecturaux qui ont des implications notables sur la parallélisation MaRS, l'architecture étant présentée plus en détails en 1.7.

En premier lieu, la mémoire est à espace unique d'adressage et les allocations de cellule sont entièrement prises en compte par le réseau. De ce fait nous nous représenterons la mémoire comme une mémoire commune unique.

De la même façon, les allocations de processus sont entièrement gérées par le réseau et les créations de processus se font donc exactement comme s'il n'y avait qu'un seul processeur d'exécution, sans spécifier le processeur d'exécution destinataire.

Il en résulte donc, au niveau de la programmation, une simplification extrême des allocations, aussi bien des cellules mémoire que des processus.

Nous pourrions reprocher à une telle approche de simplifier outrancièrement le modèle en le rapprochant d'un mono-processeur multi-tâche classique. Toutefois cette vision (aisée à mettre en œuvre) est celle que nous voulons présenter au programmeur, profitant ainsi des propriétés de l'architecture : la distance entre un processus et la mémoire est indépendante des processeurs concernés, ce qui rend inutile toute différenciation explicite au niveau de la programmation entre processeurs de même type.

Il découle de ces constatations que des annotations similaires à celles présentées dans *[BURTON 84]* ou *[HUDAK 86]* (et qui visent à expliciter le découpage des programmes sur l'architecture cible) ne sont absolument pas justifiées pour MaRS. De la même façon, le code source (pas plus que le code compilé) ne fait pas référence à la configuration de la machine sur laquelle il doit être exécuté.

Par contre, les processus ont une caractéristique peu commune : les choix architecturaux ont conduits à des processus non préemptibles dont le coût de gestion est minimal, mais cela au prix d'un nombre limité de processus allouables.

De ce fait, la préoccupation première lors de la parallélisation sera bien plus de déclencher des processus à fort taux d'activité (car un processus alloué mais non actif bloque une ressource processus) que des processus à gros grain.

Le premier corollaire de ce principe est de rejeter toute parallélisation spéculative, contrairement à ce qui est possible pour des modèles disposant de processus interruptibles en nombre non borné *[BURTON 86]*.

1.5.1.b Pourquoi ne pas mettre des processus partout?

La première idée généralement exprimée dans les articles traitant de parallélisation de langages fonctionnels est de placer des processus partout *[GRAY 86]*. En effet, évaluer en parallèle toutes les expressions évaluables est une stratégie d'évaluation sure (qui fournit le résultat de tout calcul défini si les ressources existent en nombre suffisant), et en première approche cela semble être le meilleur moyen d'engendrer le parallélisme maximum.

Outre le défaut couramment constaté pour une telle approche (la durée de création d'un processus peut être supérieur au calcul de l'expression qu'il évalue en parallèle) en supposant le nombre de processus virtuellement infini, elle est tout à fait contraire à notre besoin de réserver les ressources processus à des processus actifs.

Cette approche est donc à rejeter.

1.5.1.c Pourquoi ne pas mettre des processus sur toute expression stricte assez grosse?

Pour répondre aux objections précédentes, nous pouvons penser à ne plus créer de processus que pour des expressions strictes (de manière à n'utiliser les ressources processus qu'à des calculs utiles) et de taille suffisante pour contre-balancer la création d'un processus.

Pourtant cette approche n'est pas non plus la bonne avec MaRS :

- Il est inutile de créer un processus qui prend le travail d'un autre, comme dans `(+ (processus e1) (processus e2))` où le processus sur `e1` réalise une évaluation qui dans `(+ e1 (processus e2))` est faite par le processus père.
- Il est inutile de lancer un processus qui va devoir attendre ses arguments comme dans `(let ((x (processus e2))) (+ e1 (processus (* 5 x))))` où le processus père peut évaluer `e1`, le processus sur `e2` peut travailler normalement, mais celui sur `(* 5 x)` va devoir attendre la fin du processus sur `x`.

Dans ces deux cas (et dans d'autres), le choix de parallélisation n'est pas bon car il entraîne le blocage d'un processus, et donc le gaspillage d'une ressource qui n'existe qu'en quantité limitée.

1.5.1.d Où mettre des processus?

Nous venons de voir des cas où il est néfaste de placer un processus. Par contre, énoncer des cas généraux où un processus est toujours bien placé n'est malheureusement pas possible (sinon il n'y aurait plus de réel problème de parallélisation).

Le programmeur doit se baser sur la connaissance des situations de blocage inter-processus (telles que celles présentées en 1.5.1.b et en 1.5.1.c) et chercher un bon compromis entre le trop faible parallélisme que peut engendrer tout refus de blocage potentiel et le trop grand taux d'attente auquel conduisent généralement trop de processus engendrés.

1.5.2 Le mode 'parallélisation automatique' du compilateur MaRS_Lisp

Dans ce mode, le compilateur est libre de placer des processus parallèles sur toute expression stricte[1] qui lui semble appropriée (après analyse heuristique). Ce mode permet donc au programmeur de ne pas se préoccuper de la parallélisation.

En contrepartie le programmeur ne dispose d'aucun contrôle sur les choix faits par le compilateur et doit se contenter des performances obtenues[2]. Celles-ci sont principalement limitées par l'absence d'heuristique visant à déterminer quelles potentialités sûres de parallélismes sont susceptibles d'avoir le plus fort taux d'activité.

Pour illustrer ce phénomène, nous considérerons la fonction de Fibonacci dans deux versions voisines :

```
(defun fib1 (n) (declare (strict n))
  (if (≤ n 2) 1
      (+ (fib1 (- n 1)) (fib1 (- n 2)))))

(defun fib2 (n) (declare (strict n))
  (if (≤ n 2) 1
      (+ (fib2 (- n 2)) (fib2 (- n 1)))))
```

Ces deux versions ne diffèrent que par l'ordre des arguments de l'addition, ce qui ne devrait pas avoir de conséquence sur la parallélisation. Pourtant dans les deux cas le compilateur choisira de lancer en parallèle le second argument (en gras) qui correspond au plus petit calcul dans le cas de `fib1` et au plus gros dans le cas de `fib2`.

Dans le cas de `fib1`, le processus père évalue le premier argument de l'addition, soit `(fib (- n 1))` qui est le plus rapide à évaluer des deux arguments

1. En l'absence d'analyseur automatique de stricticité le compilateur considère strictes les expressions déclarées comme telles par le programmeur.

2. Le programmeur peut toutefois placer des processus lui-même dans le programme en étant sûr que ses choix seront respectés. Le compilateur ne fait alors qu'ajouter d'autres processus où bon lui semble.

de l'addition, pendant que son fils évalue le second, soit `(fib (- n 2))`, et donc (en supposant une vitesse de progression identique pour les deux processus) le processus fils termine avant le processus père. Il en résulte que lorsque le père termine `(fib (- n 1))` il peut effectuer l'addition sans attendre, l'évaluation du second argument étant terminée.

Par contre dans le cas de `fib2`, le processus père qui évalue `(fib (- n 2))` pendant que son fils évalue `(fib (- n 1))` termine avant son fils et ne peut donc pas effectuer l'addition tant que celui-ci est en cours d'évaluation. Il en résulte une phase d'attente préjudiciable au rendement du processus père pendant laquelle la ressource correspondante est inutilement occupée, ce qui conduit à un programme plus lent dès lors que la machine atteint l'état de saturation (un tiers de durée supplémentaire dans le cas de `(fib 20)` sur 64 processus).

1.5.3 Le mode 'parallélisation manuelle' du compilateur MaRS_Lisp

Dans ce mode, le compilateur n'engendre aucun autre processus que ceux placés dans le code du programme par le programmeur qui garde ainsi tout contrôle sur le déclenchement du parallélisme (au prix d'efforts supplémentaires).

Par exemple, une version parallèle de la fonction de fibonacci peut être :

```
(defun fib (n) (declare (strict n))
  (if (≤ n 2) 1
      (+ (fib (- n 1)) (processus (fib (- n 2))))))
```

Qui produit un code compilé identique à la version `fib1` présentée en 1.5.2.

1.5.4 Le passage de paramètres

Le but de la compilation est de transformer le programme source contenant des variables locales en un programme objet n'en contenant plus, et cela en y introduisant des combinateurs.

Ce processus d'abstraction compile une définition `(lambda (`x_1 ... x_n`) e)` en un code `E` ne contenant aucune référence à un des x_i et tel que `(E` v_1 ... v_n`)` soit équivalent à `((lambda (`x_1 ... x_n`) e)` v_1 ... v_n`)`.

Par exemple la fonction `(lambda (x) (+ x x))` se compile (sans optimisation) en `(!W2 +)` qui ne référence plus `x`, et à partir de la définition de `!W2` nous vérifions que `((!W2 +) n) = (+ n n) = ((lambda (x) (+ x x)) n)`.

Grâce à l'absence de variables locales dans le code compilé, il ne sera pas nécessaire de gérer un environnement tout au long de l'exécution, ce qui dans une architecture parallèle constitue généralement une des sources principales de contention. Nous dirons que le code combinatoire réalise une gestion répartie du passage de paramètres.

Ainsi de nombreux mécanismes mettant à l'œuvre un environnement dans des implémentations classiques (ce qui rend ces mécanismes coûteux sur des machines massivement parallèles) seront ici possible sans mécanisme centralisé de ce type. Par exemple les applications partielles nécessitent habituellement la construction d'une fermeture ce qui pour nous est parfaitement inutile : `(lambda (x) (- x n))` où `n` est une variable globale se compilera simplement en $(!M_2\ -\ n)$ avant optimisation et sera optimisé en un seul combinateur réalisant le déplacement d'argument dans la pile et la soustraction (cf. 1.4).

1.5.5 Compilation et stricticité

L'utilisateur peut, grâce à des déclarations de stricticité, indiquer au compilateur quels sont les paramètres stricts du calcul. De plus, la stricticité de chaque opérateur primitif du langage est connue du compilateur (par exemple l'addition admet deux arguments stricts).

La source d'informations que constituent les déclarations du programmeur (pour autant qu'elle soit fiable) permet au compilateur de faire comme s'il disposait d'une analyse automatique de la stricticité qui annoterait automatiquement le programme. Ce choix, qui impose un travail supplémentaire lors de l'écriture des programmes, est dû à l'effort important qu'aurait nécessitée l'écriture d'un analyseur automatique, ce qui a justifié la relégation au second plan de priorité d'un tel développement.

1.5.6 Compilation de l'évaluation paresseuse

L'influence première des informations de stricticité est de permettre au compilateur d'utiliser localement un appel par valeur pour les paramètres stricts car cela ne diminue pas la définition du langage, le mode d'évaluation par défaut étant l'évaluation paresseuse.

Le mode d'évaluation choisi se concrétise non pas à l'intérieur du code de la fonction concernée, mais au niveau de l'application de la fonction à ses arguments. Ainsi à chaque appel de cette fonction les arguments correspondants à un paramètre strict pourront être évalués alors que les autres donneront lieu à la construction d'un graphe qui sera réduit selon sa nécessité.

Considérons par exemple les définitions qui suivent (chacune étant suivie du code compilé correspondant) :

```
(defun f1 (n) (cons n n))
> (!W2 CONS)

(defun f2 (n) (declare (strict n)) (cons n n))
> (!W2 CONS)

(defun g1 (x) (f1 (* 3 x)))
> (!B3 F1 (* 3))

(defun g2 (x) (f2 (* 3 x)))
> (*SEQ F2 3)
```

Les fonctions f1 et f2 donnent lieu au même code compilé, mais lorsqu'elles sont appliquées à leur argument elles conduisent à des codes distincts. Nous voyons dans le code de g1 le combinateur !B3 qui construit un graphe, alors que le code de f2 contient le combinateur *SEQ qui réalise la multiplication séquentiellement et évite ainsi toute création de graphe en effectuant directement la multiplication sur les arguments présents dans la pile.

Les premières étapes de réduction de (g1 0) et de (g2 0) sont présentées ci-dessous :

```
(g1 0)
= (!B3 F1 (* 3) 0)
= (F1 @1) ;; où @1 = ((* 3) 0)
;; un graphe non réduit est crée (à l'adresse @1)
;; pour passer l'expression (* 3 0) non évaluée à F1

(g2 0)
= (*SEQ F2 3 0) ;; (*SEQ k a b) = (k r) où r=a*b
= (F2 0)
;; la multiplication entre 3 et 0 est faite
;; puis son résultat passé à F2
```

Nous voyons donc que la conséquence première d'une évaluation paresseuse est d'imposer la création d'un graphe, création qui peut être évitée lors d'un appel par valeur.

Le coût à l'exécution correspondant à l'évaluation paresseuse ne peut être évalué précisément. Toutefois, l'ensemble des coûts liés au passage de paramètre réparti, à l'évaluation paresseuse et à la construction des applications présentes dans le programme a été évalué sur divers programmes de test *[COUSIN 91]*. Ce coût global correspond à l'expansion dynamique, c'est à dire au rapport entre le nombre de combinateurs et le nombre d'opérateurs Lisp exécutés qui est 1,33 (pour la compilation standard que nous avons utilisé tout au long de nos expérimentations), ce qui signifie que plus de 40% du code machine exécuté correspond directement à l'exécution des primitives du langage placées par le programmeur dans son programme.

Ce taux d'expansion très faible nous conforte dans l'idée que les solutions techniques retenues pour la compilation sont tout à fait adaptées à une machine parallèle.

D'autres approches, telles que la Categorical Abstract Machine *[COUSINEAU 85]*, *[COUSINEAU 86]* qui font appel à la notion d'environnement compilent l'évaluation paresseuse par des fermetures *[MAUNY 86]*.

Cette technique peut avoir un coût limité dans le cas d'un modèle séquentiel, toutefois dans le cas de nombreux accès parallèles la ressource que constitue un environnement centralisé pose des problèmes de contention.

Nous pensons donc que dans le cas d'un modèle parallèle, tel le modèle MaRS, il est préférable d'éviter le recours à un environnement centralisé pour préférer un environnement réparti, que réalise la compilation des lambda-expressions en code combinatoire MaRS.

1.6 La réduction de graphe

1.6.1 La synchronisation

La présentation rapide de la réduction de graphe faite en 1.2.3 est incomplète : il y manque un mécanisme pour synchroniser les accès concurrents aux noeuds du graphe.

Pour réaliser cette synchronisation, nous utilisons un mécanisme classique basé sur des états du graphe :

- Noeud libre (état L).

  Ne participe à aucune expression du graphe, c'est l'état initial de tout noeud au démarrage de la machine et de tout noeud qui vient d'être récupéré par le ramasse-miettes (*garbage collector*).
- Noeud non réduit (état NR).

  Participe à une expression du graphe mais n'est pas en forme normale : pour connaître sa valeur un processus devra le réduire.
- Noeud réduit (état R).

  Participe à une expression du graphe et est en forme normale : pour connaître sa valeur il suffit de lire son contenu qui ne peut plus varier.
- Noeud cons (état C).

  Participe à une expression du graphe qui est une cellule cons. Cet état est une simple variante de l'état R.
- Noeud en réduction (état ER).

  Participe à une expression du graphe. Cet état est celui qui sert à synchroniser : pendant tout le temps où un noeud est en réduction, les demandes de lecture de son contenu sont différées pour n'être satisfaites qu'après sa récriture.

L'ensemble des transitions autorisées entre états de noeuds du graphe est représenté Figure 9.

1.6.2 Implications de la synchronisation au niveau des processus

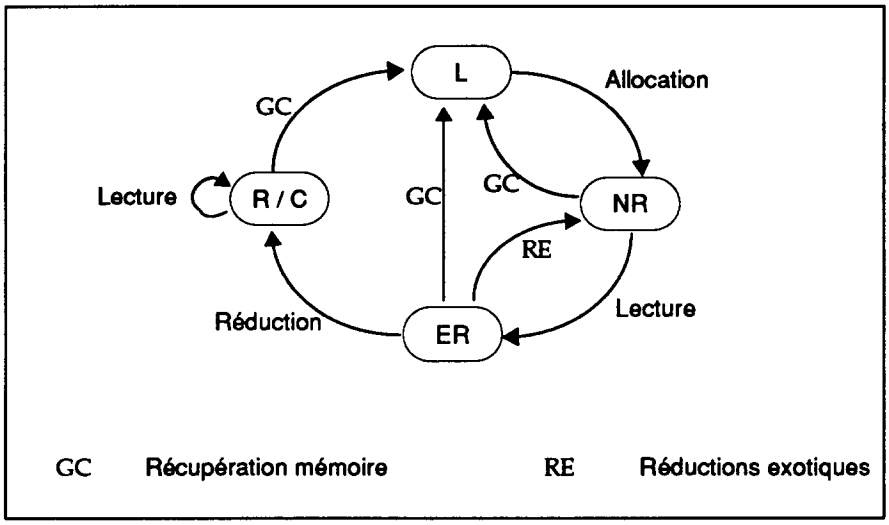
Après avoir présenté la synchronisation du point de vue du graphe, voyons la du point de vue des processus.

1.6.2.a Cas de lecture d'un noeud par un processus

- Un processus lit un noeud R (ou C).

  Le noeud a déjà été réduit, le processus accède directement à son contenu qui est sa forme normale.

Figure 9 Cycles des états du graphe



- Un processus lit un noeud ER.

  Le processus reste bloqué jusqu'à ce que le noeud soit récrit par un autre processus. A ce moment là, la demande de lecture pourra être satisfaite, et la mémoire le débloquera.

- Un processus lit un noeud NR.

  Le noeud n'est pas évalué, le processus va le réduire lui même. Pendant ce temps le noeud est mis ER pour synchroniser tout autre accès. Lorsque la réduction est terminée, le résultat trouvé par le processus (qui est la forme normale du noeud) est récrit à la place de l'ancienne valeur à l'intérieur du noeud. Si d'autres processus sont bloqués en attente sur ce noeud, ils sont alors libérés par la mémoire.

1.6.2.b Conséquences

Ce mécanisme de synchronisation par l'état des noeuds du graphe garantit l'atomicité des accès aux cellules du graphe[1] telle que *[DIJKSTRA 65]* en montra la nécessité. En effet, si plusieurs processus accèdent en même temps à un noeud :

- Si le noeud est NR seul le premier à effectuer la lecture accède à son contenu (l'ordre d'arrivée des requêtes à la mémoire impose un ordre total), grâce à la synchronisation par l'état ER les autres accès ne pouvant avoir lieu qu'après la fin de l'accès par le premier processus.

1. Un accès à la mémoire est atomique s'il ne peut être interrompu. Dans le cas contraire (par exemple si `{x:=x+1}` est effectué en plusieurs étapes en mettant en œuvre un registre comme dans la séquence `{r:=x; r:=r+1; x:=r}`) la séquence peut être interrompue entre deux instruction ce qui rend non déterministes plusieurs accès réalisés en parallèle.

Dans ce cas il y a donc atomicité de la lecture

- Si le noeud est ER, ces demandes sont différées jusqu'a la récriture du noeud. Ces demandes sont prises en compte[1] après la récriture du noeud.

  Dans ce cas il n'y a donc pas de lecture, ce qui est plus fort que l'atomicité recherchée.
- Si le noeud est R (ou C), le contenu du noeud est en forme normale et ne pourra donc plus changer, toutes les demandes de lectures ultérieures donneront donc le même résultat et il n'y a plus besoin de synchronisation.

  Dans ce cas l'atomicité n'est plus nécessaire.

1.7 L'architecture MaRS

1.7.1 Généralités

Pour exécuter le code combinatoire dans lequel sont compilés les programmes MaRS_Lisp, Guy DURRIEU et Michel CUBERO-CASTAN ont défini une architecture multi-processeur à base de circuits spécifiques.

La machine est essentiellement constituée de trois types de processeurs différents (cf. Figure 10) : les processeurs de réduction qui réduisent le code combinatoire, les processeurs mémoire qui gèrent le graphe et les processeurs de communication qui constituent le réseau de communication reliant entre eux processeurs mémoire et de réduction (cf. Figure 11) et dont le rôle est également d'assurer une gestion décentralisée des ressources de la machine.

Les communications entre processus se font par messages, tout processus en attente d'une réponse est interrompu et ne sera réactivé que suite à la réception de la réponse.

Pour assurer un recouvrement des attentes liées aux communications, chaque processeur de réduction gère plusieurs processus qui pour des raisons d'efficacité lors des fréquentes commutations de contexte sont physiquement implantés sur le silicium (dans l'architecture actuelle ce nombre est fixé à huit). La conséquence première de ce choix est de limiter le nombre total de processus.

Pour assurer l'autorégulation du parallélisme ainsi que la répartition des allocations qui lui incombent, le réseau utilise des informations de charge que lui transmettent en sens inverse des messages les différents processeurs qui lui sont connectés. Ainsi toute allocation ou libération de ressource[2] induit

1. Pour éviter une famine, les requêtes sont stockées dans une file.

2. La notion de ressource regroupe aussi bien les processus (gérés par les processeurs de réduction) que les cellules mémoire (gérées par les processeurs mémoire).

Figure 10 Architecture simplifiée de MaRS

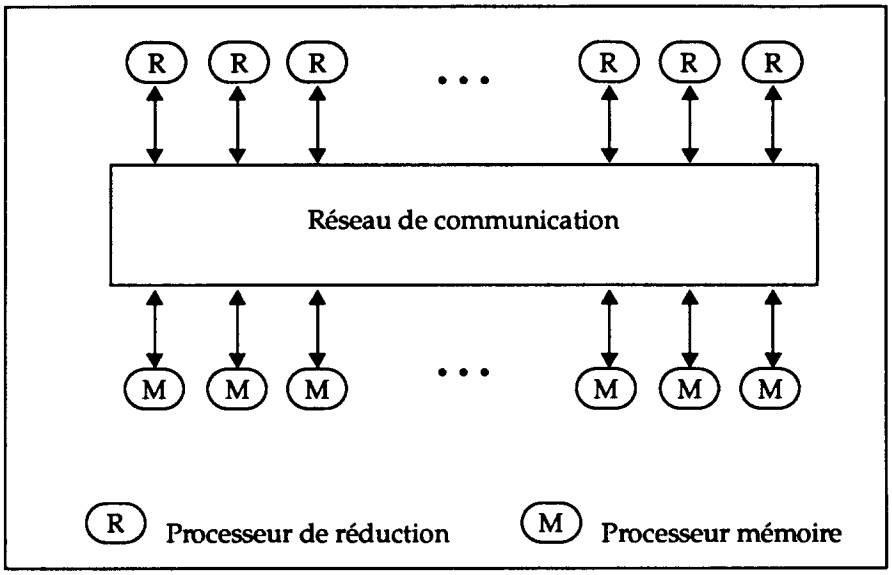
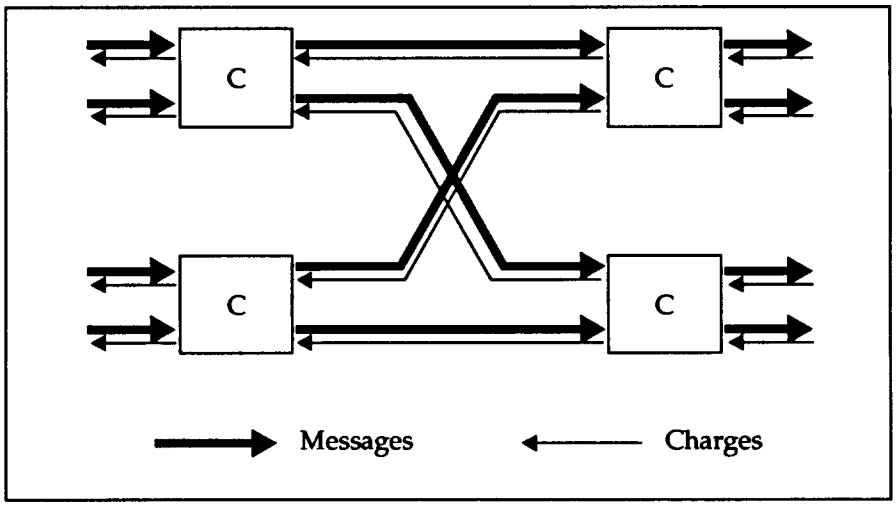


Figure 11 Le réseau de communication



l'émission par le processeur concerné d'une variation de charge sur le réseau. Ces variations de charges, qui se propagent de proche en proche sur l'ensemble des processeurs de communication du réseau, permettent :

- A tout processeur de réduction de connaître la charge mémoire globale et donc de freiner le parallélisme lorsque cette charge est trop haute ou de retourner à un fonctionnement normal lorsqu'elle est redescendue.

- A tout processeur de communication de connaître laquelle de ses deux sorties mène au processeur le moins chargé, ce qui lui permet de répartir les demandes d'allocation qui lui sont faites, non pas sur adresse mais en fonction de la charge, sur le processeur le moins chargé.
- A tout processeur mémoire de connaître la charge de réduction globale et donc de différer les créations de processus[1] lorsqu'un certain seuil est franchi.

1.7.2 Les caractéristiques du processus

1.7.2.a Granularité

Notre modèle de machine parallèle est basé sur le pari suivant : le parallélisme massif ne peut s'envisager pour le plus grand nombre d'algorithmes qu'à la condition de pouvoir prendre en compte des processus de granularité aussi diversifiée que possible.

Ainsi les choix architecturaux du projet MaRS visent généralement à diminuer autant que possible les coûts associés aux processus, de manière à ce qu'un processus soit rentable même pour des tâches de granularité faible.

Toutefois, nous ne devons pas perdre de vue qu'il existe une limite inférieure (même très faible) à la granularité accessible : si la latence de création du processus est supérieure au temps que prendrait son exécution séquentielle, celui-ci ralentit les calculs.

Nous pourrons parfois simplifier en disant que MaRS n'a pas de problème de granularité, il nous faudra malgré tout garder à l'esprit que ceci n'est plus vrai pour des processus n'exécutant qu'un très petit nombre d'instructions (nous verrons par exemple au Chapitre 4 le cas de processus mis en œuvre pour le paradigme pipe-line et n'exécutent qu'une instruction cons).

1.7.2.b Coût de création

La création d'un nouveau processus par un processus existant consiste simplement en l'envoi d'un message à la mémoire lui demandant de mettre le graphe correspondant en réduction.

De ce fait, le coût associé à la création d'un processus est minime, aussi bien pour le processus créateur que pour les autres processus existant. Par contre la création est soumise à une certaine latence, qui est assez faible si une ressource processus est disponible (temps des traversées du réseau et de prise en compte des messages), mais peut être arbitrairement grande si la demande de création a lieu lorsqu'aucune ressource n'est disponible et que la création doit ainsi être différée.

1. Un processus est demandé par un processeur de réduction, mais son démarrage effectif est à la charge du processeur mémoire qui gère la cellule mémoire correspondante (le processeur mémoire la fait alors passer en réduction).

1.7.2.c Coûts de gestion

Les processus gérés par les différents processeurs de réduction sont physiquement câblés. Ainsi la gestion de processus, y compris les commutations entre processus a un coût très faible.

Il en découle que la granularité des différentes tâches (dont dépendent directement les fréquences de commutation, d'allocation et de libération des processus) n'est pas d'une importance primordiale, à l'inverse des préoccupations majeures de nombre de projets ayant trait au parallélisme (*[GOLDMAN 88], [HALSTEAD 86], [MOHR 90]* par exemple pour ne citer que des projets du domaine fonctionnel).

1.7.2.d Interruptions

Les processus ne sont interrompus que lorsqu'ils ont besoin d'une information extérieure (essentiellement résolution d'une référence à la mémoire).

En fait, le mode de fonctionnement du processeur de réduction est *message driven* : il n'y a pas de compteur ordinal désignant une prochaine instruction à exécuter, le processus réduit le code se trouvant dans sa pile et s'arrête lorsqu'il doit attendre un message en réponse à une de ses requêtes.

A ce moment là, c'est le contenu du message arrivant pour le processus qui d'une part le rend à nouveau éligible et d'autre part détermine par son contenu la nouvelle portion de micro-code que le processus va exécuter.

Un processus MaRS n'est donc ni interruptible ni stoppable, et ce par impossibilité de lui envoyer un message qu'il n'attend pas.

La conséquence première au niveau algorithmique est que le programmeur doit absolument éviter l'utilisation du parallélisme spéculatif, faute de savoir stopper les processus travaillant à réduire des expressions inutiles.

1.7.3 La récupération mémoire

Le principe retenu pour la récupération est le comptage de références optimisé à l'aide de la notion de pointeurs fantômes qui sont des références non comptabilisées *[CASTAN 88]*.

La récupération mémoire est faite en parallèle avec l'exécution par les processeurs mémoire et n'a donc qu'un coût indirect pour l'exécution qui est de diminuer la disponibilité des processeurs mémoire et d'augmenter le débit de messages sur le réseau.

1.8 Le simulateur

1.8.1 Généralités

Le simulateur (écrit en Common_Lisp) exécute du code MaRS_Lisp tel qu'il serait exécuté sur la machine. Toutes les phases de traduction et d'optimisation de codes ainsi que de chargement du graphe en mémoire sont donc faites identiquement pour la machine et le simulateur.

La configuration simulée est paramétrable en nombre maximum de processus, ce qui permet d'étudier l'influence de la quantité de ressources disponibles sur l'exécution d'un programme.

Les informations obtenues en fin de simulation sont le résultat du programme, diverses statistiques d'exécution, le détail des primitives exécutées, les dates de début et de fin ainsi que le taux d'activité des processus mis en œuvre, et le profil parallèle.

1.8.2 Les statistique d'exécution

Les statistiques regroupent le nombre de processus ayant participé au calcul, le taux de parallélisme (nombre moyen de processus actifs au cours de l'exécution), le taux d'attente (nombre moyen de processus en attente), et le nombre total de primitives exécutées.

1.8.3 Le profil parallèle

Il est la représentation graphique de l'activité des processus au cours de l'exécution. Pour chaque intervalle de temps (dont la durée dépend de la durée totale du programme) est reporté le nombre moyen de processus en attente (représentés par des -) et le nombre moyen de processus actifs (représentés par des @).

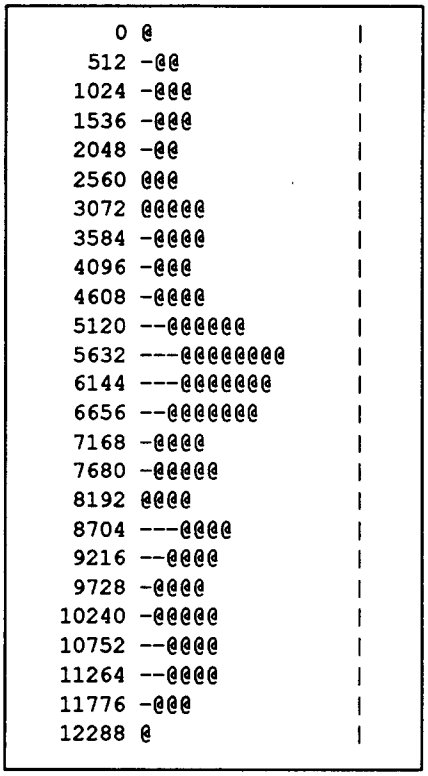
Nous disposons en outre d'une autre information capitale dans la lecture du profil : le nombre de processus de la configuration simulée détermine un seuil de saturation de la machine (lorsque tous les processus possibles sont alloués) ce qui avec un programme suffisamment parallèle donne à la courbe un aspect tronqué. Le seuil de saturation sera représenté sur les profils par |.

L'allure générale d'un profil parallèle est un histogramme représenté verticalement avec la date origine en haut à gauche. Nous pouvons lire sur le profil Figure 12 que la configuration simulée de 16 processus n'atteint jamais la saturation et qu'entre les cycles 5632 et 6144 il y a eu en moyenne 3 processus en attente et 8 processus actifs.

1.8.4 Le mode trace

En plus du fonctionnement normal que nous venons de décrire, le simulateur peut fonctionner en mode trace, c'est à dire détailler à chaque cycle de

Figure 12 Exemple de profil parallèle



réduction le contenu de la pile de chaque processus présent ainsi que les différents noeuds mémoires créés ou récrits.

Ce mode permet donc de visualiser de façon précise l'évolution parallèle du graphe tout au long de la séquence de réduction.

1.8.5 Différences principales par rapport au modèle

Notre simulateur, comme tout simulateur n'est qu'une vision simplifiée du modèle. Parmi ces simplifications les plus marquantes sont :

- Le réseau simulé est à temps de transit nul.
- La mémoire simulée est à temps de réponse nul.
- La configuration simulée n'est pas `n` processeurs à `8` processus chacun mais `8n` processeurs à un processus chacun. Il en résulte que tous les processus prêts sont activés à chaque cycle, et non pas un processus prêt par processeur.
- La durée d'exécution des primitives ne correspond pas précisément à leur complexité réelle.

- La latence de création d'un nouveau processus est arbitrairement fixée à deux cycles par étage du réseau ce qui ne tient pas compte des variations de disponibilité du réseau et de la mémoire.

1.8.6 Conséquences

Notre simulateur restitue une vision tronquée mais réaliste du modèle. S'il n'est pas possible de faire correspondre précisément une durée à un nombre de cycles, les profils d'exécution sont malgré tout une donnée très intéressante dans l'analyse des algorithmes.

C'est cette dernière caractéristique qui sera principalement utilisée en conjonction avec la possibilité de fixer le nombre de processus simulés. Les profils obtenus donnent une idée précise du comportement de l'algorithme simulé et permettent de véritables études algorithmiques.

Les différentes qualités du simulateur pour ce qui est de visualiser le comportement d'un programme, et la persistance de certains problèmes de mise au point du prototype qui lui interdisent l'exécution de gros programmes font que le simulateur sera préféré à la machine lors de la mise au point et de la parallélisation des algorithmes.

1.9 Acquis et limites de la parallélisation MaRS_Lisp

1.9.1 Les moyens de la parallélisation

La parallélisation peut être faite par le compilateur comme nous l'avons vu en 1.5.2. Toutefois, les heuristiques mises en œuvre lors de la parallélisation automatique de programmes ne sont pas encore assez mûres pour prétendre satisfaire tous les besoins.

L'exemple donné en 1.5.2 illustre un cas dans lequel les heuristiques du compilateur peuvent être prises en défaut, mais bien d'autres illustrations sont possibles qui sont susceptibles de tromper bien des heuristiques.

Ne pouvant nous contenter de laisser faire le compilateur, il nous reste à effectuer une parallélisation *à la main*.

Deux options sont alors envisageables : soit paralléliser à la main et laisser le compilateur refaire une parallélisation supplémentaire (qui conservera alors tous les processus placés par le programmeur), soit ne faire que la compilation à la main.

Le seul choix possible parmi ces deux options est de ne faire que la compilation à la main : L'autre choix ne peut laisser au compilateur que la possibilité de rajouter des processus écartés par le programmeur. Sauf à supposer que celui-ci ait fait les mauvais choix (mais dans ce cas là, le résultat est de toute façons fort compromis), un tel rajout ne peut que conduire à ajouter des processus peu actifs et à perturber le bon déroulement parallèle du programme.

Quant à l'éventualité de retrouver, à partir d'un code parallélisé par le seul compilateur, les potentialités de parallélisme retenues par celui-ci pour en dériver une version parallélisée à la main, nous devons la rejeter pour trois raisons : Premièrement retrouver les processus dans le code compilé n'est pas très aisé (impossible pour un novice), deuxièmement cette technique complique largement la parallélisation, enfin elle n'a que peu de chances d'aboutir à un code meilleur, même si le programmeur (sachant où lui-même placerait les processus) essaye de comprendre les choix du compilateur.

Les moyens à notre disposition pour paralléliser à la main sont essentiellement l'utilisation de méthodes formelles, telles qu'une analyse de stricticité permettant de détecter des opportunités acceptables de parallélisme, l'utilisation des paradigmes classiques de la programmation fonctionnelle parallèle (diviser pour régner et pipe-line) pour introduire le parallélisme au niveau des opportunités détectées, et l'utilisation de transformations de programme essentiellement centrées sur les fonctions d'ordre supérieur de manipulation de liste (`map`, `reduce`) et de leurs versions parallèles (`p-map`, `p-reduce`) *[MARRE 92]*.

Pour nous guider lors de l'utilisation de ces méthodes ou paradigmes, la seule aide réellement efficace en toutes circonstances est une bonne connaissance du mécanisme de synchronisation implicite (qui est somme toute fort simple) et de ses conséquences, telles qu'elles sont présentées en 1.5.1.

1.9.2 Quelques objectifs poursuivis lors de parallélisations

Au vu des propriétés du modèle, le but le plus souvent poursuivi lors de la parallélisation est d'éviter une prolifération des processus en attente qui utilisent une ressource non préemptible inefficacement (voir en 1.5.1 et en 1.7). Ce problème ne se pose bien évidemment que pour les algorithmes réellement parallèles, mais compte tenu de l'aptitude de MaRS à prendre en compte tous les grains de parallélisme, les problèmes de ce type sont majoritaires.

Un autre objectif, souvent antinomique, et qui prévaut dans certains cas d'algorithmes peu parallèles est l'introduction de processus supplémentaire par décomposition et introduction de structures de données intermédiaires (paradigme pipe-line) ou par découpage du problème en sous-problèmes (paradigme diviser pour régner).

Le cas le plus délicat correspond bien entendu à des algorithmes dont le degré de parallélisme varie au gré des données, que ce soit d'une exécution à l'autre ou même au cours d'une seule exécution.

1.9.3 Conclusions relatives à la parallélisation

Les conclusion que nous pouvons tirer d'une longue pratique en parallélisation de programmes MaRS_Lisp (*[MaRS 89a]*, *[MaRS 89b]*, *[MaRS 90b]*, *[MARRE 92]*) sont que les processus en attente posent un problème délicat

car il est souvent difficile d'en diminuer le nombre sans faire également chuter le nombre de processus actifs.

La limitation principale pour aller vers plus de parallélisme dans le cadre de MaRS est la perte de ressources due aux processus en attente. Une voie suggérée dans *[MARRE 92]* pour en limiter le nombre est de rompre l'équivalence entre processus logique (processus placé dans le code par le programmeur) et processus réel (processus généré à chaque application de la primitive parallèle) en introduisant de nouveaux opérateurs parallèles à comportement dynamique (qui engendre ou non un processus en fonction de certains critères liés à l'exécution).

Une limitation a également été observée dans l'utilisation du paradigme pipe-line : celui-ci correspond à un des styles les plus naturels de programmation fonctionnelle parallèle et il conduit à des programmes très clairs, mais il met en œuvre un tel nombre de processus (un processus par cellule de structure intermédiaire) que son utilisation n'est pas aussi profitable sur le plan des performances que sur celui de l'expressivité. En fait le paradigme pipe-line amène bien souvent à franchir le plancher de la granularité minimum accessible au processus MaRS.

Enfin une lacune à été observée dans la mise en œuvre d'algorithmes nécessitant des structures de données autres que des listes, le langage ne possédant à la base que cette structure de donnée.

1.10 Objectifs de cette thèse

Dans la suite de cette thèse nous allons proposer diverses solutions aux problèmes que nous venons de soulever.

Ainsi le Chapitre 2 est consacré à l'introduction de nouvelles structures de données (valeurs multiples, structures - *record*- et vecteurs), basée sur l'interprétation de ces structures en tant qu'objet actif, c'est à dire en tant que fonction, et sur leur expression en terme de lambda-expressions.

Le Chapitre 3 traite de deux façons de diminuer le coût associé aux processus en attente : introduire des primitives de création dynamique de processus, aptes à adapter leur comportement à l'évolution des calculs, et introduire de nouvelles constructions parallèles, syntaxiquement semblables à de la programmation impérative mais n'introduisant aucun effet de bord, qui permettent de s'affranchir de certaines synchronisations de processus. Dans les deux cas, le but poursuivi sera l'augmentation de l'activité parallèle du programme par diminution de la part relative des processus en attente.

Le Chapitre 4 traite du paradigme pipe-line, qui est très expressif mais dont le rendement en terme d'augmentation du parallélisme effectif est compromis par le très grand nombre de processus qu'il met en œuvre, et présente une nouvelle approche visant à conserver l'effet pipe-line tout en ne mettant en œuvre qu'un seul processus par traitement.

Enfin, dans la Conclusion nous essaierons de faire la synthèse de ce travail et de présenter les perspectives qu'il peut offrir.

Chapitre 2

Des structures de données en logique combinatoire

Objectif : Détailler comment une structure de données adaptée à l'algorithme peut améliorer celui-ci en donnant des exemples déjà maîtrisés. Montrer comment réaliser des structures plus complexes que des listes ou des arbres.

2.1 Nécessité

Est-il indispensable de disposer de structures de données complexes, du moment que nous disposons des cellules cons classiques?

Il est possible de réaliser toutes sortes de structures et de fonctions d'accès à la simple condition qu'il n'y ait pas de limitation due au typage et de disposer de la cellule cons. Dans le cas où il y a un typage fort *[MITCHEL 90]*, il est quand même possible d'offrir des primitives implémentant les structures désirées, puis de les compiler sous forme de manipulations de listes.

Cependant, disposer de nombreuses structures de données n'est pas suffisant : diverses études *[CARCENAC 90] [CARCENAC 91] [MARRE 92]* ont mis en évidence avec notre modèle le lien entre le gain obtenu lors de la parallélisation d'une application et le choix initial des structures de données. On peut également citer des études à caractère plus général *[ROE 91]*, qui montrent le lien entre le parallélisme effectif des algorithmes et les performances des structures de données utilisées.

En fait, si le parallélisme potentiel d'un algorithme est bien indépendant de l'implémentation des structures de données, ce parallélisme sera d'autant moins efficace que les différents processus se bloqueront plus longtemps lors des accès aux données[1]. Ceci n'est dû qu'à la différence entre le taux de parallélisme d'un programme (nombre moyen de processus présents pendant l'exécution) et son speed-up (rapport des temps d'exécution de la version séquentielle et de la version parallèle) : augmenter le taux de

parallélisme ne signifie pas automatiquement diminuer le temps d'exécution en proportion.

Ainsi *[ROE 91]* montre sous certaines hypothèses simplificatrices qu'une utilisation efficace du paradigme *diviser pour régner* n'est envisageable qu'à certaines conditions : si les phases de décomposition et de recomposition des données ont un coût constant, le parallélisme peut être proportionnel à la taille des données, alors que si ce coût est proportionnel à la taille des données le parallélisme n'est qu'un logarithme de cette taille, ce qui est un très mauvais résultat. Le coût maximum admissible exhibé pour ces phases est en fait un logarithme de la taille des données qu'elles manipulent.

Nous ne serons donc pas surpris de vérifier cela dans *[MARRE 92]* qui calcule le taux de parallélisme atteint par différents programmes mettant en œuvre le paradigme *diviser pour régner*. Il en ressort qu'une bonne utilisation en est l'inversion récursive d'un arbre binaire où les opérations de division et de composition ont un coût constant, et qu'une mauvaise utilisation en est le quicksort d'une liste pour lequel l'opération de division a un coût proportionnel à la taille des données.

Pour sa part *[CARCENAC 90] [CARCENAC 91]* montre comment le remplacement de tableaux de facettes par des arbres octaux paresseux d'éléments de volume permet de gagner plusieurs ordres de grandeur dans la synthèse d'images à précision fixée.

Le gain de la phase de parallélisation étant lié au choix des structures de données, la nécessité de disposer de structures puissantes et variées est donc incontestable. Nous allons montrer comment introduire certaines structures non triviales dans notre modèle qui ne contient initialement que la cellule `cons` et l'application (cf. 1.3.1 page 23).

2.2 Les valeurs multiples

2.2.1 Description

Une des caractéristiques de la programmation purement fonctionnelle est que les fonctions ne peuvent utiliser de variable globale pour transmettre des résultats. De ce fait, la seule valeur récupérée par l'appelant est le résultat de l'appel de fonction. Pour calculer et retourner plusieurs résultats à l'aide d'une seule fonction, nous devrons mettre en œuvre une construction spécifique, comme le fait Lisp avec les valeurs multiples qui permettent à une fonction de retourner un agrégat de résultats. Pour ce faire, il faut disposer de deux constructions que nous nommerons à la Common Lisp, une

1. Par exemple, le parallélisme potentiel d'un pipe-line est deux, cependant l'accès aux données peut faire chuter le parallélisme réel bien en dessous.

`values` pour construire l'agrégat et l'autre pour en extraire les constituants à la manière d'un `let`, `multiple-value-bind`.

[HENDERSON 80] présente diverses possibilités à propos des valeurs multiples (utilisation de listes (cf. 2.2.2 page 54), utilisation de plusieurs fonctions, utilisation de continuations également présentée dans *[MaRS 88a]*) nous pouvons y ajouter l'utilisation de tuples à la Miranda *[WADLER 85b]* qui n'est rien d'autre que la solution à base de listes. Il ressort de cette analyse une nécessité d'implanter les valeurs multiples dans un langage fonctionnel pour des raisons de clarté et d'expressivité. Ce fait étant acquis, nous nous intéresserons dans la suite de ce paragraphe aux performances du mécanisme.

Enfin, et indépendamment des contraintes de types pouvant exister au niveau des constituants de la valeur multiple, il faut assurer que la construction `multiple-value-bind` soit bien appliquée à une valeur multiple.

Dans ce but, et après avoir présenté la syntaxe de nos deux constructions, nous proposerons deux ensembles de règles de typage qui permettent d'introduire un type pour les valeurs multiples et qui sont plus ou moins restrictives.

2.2.1.a Syntaxe

La construction d'une valeur multiple contenant les expressions $e_1..e_n$ se fait par `(value` $e_1..e_n$`)`.

L'expression `(multiple-value-bind (`$x_1..x_n$`) v corps)` lie en parallèle les valeurs de la valeur multiple `v` aux variables $x_1..x_n$ et évalue l'expression `corps` dans le contexte en résultant. L'évaluation de `v` (qui donne une valeur multiple comme résultat) est faite dans le contexte d'évaluation de l'expression `(multiple-value-bind (`$x_1..x_n$`) v corps)` toute entière, et ne peut donc référencer les x_i introduits par `multiple-value-bind`.

2.2.1.b Typage restrictif

En utilisant un typage à la ML *[DAMAS 82]* nous exprimerons le typage des valeurs multiples par les deux règles ϑIntro et ϑElim qui respectivement introduit et élimine le type valeur multiple ϑ. Les règles sont écrite en sémantique naturelle (ou sémantique *à la Plotkin*) dont une présentation peut être trouvée dans *[KAHN 87]*.

$$\vartheta\text{Intro}: \frac{\Gamma \vdash (x_1 : t_1) \quad \ldots \quad \Gamma \vdash (x_n : t_n)}{\Gamma \vdash (\text{values}\, x_1 \ldots x_n) : (\vartheta\ t_1 \ldots t_n)}$$

$$\vartheta\text{Elim}: \frac{(\Gamma \vdash v : (\vartheta\ t_1 \ldots t_n)) \qquad (\Gamma, x_1 : t_1, \ldots, x_n : t_n \vdash (e : t))}{\Gamma \vdash (\text{multiple-value-bind}\ (x_1 \ldots x_n)\ v\ e) : t}$$

La règle ϑIntro se lit "Si du contexte Γ d'assertions de type il est possible de déduire que x_1 est de type t_1 ... x_n est de type t_n, alors il est possible d'en déduire que (values $x_1 \ldots x_n$) est de type (ϑ $t_1 \ldots t_n$)".

La règle ϑElim se lit "Si du contexte Γ d'assertions de type il est possible de déduire que v est de type (ϑ $t_1 \ldots t_n$) et que de Γ augmenté des assertions x_1 est de type t_1 ... x_n est de type t_n, il est possible de déduire que e est de type t, alors il est possible d'en déduire que (multiple-value-bind ($x_1 \ldots x_n$) v e) est de type t".

Il résulte de la règle ϑElim une interdiction de ne lier qu'une partie des valeurs de l'agrégat.

2.2.1.c Typage moins restrictif

Il est similaire au typage restrictif sauf la règle d'élimination qui autorise à ne lier qu'une partie des valeurs.

$$\vartheta\text{Intro}: \frac{\Gamma \vdash (x_1 : t_1) \quad \ldots \quad \Gamma \vdash (x_n : t_n)}{\Gamma \vdash (\text{values}\, x_1 \ldots x_n) : (\vartheta\ t_1 \ldots t_n)}$$

$$\vartheta\text{Elim}: \frac{(\Gamma \vdash v : (\vartheta\ t_1 \ldots t_{n+k})) \quad (\Gamma, x_1 : t_1, \ldots, x_n : t_n \vdash (e : t))}{\Gamma \vdash (\text{multiple-value-bind}\ (x_1 \ldots x_n)\ v\ e) : t}$$

La règle ϑElim permet ici de ne lier que les n premières valeurs d'une valeur multiple qui en compte $n+k$; ($k \geq 0$), les autres étant ignorées.

2.2.1.d Sémantique intuitive

Informellement, et si l'expression est correctement typée (ce qui va amener une des contraintes ($n = k$) ou ($n \leq k$) dans l'expression qui suit), la sémantique de ces constructions est :

```
(multiple-value-bind (x1..xn) (values e1..ek) corps)

= (let ((x1 e1) .. (xn en)) corps)
```

Dans cet exemple volontairement simplifié, la valeur multiple est directement construite par `values`, mais dans le cas général nous pouvons introduire à la place tout calcul dont le résultat est de type correct. Nous vérifions alors que l'environnement n'est enrichi par les liaisons entre les x_i et les e_i qu'au niveau de `corps`.

De plus, et pour être homogène avec notre modèle, les valeurs multiples devront être paresseuses, parallélisables et à moindre coût.

2.2.1.e Parallélisation des valeurs multiples

La source de parallélisme potentiel pour la construction `multiple-value-bind` n'est pas la même que celle du `let`. En effet s'il est possible dans ces deux constructions de calculer les valeurs e_i liées aux variables x_i à l'aide de `processus`, ce qui s'écrit respectivement :

```
(multiple-value-bind (x1…xn)
   (values (processus e1) … (processus ek))
   corps)
```

et

```
(let ((x1 (processus e1)) … (xn (processus en))) corps)
```

ces deux expressions ont un fonctionnement parallèle fondamentalement différent conformément à la règle d'évaluation des expressions (cf. 1.3.4 page 26).

Dans le cas du `let`, les e_i seront bien calculés en parallèle avec `corps` car ils sont touchés quand le `let` est touché.

Par contre, dans le cas du `multiple-value-bind`, les e_i ne seront touchés que lorsque sera touché le `values`, ce qui selon les cas (il peut y avoir un calcul important englobant cette expression) peut avoir lieu bien plus tard (cf. Figure 1).

Figure 1 Mauvaise parallélisation d'un multiple-value-bind

```
;; calcul du min et du max d'une liste de nombres non vide
(defun bornes (l))
  (bornes2 (rest l) (first l) (first l)))

(defun bornes2 (l min max)
  (cond ((null l) (values (processus min) (processus max)))
        ((< (first l) min) (bornes2 (rest l) (first l) max))
        ((> (first l) max) (bornes2 (rest l) min (first l)))
        (t (bornes2 (rest l) min max))))

(multiple-value-bind (min max)
  (bornes l)
  (gros-calcul max min))
```

Les processus de la fonction `bornes2` seront déclenchés alors que le calcul des bornes est terminé : ils ne servent pas à calculer `(bornes l)` et `(gros-calcul max min)` en parallèle.

La parallélisation d'un `multiple-value-bind` se fera donc tout naturellement en produisant l'évaluation en parallèle de la valeur multiple et du corps, ce qui n'empêche aucunement de placer des `processus` pour construire certains e_i en parallèle.

Pour cela, nous introduirons une version parallèle du `multiple-value-bind` que nous noterons `p-multiple-value-bind`, dont la syntaxe restera identique et qui déclenchera un processus pour évaluer la valeur multiple avant d'effectuer son traitement, à la manière de `(multiple-value-bind (`x_1`..`x_n`) (processus v) corps)`.

Voyons maintenant comment introduire les valeurs multiples dans notre modèle.

2.2.2 Une solution simple

La solution la plus simple pour construire des valeurs multiples est de les compiler sous forme de listes[1]. Le fonctionnement qui en découle est assimilable à ce qui suit :

`(multiple-value-bind (`x_1`..`x_n`) v corps)`

`=` `(let ((`x_1 `(car v)) ... (`x_n `(cad..dr v))) corps)`

Cette solution est très simple mais a de nombreux défauts :

- elle est coûteuse en terme de graphe car elle utilise de nombreuses cellules `cons`.
- le temps d'accès aux valeurs agglomérées est en $O(n^2)$ puisque les accès se font par `car`, `cadr`, `caddr`, ..., `cad..dr` dont les coûts sont respectivement 1,2,...,n.

Par contre :

- elle est facile à macrogénérer en version parallèle :

  `(p-multiple-value-bind (`x_1`..`x_n`) v corps)`

  `=` `(let ((w (processus v)))`
  `(let ((`x_1 `(car w)) ... (`x_n `(cad..dr w))) corps))`

  où w est une variable jamais encore utilisée.
- elle est compatible avec les deux ensembles de règles de typage.
- elle ne limite pas le nombre de valeurs agglomérables par `values`.

2.2.3 Une première approche combinatoire

2.2.3.a Principe

Le principe utilisé est connu : il s'agit de traduire le type désiré en lambda expression et de réaliser ensuite l'abstraction de ces lambda expressions sous forme de combinateurs. Un exemple célèbre de cette approche sont les

1. L'utilisation de tuples est tout à fait similaire.

nombres de Church, représentation des entiers à l'aide des combinateurs S et K *[CHURCH 41]*.

2.2.3.b Définitions

Une première solution utilisant une compilation en graphe de combinateurs a été proposé dans *[COUSTET 90]*, elle repose sur les définitions :

(multiple-value-bind (x_1..x_n) v corps)

= (v (lambda (x_1..x_n) corps))

(values e_1..e_n)

= (lambda (f) (f e_1..e_n))

Avec ces définitions, nous vérifions aisément que :

(multiple-value-bind (x_1..x_n) (values e_1..e_n) corps)

= ((lambda (f) (f e_1..e_n)) (lambda (x_1..x_n) corps))

= ((lambda (x_1..x_n) corps) e_1..e_n)

= (let ((x_1 e_1) ... (x_n e_n)) corps)

2.2.3.c Compilation

`multiple-value-bind` ne procède qu'à un réarrangement de ses arguments et peut être simplement une macro :

(multiple-value-bind (x_1..x_n) v corps)

= (v (lambda (x_1..x_n) corps))

`values` peut être directement compilé en utilisant le combinateur $!MS_i$:

$!MS_i$ x_1..x_i = x_i x_1..x_{i-1}

(values e_1..e_n)

= (lambda (f) (f e_1..e_n))

= ($!MS_{n+1}$ e_1..e_n)

Remarquons que l'indice des combinateurs étant forcément borné, le nombre de valeurs que `values` peut agglomérer est également borné. De plus, l'expression (values e_1..e_n) étant irréductible (au sens de la réduction de graphe), l'évaluation des e_i ne pourra être que paresseuse, comme désiré.

2.2.3.d Analyse

Les avantages de cette approche sont :

- un temps d'accès minimum (1 combinateur exécuté).
- un coût optimum en terme de graphe par utilisation d'un nombre minimum de noeuds.

Par contre, elle a des limitations :

- elle n'est compatible qu'avec le typage restrictif.
- de par la compilation sous-jacente, elle limite le nombre de valeurs admissibles dans `values`.
- elle n'admet pas de parallélisation satisfaisante car, dans l'expression `(multiple-value-bind` $(x_1..x_n)$ `(values` $e_1..e_n$`) corps)` qui est représentée par `((values` $e_1..e_n$`) (lambda` $(x_1..x_n)$ `corps))`, seul le calcul qui produit `(values` $e_1..e_n$`)` est initialement possible, `(lambda` $(x_1..x_n)$ `corps)` étant irréductible.

Au vu de ces défauts, nous proposons donc une nouvelle solution à coût intermédiaire mais gardant le maximum des qualités des deux autres solutions.

2.2.4 Une meilleure approche combinatoire

2.2.4.a Définitions

Le défaut le plus grave de la solution précédente est l'impossibilité de générer une version parallèle, c'est donc ce défaut là que nous allons supprimer.

Pour cela nous allons rendre possible le calcul de `(lambda` $(x_1..x_n)$ `corps)` en lui fournissant ses arguments sélectionnés dans la valeur multiple à l'aide du mécanisme `((values` $e_1..e_n$`) select`$_i$`)` = e_i qui utilise un nouvel opérateur `select`$_i$ que nous allons décrire. Si nous voulons avoir accès au typage non restrictif et comme le nombre n de valeurs dans `values` n'est pas forcément connu lors de la compilation de `select`$_i$, il nous faut définir cet opérateur indépendamment de n. Cela va nous amener à de nouvelles définitions basées sur les combinateurs que nous rappelons :

$!MS_i\ x_1..x_i = x_i\ x_1\ x_2\ x_3..x_{i-1}$

$!R_i\ x_1..x_i = x_1$

$!M_i\ x_1..x_i = x_1\ x_i\ x_2\ x_3..x_{i-1}$

$values\ e_1..e_n = !MS_{n+2}\ !R_n\ e_1..e_n$

$select_i = !M_{i+1}$

L'association de ces deux définitions permet effectivement la sélection d'une des valeurs agglomérées par l'exécution de trois combinateurs :

$((values\ e_1..e_n)\ select_i)$

$= ((!MS_{n+2}\ !R_n\ e_1..e_n)\ !M_{i+1})$

$= (!M_{i+1}\ !R_n\ e_1..e_n)$

$= (!R_n\ e_i\ e_1..e_{i-1}\ e_{i+1}..e_n)$

$= e_i$

C'est cette propriété que nous allons utiliser dans les définitions de `multiple-value-bind` et de `p-multiple-value-bind` sans omettre d'éviter la duplication de l'expression `v` (qui peut être complexe) en la liant à une variable jamais encore utilisée (ici `w`) et partagée par tous les calculs :

(multiple-value-bind (x_1..x_n) v corps)

= (let ((w v)) ((lambda (x_1..x_n) corps) (w select$_1$) .. (w select$_n$)))

(p-multiple-value-bind (x_1..x_n) v corps)

= (let ((w (processus v)))
    ((lambda (x_1..x_n) corps) (w select$_1$) .. (w select$_n$)))

Nous pouvons vérifier que cette définition conduit au résultat souhaité :

(p-multiple-value-bind (x_1..x_n) (values e_1..e_n) corps)

= (let ((w (processus (values e_1..e_n))))
    ((lambda (x_1..x_n) corps) (w select$_1$) .. (w select$_n$)))

= ((lambda (x_1..x_n) corps) (w select$_1$) .. (w select$_n$))

; w = (processus (values e_1..e_n))

= ((lambda (x_1..x_n) corps) e_1..e_n)

= (let ((x_1 e_1) .. (x_n e_n)) corps)

De plus, le calcul qui produira (values e_1..e_n) peut être lancé en parallèle avec ((lambda (x_1..x_n) corps) (w select$_1$) .. (w select$_n$)) qui est une lambda expression appliquée à suffisamment d'arguments pour être réduite.

2.2.4.b Compilation

Notre compilation se réfère toujours au 1.4 page 27. La nouvelle définition de values introduite en 2.2.4.a est :

(values e_1..e_n)

= (!MS$_{n+2}$!R$_n$ e_1..e_n)

`multiple-value-bind` et `p-multiple-value-bind` peuvent être tout simplement macrogénérés en utilisant les combinateurs !M$_i$ et !R$_i$ ainsi que les combinateurs ad-hoc !MV$_i$, !P-MV$_i$ et !ACC-C$_i$:

!M$_i$ x_1..x_i = x_1 x_i x_2 x_3..x_{i-1}

!R$_i$ x_1..x_i = x_1

!MV$_i$ x_1..x_i = x_1 (x_i x_2) x_3..x_i

!P-MV$_i$ x_1..x_i = x_1 (x_i x_2) x_3..x_i ; x_i en réduction

!ACC_C$_i$ x_1..x_i = x_1 (x_i x_2) x_3..x_{i-1}

La définition de `multiple-value-bind` est :

`(multiple-value-bind (`$x_1..x_n$`) v corps)`

= `(let ((w v)) ((lambda (`$x_1..x_n$`) corps) (w select`$_1$`) ... (w select`$_n$`)))`

= `(let ((w v)) ((lambda (`$x_1..x_n$`) corps) (w !`M_2`) ... (w !`M_{n+1}`)))`

= `(!MV`$_3$ `(... (!MV`$_{n+1}$ `(!ACC_C`$_{n+2}$ `(lambda (`$x_1..x_n$`) corps) !`M_2`) !`M_3`) ...) !`M_{n+1} `v)`

La définition de `p-multiple-value-bind` est :

`(p-multiple-value-bind (`$x_1..x_n$`) v corps)`

```
= (let ((w (processus v)))
     ((lambda (x1..xn) corps) (w !M2 ) ... (w !Mn+1)))
```

= `(!P-MV`$_3$ `(!MV`$_4$ `(... (!MV`$_{n+1}$ `(!ACC_C`$_{n+2}$ `(lambda (`$x_1..x_n$`) corps) !`M_2`) !`M_3`) ...) !`M_n`) !`M_{n+1} `v)`

Ici aussi, `values` ne peut agglomérer qu'un nombre borné de valeurs, mais est paresseuse comme désiré. De plus le coût du `let` est égal à n combinateurs exécutés.

La compilation retenue n'impose aucun lien entre le nombre `n` de variables locales au `multiple-value-bind` et le nombre de valeurs agglomérées dans la valeur multiple `v`. Elle autorise donc aussi bien le typage restrictif présenté en 2.2.1.b (et qui impose que ces deux nombres soient égaux) que le typage non restrictif présenté en 2.2.1.c (et qui permet de ne lier qu'une partie des valeurs agglomérées, les autres n'étant pas évaluées, grâce au mécanisme d'évaluation paresseuse).

Le nombre de combinateurs effectivement exécutés pour lier les variables $x_1..x_n$ va donc dépendre (grâce à l'évaluation paresseuse) du nombre n'; ($n' \leq n$) de ces variables effectivement utilisées par `corps` et est $n + 3 \times n'$.

2.2.4.c Analyse

Il résulte de cette approche :

- un temps d'accès en $O(n)$, chaque accès à un des e_i ayant un coût constant.
- un coût optimum en terme de graphe par utilisation d'un nombre minimum de noeuds.
- une parallélisation effective.
- elle est compatible avec les deux ensembles de règles de typage.

La seule limite restante est :

- de par la compilation sous-jacente, elle limite le nombre de valeurs admissibles dans `values`.

2.2.5 Conclusion

Nous avons donc vu trois versions des valeurs multiples. La première à base de `cons` a un coût en $O(n^2)$ et la seconde n'est pas parallélisable, en revanche la troisième (cf. 2.2.4 page 56) offre un bon compromis notamment par son coût en $O(n)$, l'interchangeabilité entre `multiple-value-bind` et `p-multiple-value-bind` (qui est nécessaire pour paralléliser simplement nos programmes) et la compatibilité avec le typage le moins restrictif (cf. 2.2.1.c page 52).

2.3 Les structures

2.3.1 Description

Nous voulons implanter dans notre modèle sous forme de graphes combinatoires des structures à la Common Lisp *[STEELE 78]*, c'est à dire des objets d'un type composé dont les composantes sont nommées.

Différentes approches visent à implanter diverses structures de données complexes dans des modèles purement fonctionnels. Nous pouvons par exemple citer *[GRUE 89]* qui donne une représentation des tableaux dans un domaine ne contenant que deux constructeurs `cons` et `nil`. Une telle approche peut tout à fait être envisagée dans un cadre plus large mais est basée sur l'existence d'un nombre fini de constructeurs. Notre modèle n'entre pas dans cette catégorie (notre lambda calcul est étendu à diverses constantes dont les nombres en quantité virtuellement infinie) et de ce fait nous ne pourrons nous inspirer de ce type d'approche.

Les opérations minimales sur les structures sont la définition de nouveaux types structure par `defstruct` et les opérateurs de manipulation des types structures nouvellement introduits. Ainsi, suite à la définition du type de structure `type` par (`defstruct type` $composante_1$..$composante_n$), la création d'une instance par `make-`*type*, l'accès à une composante d'une instance par *type-composante*$_i$ et la modification d'une composante d'une instance par `set-`*type-composante*$_i$.

Cette dernière opération diffère par sa syntaxe des structures Common Lisp qui réalisent la modification par la méthode générique `setf` appliquée à l'accès à la composante par *type-composante*$_i$. De plus, notre cadre étant fonctionnel il s'agira pour nous d'une modification par construction d'une copie et non par modification physique de l'instance initiale comme c'est le cas en Common Lisp.

Il faut également remarquer que `defstruct` n'est pas un opérateur mais ne sert qu'à introduire les opérateurs sur les instances. Nous n'aurons donc pas à exprimer de définition pour `defstruct`, et nous nous contenterons de définir les autres opérateurs tels que devra les générer l'expression (`defstruct type` $composante_1$..$composante_n$).

Remarquons enfin que cette définition des structures correspond à une vision non typée, une vision typée impliquant une définition telle que (defstruct type (composante$_1$ type$_1$) ... (composante$_n$ type$_n$)). Nous ne nous intéresserons qu'au cas non typé, le cas typé n'introduisant aucune contrainte supplémentaire au niveau de la réalisation des structures.

2.3.2 Une solution simple

Comme pour les valeurs multiples, il est possible de construire des structures en utilisant uniquement des cellules cons par exemple à l'aide des définitions suivantes :

```
(make-type valeur1...valeurn)

= (list valeur1...valeurn)

= (cons valeur1 (cons ... (cons valeurn nil)))


(type-composantei instance)

= (nth i instance)


(set-type-composantei instance valeur)

= (letrec ((replace (lambda (ins n) ;; letrec est un let récursif
                      (if (eq0 n)
                          (cons valeur (cdr ins))
                          (cons (car ins)
                                (replace (cdr ins) (sub1 n)))))))
    (replace instance i))
```

Nous ne détaillerons pas cette solution d'avantage du fait de son coût prohibitif lors des accès et modifications.

2.3.3 Une solution combinatoire

2.3.3.a Définitions

Cette approche a été présentée dans *[COUSTET 90]*. Les opérateurs sont définis comme suit :

```
(make-type valeur1...valeurn)
= (lambda (f) (f valeur1...valeurn))

(type-composantei instance)
= (instance (lambda (x1...xn) xi))

(set-type-composantei instance valeur)
= (instance (lambda (x1...xn f) (f x1...xi-1 valeur xi+1...xn)))
```

Il est facile de vérifier que ces définitions implémentent bien des structures, par exemple :

(type-composante$_i$ (make-type valeur$_1$..valeur$_n$))

= ((make-type valeur$_1$..valeur$_n$) (lambda (x_1..x_n) x_i))

= ((lambda (f) (f valeur$_1$..valeur$_n$)) (lambda (x_1..x_n) x_i))

= ((lambda (x_1..x_n) x_i) valeur$_1$..valeur$_n$)

= valeur$_i$

2.3.3.b Compilation

La compilation de ces définitions peut être faite par le compilateur MaRS_Lisp qui utilisera alors le jeux standard de combinateurs. Toutefois l'utilisation de combinateurs ad-hoc, tirant partit des possibilités de micro-codage, est nécessaire à l'obtention de performances raisonnables.

Cette compilation est présentée Annexe A dont la partie A.3 contient un ré[illegible] des codes combinateurs des différents opérateurs.

2.3.3.c Performances de l'approche

Les performances décrites ici découlent de la compilation optimisée présentée en Annexe A.

[illegible]

sidéré dans *[ROE 91]* comme un coût raisonnable en vue d'exécutions parallèles et dont la taille est telle que pour *Lmax* composantes, il y a environ 1 pointeur et 3 combinateurs.

2.4 Les vecteurs

Les vecteurs ont de grandes similitudes avec les structures (dont la compilation est en Annexe A), la différence essentielle étant que les accès ne se font plus par le nom de la composante, opérateur figé à la création du nouveau type structure, mais par son numéro. Nous allons donc nous inspirer des résultats des structures en montrant comment construire les séquences d'accès à partir de l'indice d'un élément, le corps du vecteur lui-même étant calqué sur le modèle des structures.

2.4.1 Le vecteur

Nous construirons nos vecteurs à l'aide de l'opérateur `make-vector` selon deux modes possibles :

- `(make-vector k $initial-element e)` construit un vecteur de taille k et dont tous les éléments sont `e`.
- `(make-vector k $initial-values l)` construit un vecteur de taille k et dont les éléments sont les éléments de la liste `l`.

Dans les deux cas, nous allons construire une arborescence à k composantes dont les k feuilles seront soit la valeur `e` (premier cas) soit les valeurs $\texttt{valeur}_1 \ldots \texttt{valeur}_k$ issues de la liste `l` (second cas). Cette arborescence sera construite sur le modèle des structures (cf. Annexe A).

La structure arborescente d'un vecteur $A[n,a,b]$ qui contient les feuilles d'indice $a...b$ et est constitué de $\eta = \lceil (1+b-a) / (Lmax)^{n-1} \rceil$ sous-arbres, est représenté récursivement par :

$\texttt{A[n,a,b]} = (\texttt{!MS}_{\eta+3}\ \texttt{!R}_{\eta+1}\ \texttt{!MS}_{\eta+3}\ \texttt{A[n-1,a+0*Lmax}^{n-1}\texttt{,a-1+1*Lmax}^{n-1}\texttt{]}\ \texttt{A[n-1,a+1*Lmax}^{n-1}\texttt{,a-1+2*Lmax}^{n-1}\texttt{]}\ \ldots\ \texttt{A[n-1,a+}(\eta-1)\texttt{*Lmax}^{n-1}\texttt{,b]})$

Un sous-arbre de profondeur 1 contenant les feuilles f_x à f_y est représenté par :

$\texttt{A[1,x,y]} = (\texttt{!MS}_{y-x+4}\ \texttt{!R}_{y-x+2}\ \texttt{!MS}_{y-x+4}\ f_x \ldots f_y)$

2.4.2 L'opérateur vref

Une fois la forme du vecteur connue, il nous reste à déterminer les opérateurs d'accès. Ici et contrairement aux structures, l'accès ne se fait plus par le nom du champ mais par son indice. Dans le cas où cet indice ainsi que la taille du vecteur sont connus dès la compilation, le problème est identique au problème des structures : du fait de l'identité de forme entre vecteurs et structures, il suffit de compiler `(vref v i)` de la même façon que nous compilerions $(\texttt{type-composante}_i\ \texttt{v})$. Par contre, si ces deux informations ne

sont pas connues dès la compilation (le problème ne se posant que pour l'indice, la taille du vecteur pouvant être comprise comme faisant partie intégrante de son type) il nous faudra être capable de générer dynamiquement lors de l'exécution la bonne séquence d'accès. Cette séquence sera la même que la séquence d'accès définie pour une structure (cf. Annexe A), soit $\{i_1 \ldots i_p\}$ et devra être traduite en le même opérateur d'accès $!MS_{1+p} !M_{2+i_1} \ldots !M_{2+i_p}$.

La profondeur `p` ne dépend que du nombre de composantes du vecteur et est donc connue, le combinateur $!MS_{1+p}$ peut donc être généré à la compilation. Le problème se pose par contre pour générer les `p` combinateurs $!M_{2+i_j}$ qui ne peuvent l'être que dynamiquement.

Une première façon simple de réaliser cette opération est de le faire avec un `cond`[1] :

```
(cond ((= i 1) !M3)
      (= i 2) !M4)
        ...
      (= i n) !M2+n)))
```

Il est également possible d'utiliser tout mécanisme ad-hoc, par exemple nous pourrons construire le combinateur $!M_5$ en utilisant un combinateur qui réalise un `ou` logique entre ses deux arguments : `(logOR !M0 5)`.

2.4.3 L'opérateur vset

L'opérateur `vset` est celui qui sera utilisé pour modifier la valeur d'un des éléments du vecteur[2].

Nous utiliserons pour cet opérateur le même mécanisme de création dynamique que pour `vref`. Dans un premier temps nous déduisons de la profondeur `p`, qui ne dépend que du type du vecteur et est donc connue statiquement, et de l'indice `i` de l'élément la séquence d'accès à la $i^{ème}$ composante du vecteur, qui est la même que celle définie en 2.3.3.b pour l'accès à $composante_i$ d'une structure.

A partir de cette séquence, il suffit ensuite de générer dynamiquement la définition en combinateurs de $set\text{-}composante_i$ telle qu'elle est donnée en Annexe A par les mêmes techniques que celles mises en œuvre pour l'opérateur `vref`.

1. Il faut noter qu'une telle solution n'est pas irréaliste car si la première exécution d'un tel code à un coût en $O(n)$, du fait du mécanisme de réduction de graphe les utilisations ultérieures auront un coût nul.

2. La remarque faite pour les structures s'applique ici également : dans notre cadre fonctionnel, la modification correspond à la construction d'un nouveau vecteur qui partage avec le vecteur original ses constituants non modifiés.

2.5 Conclusion

En conclusion, nous avons montré la nécessité de structures de données diverses et performantes par le lien existant entre le coût d'accès aux données et le taux de parallélisme envisageable par un algorithme. Nous avons vu que les structures de données construites à base de cellules cons ne pouvaient satisfaire ces besoins et nous avons donc montré comment les graphes de combinateurs indicés peuvent y répondre.

Il apparaît ainsi que notre modèle peut avantageusement être étendu selon les besoins avec toute structure de donnée compilée en graphes de combinateurs, à la seule condition de pouvoir en exprimer le fonctionnement en lambda expressions ou toute autre expression équivalente.

Chapitre 3

Augmenter le parallélisme effectif par diminution des phases de synchronisation

Objectif : Exposer les acquis en parallélisation, montrer qu'il n'y a pas de solution à certaines situations défavorables. Donner une solution simple à deux cas non encore résolus. Montrer le gain obtenu sur des exemples conséquents.

3.1 Le coût des processus en programmation fonctionnelle

Dans toute machine, la gestion des processus a un coût en temps système et en matériel qui est très lié au type de parallélisme pris en compte par le modèle et aux choix d'implémentation.

Nous trouverons par exemple dans *[CHEN 88]* l'étude d'un modèle d'hypercube à mémoire distribuée avec espace d'adressage commun dans lequel l'essentiel du problème est lié à la métrique des coûts associée aux communications, dans *[KELLY 89]* une étude complète des architectures faiblement couplées, et dans *[HUDAK 86]* un ensemble d'annotations permettant d'exprimer une répartition de processus sur un réseau de processeurs.

Pour sa part, *[BURTON 84]*, *[BURTON 85]*, *[BURTON 86]* et *[BURTON 87]* étudie comment mélanger les différentes stratégies d'évaluation (par nom, par valeur séquentiel, par valeur parallèle et par spéculation) pour contrôler les coûts de communications entre processus, l'espace mémoire consommé ou le nombre d'instructions exécutées dans une architecture répartie.

Nous ne nous intéresserons dans la suite qu'aux problèmes concernant notre modèle.

3.1.1 Le problème général

Si la source de performances en algorithmique parallèle provient de la possibilité de lancer des processus parallèles exécutant ainsi plusieurs parties

du programme concurremment, cette possibilité est également la source de nombreux problèmes liés à la bonne utilisation de ces ressources.

La nature du problème est fortement lié au modèle d'exécution visé et notamment au nombre de processeurs disponibles, mais nous pouvons néanmoins isoler quelques constantes. Ainsi le parallélisme effectif d'un programme est très lié, non seulement au nombre de processeurs, mais également à tous les coûts de gestions associés aux processus. De ce coût nous pouvons déduire une granularité minimum accessible qui correspond à la taille de processus en dessous de laquelle le parallélisme engendré ne compense plus les coûts associés. Par exemple l'implémentation de QLISP sur ALLIANT FX/8 est limitée au parallélisme à grains moyen et gros *[GOLDMAN 88]*.

Ce problème de granularité limite (parfois fortement) les potentialités de parallélisme des algorithmes et peut ainsi conduire à l'impossibilité d'engendrer un parallélisme important, ce qui interdit d'envisager le parallélisme massif. De plus, si le coût d'une tâche mise en attente d'une ressource processeur est élevé, cela impose de limiter le nombre de processus générés.

De nombreuses études ont porté sur ces points, *[HALSTEAD 86]*, *[HALSTEAD 87]*, *[MOHR 90]* étudient notamment les problèmes liés au coût des créations de processus sur diverses implémentations de Multilisp.

De plus le nombre de processus engendrés et leur granularité lors de l'évaluation d'un programme fonctionnel est généralement très lié aux données particulières utilisées. Une parallélisation effectuée avec quelques jeux de test peut ainsi être totalement inadaptée à des données d'un autre ordre de grandeur.

3.1.2 Les particularités du modèle MaRS

Vis à vis de la granularité, notre modèle est particulier car il fait le pari que le parallélisme massif ne pourra être envisagé qu'à la condition de pouvoir prendre en compte des processus de grain suffisamment fin. De cette manière nous pensons pouvoir alimenter en calculs un grand nombre de processeurs avec la plupart des algorithmes et non pas en étant limité à quelques algorithmes favorables. Sans autoriser un grain aussi fin, cette démarche nous rapproche du grain de parallélisme accessible au modèle dataflow *[LAU 80]*, *[DENNIS 74]* (pour sa part la Connection Machine *[HILLIS 85]* *[HILLIS 86]* autorise également un grain très fin mais de par son architecture hypercube SIMD elle n'est pas apte à traiter tous les types d'algorithmes).

Pour pouvoir atteindre cet objectif, les architectes de MaRS ont choisi de câbler tous les processus dans les processeurs (à raison de huit processus par processeur pour recouvrir les temps de communications). De cette façon la gestion des processus est très peu coûteuse : le processeur pouvant activer un processus prêt en un seul cycle machine et la création d'un processus ne

prend guère plus que le temps de transit sur le réseau du message de création de processus (cf. 1.7.2 page 41).

Ce choix architectural, qui nous permet d'avoir une gestion des processus à coût minimal, introduit par contre des contraintes nouvelles. Ainsi les processus étant physiquement implantés dans le processeur de réduction, leur nombre est obligatoirement limité et il ne peuvent libérer la ressource que lorsqu'ils terminent l'évaluation de leur expression.

La conséquence la plus importante est qu'un processus en attente bloque sa ressource sans réaliser le moindre calcul. Si la proportion de ces processus est faible, chaque processeur peut quand même être actif s'il dispose d'assez de processus éligibles pour occuper toute l'activité. Mais lorsque cette proportion croît, de plus en plus de processeurs voient leur taux d'activité diminuer jusqu'à éventuellement ne plus gérer que des processus en attente et devenir ainsi totalement inactifs.[1]

Dans notre modèle nous pourrons donc quasiment ignorer les problèmes de granularité des processus mais nous devrons toujours veiller à ne générer que des processus le plus actifs possibles, quitte à en générer moins.

3.1.3 Eléments de solutions

3.1.3.a La compilation automatique

Dès les premières études algorithmiques menées dans le cadre de notre modèle, l'accent a été mis sur le taux d'activité des processus générés.

Ainsi une première approche consistant en une parallélisation automatique de l'algorithme par le compilateur est possible *[COUSIN 88]*, *[COUSIN 91]*. Les processus générés par ces algorithmes sont non spéculatifs (la création d'un processus est conditionnée par la stricticité des expressions) mais ils peuvent avoir un taux d'activité très faible : seuls sont éliminés les processus candidats sur des arguments *strict immédiat*, c'est à dire dont la valeur est immédiatement utilisée par la fonction (qui ne peuvent conduire qu'à un taux d'activité proche de zéro).

D'autres études ont cherché à modéliser la durée d'évaluation d'une expression (permettant ainsi de décider de l'opportunité de certains processus) *[GOLDBERG 88]*, *[GRAY 86]* mais à notre connaissance il n'est, dans le cas général, pas possible de qualifier les expressions strictes plus finement qu'en *rapide* (plus rapide à évaluer en séquentiel qu'en parallèle), *strict* (nécessaire au résultat) et *strict immédiat* (nécessaire au résultat mais ne doit pas être évalué en parallèle car utilisé aussitôt) comme le font les algorithmes mis en œuvre dans *[COUSIN 91]*.

1. L'allocation des processus aux processeurs, qui est automatique et dynamique, est faite en cherchant une équi-répartition. Il se peut très bien qu'un processeur n'ait que des processus inactifs à gérer, et soit donc lui-même inactif, alors que d'autres processeurs gèrent plus de processus éligibles que nécessaire pour assurer leur pleine activité.

D'autres approches, qui sont également tributaires d'une estimation de la durées des expressions, telles que les *Serials Combinators [HUDAK 85]* permettent d'utiliser cette estimation pour générer le grain de parallélisme le plus adapté à la machine cible.

Le vrai problème reste de déterminer quand une expression est *rapide*, ce qui est approximé le plus souvent en considérant comme rapide :

- toute primitive
- toute constante
- toute combinaison d'expressions *rapides*
- De plus il peut être pris en compte le fait qu'une expression partagée déjà évaluée par ailleurs est *rapide*.

Il n'en reste pas moins que toute compilation automatique basée sur ce type d'analyse engendrera selon l'algorithme un certain taux de processus en attente qui avec notre modèle sont préjudiciables au taux de parallélisme effectif.

3.1.3.b L'utilisation de paradigmes

Une deuxième approche a été développée *[MaRS 89a], [MaRS 89b], [MARRE 92]* visant à identifier les paradigmes de l'algorithmique parallèle adaptés à notre modèle et à définir leurs conditions de bonne application. Il en ressort la nécessité de faire appel à des heuristiques pour choisir entre diverses stratégies de parallélisation.

- Pas de parallélisation des appels de fonction

  Si la fonction n'a qu'un seul argument ou n'a pas d'argument strict.
- Parallélisation horizontale des appels de fonction

  Il s'agit de lancer en parallèle des arguments stricts d'une fonction avant que le corps de la fonction ne soit exécuté, le processus père se chargeant d'évaluer lui-même au moins un des arguments stricts.

  Cette stratégie sera utilisée si tous les arguments stricts sont *strict immédiat*.
- Parallélisation verticale des appels de fonction

  Il s'agit de lancer en parallèle des arguments stricts d'une fonction avant que le corps de la fonction ne soit exécuté, le processus père évalue alors le corps de la fonction.

  Cette stratégie sera utilisée si certains arguments stricts ne sont pas *strict immédiat*.
- Parallélisme producteur-consommateur sur les données

  Ce parallélisme se met en œuvre avec les constructeurs de données paresseux (`ll-cons` par exemple, qui est introduit en 1.4.2). Cette technique réalise un étage de pipe-line entre deux traitements à travers une structure de données intermédiaire.

- Fusion-scission des pipe-lines

  Bien que le coût de gestion des processus soit très faible dans notre modèle, le parallélisme pipe-line peut nécessiter une fusion regroupant deux étages en un seul de manière à éviter une structure de données intermédiaire dans le cas où le parallélisme engendré est à trop petit grain.

  Certaines transformations *[BIRD 88], [LEMAITRE 90]* à base de fonctions d'ordre supérieur permettent de regrouper des étages de pipe-line en un seul traitement. Nous pouvons également utiliser les transformations proposées dans *[WADLER 84]* et *[WADLER 87a]* dans le cadre du langage séquentiel Miranda *[TURNER 85a], [TURNER 85b]*.

  A l'inverse, si un étage de pipe-line effectue de gros calculs, il peut être intéressant de le scinder en deux étages, le parallélisme engendré compensant alors le coût de la nouvelle structure de données intermédiaire.

- Limitation du parallélisme explosif

  Il s'agit de modifier un programme au parallélisme explosif, qui engendrerait des processus à mauvais taux d'activité, de manière à limiter le nombre de processus créés. Notre modèle permet une gestion dynamique et peu coûteuse de ces processus, le but visé est ici d'éviter l'accumulation des processus en attente qui ont tendance avec de tels programmes à occuper une part croissante des ressource au cours de l'exécution.

  D'autres modèles, où les processus ne sont pas en nombre limité mais où leur coût est élevé ont le même problème et nécessitent également de pouvoir freiner la génération de processus au dessus d'un certain seuil (voir la notion de *lazy task creation* en Multilisp *[MOHR 90]* ou la création conditionnelle des processus en QLISP *[GABRIEL 84], [GABRIEL 88]*).

L'utilisation de cet ensemble de techniques pour paralléliser un algorithme peut donner des résultats bien meilleurs qu'une compilation automatique, cependant l'absence d'outils automatique d'analyse (certains points, comme estimer quel argument d'une fonction sera généralement le plus long à évaluer, ne sont pas automatisables) rend cette approche très lourde du fait des nombreux choix possibles à chaque niveau (parallélisation horizontale ou verticale des appels de fonction, fusion ou scission des pipe-lines, ...).

Il en ressort donc que le gain en performances que nous pouvons obtenir par utilisation de ces paradigmes ne peut se faire qu'au prix d'un important coût de développement et de mise au point. De plus certains cas défavorables n'ont pas de solution satisfaisante à travers ces techniques et doivent être résolus par une autre approche.

3.2 Contrôler le déclenchement des processus

3.2.1 Les moyens du contrôle

Limiter le nombre de processus générés par un programme est une préoccupation commune à de nombreux modèles fonctionnels parallèles. Par exemple, toute directive de parallélisation exprimée en QLISP *[GABRIEL*

84], est conditionnée par une expression booléenne. Le programmeur peut ainsi exprimer un calcul qui va selon certains paramètres être fait en séquentiel ou en parallèle[1].

Nous pouvons écrire en QLISP une fonction de fibonacci parallèle dont le nombre de processus générés est contrôlé (cf. Figure 1).

Figure 1 Fibonacci en QLISP avec génération contrôlée du parallélisme

```
(defun qfib (m n)
  (if (≤ m 1) 1
      (qlet (> n 0)
            ((a (qfib (- m 1) (- n 1)))
             (b (qfib (- m 2) (- n 1))))
        (+ a b))))
```

Cette version calcule `(fib m)` avec au plus 2^n processus. L'opérateur remarquable dans cette fonction est `qlet` qui est similaire à un `let` si son premier argument s'évalue à faux et est un `let` parallèle (un processus par variable locale) s'il s'évalue à vrai.

D'autre approches visent à limiter la génération effective de processus indépendamment des choix de parallélisation du programmeur. Ainsi Multilisp ne permet pas d'exprimer directement une création conditionnelle de processus mais en limite quand même dynamiquement le nombre grâce à la notion de *lazy task creation [MOHR 90]*.

Ainsi les opportunités de parallélisme désignées par le programmeur grâce à la directive `future` peuvent ou non donner lieu à la création effective d'un processus parallèle selon la charge d'exécution au moment où l'expression est touchée.

De la même manière avec notre modèle, les expressions annotées avec la directive `processus` peuvent ou non et selon la charge d'exécution engendrer du parallélisme (cela correspond au phénomène de saturation qui se produit lorsque toutes les ressources en processus sont allouées).

3.2.2 Conséquences du contrôle

Si ces modèles prévoient une limitation du parallélisme généré, la justification en est variable. Dans la plupart des cas, il s'agit de diminuer globale-

1. En QLISP, il n'est pas possible de faire intervenir parmi ces paramètres des grandeurs telles que la charge d'exécution. La parallélisation obtenue ne peut donc s'adapter à une variation des ressources disponibles.

ment le coût cumulé des créations de processus, et ainsi d'accélérer le programme.

Le cas de Multilisp est typique puisque l'objectif assigné au mécanisme de *lazy task creation* est de diminuer le coût moyen de l'annotation `future` en fusionnant dynamiquement certains processus. Ainsi un grain de parallélisme plus fin se trouve accessible au niveau algorithmique, ce qui augmente le taux de parallélisme effectif et rend l'approche plus universelle.

Les choix faits pour QLISP sont tout autres et privilégient un contrôle fin du parallélisme généré par rapport à la possibilité de prendre en compte du parallélisme de grain fin. Chaque création (ou non création) de processus correspondant à l'évaluation d'une primitive parallèle donne lieu à l'évaluation préalable d'une expression booléenne qui accroit le coût moyen des processus générés. De ce fait le parallélisme à grain fin est inaccessible ce qui fait de QLISP un langage parallèle limité au grain moyen sur des architectures peu parallèles à processeurs puissants (par exemple ALLIANT FX/8). Le défaut majeur de tels choix est que certains algorithmes qui disposent d'un important potentiel de parallélisme, mais à grain fin, ne peuvent tirer aucun profit d'une parallélisation en QLISP.

A l'opposé de ces deux approches, notre modèle vise à prendre en compte un parallélisme de grain aussi diversifié que possible de manière à répondre aux potentialités d'une majorité d'algorithmes. Nous avons alors choisi de diminuer au maximum le coût de gestion des processus en acceptant en contrepartie une limitation en nombre des processus crées (cf. 1.7 page 39). Cela nous oblige donc à limiter la création des processus, non pas pour en diminuer le coût total de gestion, mais pour réserver les ressources aux plus actifs.

Les demandes de processus qui sont effectuées lorsque la charge d'exécution ne permet pas de les satisfaire (toutes les ressources étant allouées) conduisent alors à une mise en attente et à un éventuel regroupement de processus au gré de l'évolution du calcul de façon similaire à ce qui se passe avec la *lazy task creation* de Multilisp.

Pourtant, si ce mécanisme permet de gérer de façon homogène une situation d'exception, son rôle n'est pas d'optimiser l'exécution, ni par diminution du coût de gestion des processus, ni en augmentant l'activité des processus alloués. Le problème principal de la parallélisation reste donc de générer des processus aussi actifs que possible.

3.2.3 Le coût de la parallélisation

Les efforts consentis par le programmeur en vue de la limitation de création des processus par son programme induisent des coûts humains supplémentaires lors de la phase de parallélisation. Ce n'est pas le cas en Multilisp (qui règle le problème implicitement et dynamiquement), en QLISP par contre le programmeur doit définir explicitement la limitation.

Le défaut de ce contrôle est qu'il induit un comportement indépendant de la charge d'exécution, et ce au prix d'un double effort : il faut détecter les opportunités de parallélisme de grain suffisant et définir les conditions de déclenchement du parallélisme.

Il en découle de nombreux inconvénients :

- Une diminution de la clarté du programme.
- La nécessité d'expérimenter les expressions de contrôle, ce qui conduit à des coûts de développement accrus.
- Une difficulté à prendre en compte les ressources disponibles au début de l'exécution et impossibilité de tenir compte d'une variation de ces ressources en cours d'exécution.
- Une augmentation du coût effectif de gestion du parallélisme.

L'ensemble de ces restriction conduit en QLISP à réserver l'évaluation parallèle à un petit nombre de tâches de grain gros ou moyen.

En Multilisp, la *lazy task creation* diminue légèrement la granularité accessible aux algorithmes les plus parallèles (car elle ne rentre en action que lorsque la machine est saturée) mais ce problème étant surtout crucial pour les algorithmes les moins parallèles, nous pouvons considérer qu'elle entraîne un gain réel mais non fondamental.

Son avantage principal est d'être totalement transparente au programmeur qui doit néanmoins effectuer une analyse de son algorithme pour y détecter les meilleures opportunités de parallélisme selon des critères de granularité des tâches engendrées, ce qui conduit à une phase délicate d'expérimentation.

Le cas de MaRS_Lisp est plus complexe. Si la gestion du parallélisme ne pose pas de problème, par contre le programmeur doit passer par une phase de parallélisation de son algorithme pour atteindre des performances optimales. Cette phase délicate dont le but est de détecter les opportunités favorables de parallélisme, non pas selon des critères de granularité suffisante mais de taux d'activité, impose aussi une expérimentation des choix.

L'apport principal de l'approche fonctionnelle dans la parallélisation étant probablement la facilité d'expression, ce qui est assez peu compatible avec ces phases d'expérimentation qui ne font que trahir une difficulté certaine à obtenir le fonctionnement parallèle désiré, nous allons (comme le suggérait déjà *[MARRE 92]*) proposer une construction introduisant du parallélisme efficacement mais avec une phase d'analyse et d'expérimentation aussi restreinte que possible.

3.2.4 Une approche optimisante

3.2.4.a Contraintes

Notre construction doit répondre à certaine contraintes :

- Le coût de gestion des processus doit rester aussi faible que possible.
- La parallélisation d'un algorithme doit s'adapter automatiquement aux ressources disponibles au début des calculs mais aussi à l'évolution de ces ressources en cours de calcul.
- La parallélisation doit se faire après une phase d'analyse aussi restreinte que possible et sans être suivie d'une phase de réglage des paramètres.

3.2.4.b Propriétés

Le déclenchement d'un processus a dans notre modèle les propriétés suivantes que nous devrons prendre en compte :

Si le nouveau processus se déroule entièrement avec la machine non complètement chargée, un taux d'activité même très faible ne peut ralentir le calcul (on gaspille une ressource qui resterait inutilisée).

Cependant cette propriété dépend des ressources disponibles et nous ne pouvons donc pas la mettre à profit pour paralléliser indépendamment des ressources.

Lorsqu'un processus fils est lancé, si son père à besoin de son résultat avant qu'il ne soit calculé, il doit attendre et bloque une ressource. Nous avons donc en première approche envie d'en conclure que si le fils est assez rapide, le parallélisme engendré est favorable.

Cependant, un processus demandé lorsqu'il n'y a plus de ressources peut soit démarrer de façon différée soit être séquentialisé. Le second cas ne pose aucun problème, par contre dans le premier cas le processus fils peut démarrer juste avant que son résultat soit nécessaire et donc bloquer une ressource pour rien. Ce comportement ne peut être prévu statiquement, de plus il est variable en fonction des ressources.

Ce type de parallélisme ne peut pourtant pas être rejeté car il en constitue la source principale. Le problème vient de ce que parmi ces processus potentiels certains s'avèrent favorables après analyse et d'autres défavorables.

Si nous voulons optimiser l'exécution, le passage obligé est donc une phase de mise au point délicate dont le résultat est un programme parallélisé statiquement qui n'est pas optimum tout au long de l'exécution et qui peut être mis en défaut par une configuration défavorable des ressources ou un jeu de données différent.

En fait il n'y a pas de source idéale de parallélisme et nous devons nous contenter de partitionner les processus en probablement favorable (que nous savons parfaitement prendre en compte avec la directive `processus`) et défavorable (ou peu favorable). Ces derniers sont souvent détectables statiquement et la seule façon de les prendre en compte est soit de les rejeter en bloc si le parallélisme ainsi obtenu est suffisant, soit de passer par la phase de mise au point en envisageant de déclencher ou non chacun des processus les moins défavorables.

Ce sont donc ces derniers processus que nous voulons prendre en compte à l'aide d'une construction au comportement dynamique et de manière à ne pas perturber le fonctionnement des processus ordinaires.

3.2.4.c La construction

Nous allons envisager une nouvelle classe de primitives parallèles qui déclenchent un processus si la charge est assez faible (par exemple la moitié de la charge maximale). Dans le cas contraire, la demande n'est pas prise en compte et le traitement reste séquentiel.

Nous définissons ainsi une nouvelle version pour chaque opérateur parallèle qui se comporte comme la version parallèle normale si la charge d'exécution est assez faible et comme la version séquentielle sinon. Dans ce dernier cas, il n'y a aucune création de processus, pas même différée jusqu'à une prochaine baisse de l'activité (cela pour ne pas compliquer, donc ralentir, le traitement des processus en attente d'une ressource).

Ainsi le comportement est dynamique, s'adaptant aux variations de ressources au cours du calcul et les processus ainsi déclenchés perturbent au minimum le déroulement des processus normaux car ils ne sont utilisés que pour compléter la charge. Nous désignerons nos processus sous le terme de *processus complémentaire.*

3.2.5 Conséquences sur la parallélisation

Avec la possibilité de lancer des processus pour compléter la charge, nous pouvons simplifier grandement la parallélisation des programmes.

La phase d'analyse du code qui vise à détecter les potentialités de parallélisme très favorables, les possibilités moins favorables et les possibilités défavorables doit être maintenue mais cette partition est moins critique car les différences de prise en compte sont plus graduées.

Une des limitations majeure de la parallélisation que nous effectuons en l'absence de construction parallèle dynamique, est que toute parallélisation d'algorithme se fait avec l'hypothèse implicite d'un certain nombre de ressources processus disponibles, ce qui dans la plupart des cas amène à une version parallèle dont le comportement n'est pas reproductible d'une configuration à l'autre.

A l'inverse la présence d'une telle construction va nous permettre d'envisager la parallélisation d'algorithmes indépendamment des ressources.

3.3 Exemple

Pour illustrer notre construction, nous allons prendre un exemple déjà longuement étudié : le programme *cecilia [MaRS 90a], [MARRE 92]* de la société

AMD-BA qui réalise la mise sous forme disjonctive minimale d'un arbre de défaillance et est utilisé pour fiabiliser des systèmes complexes.

Les deux études approfondies portant sur la parallélisation de l'application sont quasi-concordante et permettent d'établir un ensemble d'opportunités de bon parallélisme, d'autres opportunités sont détectées comme moins favorables car susceptibles d'induire des attentes, d'autre enfin sont détectées comme défavorables.

Ces analyses, basées sur la notion de stricticité immédiate sont chacunes suivies d'une évaluation comparative de deux versions parallèles du programme, une qui ne retient que les opportunités les plus favorables de parallélisme (version prudente), l'autre qui retient également les opportunités presque favorables (version moins prudente).

L'évaluation est faite sur plusieurs jeux de données qui donnent des résultats variables quant à la meilleure version. Il en ressort qu'à partir d'une taille suffisante des calculs, la version la plus performante est la version prudente. L'explication qui en est faite est qu'au delà d'une certaine taille des données, les meilleures opportunités de parallélisme suffisent à saturer la machine.

Nous conserverons les processus de la version prudente donnée dans *[MaRS 90a]* et rajouterons deux processus complémentaires dont l'emplacement à été déterminé comme étant presque favorable lors des deux analyses précédentes.

3.3.1 Le programme

Le programme tel que nous le présentons (cf. Figure 2) est directement dérivé de *[MaRS 90a]* et est testé en trois versions qui sont la version prudente, la version moins prudente et la version avec processus complémentaires.

Sont figurés en majuscules les processus communs aux trois versions et en minuscules les processus qui sont absents de la version prudente, qui figurent dans la version moins prudente en tant que processus et en tant que processus complémentaire dans la troisième version.

Nous avons réalisé les tests des trois versions sur trois jeux de données (arb-p qui est petit, arb-m qui est moyen et arb-g qui est aussi gros que nous le permet le simulateur[1]) et sur trois configurations de ressources distinctes (128 processus, 64 processus et 32 processus) de manière à faire ressortir les qualités de nos trois versions indépendamment des données et des ressources.

1. De l'ordre du million d'opérateurs Lisp (de primitives du langage source) exécutés.

Figure 2 Le programme cecilia

```
(defun p-map (f l)
  (reduce (lambda (a r) (cons (PROCESSUS (f a)) (PROCESSUS r)))
          l nil))

(defun dans? (a p)  (declare (strict a p))
  (reduce (lambda (x r) (declare (strict x))
            (or (eq a x) r))
          p nil))

(defun inclus? (p1 p2) (declare (strict p1 p2))
  (reduce (lambda (x r) (declare (strict x))
            (and (dans? x p2) r))
          p1 t))

(defun term (t1) (list (list t1)))

(defun ote (t1 p) (declare (strict t1 p))
  (cond ((null p) nil)
        ((eq t1 (first p)) (rest p))
        (t (cons (first p) (PROCESSUS (ote t1 (rest p)))))))

(defun T-et-P (t1 p) (declare (strict t1 p))
  (cons t1 (PROCESSUS (ote t1 p))))

(defun P-et-P (p1 p2) (declare (strict p1 p2))
  (reduce (lambda (x r) (T-et-P x (PROCESSUS r)))p1 p2))

(defun P-ou-S (p s) (declare (strict p s))
  (cond ((null s) (list p))
        ((inclus? (first s) p) s)
        (t (let ((slp (PROCESSUS (P-ou-S p (rest s)))))
             (if (inclus? p (first s)) slp
               (cons (first s) slp))))))

(defun P-et-S (s p) (declare (strict s p))
  (reduce P-ou-S (p-map (P-et-P p) s) nil))

(defun ou (s1 s2) (declare (strict s1 s2))
  (reduce (lambda (x r) (P-ou-S x (processus r)))
          s1 (processus s2)))

(defun et (s1 s2) (declare (strict s1 s2))
  (reduce ou (p-map (P-et-S ps1) s2) nil))
```

3.3.2 Expérimentations

Nous reproduisons ci-après les résultats d'exécution. Pour chaque donnée traitée, les résultats correspondent respectivement à la version prudente, à la version avec processus complémentaires et à la version moins prudente.

La colonne *cycles* correspond à une estimation relative des durées d'exécution, *parallélisme* est le nombre moyen de processus actifs pendant la durée du calcul, *attente* est le nombre moyen de processus en attente pendant la durée du calcul, *efficacité* est le rapport entre ces deux dernières valeurs et *durée* est le rapport du nombre de cycles entre la version considérée et la version la plus rapide (- pour la version la plus rapide).

Tableau 1 Exécution avec 128 processus

	Cycles	Parallélisme	Attente	Efficacité	Durée
	7821	2.19	0.38	85	1,03
Arb-p	7605	2.25	0.62	78	-
	7605	2.25	0.62	78	-
	45444	4.35	1.53	74	1,03
Arb-m	44051	4.48	3.43	75	-
	44051	4.48	3.43	75	-
	340688	17.96	22.38	45	1,11
Arb-g	307837	19.87	28.32	41	-
	308105	19.86	32.30	38	1,00

Dans le cas d'une exécution avec `128` processus la version prudente est la moins rapide, surtout sur `Arb-g` qui constitue le plus gros exemple. Les versions avec processus complémentaires et moins prudente sont sensiblement équivalentes.

Tableau 2 Exécution avec 64 processus

	Cycles	Parallélisme	Attente	Efficacité	Durée
	7696	2.22	0.39	85	1,03
Arb-p	7471	2.29	0.62	79	-
	7471	2.29	0.62	79	-
	44772	4.41	1.55	74	1,03
Arb-m	43352	4.56	3.44	57	-
	43352	4.56	3.44	57	-
	374153	16.36	20.28	45	1,06
Arb-g	353355	17.32	21.74	44	-
	353613	17.32	26.34	40	1,00

Dans le cas d'une exécution avec `64` processus la version prudente reste la moins rapide surtout sur `Arb-g` et les versions avec processus complémentaires et moins prudente restent équivalentes.

Tableau 3 Exécution avec 32 processus

	Cycles	Parallélisme	Attente	Efficacité	Durée
	7568	2.26	0.38	86	1,03
Arb-p	7334	2.33	0.62	79	-
	7334	2.33	0.62	79	-
	44076	4.48	1.57	74	1,03
Arb-m	42644	4.63	3.08	60	1,00
	42638	4.63	3.45	57	-
	487249	12.56	13.54	48	-
Arb-g	491759	12.45	14.34	46	1,01
	787772	7.78	21.93	26	1,62

Dans le cas d'une exécution avec 32 processus la version prudente est encore la moins rapide sur les deux petits exemples, par contre sur `Arb-g` elle est légèrement plus rapide (de 1% environ), la version moins prudente y étant de loin moins rapide que les deux autres.

Il découle de ces simulations que la version la plus rapide est celle à processus complémentaires qui est apte à adapter son fonctionnement aux variations de ressources disponibles.

La version moins prudente dont les résultats sont dans l'ensemble corrects s'écroule complètement lors du calcul de arb-g avec 32 processus car elle génère alors trop de processus peu actifs qui bloquent la progression du calcul.

La version prudente a de bonnes performances mais elle n'est pas toujours apte à utiliser un quota important de ressources, d'où une durée toujours supérieure à `1` avec `64` ou `128` processus qui atteint `1,11` sur `arb-g` avec `128` processus.

3.3.3 Conclusion

Nos expérimentations montrent l'avantage (au niveau de la durée) du programme utilisant des processus complémentaires pour les opportunités de parallélisme de moindre rendement. Nous devons particulièrement insister sur le fait que ces performances supérieures ne sont pas liées à une configuration de test particulière (et donc non significative), mais se confirment avec plusieurs jeux de données et avec divers quotas de ressources.

Un autre avantage que les chiffres ne peuvent faire ressortir est la plus grande facilité de parallélisation, celle-ci devenant indépendante des ressources et des données. La phase de parallélisation se réduit alors à détecter les opportunités de parallélisme de meilleur rendement et à les lancer par

un processus, et à détecter les opportunités de parallélisme dont le rendement dépend des ressources disponibles et à les lancer par un processus complémentaire.

Si cette parallélisation n'est pas infiniment plus simple qu'une classique parallélisation à base de `processus`, elle a le gros avantage d'être indépendante des ressources et donc de rendre le programme ainsi parallélisé réutilisable plus facilement dans d'autres contextes sans en remettre en cause la phase de parallélisation.

Nous pourrions être tentés de généraliser cette approche en modifiant la directive processus qui serait alors paramétrée par le seuil d'activité au delà duquel elle serait ignorée. Toutefois, s'il est probable que la performance maximum envisageable par une telle approche serait supérieure à ce que nous pouvons atteindre, son effet sur la phase de parallélisation (et donc sur les coûts humains à y investir) serait désastreuse pour des raisons de complication excessive. Nous préférerons donc renoncer à une telle approche.

En conclusion, le mécanisme de processus complémentaires que nous avons étudié ne peut prétendre régler tous les problèmes liés à la parallélisation des algorithmes mais son apport peut être réel dans de nombreux cas d'algorithmes délicats à paralléliser.

3.4 Une source d'inactivité : la recomposition de résultats

Nous venons d'analyser l'impact des processus complémentaires dans notre modèle et nous avons constaté qu'ils sont aptes à améliorer le taux de parallélisme global en supprimant certains processus des moins actifs.

Nous allons maintenant étudier une autre approche qui ne diminue pas le nombre de processus en attente en supprimant certains processus peu actifs à certains moments de l'exécution, mais qui diminue le nombre moyen de processus en attente en diminuant le taux d'attente des processus générés.

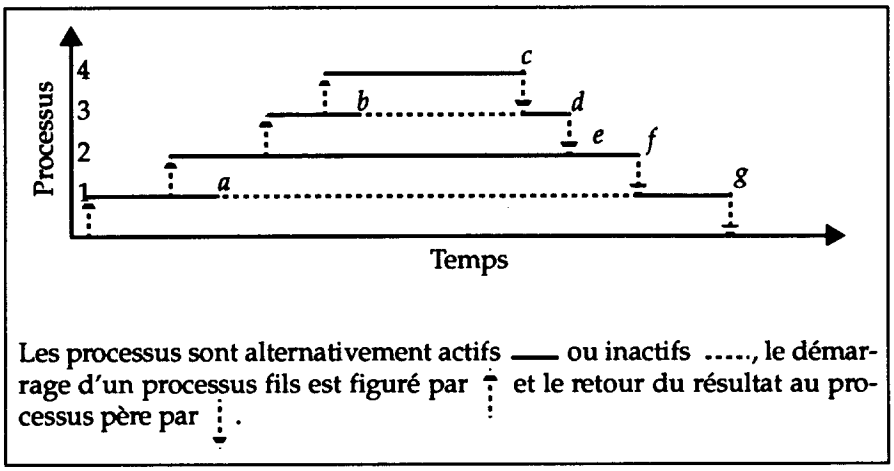
Un grand nombre d'algorithmes parallèles calculent des parties du résultat global à l'aide de plusieurs processus et recomposent le résultat global à partir de ces éléments de solution. Ce schéma correspond par exemple à la phase de recomposition du paradigme *diviser pour régner.*

La phase de composition introduit alors une synchronisation entre processus, le temps que l'opérateur de composition ait ses arguments disponibles.

Ce phénomène est illustré Figure 3 sur laquelle nous pouvons observer sept instants particuliers :

- *a* : le premier processus a calculé son résultat r_1 et n'attend plus que le résultat de son fils pour composer le résultat final.
- *b* : le troisième processus a calculé son résultat r_3 et n'attend plus que le résultat de son fils pour composer son résultat.

Figure 3 Composition synchronisée



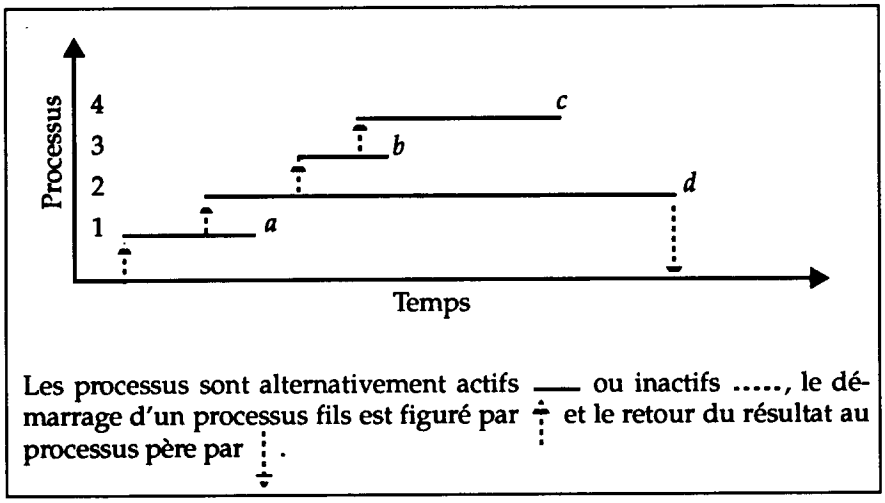
Les processus sont alternativement actifs ——— ou inactifs ....., le démarrage d'un processus fils est figuré par ⋮ et le retour du résultat au processus père par ⋮.

- *c* : le quatrième processus a calculé son résultat r_4 et le retourne à son père, celui-ci (le troisième) qui était en attente depuis l'instant *b* peut alors commencer à composer r_3 et r_4 pour former r_3' qu'il retournera à son propre père.
- *d* : le troisième processus a calculé son résultat r_3' et le retourne à son père puis termine.
- *e* : le deuxième processus a calculé son résultat r_2 et le résultat de son fils étant disponible, il peut commencer à calculer la composition de r_2 et r_3' pour former r_2' qu'il retournera à son propre père.
- *f* : le second processus à calculé r_2' et le retourne à son père, celui-ci qui était en attente depuis l'instant *a* peut alors commencer à composer r_1 et r_2' pour former r_1'
- *g* : le calcul est terminé, le résultat est r_1'.

Dans le cas général, cette synchronisation père / fils est incontournable. Cependant dans certains cas une synchronisation moins rigide peut être suffisante pour rendre le résultat déterministe. A l'extrême nous trouvons des cas où aucun ordonnancement n'est nécessaire, par exemple si les processus calculent un ensemble de solutions, la synchronisation ne sert qu'à imposer inutilement un ordre aux éléments de cet ensemble. En fait toute opération commutative (comme ajouter un élément à un ensemble) peut être utilisée sans synchronisation.

L'intérêt d'un tel désordonnancement réside dans la possibilité ainsi offerte aux processus qui ont trouvé une partie du résultat de le composer au résultat tel qu'il existe à ce moment là et de terminer sans plus attendre, ce qui libère la ressource processus correspondante.

Figure 4 Composition commutative non synchronisée



Ce phénomène est illustré Figure 4 sur laquelle nous pouvons observer quatre instants particuliers :

- *a* : le premier processus a calculé son résultat r_1 et réalisé une partie de la composition. Il peut terminer.
- *b* : le troisième processus a calculé son résultat r_3 et réalisé une partie de la composition. Il peut terminer.
- *c* : le quatrième processus a calculé son résultat r_4 et réalisé une partie de la composition. Il peut terminer.
- *d* : le second processus a calculé son résultat r_3' et réalisé une partie de la composition. Il peut terminer et le calcul est alors totalement terminé.

Nous allons donc chercher à résoudre le problème suivant : comment éviter certaines phases de synchronisation entre processus, qui diminuent le taux de parallélisme des algorithmes, et dont nous pouvons déterminer qu'elles sont inutiles au déterminisme du résultat.

3.4.1 Un cadre pour résoudre ce problème

Le fonctionnement que nous venons de décrire correspond directement au fonctionnement d'un programme impératif : le résultat, partagé par tous les processus, est modifiable sans autre synchronisation que pour en assurer la cohérence. Les processus n'ont d'autre effet que de modifier le contenu des variables.

La transcription d'algorithmes impératifs dans un style fonctionnel est connue depuis *[Mc CARTHY 60]* qui présente une technique basée sur l'utilisation de fonctions intermédiaires. Cette technique est également présentée entre autres dans *[HENDERSON 80]* et *[MaRS 88a]*.

3.4.1.a De l'impératif en MaRS_Lisp

Le langage MaRS_Lisp contient des constructions permettant une programmation de style impératif avec une syntaxe proche de Common Lisp et une sémantique stricte (évaluation par valeur) présentées dans *[MaRS 88a]*. La technique de transformation utilisée est basée sur des fonctions utilisant des paramètres supplémentaires pour les variables affectables et elle permet l'écriture de tout algorithme impératif à variables statiques. Par exemple, nous pouvons programmer la fonction de fibonacci par l'algorithme linéaire (cf. Figure 5).

Figure 5 La fonction de fibonacci en style MaRS_Lisp impératif

```
(defun fib (n)
   (program ((c n) (a 0) (d 1)) ;; variables affectables
      d ;; résultat du bloc program retourné en fin de calcul
      () ;; procédures locales au bloc program
      (while (not (= c 1)) ;; calcul lancée par le bloc program
             (psetf ;; affectation parallèle
                c (- c 1)
                a d
                d (+ a d)))))
```

La limitation principale de cette approche pour notre utilisation est son caractère fortement séquentiel, la seule source de parallélisme possible étant l'affectation parallèle par `psetf`. En particulier, il est impossible de lancer en parallèle deux tâches qui manipulent les mêmes variables. Nous devrons donc abandonner cette voie pour l'implémentation d'algorithmes impératifs parallèles, tels que ceux envisagés ici.

De plus la syntaxe admise à l'intérieur d'une construction `program` MaRS_Lisp connaît quelques contraintes syntaxiques par rapport à son équivalent fonctionnel[1], ce qui est une source constante d'insatisfaction du programmeur.

3.4.1.b Autres approches

De nombreux langages parallèles à sémantique stricte de style fonctionnel incluent la possibilité de faire des effets de bord et fournissent des constructions de plus ou moins haut niveau pour en contrôler le fonctionnement.

Une première approche consiste à ne fournir que des primitives de bas niveau telles que les verrous, affectations ou mutations de structures de données et à laisser le programmeur écrire la synchronisation qui lui convient. Dans cette catégorie, nous trouvons par exemple Multilisp *[HALSTEAD 85]*

1. Ces contraintes proviennent classiquement de l'introduction d'une partition du langage en deux catégories syntaxiques : les expressions et les commandes.

et Multischeme *[MILLER 87]*. Cette approche ne peut conduire qu'à renoncer à utiliser les effets de bords à des fins algorithmiques, ce qui explique l'absence de publication sur la bonne programmation parallèle avec effets de bord dans un tel cadre. Ceci est confirmé dans *[HALSTEAD 87]* où il est conseillé de confiner les effets de bords de telle sorte que n'importe quel module du programme puisse être exécuté en parallèle, ce qui est un principe plein de sagesse mais peu à même de guider le programmeur.

Une autre approche consiste à fournir des outils plus évolués mais à laisser le programmeur écrire la synchronisation qui lui semble nécessaire. Nous pouvons citer par exemple QLISP *[GABRIEL 84]*, *[GOLDMAN 89]* qui outre des mécanismes de bas niveau tels les verrous ou les signaux fournit la possibilité d'écrire des sections de code en exclusion mutuelle par `qlambda`, ce qui doit éviter tout recours aux mécanismes de base. Cependant *[GOLDMAN 89]* reconnaît que ce mécanisme est encore de trop bas niveau pour être aisément utilisable.

Une voie diamétralement opposée consiste à ce que le programmeur n'ait pas à spécifier la moindre synchronisation pour obtenir un fonctionnement cohérent de ses effets de bord, ce qui constitue un idéal du point de vue de celui-ci. Dans cette catégorie, nous trouvons le compilateur Parcel *[HARRISON 89a]* qui extrait par analyse des dépendances un parallélisme exploitable à partir de programmes Scheme séquentiels avec effets de bord. Le défaut majeur de Parcel est d'être figé et coûteux en temps d'analyse *[HARRISON 89b]*. De plus on peut s'interroger sur la qualité réelle du code généré et sur la taille des programmes compilables par cette approche.

Il semble donc que la seule voie actuellement exploitable consiste à fournir à l'utilisateur des primitives lui permettant d'écrire un programme au fonctionnement cohérent. Cependant, une telle approche pour être réellement utilisable nécessite des construction puissantes et peu nombreuses ainsi que l'identification de schémas algorithmiques de mise en œuvre des effets de bord.

3.4.1.c Des constructions pour résoudre le problème

Nous allons présenter un ensemble minimal de constructions en vue d'étendre notre langage à certains algorithmes impératifs parallèles tout en conservant autant que possible la programmation MaRS_Lisp classique pour tout ce qui ne relèvera pas de ces constructions[1]. Tout comme pour la construction `program` déjà existante, nous choisirons d'encapsuler les parties impératives dans un bloc de manière à en clarifier la portée. Dans le cas d'un programme totalement impératif, il suffira d'utiliser un bloc englobant l'ensemble du programme.

1. Conserver le style MaRS_Lisp applicatif consiste essentiellement à refuser la partition du langage en expressions et commandes.

Pour prendre en compte l'aspect impératif, nous devrons disposer de quoi introduire une variable, la consulter et la modifier, la prise en compte du parallélisme nécessitant également une construction particulière.

- p-program

  La construction `p-program` encapsule un bloc impératif :

  `(p-program ((`v_1 e_1`) ... (`v_n e_n`)) r ((`p_1 d_1`) ... (`p_m d_m`)) a)` où les v_i sont les variables affectables locales et les e_i leur initialisation (qui restent non évaluées), les p_i les noms des procédures locales (qui sont utilisables dans `r`, `a` et les d_i) et les d_i les lambda expressions les définissant, `a` l'action qui sera exécutée et `r` le résultat qui sera calculé avec les variables telles que `a` les aura laissées. Il n'y a aucune contrainte syntaxique spécifique aux expressions e_i, `a` et `r` ni pour les lambda-expressions d_i (les v_i étant des symboles), par contre des contraintes spécifiques existent au niveau du typage (cf. 3.4.3).

- newvar

  Permet d'introduire une nouvelle variable dynamiquement :

  `(newvar v e r)` crée dynamiquement une variable affectable nommée `v` initialisée avec `e` (qui reste non évalué) et évalue `r` (qui logiquement référence `v`). Chaque instance de `r` en cours de calcul utilise bien sûr une variable `v` différente. Les variables crées par `newvar` sont du même type que celles introduites par `p-program`.

- use

  Réalise l'accès à la valeur d'une variable modifiable :

  `(use v k)` applique `k` à la valeur de `v`. Cela garantit que la valeur de `v` utilisée par `k` (via son paramètre) est celle de `v` au moment où `(use v k)` a été touché.

- affect

  Permet de modifier le contenu d'une variable :

  `(affect v f k)` affecte à `v` la valeur obtenue en appliquant `f` sur l'ancienne valeur puis continue en évaluant `k`. Cela garantit que `k` va utiliser la variable `v` après sa modification.

  Modifier la variable en lui appliquant une transformation `f` permet de se dispenser de l'utilisation d'une section critique pour de nombreux traitements tels que incrémenter la variable.

- prog2

  Permet de composer deux actions en séquence :

  `(prog2 e1 e2)` réduit `e1` puis `e2` qui est le résultat du `prog2`.

- fork

  Permet de lancer un nouveau processus dont le résultat est indifférent, mais qui peut modifier les variables concurremment aux autres processus :

  `(fork p k)` lance un processus pour évaluer `p` puis continue en évaluant `k`.

3.4.1.d Premier typage

Nous allons étudier un typage naïf de nos nouveaux opérateurs. Pour cela nous donnons des règles d'inférence de type relatives à chacun d'eux. Les limitations dues à cette approche seront discutées (cf. 3.4.2 page 90) et un nouveau typage sera alors proposé pour y remédier.

Nous avons choisi d'affecter un type référence aux variables affectables comme c'est le cas en ML *[COUSINEAU 89]* ce qui évite toute confusion entre une variable et son contenu.

Nous donnons ci-dessous uniquement les règles de typage qu'il est nécessaire d'ajouter aux règles classiques du typage fonctionnel pour typer des expressions contenant nos opérateurs. Le style retenu pour exprimer nos règles est la sémantique naturelle (ou sémantique *à la Plotkin*) dont une bonne présentation ainsi que les règles d'inférence de type pour un langage fonctionnel peuvent être trouvées dans *[KAHN 87]*.

r.1 `p-program` :
$$\frac{\varepsilon_1 \quad \varepsilon_2 \quad \varepsilon_3 \quad \varepsilon_4}{\Gamma\vdash (\text{p-program } ((v_1\ e_1)\ldots(v_n\ e_n))\ r\ ((p_1\ d_1)\ldots(p_m\ d_m))\ a) : t}$$

Où :

$$\varepsilon_1 \equiv \Gamma\vdash (e_1 : t_1) \wedge \ldots \wedge (e_n : t_n)$$

$$\varepsilon_2 \equiv (\Gamma, v_1 : (\Re\ t_1), \ldots, v_n : (\Re\ t_n)) \vdash (d_1 : (t'_1 \to t''_1)) \wedge \ldots \wedge (d_m : (t'_m \to t''_m))$$

$$\varepsilon_3 \equiv (\Gamma, v_1 : (\Re\ t_1), \ldots, v_n : (\Re\ t_n), p_1 : (t'_1 \to t''_1), \ldots, p_m : (t'_m \to t''_m)) \vdash r : t$$

$$\varepsilon_4 \equiv (\Gamma, v_1 : (\Re\ t_1), \ldots, v_n : (\Re\ t_n), p_1 : (t'_1 \to t''_1), \ldots, p_m : (t'_m \to t''_m)) \vdash a : t_a$$

Dans ces équations, le type $(\Re\ t)$ est celui d'une variable affectable dont le contenu est de type t (référence à un élément du type t). Notons que ε_4 ne sert qu'à assurer l'existence d'un type pour `a`, mais celui-ci est indifférent.

r.2 `newvar:`
$$\frac{(\Gamma\vdash e : t_1) \qquad (\Gamma, v : (\Re\ t_1) \vdash r : t_2)}{\Gamma\vdash (\text{newvar } v\ e\ r) : t_2}$$

r.3 `use:`
$$\frac{(\Gamma\vdash v : (\Re\ t_1)) \qquad (\Gamma\vdash k : (t_1 \to t_2))}{\Gamma\vdash (\text{use } v\ k) : t_2}$$

r.4 `affect:`
$$\frac{(\Gamma\vdash v : (\Re\ t_1)) \qquad (\Gamma\vdash f : (t_1 \to t_1)) \qquad (\Gamma\vdash k : t_2)}{\Gamma\vdash (\text{affect } v\ f\ k) : t_2}$$

Notons que cette équation impose à la nouvelle valeur de la variable, soit `f` appliquée à l'ancienne valeur, de garder le même type. Ce choix est clairement le seul possible et ne doit donc pas être perçu comme une limitation.

r.5 `prog2:`
$$\frac{(\Gamma\vdash p : t_1) \qquad (\Gamma\vdash k : t_2)}{\Gamma\vdash (\text{prog2 } p\ k) : t_2}$$

$$\textbf{r.6}\quad \texttt{fork:}\ \frac{(\Gamma\vdash p:t_1)\qquad (\Gamma\vdash k:t_2)}{\Gamma\vdash (\text{fork } p\ k):t_2}$$

3.4.1.e Réalisation

Le mécanisme que nous venons de décrire pour modifier les variables a diverses implications pour ce qui est de leur réalisation.

En premier lieu, une synchronisation correcte lors de diverses consultations et modifications parallèles ne peut se faire que grâce à une sérialisation de ces accès. Le cycle de vie des nœuds mémoires (cf. 1.6 page 37) nous offre une solution élégante puisque lorsqu'un nœud est *en réduction* toutes les demandes d'accès ultérieures sont mises en file d'attente jusqu'à ce qu'il soit réécrit. Il suffit donc que l'accès à une variable la fasse passer *en réduction* pendant toute l'opération pour assurer une synchronisation cohérente.

Ensuite, chaque variable pouvant être modifiée un nombre quelconque de fois, une variable qui n'est pas en cours d'accès doit toujours être à l'état *non réduit*, seul état autorisant un passage à l'état *en réduction* et une réécriture ultérieure.

Une variable affectable doit donc être représentée par un graphe au cycle de vie atypique, alternativement *en réduction* lors des accès par un processus et *non réduit* entre deux de ces accès[1].

Pour des raisons d'orthogonalité, nous ne coderons pas chaque opérateur directement mais nous utiliserons deux combinateurs de manipulation de graphe qui factorisent une large part des besoins. Ces manipulations de graphe vont donc être assurées par deux primitives de synchronisation : `lock` et `unlock` qui seront utilisées dans la compilation de nos opérateurs mais n'ont pas à être connues de l'utilisateur.

- lock

  Dans le cas où nous choisirions d'implémenter un constructeur de type variable affectable *make*$\Re$: $(t \to (\Re\ t))$, cela nous obligerait à assurer qu'une expression utilisant de telles variables à l'aide de `use` et `affect` conduise à un ordonnancement correct des accès. Pour cela, nous pourrions être amenés à imposer une stratégie d'évaluation par valeur, ce qui n'est pas souhaitable. Pourtant `(use v k)` ne peut avoir un sens que si le contenu de `v` utilisé par `k` est celui qui est valide au moment où `(use v k)` est touché.

  Cette constatation nous amène non pas à définir des opérateurs *make*$\Re$: $(t \to (\Re\ t))$ et *deref*: $((\Re\ t) \to t)$ pour déréférencer la variable et

1. Contrairement à tout autre graphe, les réécritures de la variable ne se font pas à valeur égale (la valeur réécrite n'est pas une autre représentation de la valeur initiale).

ainsi récupérer son contenu, mais un opérateur *lock*: $(t_1 \rightarrow (t_1 \rightarrow t_2) \rightarrow t_2)$ combinant les deux.

Une expression de la forme `(lock e)` est alors une fonction à un argument qui doit être l'expression utilisant la valeur `e`. L'évaluation de ce graphe le met bien entendu *en réduction*, ce qui réalise la partie synchronisation des accès que nous désirons. Si `v = (lock e)`, le graphe `(v f)` va se réduire comme suit :

```
v f ;; v = lock e

= f e ;; v en réduction
```

- unlock

  Pour des raisons similaires à celles évoquées pour `lock`, nous n'implémenterons pas une primitive *setq*: $((\Re\ t) \rightarrow t \rightarrow t)$ de modification d'une variable, mais une primitive *unlock*: $(t_1 \rightarrow (t_2 \rightarrow t_3) \rightarrow t_3 \rightarrow t_2 \rightarrow t_1)$ combinant et sérialisant la modification de la variable et son utilisation ultérieure.

  La primitive `unlock` sert à restituer une variable *en réduction* dans un état utilisable pour un accès ultérieur et va donc être utilisée à l'intérieur d'un `lock`.

  Une expression `(unlock k f a e)` restitue le nœud du graphe situé à l'adresse `a` (qui doit initialement être *en réduction*) dans l'état *non réduit* avec `f` en partie fonction et `e` en partie argument; après cette modification l'évaluation se poursuit avec `k` qui est la valeur de l'expression toute entière. Notons que les contraintes de typage appliquées à une variable affectable veulent que l'argument `f` soit égal à `lock`. Si `v` est *en réduction*, le graphe `(unlock k f v e)` se réduit comme suit :

A l'aide de ces deux primitives nous pourrons réaliser les diverses manipulations de variables :

- use

  `(use v k)` où `v` est une variable, et donc de la forme `(lock e)`, et `k` est la fonction à un argument qui va utiliser la valeur de `v` va s'écrire :

  ```
  use v k = (v (lambda (w) (unlock (k w) lock v w)))
  ```

  Où `w` est une variable non encore utilisée.

  Observons la réduction de l'expression `(use v k)` où `v = (lock e)` :

  ```
  use v k ;; v = lock e

  = v (lambda (w) (unlock (k w) lock v w)) ;; v = lock e

  = (lambda (w) (unlock (k w) lock v w)) e ;; v en réduction

  = unlock (k e) lock v e ;; v en réduction

  = k e ;; v = lock e
  ```

  Le résultat obtenu est bien `(k e)` et la variable `v` reste inchangée après un cycle *non réduit / en réduction / non réduit* qui assure la synchronisation nécessaire.

- affect

  `(affect v f k)` où `v` est une variable, et donc de la forme `(lock e)`, `f` est la transformation qui va s'appliquer à la valeur de la variable et `k` est l'expression à évaluer après cette modification va s'écrire :

```
affect v f k

= (v (lambda (w) (unlock k lock v (f w))))
```

  Où `w` est une variable non encore utilisée.

  La réduction de `(affect v f k)` où `v = (lock e)` se fait comme suit :

```
affect v f k ;; v = lock e

= v (lambda (w) (unlock k lock v (f w)))
;; v = lock e

= (lambda (w) (unlock k lock v (f w))) e
;; v = en réduction

= (unlock k lock v (f e)))
;; v = en réduction

= k ;; v = (lock (f e))
```

  Le résultat obtenu est bien `k` et la variable `v` voit sa valeur passer de `e` à `(f e)` après un cycle *non réduit / en réduction / non réduit* qui assure la synchronisation nécessaire.

  Remarquons qu'une écriture telle que `(setq v e) ; k` pourrait être autorisée et compilée comme `(affect v (lambda (v) e) k)`.

  Remarquons aussi qu'une compilation optimisée peut être facilement introduite si `f` n'utilise pas son paramètre, nous pouvons traduire :

```
affect v (lambda (v) e) k ;; e n'utilise pas v

= (v (lambda (w) (unlock k lock v e)))
```

La création d'une variable affectable se fera de la façon suivante :

- newvar

  `(newvar v e r)` où `v` est le nom de la nouvelle variable, `e` son contenu et `r` l'expression qui l'utilise va s'écrire à l'aide du combinateur `new-app` tel que `(new-app f e k)` crée un nouveau nœud *non réduit* de contenu `(f e)` et fait l'application `(k (f e))` entre `k` et ce nœud. Par suite :

```
newvar v e r = (new-app lock e (lambda (v) r))
```

  Remarquons que cette manipulation ne pourrait se faire par `(newvar v e r) = (r (lock e))` car dans une telle expression chaque processus exécutant un même `(newvar v e r)` partagerait la même variable.

La macroexpansion de `p-program` à l'aide des opérateurs précédents se fait comme suit :

- p-program

  (p-program ((v$_1$ e$_1$) … (v$_n$ e$_n$)) r ((p$_1$ d$_1$) … (p$_m$ d$_m$)) a) où les v$_i$ sont les variables affectables locales au bloc et les e$_i$ leur initialisation, les p$_i$ les noms des procédures locales et les d$_i$ leur définition, a l'action exécutée et r le résultat tel qu'il sera calculé à partir des variables positionnées par a sera macrogénéré comme suit :

```
p-program ((v1 e1) … (vn en)) r ((p1 d1) … (pm dm)) a =

(newvar cpt 0
  (newvar syn t
    (newvar v1 e1 …
        (newvar vn en
          (letrec ((p1 d1) … (pm dm))
            (syn (lambda (w1) ;; la synchro devient bloquante
              (prog2 (prog2 a
                            (cpt (lambda (v)
                                   (unlock
                                     (when (= v 0)
                                       (unlock t lock syn t))
                                     lock cpt (- v 1))))))
                     (syn (lambda (w2) ;; point de synchro
                          (unlock r lock syn t))))))))) …)))
```

  Où cpt, syn, w1 et w2 sont des variables non encore utilisées.

  La variable syn se caractérise par le fait qu'elle se trouve toujours *en réduction* lors du calcul de a. Ainsi le processus initial qui commence son calcul et qui peut terminer sa part du travail (laissant ses fils terminer) sera bloqué au point de synchronisation jusqu'a la fin du dernier processus actif. La variable cpt sert à compter les processus en activité, lorsqu'elle est à zéro le processus qui termine sait qu'il est le dernier et peut ainsi rendre la synchronisation non bloquante.

- prog2

  Est directement disponible dans le jeu de combinateurs.

- fork

  Cet opérateur se réalise directement à partir du combinateur !fork qui prend comme premier argument un compteur de processus présents (créé par new-var dans le p-program englobant), comme second argument une synchronisation initialement *en réduction* (également introduite par le p-program englobant) et comme troisième et quatrième arguments les deux expressions à évaluer.

  ```
  fork p k = (!fork cpt syn p k)
  ```

  Où cpt et syn sont les mêmes variables que celles utilisées dans le bloc p-program englobant.

  La primitive !fork garantit qu'un processus sera lancé pour évaluer son troisième argument (l'expression à lancer en parallèle), soit immédiate-

ment si une ressource est disponible, soit à son tour si une attente est nécessaire.

Chaque appel à `!fork` commence par incrémenter le compteur de processus avant de créer le processus fils, et ne le décrémente que lorsqu'il a terminé. Si `cpt` contient zéro, c'est que le processus est le dernier et `!fork` débloque alors la synchronisation.

Nous n' avons décrit qu'un nombre limité d'opérateurs, sachant que les primitives `lock` et `unlock` permettent d'écrire simplement des extensions. Ainsi nous pouvons définir un opérateur inspiré du `qlambda` de QLISP *[GOLDMAN 89]* qui est un `lambda` dont le corps est protégé par exclusion mutuelle :

```
(qlambda (x1..xn) corps) =

(lambda (x1..xn)
  (let ((X (lock t))) ; où X est une variable non encore utilisée
    (X (lambda (v) (el-unlock corps lock X v)))))
```

Où `el-unlock` est une variante de unlock qui évalue son premier argument par valeur, donc avant de récrire le nœud passé en argument.

De la même façon il serait possible d'écrire des variantes de `affect` pour lesquelles `k` serait appliqué à l'ancien contenu de la variable ou à son nouveau contenu.

3.4.2 Sémantique de p-program

A l'aide d'un bloc p-program tel que nous venons de le définir, il est possible d'écrire des expressions à la sémantique on ne peut moins claire :

```
(affect v (lambda (v!) 0)
        (+ (affect v (lambda (v!) (+ v! 2)) 2)
           (use v (lambda (v!) v!)))))
```

Cette expression peut signifier quelque chose comme "Affecter `v:=0` puis faire l'addition entre `{v:=v+2 ; 2}`, qui affecte `v:=v+2` et vaut `2`, et le contenu de `v`". Le problème provient de ce que selon l'ordre d'évaluation des arguments de `+`, l'affectation de `v+2` à `v` sera ou non effective lors de l'évaluation de la seconde branche qui peut donc valoir `0` (évaluation de droite à gauche des argument de l'addition) ou `2` (évaluation de gauche à droite). Une écriture non ambiguë peut être :

```
(affect v (lambda (v!) 0)
        (affect v (lambda (v!) (+ v! 2))
                (use v (lambda (v!!) (+ 2 v!!))))))
```

Dans ce cas, la valeur de l'expression est claire ("Affecter `v:=0` puis affecter `v:=v+2` puis faire l'addition entre `2` et le contenu de `v`") et indépendante de détails d'implémentation dont le programmeur n'a que faire, ce qui est tout à fait satisfaisant.

Comment pouvons nous rendre illégales les expressions telles que la première et ne tolérer que les expression raisonnables telles que la seconde?

3.4.3 Un nouveau typage

Un début de solution nous est offert dans *[SWARUP 91]* qui définit un ensemble d'opérateurs équivalent au nôtres pour manipuler des références dans un cadre fonctionnel[1] paresseux séquentiel et un système de types du premier ordre (donc non polymorphe) décomposé en trois couches :

- Types fonctionnels (ou applicatifs) :
  $\tau ::= \beta \mid (\tau_1 \times \tau_2) \mid (\tau_1 \rightarrow \tau_2)$
  Correspondent aux types fonctionnels classiques, β étant un type primitif quelconque.
- Types mutables :
  $\theta ::= \tau \mid (\mathrm{Ref}\,\theta) \mid (\theta_1 \times \theta_2) \mid (\theta_1 \rightarrow \theta_2)$
  Etendent le niveau applicatif aux références.
- Types observeurs :
  $\omega ::= \theta \mid (\mathrm{Obs}\,\tau) \mid (\omega_1 \times \omega_2) \mid (\omega_1 \rightarrow \omega_2)$
  Types des expressions qui utilisent les références.

Les trois niveaux de types ainsi définis ne sont pas indépendants mais ils sont au contraire imbriqués selon le schéma $\omega \supseteq \theta \supseteq \tau$: tout type fonctionnel peut être considéré comme mutable ou observeur, tout type mutable peut être considéré comme observeur. A l'inverse l'appartenance d'un élément du type observeur au type fonctionnel n'est pas systématique.

Les constructeurs $\times$ et $\rightarrow$ sont les mêmes pour les trois niveaux de types. Le niveau des types mutables n'a qu'un seul constructeur : Ref qui introduit le type des références. Le niveau observeur a également un constructeur : Obs introduit le type des expressions qui utilisent l'état (pour le lire ou le modifier).

Nous allons adapter ce système de types à notre langage, mais de manière à limiter l'utilisation des objets mutables et des observeurs à l'intérieur de régions bien définies et en y introduisant (de manière limitée) le polymorphisme à la ML. Nous verrons également l'extension de ces principes afin d'y introduire le parallélisme de façon homogène.

3.4.3.a Références et polymorphisme

Inclure sans précautions le polymorphisme et les références à un système de types est une source d'incohérence, comme le montre l'exemple présenté Figure 6 tiré de *[PAULSON 91]* et conservé dans sa syntaxe à la Standard ML (*à la* parce que mal typé).

1. Le langage ILC utilisant ce système de typage est fonctionnel tout en autorisant les affectations et mutations de données : ce typage garantit en effet que toutes les modifications de données se font sans effet de bord (la transparence du référentiel est maintenue).

Figure 6 Une session SML imaginaire

```
fun I x = x;
> val I = fn : 'a -> 'a
```

I est la fonction identité de type polymorphe ('a représente une variable de type) et peut être appliquée à des arguments de tout type.

```
val fp = ref I;
> val fp = ref fn : ('a -> 'a) ref
```

fp est une référence à I.

```
(!fp true, !fp 5);
> (true, 5) : bool * int
```

Le contenu de fp (désigné par !fp) qui est I s'applique aussi bien à un booléen qu'à un nombre.

```
fp := not;
```

not est de type bool -> bool et peut donc être affecté à fp qui est de type polymorphe ('a -> 'a) ref.

```
!fp 5;
> ERROR!
```

Le contenu de fp ne s'applique plus à un nombre!!!

Le problème dans cet exemple provient de ce que le type de fp varie au cours de la session, ce qui devra donc être interdit pour maintenir la cohérence (la solution adoptée dans SML est de restreindre le polymorphisme dans les références au polymorphisme ne pouvant entraîner ce style de changements de type).

Nous adopterons une solution plus simple et radicale qui est de restreindre les références aux types non polymorphes.

3.4.3.b Les types

Nous allons donc introduire le constructeur de types $\forall$ dans le système de types de *[SWARUP 91]* en veillant à ne pas autoriser les références polymorphes. Pour cela nous décomposerons le niveau fonctionnel en deux univers : les types non polymorphes qui correspondent aux types du lambda-calcul simplement typé et les types polymorphes. Les variables de type ν introduites par $\forall$ prennent valeur dans les types non polymorphes.

- Types fonctionnels (ou applicatifs) :

$$\tau ::= \Upsilon \mid (\forall \nu : \Upsilon \cdot \tau)$$
$$\Upsilon ::= \beta \mid \nu \mid (\Upsilon_1 \times \Upsilon_2) \mid (\Upsilon_1 \rightarrow \Upsilon_2)$$

  Correspondent aux types fonctionnels classiques, β étant un type primitif quelconque et ν une variable de type prenant valeur dans Υ.

- Types mutables :

  $\theta ::= \Upsilon \mid (\mathrm{Ref}\,\theta) \mid (\theta_1 \times \theta_2) \mid (\theta_1 \rightarrow \theta_2)$

  Etendent le niveau applicatif non polymorphe aux références.

- Types observeurs :

  $\omega ::= \theta \mid (\mathrm{Obs}\,\tau) \mid (\omega_1 \times \omega_2) \mid (\omega_1 \rightarrow \omega_2)$

  Types des expressions qui utilisent les référence.

3.4.3.c L'environnement

Nous manipulerons des environnements constitués d'assertions $x:\pi$, "la variable x est de type π". L'environnement $\Gamma + \Gamma'$ représente une concaténation et $\Gamma, x:\pi$ un enrichissement. Les environnements ne sont pas des ensembles, si plusieurs assertions $x:\pi$ sont faites sur une même variable seule la dernière doit être prise en compte. Ainsi dans $\Gamma + \Gamma'$ le type d'une variable ne sera cherché dans Γ que s'il n'est pas précisé par Γ', de la même façon le type de x dans $\Gamma, x:\pi$ est π indépendamment de Γ.

Le type des opérateurs primitifs est inchangé, un schéma de type tel que $\forall x \cdot T$ devant simplement être compris "quelque soit x de type observeur" et non plus "quelque soit x de type fonctionnel".

Si nous prenons l'exemple du IF, son type reste $\mathrm{IF}: \forall \pi \cdot \mathrm{Bool} \rightarrow \pi \rightarrow \pi \rightarrow \pi$, mais la variable π peut être instanciée avec tout type observeur (ce qui comprend les types mutables et fonctionnels).

Grâce à notre nouveau système de type, il est possible d'admettre des expressions dont le type n'est plus fonctionnel. Toutefois ces expressions ne peuvent être évaluées dans notre modèle, leur seule possibilité d'utilisation est en tant que sous-expression d'une expression de type fonctionnel.

3.4.3.d Règles de typage

Le style retenu pour nos règles de typage reste la sémantique naturelle *[PLOTKIN 81]*. Les règles d'inférence de type sont listées ci-dessous.

R.1 $\Gamma \vdash k:\tau$; si k objet primitif de type fonctionnel τ

Cette règle s'applique pour tous les objets primitifs y compris les opérateurs.

R.2 $\Gamma, x:\pi \vdash x:\pi$

R.3 $\dfrac{\Gamma \vdash x:\pi_1}{\Gamma, y:\pi_2 \vdash x:\pi_1}$; (si $x \neq y$)

R.4 $$\frac{\Gamma, x:\pi_1 \vdash e:\pi_2}{\Gamma \vdash (\lambda x:\pi_1 . e) : (\pi_1 \to \pi_2)}$$

R.5 $$\frac{(\Gamma \vdash f: (\pi_1 \to \pi_2)) \qquad (\Gamma \vdash e:\pi_1)}{\Gamma \vdash (f\ e) : \pi_2}$$

R.6 $$\frac{(\Gamma \vdash e_1:\pi_1) \qquad (\Gamma \vdash e_2:\pi_2)}{\Gamma \vdash (e_1, e_2) : \pi_1 \times \pi_2}$$

R.7 $$\frac{\Gamma \vdash e:\pi_1 \times \pi_2}{\Gamma \vdash e.i:\pi_i}; (\text{où } i \in \{1;2\})$$

$e.i$ représente la i$^{\text{ème}}$ projection.

R.8 $$\frac{\Gamma \vdash t:\tau}{\Gamma \vdash t:\text{Obs}\,\tau}$$

R.9 $$\frac{\Gamma \vdash t:\text{Obs}\,\tau}{\Gamma \vdash t:\tau}$$; si Γ ne contient que des assertions sur des types fonctionnels

R.10 $$\frac{\varepsilon_1 \qquad \varepsilon_2 \qquad \varepsilon_3 \qquad \varepsilon_4}{\Gamma \vdash (\text{p-program}\ ((v_1\ e_1) \ldots (v_n\ e_n))\ r\ ((p_1\ d_1) \ldots (p_m\ d_m))\ e) : \text{Obs}\,\tau_1}$$

où :

$$\varepsilon_1 \equiv \Gamma \vdash (e_1:\theta_1) \wedge \ldots \wedge (e_n:\theta_n)$$

$$\varepsilon_2 \equiv (\Gamma, v_1:\text{Ref}\,\theta_1, \ldots, v_n:\text{Ref}\,\theta_n) \vdash (d_1:(\omega_1 \to \omega'_1)) \wedge \ldots \wedge (d_m:(\omega_m \to \omega'_m))$$

$$\varepsilon_3 \equiv (\Gamma, v_1:\text{Ref}\,\theta_1, \ldots, v_n:\text{Ref}\,\theta_n, p_1:(\omega_1 \to \omega'_1), \ldots, p_m:(\omega_m \to \omega'_m)) \vdash r:\text{Obs}\,\tau_1$$

$$\varepsilon_4 \equiv (\Gamma, v_1:\text{Ref}\,\theta_1, \ldots, v_n:\text{Ref}\,\theta_n, p_1:(\omega_1 \to \omega'_1), \ldots, p_m:(\omega_m \to \omega'_m)) \vdash e:\text{Obs}\,\tau_2$$

R.11 $$\frac{(\Gamma \vdash e:\theta) \qquad ((\Gamma, v:\text{Ref}\,\theta) \vdash r:(\text{Obs}\,\tau))}{\Gamma \vdash (\text{newvar}\ v\ e\ r) : (\text{Obs}\,\tau)}$$

R.12 $$\frac{(\Gamma \vdash v:(\text{Ref}\,\theta)) \qquad (\Gamma \vdash k:(\theta \to (\text{Obs}\,\tau)))}{\Gamma \vdash (\text{use}\ v\ k) : (\text{Obs}\,\tau)}$$

R.13 $$\frac{(\Gamma \vdash v:(\text{Ref}\,\theta)) \qquad (\Gamma \vdash f:(\theta \to \theta)) \qquad (\Gamma \vdash k:(\text{Obs}\,\tau))}{\Gamma \vdash (\text{affect}\ v\ f\ k) : (\text{Obs}\,\tau)}$$

Les cas de `prog2` et de `fork`, qui n'admettent pas d'équivalent dans *[SWA-RUP 91]* seront discutés ultérieurement.

3.4.3.e Commentaires

La règle R.1 permet de typer les constantes et primitives du langage, les règles R.2 et R.3 correspondent à une recherche dans l'environnement, la règle R.4 introduit les types fonction et la règle R.5 permet de typer les applications, les règles R.6 et R.7 introduisent et éliminent les types produit, les règles R.8 et R.9 permettent la coercition entre types fonctionnels et types observeurs, les règles R.10 et suivantes permettent d'inférer un type pour nos nouveaux opérateurs.

La règle R.8 permet d'inférer un deuxième type pour toute expression de type fonctionnel. Toutefois cela ne pose pas de problème car la phase de typage à pour but d'inférer un type fonctionnel pour toute forme (expression de premier niveau autres que définition) du programme et cette règle n'est qu'un moyen pour y arriver : partout où une expression de type observeur est désirée une expression de type fonctionnel peut convenir.

Tout bloc `p-program` de plus haut niveau (non inclus dans un autre) est typé dans un contexte Γ qui ne contient que des assertions sur des types fonctionnels, ce qui autorise l'application de la règle R.9 pour affecter au bloc un type fonctionnel.

Nos règles d'inférence ne permettent plus d'affecter un type à l'expression

```
(affect v (lambda (v!) 0)
        (+ (affect v (lambda (v!) (+ v! 2)) 2)
           (use v (lambda (v!) v!))))
```

car les deux arguments de l'addition ne peuvent se voir affecter qu'un type (Obs Num) et que l'addition ne peut se faire qu'entre arguments de type Num . A l'inverse l'expression

```
(affect v (lambda (v!) 0)
        (affect v (lambda (v!) (+ v! 2))
                (use v (lambda (v!!) (+ 2 v!!))))))
```

admet le type (Obs Num) si v: Ref Num .

De plus l'introduction des règles correspondant à nos opérateurs dans le système de récriture lui conserve sa confluence, ce qui nous permet d'utiliser toute stratégie d'évaluation (aux problèmes de terminaison près) et notamment l'évaluation paresseuse.

3.4.3.f Le cas de prog2

$$\textbf{R.14}\quad \frac{(\Gamma \vdash p : (\text{Obs } \tau_1)) \qquad (\Gamma \vdash k : (\text{Obs } \tau_2))}{\Gamma \vdash (\text{prog2 } p\ k) : (\text{Obs } \tau_2)}$$

3.4.3.g Le cas de fork

$$\textbf{R.15}\quad \frac{(\Gamma \vdash p : (\text{Obs } \tau_1)) \qquad (\Gamma \vdash k : (\text{Obs } \tau_2))}{\Gamma \vdash (\text{fork } p\ k) : (\text{Obs } \tau_2)}$$

L'introduction du parallélisme (par utilisation de `fork`) dans un cadre fonctionnel impératif peut conduire à des interférences entre instructions et par suite à une exécution non déterministe.

Ce type de fonctionnement est à proscrire car il invalide toutes les propriétés formelles sur lesquelles est basé notre modèle, nous allons donc étudier des conditions permettant de garantir le déterminisme.

3.4.4 Déterminisme

3.4.4.a Objectif

Le but poursuivi à travers le déterminisme est de pouvoir garantir $\forall\ p\ k$ (fork $p\ k \equiv$ prog2 $p\ k$), ce qui garantit que toutes les propriétés initiales de notre modèle (notamment la confluence) sont conservées.

3.4.4.b Atomicité des opérations sur la mémoire

Le cycle des nœuds mémoire garantit l'atomicité des accès aux variables modifiables partagées telle que *[DIJKSTRA 65]* en montra la nécessité (cf. 1.6.2.b page 38), ce qui nous permet de nous contenter de la notion d'entrelacement pour appréhender le parallélisme (voir par exemple *[MANNA 91]* pour une présentation des notions d'*entrelacement* et de *recouvrement* d'exécutions).

Par contre, en l'absence de restrictions supplémentaires, toutes les exécutions parallèles pouvant résulter de l'entrelacement des différents processus sont possibles ce qui rend l'exécution globalement non déterministe. Ce cas de figure ne pouvant être toléré, nous allons introduire un ensemble de restrictions syntaxiques proche de celui présenté dans *[STEELE 90]* permettant de garantir une exécution indépendante de l'entrelacement.

3.4.4.c Conditions de déterminisme

Commençons par considérer deux opérations sur la mémoire qui appartiennent au même processus (nous qualifierons de telles expressions d'*expressions sérialisées*) : leur ordre relatif d'exécution est fixe ce qui évite tout problème (donc si nous restons en séquentiel le comportement est déterministe, comme nous pouvions nous y attendre).

Si les deux opérations appartiennent à deux processus différents (quelle que soit leur filiation), leur ordre relatif ne peut être prévu et il faut donc qu'il soit indifférent relativement à l'état final de la mémoire. Dans un tel cas, nous dirons que les deux opérations commutent.

De manière à simplifier les vérifications nous prendrons une restriction plus forte et exigerons que ces deux opérations commutent quel que soit le contenu initial de la mémoire.

Nous pouvons déjà observer que dans le cas général :

- Deux opérations sur deux variables distinctes commutent.

- Une écriture et une lecture de la même variable ne commutent pas.
- Deux lectures d'une même variable commutent.

La définition formelle de la commutation de deux écritures d'une même variable sera finalement : (affect v f k) et (affect v f' k') commutent si et seulement si $\forall x \ (f(f' x)) = (f'(f x))$ [1].

Plus généralement deux processus `P` et `P'` peuvent être exécutés parallèlement sans risque de non déterminisme si et seulement si toute variable `v` utilisée par `P` et par `P'` est soit utilisée uniquement en lecture (par `use`) soit si tout couple d'expressions non sérialisées utilisant `v` commute (cela impose que toutes ces expressions soient soit des lectures par `use`, soit des écritures par `affect`).

3.4.4.d Restrictions d'utilisation de nos opérateurs

Nous allons traduire les restrictions précédentes en restrictions d'utilisation des nos opérateurs :

- Dans le cas d'une utilisation séquentielle (aucune occurrence de `fork` dans la portée du `p-program`), aucune restriction n'est nécessaire, ce qui permet de prendre en compte tous les algorithmes séquentiels impératifs classiques. Toutefois, il existe déjà une construction similaire a `p-program` et qui est séquentielle (la construction `program` présentée en introduction de cette partie de chapitre). Nous n'envisagerons donc pas de prendre en compte les optimisations possibles dans le cas particulier où `p-program` est utilisé en séquentiel.
- Pour écrire des (`p-program` ((`v`$_1$ `e`$_1$)..(`v`$_n$ `e`$_n$)) `r` ((`p`$_1$ `d`$_1$)..(`p`$_m$ `d`$_m$)) `a`) parallèles et déterministes nous devrons limiter les utilisations conjointes de `fork`, `affect` et `use` :

  Le rôle de `r` (y compris le corps des procédures que peut appeler `r`[2]) est uniquement d'exploiter le contenu de la mémoire tel que l'a laissée `a` pour en extraire le résultat du bloc et non de lancer des processus parallèles sur des expressions de type observeur. Nous interdirons donc l'utilisation de `fork` dans `r`, ce qui nous laissera toute latitude pour y employer `use` ou `affect`.

  Le rôle de `a` (y compris le corps des procédures que peut appeler `a`[3]) est de modifier l'état de la mémoire pour permettre à `r` d'en extraire le résultat final. Il n'est donc pas question d'y interdire l'utilisation de `affect`. Dans le cas d'une utilisation parallèle de `p-program`, la restriction que nous retiendrons pour `a` sera donc la non-utilisation de `use`[4] et la vé-

1. Nous pourrons utiliser la commutativité des opérateurs tels que AND, OR, UNION, ... pour monter de façon immédiate la commutation des transformations de variables.

2. Remarquons que les procédures de type fonctionnel ne sont pas concernées par ces problèmes de déterminisme, vu qu'elles ne peuvent modifier l'état.

3. Même remarque.

4. Une variable affectable partagée accessible en lecture ne peut aussi être accédée en écriture. Ce type de variable, qui tient le rôle d'une constante, nous est inutile.

rification que toute les affectations non sérialisées d'une même variable commutent.

La condition de commutation des transformation n'est pas aisément vérifiable et n'est actuellement pas vérifiée automatiquement. La solution retenue pour remédier à ce problème est similaire à celle adoptée pour la stricticité : l'utilisateur doit indiquer les paramètres stricts (et par différence les paramètres non stricts) au compilateur, de la même façon il doit indiquer les transformations qui commutent. La seule nuance est que des transformations qui ne commutent pas ne sont pas utilisables à l'intérieur de la partie parallèle de `p-program` ce qui rend inutile une déclaration de commutation : le simple fait d'utiliser une transformation dans la partie parallèle de `p-program` vaut déclaration de commutation.

- Remarque : dans le cas où une procédure est appelée depuis `a` et depuis `r`, celle-ci devrait normalement vérifier les deux ensembles de contraintes. Ce problème est similaire à celui qui peut arriver pour la stricticité d'une fonction : lorsqu'une fonction est appelée dans différents contextes elle peut avoir plusieurs stricticités différentes, ce qui se règle généralement en dupliquant la définition en autant d'exemplaires qu'il y a de stricticités différentes, chaque exemplaire étant utilisé là où il convient. La solution à notre problème est similaire : il suffit de dupliquer la fonction, l'exemplaire utilisé depuis `a` vérifiant les contraintes attachées à `a`, celui utilisé par `r` vérifiant les contraintes attachées à `r`.

3.4.5 Discussion

3.4.5.a p-program dans notre modèle

Quelles sont les interférences entre les constructions impératives que nous venons d'introduire et le reste de notre langage, c'est à dire quelles contraintes devons nous supporter à cause de `p-program`?

Un programme n'utilisant pas nos constructions ne supporte aucune contrainte d'écriture ou de performance liée à l'existence de `p-program`.

Le calcul de `r`, résultat du bloc `(p-program v r p a)` se fait après la fin du dernier calcul parallèle lancé par `a` puis par des `fork` éventuels. De ce fait toute expression parallèle qui serait lancée depuis `a` par `processus`, comme cela se fait classiquement, est susceptible de survivre à la fin du calcul de `a` et ainsi de modifier les variables dont dépend `r` après la détermination de ce dernier.

De façon plus générale, tout ce qui peut amener une modification de l'ordre d'évaluation d'une expression non fonctionnelle doit être proscrit. Cela comprend l'emploi des directives `eager` et `processus` sur une expression non fonctionnelle (qui conduisent respectivement un appel par valeur séquentiel et parallèle et donc provoquent une anticipation de l'évaluation) et le déclenchement d'un processus sur une expression non fonctionnelle à l'aide d'une primitive parallèle (effet identique à celui de `processus`).

Cette interdiction est tout simplement garantie par le système de types en limitant toutes ces constructions aux expressions de type fonctionnel.

De plus, la déclaration de stricticité (faite pour pallier à l'absence d'analyse automatisée) qui est habituellement utilisée pour signaler au compilateur des opportunités d'appel par valeur n'a pas lieu d'être pour les paramètres de type non fonctionnel.

Enfin, le coût d'un processus lancé par un `fork` étant plus élevé que le coût d'un processus classique, le grain le plus fin ne sera pas accessible, au parallélisme entre observeurs. Par contre, la directive `processus` restera bien entendu utilisable pour déclencher du parallélisme au niveau fonctionnel à l'intérieur même d'expressions de type non fonctionnel.

3.4.5.b p-program et les autres approches impératives

Par rapport à une approche automatique (ne faisant pas appel à l'utilisateur) telles que Parcel, nous gagnons en efficacité du code généré et en temps de compilation.

Par rapport aux approches plus classiques telles que QLISP, dans lesquelles l'utilisateur spécifie les synchronisations à l'aide d'outils plus ou moins évolués, nous gagnons une synchronisation automatique par le graphe des accès aux variables et la possibilité de conserver une évaluation par nécessité comme mode d'évaluation par défaut grâce aux propriétés des accès aux variables.

Ce dernier point est à notre avis fondamental, la puissance de l'évaluation paresseuse en faisant un des outils majeurs de l'algorithmique fonctionnelle parallèle.

3.4.6 Conclusion

L'utilisation pratique d'une construction telle que `p-program` qui autorise un emboîtement de programmation parallèle fonctionnelle et impérative permet d'utiliser localement dans un algorithme le style le plus adapté. La sémantique claire du petit nombre d'opérateurs introduits permet de les utiliser facilement.

De plus la mise en évidence de restrictions peu contraignantes pour obtenir un fonctionnement parallèle déterministe assure à cette construction un statut d'outil algorithmique à part entière, comme nous allons le voir sur un exemple.

3.5 Un exemple significatif : IDA*

3.5.1 Présentation du problème

Nous allons mettre en œuvre notre méthode sur un exemple conséquent qui consiste à calculer des motifs consensus sur des chaînes d'ADN[1].

Un nucléotide est une des quatre lettres de l'alphabet génétique. Leurs noms sont Adénine, Cytosine, Guanine et Thymine, qui sont symbolisés respectivement par A, C, G et T.

Une succession de nucléotides est appelée une séquence. Par exemple la séquence : A A C T G.

Si l'information comporte une alternative entre nucléotides (par exemple une alternative entre A et G), nous qualifierons cette information de dégénérée. Un nucléotide dégénéré est appelé un code. Parmi tous les codes possibles, seuls sont pris en compte par l'algorithme les six codes à deux possibilités (symbolisés par AC, AG, AT, CG, CT, GT) et celui à quatre possibilités symbolisé par ACGT[1].

Si une chaîne d'ADN contient un (ou plusieurs) code, elle est appelé un motif, la dégénérescence du motif est le produit des cardinaux des codes qui le composent. Par exemple le motif A CG ACGT T a une dégénérescence égale à $1 \times 2 \times 4 \times 1 = 8$. Enfin, nous appellerons consensus un sous-motif commun à un ensemble de séquences.

Le problème posé par les généticiens consiste à retrouver, à partir de séquences d'ADN induisant des caractéristiques communes, la sous-séquence responsable de cette caractéristique. Cette sous-séquence ne se retrouve pas identiquement dans toutes les séquences, d'où le besoin d'introduire la notion de dégénérescence qui indique alors que la caractéristique étudiée est induite par une certaine famille de sous-séquences.

La localisation des motifs solution est arbitraire et n'est pas forcément la même dans les différentes séquences. Enfin, plus la dégénérescence des solutions est faible, plus le résultat est significatif (plus les séquences ont effectivement une composante commune qui peut expliquer la caractéristique induite) et la dégénérescence des meilleures solutions est initialement inconnue.

Les ordres de grandeurs visés par les généticiens sont de 15 à 50 motifs de 1000 à 3000 codes chacun, la taille des motifs consensus souhaités est supérieure à 10 codes. Pour donner une idée de l'ampleur du problème, remarquons qu'il existe 11 codes distincts et qu'il existe donc 11^n motifs de taille n.

1. Ce problème nous a été fourni par ELF SANOFI *[RAMOND 89]*.

1. Ce choix est celui de l'algorithme original. Il est justifié par la faible différence d'information (génétiquement parlant) entre un code à trois lettres et le code à quatre lettres, de plus un choix différent conduirait à une explosion combinatoire de l'espace de recherche.

3.5.2 Présentation de l'algorithme

L'algorithme utilisé consiste à partir d'un motif initialement vide et à accroître sa taille jusqu'à la taille désirée, les seuls accroissements envisagés étant bien sûr ceux de dégénérescence minimum.

Nous allons donc explorer un arbre de recherche dont la profondeur est la taille souhaitée pour le motif consensus et dont le facteur de branchement est 11.

Ce problème consistant à explorer un espace de recherche, nous avons choisi d'utiliser l'algorithme IDA* *[KORF 85], [KUMAR 90]* qui est une variante de A* et réalise une recherche itérative de toutes les meilleures solutions dans un espace de recherche.

Le but étant ici de rechercher les solutions les moins dégénérées, la mesure du coût est tout simplement la dégénérescence des motifs candidats.

Chaque itération consiste à chercher des solutions en supposant leur coût inférieur à un seuil (le seuil initial peut être pris égal au coût de la racine) qui va donc limiter l'espace de recherche envisagé. L'itération produit un ensemble de solutions (éventuellement vide) et le seuil à appliquer pour l'itération suivante. L'algorithme s'arrête à la première itération produisant un résultat.

La limitation de l'espace de recherche par le seuil est due au fait que l'algorithme élimine toutes les branches de coût supérieur au seuil (celles-ci seront éventuellement envisagées lors d'une itération ultérieure faite avec un seuil supérieur), ce qui définit un sous-espace de recherche à l'intérieur de l'espace constitué de tous les motifs possibles de la longueur recherchée.

Le nouveau seuil utilisé suite à une itération qui n'a produit aucun résultat est le plus petit coût parmi les branches qui ont été précédemment abandonnées (il est donc le coût minimum envisageable pour une solution).

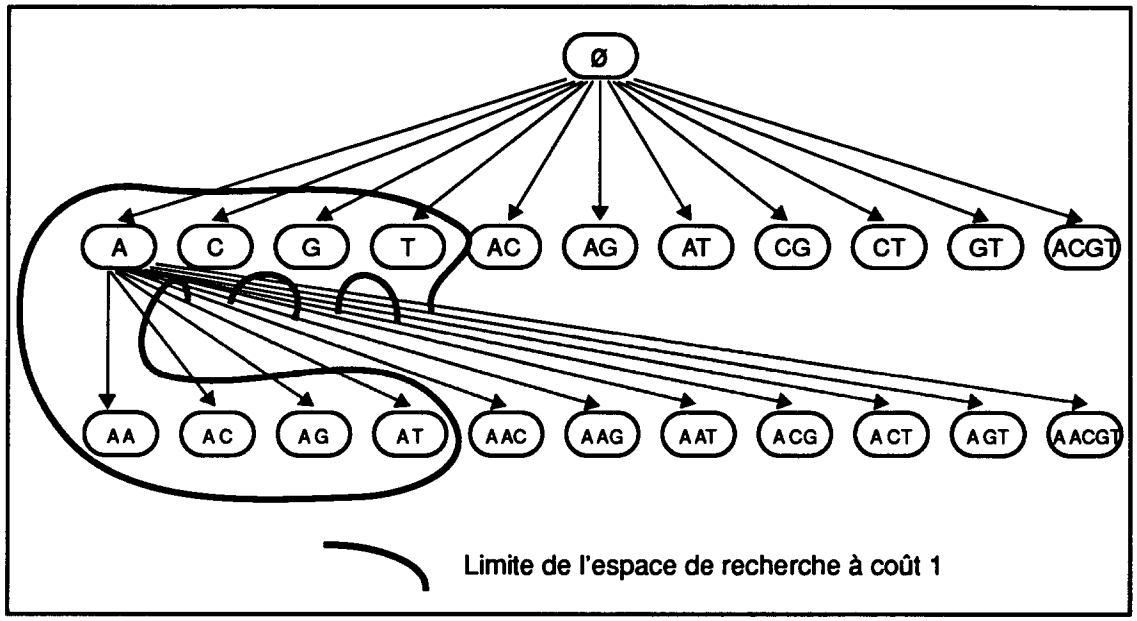
Si une itération produit des résultats, ils sont tous de coût minimum. Dans le cas contraire, celles des solutions qui ont le plus faible coût auraient été détectées lors d'une itération précédente.

Les caractéristiques d'IDA* sont qu'il réalise une recherche en profondeur d'abord à l'intérieur de l'espace de recherche à coût borné correspondant à chaque itération (cf. Figure 7). La taille totale de tous les espaces de recherche envisagés pour trouver la solution tend asymptotiquement vers la taille de la dernière itération, l'espace mémoire nécessaire est donc en $O(l)$ où l est la longueur du motif recherché.

De plus à chaque nœud IDA* n'explore que les branches de coût optimal, ce qui garantit l'optimalité pour toute heuristique admissible. Enfin, et contrairement à A* qui est un algorithme séquentiel dur car il gère les nœuds en attente d'être envisagés, IDA* abandonne définitivement les nœuds dont le coût est supérieur au seuil, ce qui supprime cette gestion des nœuds en

attente et fait de IDA* un algorithme parallélisable avec un *speed-up* qui tend à être linéaire lorsque la taille du problème augmente *[KUMAR 90]*.

Figure 7 L'espace de recherche envisagé par IDA*



A l'intérieur d'une itération, l'exploration d'un nœud produit son coût pour la détermination du prochain seuil, sa contribution à l'ensemble des solutions et ses fils à explorer. La source de parallélisme envisagée est bien sûr d'explorer plusieurs nœuds en parallèle, le problème étant que le processus père doit attendre ses fils pour recomposer seuil et ensemble de résultats de ses fils.

3.5.3 La solution fonctionnelle

Le programme fonctionnel présenté Figure 8 est tout ce qui est de plus classique. Les fonctions secondaires telles que `fils` qui produit les fils d'un nœud ont été omises, `main` lance le calcul à partir de la racine avec la taille du consensus recherché et un degré de dégénérescence (seuil) initial, `ida` réalise la recherche de solutions à partir de cette dégénérescence et jusqu'à une dégénérescence suffisante et `dfs` calcule la liste des solutions qui correspondent à une taille de solution et à une dégénérescence données.

Le parallélisme provient de la directive `processus` dans `dfs` qui lance en parallèle un nouveau `dfs` sur les fils du nœud courant. Notons que cette première solution conduit à une parallélisation excellente comme le montrent les résultats de simulations sur une configuration à 32 processus (cf. Figure 9).

Figure 8 Version fonctionnelle parallèle classique de IDA*

```
(defun compose (sol1 cout1 k)
   (multiple-value-bind (sol2 cout2) k
      (values (append sol1 sol2) (min cout1 cout2))))

(defun fusion (r1 r2)
  (multiple-value-bind (sol1 cout1) r1
    (multiple-value-bind (sol2 cout2) r2
      (values (append sol1 sol2) (min cout1 cout2)))))

(defun dfs (cout-max taille pile resultat)
  (reduce
    (lambda (fils-cour dfs-cdr)
      (if (consensus fils-cour)
          (let ((heur (heuristique fils-cour)))
            (if (≥ cout-max heur)
                (if (= taille (length (motif fils-cour)))
                    (compose (list (reverse (motif fils-cour)))
                             max_int dfs-cdr)
                    (fusion dfs-cdr
                            (processus
                               (dfs cout-max taille
                                    (fils fils-cour)
                                    resultat))))
                (compose nil heur dfs-cdr)))
          dfs-cdr))
    pile resultat))

(defun ida (solution cout-max-init taille)
  (if (null solution) (main taille cout-max-init) solution))

(defun main (taille degre)
  (multiple-value-bind (solution cout-max-suivant)
    (dfs degre taille (list (racine)) (values nil max_int))
    (ida solution cout-max-suivant taille)))
```

Le taux de parallélisme sur cet exemple est de 16.55, et le taux de processus en attente 4.39, ce qui correspond à 26% des processus environ, ce qui est un très bon résultat.

Toutefois, le speed-up théorique linéaire ne peut être atteint par cette approche à cause du nombre de processus en attente croissant tout au long des trois itérations de l'algorithme.

3.5.4 La solution avec p-program

3.5.4.a Présentation

Le code correspondant à celui présenté pour la solution fonctionnelle est consigné Figure 10.

Figure 9 Résultats d'exécution de la solution fonctionnelle

```
(main 4 0)
=> ((CT G A CT) (G A GT AC) (G A T ACGT))

220 processus, 138639 cycles, parallelisme = 16.55,
inactivite = 4.39, 615577 primitives executees.

     0 -@@                                |
  4096 -@@@@@                             |
  8192 --@@@@@@@@                         |
 12288 --@@@@@@@@                         |
 16384 --@@                               |
 20480 -@                                 |
 24576 -@@@@                              |
 28672 --@@@@@@@@@@@@                     |
 32768 --@@@@@@@@@@@@@@@@@@@@@@@@@@@@     |
 36864 ---@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 40960 ------@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 45056 ---------@@@@@@@@@@@@@@@@@@@@@@   |
 49152 --------@@@@@@@@@@@@               |
 53248 ---@@                              |
 57344 -@@                                |
 61440 -@@@@@                             |
 65536 --@@@@@@@@@@@@@@@                  |
 69632 --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   |
 73728 --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 77824 ----@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 81920 -----@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 86016 ---@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 90112 -----@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 94208 ------@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 98304 ------@@@@@@@@@@@@@@@@@@@@@@@@@@@|
102400 -----@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
106496 ------@@@@@@@@@@@@@@@@@@@@@@@@@@@|
110592 -------@@@@@@@@@@@@@@@@@@@@@@@@@@|
114688 --------@@@@@@@@@@@@@@@@@@@@@@@@@|
118784 ---------@@@@@@@@@@@@@@@@@@@@@@@@|
122880 -------------@@@@@@@@@@@@@@@@@@@@|
126976 ------------@@@@@@@@@@@@@@@      |
131072 -----@@@@@                         |
```

Les fonctions `compose` et `fusion` qui servaient à la gestion des valeurs multiples retournées par `dfs` ne sont plus nécessaires et n'ont pas d'équivalent dans cette version.

La fonction principale `main` contient un bloc `p-program` dont les variables locales sont le résultat et le seuil de l'itération suivante.

Figure 10 Le programme avec des variables modifiables

```
(defun main (taille cout-max-init)
  (p-program
    ;; Les variables locales
    ((solution nil) (cout-max-suivant Max_int))
    ;; Le résultat
    (use solution
         (lambda (sol)
           (use cout-max-suivant (lambda (cout) (ida sol cout)))))
    ;; Les procédures locales
    ((dfs (lambda (cout-max taille pile)
            (reduce
              (lambda (fils-cour dfs-cdr)
                (if (consensus fils-cour)
                    (let ((heur (heuristique fils-cour)))
                      (if (≥ cout-max heur)
                          (if (= taille
                                 (length (motif fils-cour)))
                              (affect solution
                                      (lambda (sol)
                                        (cons (reverse
                                                (motif fils-cour))
                                              sol))
                                      dfs-cdr)
                              (fork (dfs cout-max taille
                                         (fils fils-cour))
                                    dfs-cdr))
                          (affect cout-max-suivant
                                  (lambda (cout) (min heur cout))
                                  dfs-cdr)))
                    dfs-cdr))
              pile t)))
     (ida (lambda (sol cout)
            (if (null sol) (main taille cout) sol)))))
    ;; le calcul initial
    (dfs cout-max-init taille (list (racine)))))
```

La fonction `dfs` devient une procédure locale pour des raisons de modularité, mais ceci n'est pas obligatoire.

La fonction `ida` devant utiliser la variable `solution` pour déterminer si le calcul est terminé doit utiliser `use` et est donc rattachée à la partie résultat du `p-program`.

La source de parallélisme est la même que celle mise en œuvre dans la solution fonctionnelle. La différence à l'exécution provient de ce que ce parallélisme est déclenché par `fork` et non plus par `processus`, ce qui nous affranchis des synchronisations inutiles entre processus père et processus fils. Ce phénomène est visible en comparant les comportements parallèles de deux versions du programme (cf. Figure 9 et Figure 12).

3.5.4.b Conformité du programme aux contraintes

- Typage

  Notre programme vérifie les contraintes de type présentées en 3.4.3, les types des principales définitions sont présentées Figure 11.

Figure 11 Types des principales définitions

```
main : Num → Num → List Solution
solution : Ref (List Solution)
cout-max-suivant : Ref Num
dfs : Num → Num → List Noeud → Obs Bool
ida : List Solution → Num → List Solution
```

- Utilisation des primitives

  La partie résultat de `p-program` utilise la procédure locale `ida`, mais celle-ci est de type fonctionnel et n'a donc pas à être considérée.

  La partie action de `p-program` utilise la procédure locale `dfs` qui ne provoque aucun appel à `use` et qui n'utilise que des transformations qui commutent.

Notre programme vérifie donc les contraintes de type et les restrictions d'utilisation des opérateurs. De plus la fonction principale `main` calcule un résultat de type fonctionnel ce qui nous permet d'évaluer toute expression `(main n m)` où `n` et `m` sont de type `Num`.

3.5.4.c Résultats

Les résultats de simulation du même exemple toujours avec une configuration à `32` processus avec cette nouvelle version du programme sont présentés Figure 12.

Outre une légère accélération du calcul de l'ordre de 15%, le taux de parallélisme monte à 18.81 et le taux de processus en attente descend à 1.02, ce qui correspond à 5% des processus environ. L'utilisation des ressources en processus est donc meilleure, ce qui permet d'envisager des gains plus importants sur des données plus grosses ou dans le cas d'une multi-programmation.

Enfin, le nombre de processus en attente durant les trois itérations de l'algorithme reste constant, ce qui nous permet d'envisager le *speed-up* théorique linéaire.

3.5.5 Conclusion

Nous pouvons observer sur cet exemple complexe qui admet une très bonne solution séquentielle que l'utilisation de `p-program` améliore encore le

Figure 12 Résultats d'exécution de la solution avec variables affectables

```
(main 4 0)
=> ((CT G A CT) (G A GT AC) (G A T ACGT))

284 processus, 120777 cycles, parallelisme = 18.81,
inactivite = 1.02, 613213 primitives executees.

     0 -@@                                |
  4096 -@@@@@                             |
  8192 -@@@@@@@@                          |
 12288 -@@@@@@@@@@                        |
 16384 -@@@                               |
 20480 -@                                 |
 24576 -@@@@                              |
 28672 -@@@@@@@@@@@@                      |
 32768 -@@@@@@@@@@@@@@@@@@@@@@@@@@       |
 36864 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 40960 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 45056 -@@@@@@@@@@@@@@@@@@@@@@@@           |
 49152 -@@@@@@@                           |
 53248 @                                  |
 57344 -@@@                               |
 61440 -@@@@@@@@@@@                       |
 65536 -@@@@@@@@@@@@@@@@@@@@@@@@@@       |
 69632 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 73728 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 77824 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 81920 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 86016 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 90112 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 94208 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
 98304 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
102400 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
106496 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
110592 -@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
114688 -@@@@@@@@@@@@@@@@@@@@                |
```

comportement parallèle de l'algorithme. De plus l'utilisation des constructions associées à `p-program` nous est apparue aisée à mettre en œuvre.

Cet exemple est particulièrement significatif car il correspond d'une part à un type de problème très important et qu'il met en lumière la possibilité d'atteindre, grâce à la construction `p-program`, le *speed-up* théorique de IDA* alors qu'une parallélisation fonctionnelle classique en est incapable.

3.6 Conclusion

Nous avons identifié la limitation majeure de notre modèle au niveau de la gestion des processus comme étant le coût associé aux processus en attente et nous avons montré certains remèdes :

Il est possible d'améliorer le taux de parallélisme d'algorithmes peu favorables à une parallélisation classique en déclenchant des processus complémentaires pour certaines occurrences de parallélisme dont la pertinence dépend du taux d'occupation de la machine ou de la taille des données. La parallélisation à l'aide de processus complémentaires du programme cecilia déjà étudié à plusieurs reprises montre le bien fondé d'une telle approche, et ce d'autant plus qu'elle en est grandement facilitée.

Il est possible d'atteindre un taux d'attente minimal sur certains algorithmes dont le parallélisme potentiel est à grain moyen ou gros en utilisant la construction parallèle `p-program`, comme nous l'avons illustré avec une parallélisation très efficace de l'algorithme général IDA*.

La classe des problèmes concernés correspond à ceux mettant en œuvre une synchronisation inutile au déterminisme du résultat (contributions commutatives à un résultat global), mais pour lesquels une parallélisation classique induit des attentes dommageables pour le taux de parallélisme effectif.

L'aspect le plus intéressant est que cette construction permet d'exploiter un parallélisme inaccessible à une parallélisation fonctionnelle classique.

Chapitre 4

Le paradigme pipe-line revu et corrigé

Objectif : Montrer comment réaliser ce paradigme et noter la complexité des choix à faire. Faire ressortir un coût non encore montré de ce paradigme : le grand nombre de processus peu coûteux rend l'ensemble coûteux. Présenter les solutions connues pour simuler le multi-pipe-line et les coûts associés. Montrer une nouvelle solution répondant à ces deux problèmes et le gain important qu'elle entraîne.

4.1 Le paradigme pipe-line

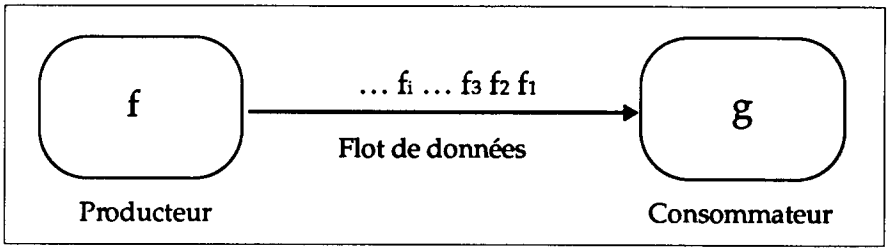
4.1.1 Généralités

Le paradigme pipe-line (ou producteur-consommateur) correspond à un des styles les plus naturels en programmation fonctionnelle et de nombreux ouvrages traitant d'algorithmique fonctionnelle lui consacrent une place de choix, les plus connus étant probablement *[HENDERSON 80]*, *[ABELSON 85]* et *[PEYTON-JONES 87b]*.

Son utilisation produit des programmes très clairs mais il est source d'inefficacité dans le cas de traitements séquentiels pour lesquels il constitue une source supplémentaire de travail à cause des structures de données intermédiaires entre traitements *[WADLER 84]*. Cependant ce point de vue ne prévaut pas dans le cas où le paradigme est introduit en vue d'une parallélisation, ce qui est notre cas.

L'effet pipe-line est mis en œuvre en connectant deux traitements qui travaillent en parallèle par un tampon. Le processus producteur écrit ses résultats dans le tampon qui est accédé en lecture par le consommateur (cf. Figure 1). Le tampon peut être de taille bornée ou non bornée, les données sont lues dans l'ordre où elles ont été écrites.

Figure 1 Le traitement pipe-line



Le buffer qui est partagé par deux processus doit être à accès synchronisé de manière à éviter toute incohérence. Par exemple *[MANNA 91]* rappelle la solution classique utilisant des sémaphores pour synchroniser les accès et éviter au consommateur de faire une lecture d'élément dans un buffer vide.

En programmation fonctionnelle le buffer est généralement implanté sous forme d'une liste non stricte (c'est à dire construite par une primitive `cons` capable de retourner une cellule que ses arguments soient ou non évalués) (cf. 1.1.2.e). La taille du buffer est alors non bornée (si ce n'est par la taille mémoire) et la synchronisation se fait non pas au niveau du buffer complet mais au niveau de la cellule mémoire.

Dans le cadre de la réduction parallèle de graphe, la synchronisation entre deux processus est assurée par la mémoire : si un processus tente d'accéder à un graphe *en réduction* (en cours de calcul par un autre processus), cette lecture va le bloquer jusqu'à la récriture du graphe (cf. 1.6 page 37).

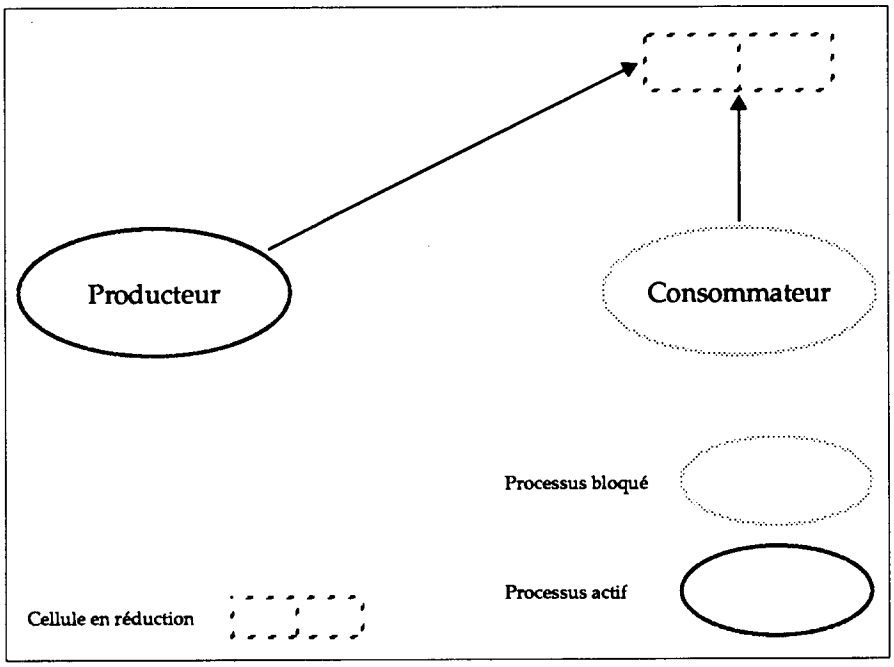
La synchronisation entre producteur et consommateur est donc faite implicitement par la mémoire. Si le consommateur est en avance sur le producteur et tente d'accéder à une donnée non encore construite (cf. Figure 2), ce qui correspond pour lui à lire un noeud *en réduction*, il sera bloqué jusqu'à la récriture du noeud par le producteur.

Sinon, le consommateur et le producteur travaillent sur des cellules distinctes et de ce fait n'interagissent pas (cf. Figure 3).

Nous qualifierons le comportement d'un processus producteur qui produit les éléments du buffer indépendamment des besoins du consommateur, au rythme ou ses propres données sont disponibles, d'*industrieux*.

A l'opposé, la production peut être assurée par le processus consommateur grâce au mécanisme d'évaluation par nécessité qui provoquera la construction des éléments du flot lorsque le consommateur voudra y accéder alors qu'ils ne sont pas encore construits (cf. 1.1.2.e page 15). Dans ce cas il n'y a pas de processus producteur et il n'y a donc ni parallélisme ni buffer.

Figure 2 Consommateur bloqué en attente du producteur



4.1.2 Description du paradigme

4.1.2.a Les briques de base

Un programme mettant en œuvre des pipe-lines peut se représenter à l'aide de boites (les traitements) et de flèches (les flots), à la manière Data-Flow.

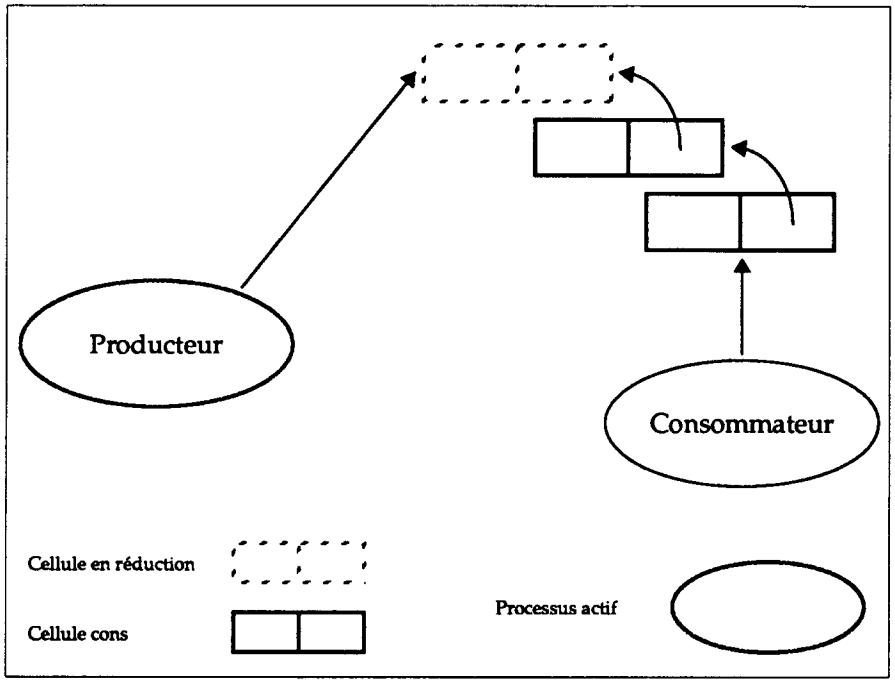
Nous différencierons quatre types principaux de traitements :

- Les générateurs, qui génèrent des flots à partir de valeurs atomiques.
- Les transformeurs, qui appliquent un opérateur à chaque élément de leur flot d'entrée pour former leur flot de sortie.
- Les filtres, qui produisent un flot de sortie en supprimant certains éléments de leur flot d'entrée.
- Les accumulateurs, qui condensent les valeurs successives de leur flot d'entrée en valeur atomique.

En plus de ces types principaux, nous pouvons dériver divers autres types par composition.

Les transformeurs, filtres et accumulateurs peuvent avoir plus d'un flot en entrée, un flot en sortie peut servir d'entrée à plusieurs traitements, mais il n'est pas possible pour un traitement d'avoir plusieurs flots distincts en sortie. Toutefois, diverses techniques permettent de simuler la production simultanée de plusieurs flots (cf. 4.1.3 page 116).

Figure 3 Producteur et consommateur actifs



4.1.2.b Correspondance entre graphe et programme

Il existe une correspondance entre graphe pipe-line et programme fonctionnel qui fait usage de certaines fonctions d'ordre supérieur bien connues.

Un transformeur qui applique une fonction `f` à un flot `E` pour produire un flot `S` peut tout simplement s'écrire à l'aide du combinateur `reduce` pour un fonctionnement paresseux (`p-reduce` pour un comportement industrieux) :

```
S=(reduce (lambda (x r) (cons (f x) r)) E nil)

S=(p-reduce (lambda (x r) (cons (f x) r)) E nil)
```

Un filtre qui supprime tout les éléments d'un flot `E` ne vérifiant pas le prédicat `p` pour former son flot de sortie `S` peut également s'écrire à l'aide de l'opérateur `reduce` (`p-reduce` pour la un fonctionnement industrieux) :

```
S=(reduce (lambda (x r) (if (p x) (cons x r) r)) E nil)

S=(p-reduce (lambda (x r) (if (p x) (cons x r) r)) E nil)
```

Un accumulateur peut souvent être réalisé à l'aide de l'opérateur `reduce` en se référant à la propriété :

```
reduce f (list x1 x2 … xn) e = (f x1 (f x2 … (f xn e)))
```

Ainsi l'accumulateur `Sigma` qui réalise la somme des éléments d'un flot `E` peut s'écrire : `Sigma E = reduce + E 0`.

Le cas des générateurs est plus complexe, bien que leur écriture sous forme de programme fonctionnel ne pose généralement aucun problème, car ils sont de forme trop variable (nombre de paramètres en entrée, critère d'arrêt, itérateur, ...) pour en tirer une abstraction générale simple.

4.1.2.c Le parallélisme

Le paradigme pipe-line consiste en une application d'un même traitement à toutes les valeurs d'un flot. Il est possible d'enchaîner ainsi plusieurs étages, chaque traitement produisant un flot en sortie qui sera lui même en entrée d'un autre traitement.

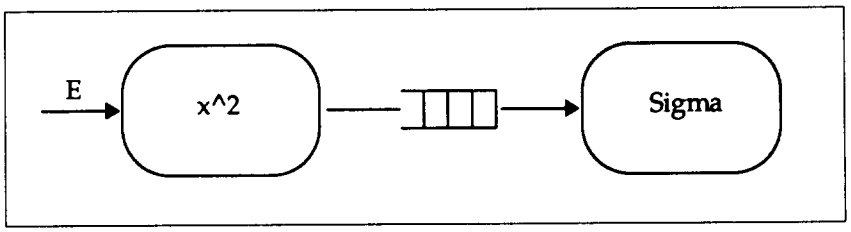
L'application des traitements des différents étages aux éléments du flot peut être réalisée soit élément par élément au fur et à mesure des besoins d'un consommateur placé en bout de chaîne (traitement paresseux), soit être réalisé indépendamment de tout besoin par certains étages qui vont ainsi produire leur propre flot de sortie aussi vite que possible (comportement industrieux).

Lorsque plusieurs traitements sont susceptibles de travailler en même temps il y a parallélisme. Les deux sources de parallélismes possibles sont d'une part le lancement d'un traitement industrieux qui s'exécutera en parallèle avec le reste du pipe-line dès lors qu'il disposera d'un élément à traiter sur son flot d'entrée, et d'autre part la mise en parallèle de plus d'un flot paresseux en entrée d'un traitement qui consomme en parallèle un élément sur chacun d'eux. Dans ce dernier cas la nécessité se propagera sur ces divers flots et les traitements correspondants se feront en parallèle (*[LIVEZEY 89]* qualifie ce parallélisme de parallélisme ET).

- Exemple de comportement industrieux

  Il s'agit de l'application de la fonction `x^2` (fonction utilisateur d'élévation au carré) à tous les nombres d'un flot `E`, lesquels nombres au carré sont accumulés pour calculer leur somme (cf. Figure 4). Ce graphe correspond au programme suivant (voir 4.1.2.b pour la dérivation du programme) :

Figure 4 Parallélisme industrieux



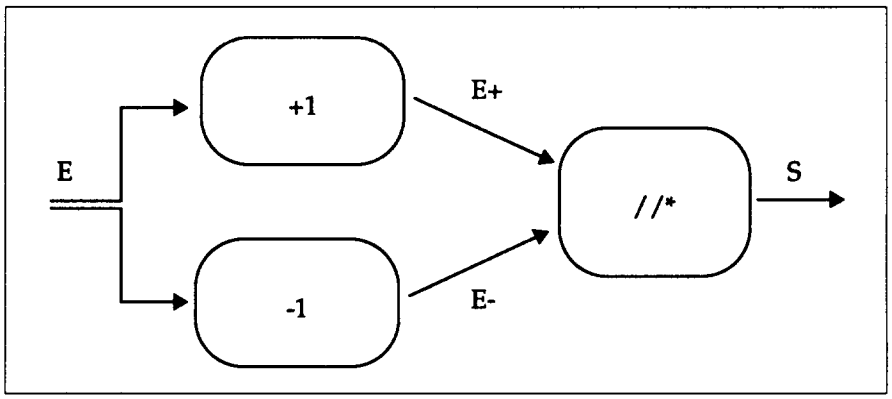
```
(Sigma (Carre E))

Carre E = (p-reduce (lambda (x r) (cons (x^2 x) r)) E nil)

Sigma E = reduce + E 0
```

Lorsque l'expression `(Sigma (Carre E))` est touchée, cela provoque le calcul de `(Carre E)` car `Sigma` est strict en son premier argument, ce qui provoque le calcul de tout le flot car sa définition est industrieuse. Le calcul de la somme et la production des carrés se feront alors en parallèle.

- Exemple de parallélisme ET

  Il s'agit de la production paresseuse du flot des `(ei + 1)*(ei - 1)` à partir du flot `E` des `ei` à l'aide de `//*` (fonction utilisateur de multiplication avec évaluation parallèle de ses arguments) (cf. Figure 5) dont le programme équivalent est présenté ci-dessous :

Figure 5 Parallélisme ET




```
(reduce2 (lambda (x y r) (cons (//* x y) r)) E+ E- nil)

E+ = (reduce (lambda (x r) (cons (1+ x) r)) E nil)

E- = (reduce (lambda (x r) (cons (1- x) r)) E nil)

reduce2 f l1 l2 e
= if (or (null l1) (null l2))
     e
     (cons (f (first l1) (first l2))
           (reduce2 f (rest l1) (rest l2) e))
```

Lorsque le consommateur demande l'élément suivant, cela entraîne l'évaluation de `(//* (first E+) (first E-))` qui provoque l'évaluation en parallèle de `(first E+)` et de `(first E-)`, c'est à dire du premier élément de `E+` et du premier élément de `E-`.

4.1.2.d Synchronisation entre producteur et consommateur

Selon que le producteur d'un flot est industrieux ou paresseux, la synchronisation avec le consommateur se fait selon divers modes.

Dans le cas paresseux, la production d'un élément supplémentaire du flot est déclenchée par le besoin du consommateur : celui-ci dispose en effet d'une référence au flot non évalué et lors du besoin, c'est à dire lorsque cette référence est touchée toute la partie de flot nécessaire pour satisfaire la référence est évaluée (cf. 1.1.2.e page 15). Par exemple, définissons un flot paresseux infini de 1 et un flot paresseux infini de 2 :

```
(def que-des-1 (cons 1 que-des-1))
;; que-des-1 = (cons 1 que-des-1)

(def que-des-2 (map 1+ que-des-1))
;; que-des-2 = (map 1+ que-des-1)
```

Puis évaluons le second élément du flot `que-des-2` :

```
(second que-des-2) => 2
;; que-des-1 = (cons 1 (cons 1 que-des-1))
;; que-des-2 = (cons 2 (cons 2 (map 1+ que-des-1)))
```

L'expression `(second que-des-2)` a joué le rôle de consommateur vis-à-vis du flot `que-des-2` qui joue lui même le rôle de consommateur vis-à-vis de `que-des-1`.

L'évaluation du second l'élément de `que-des-2` à également provoqué l'évaluation du premier élément de ce flot et des deux premiers éléments de `que-des-1` qui étaient nécessaires, mais toute la fin de ces deux flots reste non évaluée car elle était inutile à l'obtention du résultat et que leur définition est paresseuse.

Dans le cas d'un producteur industrieux, l'évaluation d'une expression qui référence le flot va entraîner son évaluation complète, élément par élément. Reprenons notre exemple précédent en donnant au flot `que-des-1` une définition industrieuse :

```
(def que-des-1 (cons 1 (processus que-des-1)))
;; que-des-1 = (cons 1 (processus que-des-1))

(def que-des-2 (map 1+ que-des-1))
;; que-des-2 = (map 1+ que-des-1)
```

Puis évaluons le second élément `que-des-2` :

```
(second que-des-2) => 2
;; que-des-1 = (cons 1 (cons 1 (cons 1 (cons 1 ...))))
;; que-des-2 = (cons 2 (cons 2 (map 1+ que-des-1)))
```

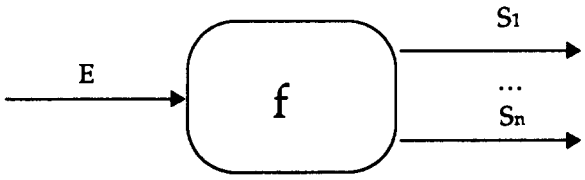
L'évaluation du second l'élément de `que-des-2` à également provoqué l'évaluation du premier élément de ce flot et des deux premiers éléments de `que-des-1` qui étaient nécessaires. Lorsque le calcul à touché `que-des-1` pour la première fois, son évaluation à commencée de manière industrieuse (en pa-

rallèle avec le reste des calculs) ce qui provoque l'évaluation de l'ensemble du flot (dans la limite de la mémoire disponible).

4.1.3 Plusieurs flots en sortie

L'utilisation de traitements produisant plus d'un flot semble tout à fait naturelle, pourtant leur écriture est impossible directement car une fonction ne peut retourner qu'un seul résultat. Nous allons voir comment contourner cette limitation avec quelques mécanismes simples.

Le schéma correspondant à notre problème est le suivant :



Diverses solutions à ce problème sont détaillées dans la littérature (par exemple *[LIVEZEY 89]*). Elles utilisent la possibilité de partager un même flot et font du seul flot en sortie un flot réel par lequel transitent autant de flots virtuels que nécessaire. Tout destinataire d'un de ces flots virtuel reçoit alors le flot réel d'où il doit extraire les données qui le concernent.

Nous allons étudier trois solutions en exprimant leur coût dans le cas où tout élément du flot en entrée produit un élément sur chaque flot virtuel (`Tous`) et dans le cas où chaque élément du flot en entrée produit un élément sur un des flots virtuels (`Un`). Nous exprimerons ce coût en fonction du nombre ℓ d'éléments en entrée et du nombre n de flots virtuels.

Pour référence, nous donnons dans le Tableau 4 le coût que pourrait avoir une production directe de ces flots.

Tableau 4 Coût de la production directe

	Cellules		Prédicat
	créations	accès	flot virtuel
Tous	$n\ell$	$(n+1)\ell$	-
Un	ℓ	2ℓ	ℓ

Les créations de cellule correspondent à la production des flots, les accès aux cellules comportent les accès du traitement `f` et des n consommateurs, le prédicat de flot virtuel est celui qui permet de décider dans quel flot virtuel doit être produit l'élément (uniquement dans le cas `Un`).

Le tableau contient le nombre de créations de cellules, d'accès à une cellule (aussi bien à celles de `E` qu'à celles des S_i) et le nombre d'appels au prédicat

qui détermine quel flot de sortie recevra les éléments du flot virtuel. Dans le cas Tous, il n'y a pas lieu de faire ce test (tous les flots reçoivent un élément), ce qui est signifié par '–'.

Par exemple, nous lisons dans le Tableau 4, cas Tous, qu'il y a $(n+1)\ell$ accès à un élément de flot qui correspondent à un accès par le producteur à chacun des ℓ éléments du flot E, et un accès par chacun des n consommateurs à chacun des ℓ éléments de son flot S_1.

4.1.3.a Produire un flot de structures.

Les élément du flot réel sont de type structure dont chaque composante correspond à un flot virtuel. Les divers consommateurs qui se partagent le flot réel n'ont qu'à sélectionner dans chaque élément la composante qui leur est destinée.

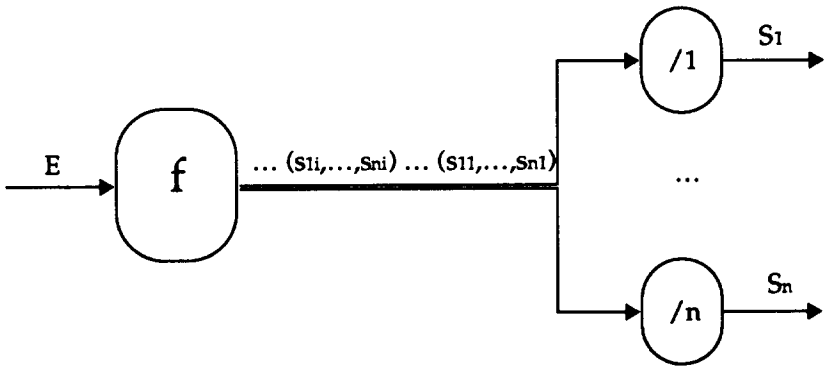


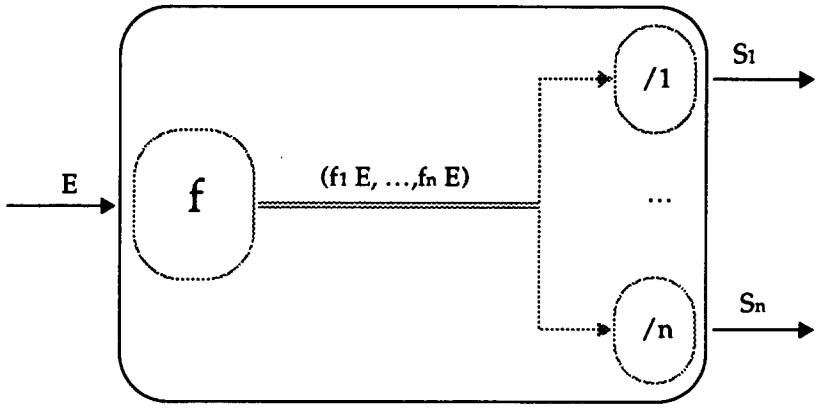
Tableau 5 Coût de la solution *flot de structures*

	Cellules		Structures	
	créations	**accès**	**créations**	**accès**
Tous	ℓ	$(n+1)\ell$	ℓ	$n\ell$

Cette solution impose la production d'un élément sur chaque flot virtuel en même temps, ce qui ne permet pas de traiter le cas Un.

4.1.3.b Produire une structure de flots.

Il s'agit simplement de renvoyer non pas un flot réel, mais une seule structure dont les composantes sont les différents flots désirés, les consommateurs sélectionnant celui qui les intéresse.



Pour cela, il est possible d'utiliser une fonction qui travaille directement sur la structure (comme représenté plus haut), mais une solution alternative est possible qui revient à partager le flot d'entrée puis à y appliquer indépendamment différents traitements comme nous pouvons le représenter (en omettant la structure) par :

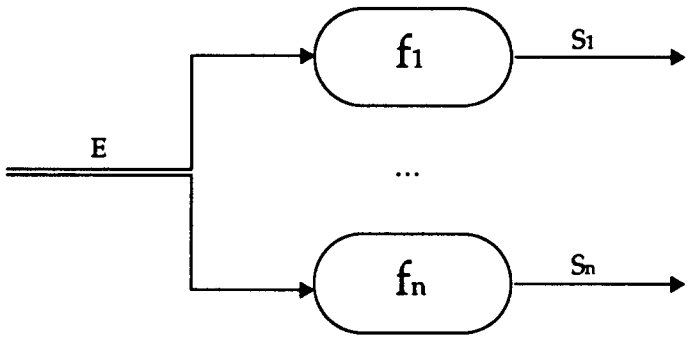


Tableau 6 Coût de la solution *structure de flots*

	Cellules		Structures		Prédicat
	créations	accès	créations	accès	flot virtuel
Tous	$n\ell$	$(n+1)\ell$	$\ell+1$	$\ell+n$	-
Un	ℓ	2ℓ	$\ell+1$	$\ell+n$	ℓ

Cette solution impose de construire une nouvelle structure pour chaque élément du flot d'entrée.

Tableau 7 Coût de la solution *structure de flots indépendants*

	Cellules		Structures		Prédicat
	créations	accès	créations	accès	flot virtuel
Tous	$n\ell$	$2n\ell$	1	n	-
Un	ℓ	$(n+1)\ell$	1	n	$n\ell$

Dans le cas Un, le prédicat permettant de décider quel flot virtuel produit un élément doit être dupliqué sur les n flots, ceux-ci étant produits indépendamment. Cependant, cette solution apporte plus de souplesse que la *structure de flots* simple car les flots sont produits indépendamment ce qui permet de suivre la nécessité imposée par chaque consommateur.

4.1.3.c Produire un flot d'éléments étiquetés.

Il s'agit de mettre dans le flot réel tous les éléments des divers flots virtuels après les avoir étiquetés au nom de celui auquel ils appartiennent. Les différents consommateurs n'ont plus alors qu'à filtrer les éléments pour ne retenir que ceux qui les concernent. Dans le cas Tous, les n éléments produits par l'unique producteur seront ajoutés au flot réel dans un ordre fixé lors du traitement de chaque élément de E.

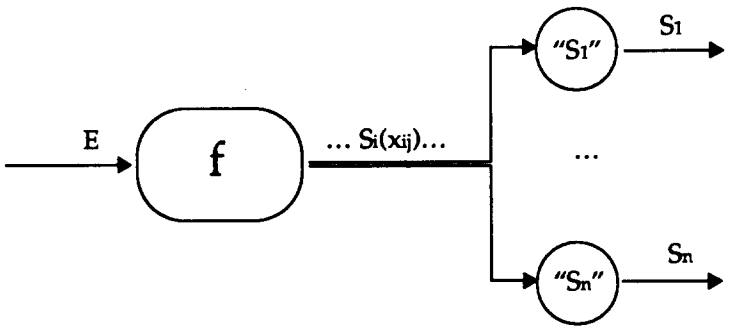


Tableau 8 Coût de la solution *éléments étiquetés*

	Cellules		Etiquettes		Prédicat
	créations	accès	créations	filtrages	flot virtuel
Tous	$n\ell$	$(n^2+1)\ell$	$n\ell$	$n^2\ell$	-
Un	ℓ	$(n+1)\ell$	ℓ	$n\ell$	ℓ

Dans le cas Tous, chaque élément en entrée entraîne la production de n éléments étiquetés qui sont tous accédés et filtrés par les n consommateurs.

Nous ne ferons pas un classement des différentes solutions présentées en fonction de leurs performances, celles-ci devant être vues en fonction de l'utilisation visée et de la souplesse nécessaire. Quoi qu'il en soit, aucune de ces solutions ne peut prétendre à un coût proche de celui que pourrait avoir une production directe.

4.1.4 Mise en œuvre

La programmation fonctionnelle permet de prendre en compte très simplement le parallélisme industrieux et le parallélisme ET en réalisant les flots à l'aide de listes paresseuses.

La production industrieuse d'un flot se réalise grâce à l'utilisation d'une primitive cons parallèle sur son second argument (en MaRS_Lisp ep-cons, lp-cons ou pp-cons)[1] alors que le parallélisme ET ne nécessite rien de particulier pour les flots qui seront produits à l'aide d'une primitive cons paresseuse sur son second argument (en MaRS_Lisp el-cons, ll-cons ou pl-cons)[2].

Chaque exécution d'une primitive cons parallèle sur son second argument entraîne la fin du processus qui l'a exécutée et lance un nouveau processus qui construira la fin de la liste. En conséquence, chaque élément de flot produit avec parallélisme industrieux le sera à l'aide d'un nouveau processus dont le coût s'ajoute à celui de la structure de donnée elle même (création de la cellule et récupération mémoire).

4.2 Critique du paradigme

4.2.1 Flots paresseux

Nous savons que la mise en série de plusieurs traitements reliés par des flots paresseux est à éviter pour des raisons de performances car l'introduction des flots intermédiaires a un coût que rien ne vient compenser (aucun parallélisme n'est introduit) *[WADLER 84]*.

Ainsi un pipe-line constitué par la mise en série de divers traitements reliés paresseusement peut avantageusement être remplacé par un seul traitement correspondant à la composition fonctionnelle des divers traitements du pipe-line initial.

4.2.2 Flots industrieux

Nous venons de voir que la mise en œuvre de programmation pipe-line à l'aide de flots industrieux nécessite un nouveau processus pour chaque élément de chaque flot.

Ces processus peuvent être également utilisés pour évaluer l'élément du flot contenu par la cellule si la primitive cons utilisée est stricte en son pre-

1. Le choix entre ces trois primitives se fait selon que l'on veut faire exécuter le traitement unitaire sur les éléments du flot par le producteur (ep-cons), par le consommateur (lp-cons) ou par un nouveau processus parallèle (pp-cons).

2. Les différences entre el-cons, ll-cons et pl-cons sont de même nature qu'entre ep-cons, lp-cons et pp-cons.

mier argument (primitive ep-cons). Dans ce cas, chaque processus produira une cellule cons et un élément du flot.

Cependant, la granularité des processus générés par une programmation pipe-line industrieuse risque d'être très faible à moins que le calcul de l'élément ne représente une tâche importante.

En fait, le parallélisme ne provenant que de la décomposition et du partage en traitements successifs (avec introduction de structures de données intermédiaires) de traitements composés, l'équilibre entre nombre de processus générés et granularité est très délicat : un petit nombre de processus générés signifie un parallélisme potentiel faible, un plus grand nombre de processus introduit plus de flots et diminue la granularité, ce qui diminue d'autant le rendement de ces processus.

Nous touchons là une des difficultés majeures rencontrée dans la mise en œuvre efficace du paradigme pipe-line : la décomposition des traitements conduit à un difficile équilibre entre nombre de processus et granularité, ce qui peut nécessiter une longue phase de mise au point.

L'impossibilité de mettre en œuvre plus d'un flot en sortie d'un traitement sans construction supplémentaire accroît encore le coût de mise en œuvre du paradigme.

Il en résulte que selon l'importance du coût associé à la gestion d'un processus et la granularité minimum accessible correspondante, le parallélisme pipe-line sera plus ou moins difficilement exploitable. Notamment, tous les exemples d'école dans lesquels un traitement parallèle est introduit pour un traitement très simple (par exemple incrémenter les éléments d'un flot) constituent une utilisation non réaliste du paradigme car le même calcul réalisé par un seul traitement ne peut être que plus rapide.

Afin de rendre le paradigme pipe-line réellement utilisable, nous devrons abaisser le coût de sa mise en œuvre de manière à étendre sa portée à une granularité plus fine.

4.3 Un multi pipe-line à coût minimum

Nous allons voir comment notre modèle de réduction et ses principes de synchronisation des processus sur les noeuds mémoire permet de produire autant de flots industrieux que désiré avec un seul processus.

4.3.1 Origine du coût

Parmi les différents coût associés à l'introduction du paradigme pipe-line, deux sont à considérer particulièrement : la gestion de la structure de données intermédiaire (allocation des cellules, récupération mémoire) et la gestion des processus nécessaires à la production de cette structure.

Selon le coût de gestion des processus, l'importance relative de ces deux charges de travail peut varier, mais pour aussi faible que soit le coût d'un processus celui-ci est prépondérant par rapport à une allocation / récupération de cellule.

Dans notre modèle, la récupération mémoire est faite par comptage de références et parallèlement à l'exécution, ce qui rend les flots entre traitements d'autant moins coûteux (il n'existe qu'un coût résiduel provenant de l'occupation mémoire et réseau). La gestion des processus, bien que particulièrement optimisée reste donc la charge principale qui limite la granularité accessible à un traitement pipe-line[1].

Un flot industrieux est construit en une seule étape, la première fois qu'un processus le touche sa construction commence et ne s'arrête qu'après sa production complète. Les différents processus mis en œuvre ne sont justifiés que par la nécessité de retourner des parties de flot au consommateur avant la fin de sa construction : sur les six primitives `cons` conduisant à une construction industrieuse du flot, celles qui sont strictes sur leur second argument (`ee-cons`, `le-cons` et `pe-cons`) ne coûtent aucun processus mais ne retournent aucun résultat, alors que celles qui sont non strictes (`ep-cons`, `lp-cons` et `pp-cons`) coûtent un processus mais retournent un élément du flot.

Pour des flots industrieux, les multiples processus sont donc justifiés par la nécessité de retourner les éléments du flot au fur et à mesure de leur production, ce qui permet au producteur et au consommateur d'être actifs en parallèle.

Cette constatation offre une première solution au problème de granularité : par dépliage du producteur et utilisation alternée de `cons` stricts et non stricts il est possible d'augmenter la granularité, comme illustré dans l'exemple qui suit.

4.3.2 Gain de performances par augmentation de la granularité : Exemple

En partant de la définition de `transformeur` qui applique industrieusement la fonction `f` à chaque élément de son flot d'entrée `E` (voir 4.1.2.b pour la dérivation de `transformeur`) :

```
transformeur E

= p-reduce (lambda (x r) (cons (f x) r)) E nil
```

1. Ce coût va en fait provenir de deux phénomènes distincts : le coût de création du nouveau processus proprement dit qui est très faible bien que réel, et la latence de cette création qui fait que le début du calcul parallèle a lieu plus tard que s'il était fait en séquentiel. Pour les petits traitements, ce retard peut facilement être du même ordre de grandeur que la durée du calcul, ce qui ralentit le calcul malgré l'introduction de parallélisme.

;; par dépliage de la définition de p-reduce

```
= if (null E)
     nil
     (cons (f (first E)) (processus (transformeur (rest E))))
```

;; en déplaçant le processus à l'intérieur du cons

```
= if (null E) nil (lp-cons (f (first E)) (transformeur (rest E)))
```

;; en remplaçant lp-cons par ep-cons pour augmenter la granularité

```
= if (null E) nil (ep-cons (f (first E)) (transformeur (rest E)))
```

;; par dépliage de transformeur et distribution sur la conditionnelle

```
= if (null E) nil
     (if (null (rest E)) (ep-cons (f (first E)) nil)
         (ep-cons (f (first E))
                  (ep-cons (f (second E))
                           (transformeur (rest (rest E))))))
```

;; par utilisation alternée de ee-cons (équivalent strict en son second argument du ep-cons initialement utilisé) nous obtenons finalement

```
transformeur2 E =

if (null E) nil
   (if (null (rest E)) (ee-cons (f (first E)) nil)
       (ee-cons (f (first E))
                (ep-cons (f (second E))
                         (transformeur2 (rest (rest E))))))
```

Examinons maintenant le fonctionnement de nos deux versions de transformeur sur un flot dont les deux premiers éléments sont e1 et e2 :

```
transformeur (cons e1 (cons e2 E)) =

((lambda (E)
    (if (null E) nil
        (if (null (rest E)) (ep-cons (f (first E)) nil)
            (ep-cons (f (first E))
                     (ep-cons (f (second E))
                              (transformeur (rest (rest E)))))))
 (cons e1 (cons e2 E)))

= (ep-cons (f e1)
           (ep-cons (f e2)
                    (transformeur E)))
```

L'application de transformeur aux deux premiers éléments du flot a nécessité deux processus.

```
transformeur2 (cons e1 (cons e2 E)) =

((lambda (E)
    (if (null E) nil
```

```
        (if (null (rest E)) (ee-cons (f (first E)) nil)
            (ee-cons (f (first E))
                     (ep-cons (f (second E))
                              (transformeur (rest (rest E)))))))
 (cons e1 (cons e2 E)))

= (ee-cons (f e1)
           (ep-cons (f e2)
                    (transformeur E)))
```

L'application de `transformeur2` aux deux premiers éléments du flot a nécessitée un processus.

Il en résulte que `transformeur` utilise un processus par élément traité alors que `transformeur2` traite deux éléments par processus mis en œuvre. Cette technique peut bien entendu être utilisée pour porter ce rapport à un processus pour `i` éléments en réalisant autant de dépliages que nécessaire.

Toutefois, il est illusoire de vouloir régler le problème posé par la granularité à l'aide d'une utilisation intensive de cette technique car celle-ci conduit à une production par paquets des éléments du flot et peut donc conduire à une diminution de l'effet pipe-line par blocage du consommateur en attente du prochain paquet.

Ce mécanisme poussé à l'extrême correspond en fait à la production du flot de façon stricte, ce qui interdit l'effet pipe-line car le producteur doit alors terminer la production du flot avant que le consommateur puisse commencer son travail.

4.4 Concilier évaluation stricte et effet pipe-line

Nous venons de voir que la panoplie de primitives `cons` (correspondant aux trois stratégies d'évaluation : par nécessité, par valeur parallèle et par valeur séquentielle) permet un compromis entre granularité et effet pipe-line dans la production industrieuse des flots.

Toutefois, il serait préférable de disposer des éléments du flot un à un (comme le permet l'évaluation paresseuse parallèle) tout en n'utilisant qu'un seul processus (comme le permet l'évaluation par valeur séquentielle).

Nous allons voir un principe de communication entre traitements qui permet de concilier ces deux propriétés.

4.4.1 Le principe

Pour nous permettre de concilier une production stricte du flot et sa consommation classique nous allons être amenés à introduire de nouveaux opérateurs et à enrichir notre système de types.

Le flot auquel accédera le consommateur sera une liste, exactement comme dans le cas d'une production classique, la synchronisation lors des accès se fera donc sur les cellules *en réduction* de la liste (cf. 1.6.1 page 37).

Pour le producteur nous allons introduire deux niveaux de types supplémentaires qui lui permettrons de travailler sur un stream (élément d'un type (Stream τ)) auquel est associé la liste produite et d'être lui même d'un type producteur (élément d'un type (Prod τ)). En travaillant sur un stream le producteur construira la liste associée à ce stream, liste à laquelle le consommateur accédera normalement.

4.4.2 Les opérateurs de construction de streams

Pour assurer la communication entre producteur et consommateur, nous aurons besoin d'un opérateur pour introduire les streams (et leurs listes associées), d'un opérateur pour ajouter un élément à la liste associée à un stream et d'un opérateur pour clôturer la communication établie à travers un stream.

4.4.2.a L'opérateur make-stream

L'expression `(make-stream (`s_1 ... s_n`) p c)` crée les nouveaux streams s_1 à s_n utilisés par le producteur `p` qui est un nouveau processus et est en charge de produire les éléments successifs des listes associées aux streams, ceux-ci étant consommés par `c`. Si aucune ressource processus n'est disponible, le processus initial réalise production et consommation en séquence.

Notons la similitude syntaxique et la proximité sémantique entre `make-stream` et `p-multiple-value-bind` (cf. 2.2 page 50) qui lance un nouveau processus pour produire la valeur multiple que consommera le processus père.

4.4.2.b L'opérateur add-stream

Les trois expressions `(l-add-stream s e k)`, `(e-add-stream s e k)` et `(p-add-stream s e k)` ajoutent l'élément `e` au flot associé au stream `s` et continuent en évaluant `k`.

La différence entre les trois opérateurs est similaire à celle qui existe entre les différentes versions de l'opérateur `cons` : `l-add-stream` (également appelé `add-stream`) conduit à une évaluation par nécessité (*Lazy evaluation*) de l'élément ajouté, `e-add-stream` conduit à une évaluation par valeur séquentielle (*Eager evaluation*) et `p-add-stream` conduit à une évaluation par valeur parallèle (*Parallel evaluation*).

4.4.2.c L'opérateur close-stream

Les trois expressions `(l-close-stream s e k)`, `(e-close-stream s e k)` et `(p-close-stream s e k)` clôturent le stream `s` (clôture la communication établie à travers `s`), c'est à dire terminent la liste associée à `s` par la liste `e` puis évalue l'expression `k`. Une fois appliqué `close-stream` sur un stream, il n'est plus possible d'y appliquer `add-stream` ou `close-stream`.

4.4.3 Interprétation en terme de listes

L'interprétation que nous pouvons donner des opérateurs de manipulation de streams d'après leurs effets sur les listes associées est assez simple. Pour cela il suffit de considérer ces listes comme des approximations de la liste complète telle que la construira le producteur, et d'exprimer l'action des opérateurs comme des raffinages de ces approximations.

Par exemple les listes $\perp$, $(0 \cdot \perp)$, $(0\ 1 \cdot \perp)$, ... peuvent être interprétées comme des approximations successives de précision croissante de la liste des entiers naturels.

Nous pouvons trouver une utilisation de cette notion dans *[BURTON 91]* qui l'utilise pour dériver des parallélisations performantes d'algorithmes séquentiels dur tel que alpha-béta.

4.4.3.a Interprétation de make-stream

La liste telle que peut la voir le consommateur après sa création par `make-stream` est $\perp$ qui en constitue l'approximation initiale. Notons que `make-stream` lance un nouveau processus producteur, le processus père jouant le rôle du consommateur.

4.4.3.b Interprétation de add-stream

L'exécution de `(add-stream s e k)` raffine l'approximation de `s` qui passe de $(e_1 ... e_n \cdot \perp)$ à $(e_1 ... e_n\ e \cdot \perp)$.

4.4.3.c Interprétation de close-stream

L'exécution de `(close-stream s e k)` permet de passer d'une approximation $(e_1 ... e_n \cdot \perp)$ à la valeur terminale de la liste $(e_1 ... e_n \cdot e)$.

4.4.4 Premier exemple d'utilisation d'un stream

Nous allons décrire un exemple très simple : un transformeur qui ajoute `1` aux éléments de son flot d'entrée.

```
add-1 e
= make-stream (s) (add-1-suite e s) s

add-1-suite e s
= if (null e)
     (close-stream s nil t)
     (e-add-stream s (1+ (first e)) (add-1-suite (rest e) s))
```

Cette écriture s'apparente à un style par continuation (Continuation Passing Style *[WAND 80]*) et est utilisable par tout programmeur familiarisé avec la notion de continuation.

4.4.5 Le système de types

Pour garantir une sémantique déterministe aux expressions utilisant nos opérateurs, nous allons utiliser un typage à trois niveaux, similaire à celui introduit pour les blocs impératifs (cf. 3.4.3 page 91).

- Types fonctionnels (ou applicatifs) :

  $$\tau ::= \Upsilon \mid (\forall \nu : \Upsilon \cdot \tau)$$
  $$\Upsilon ::= \beta \mid \nu \mid (\Upsilon_1 \times \Upsilon_2) \mid (\Upsilon_1 \to \Upsilon_2)$$

  Correspondent aux types fonctionnels classiques, β étant un type primitif quelconque et ν une variable de type prenant valeur dans Υ.

- Types streams :

  $\sigma ::= \tau \mid (\text{Stream}\,\tau) \mid \text{Closed} \mid (\sigma_1 \times \sigma_2) \mid (\sigma_1 \to \sigma_2)$

  Etendent le niveau applicatif aux streams. Un élément de type $(\text{Stream}\,\tau)$ est un stream non clôturé dont les éléments sont de type τ, un élément de type Closed est un stream clôturé.

- Types producteurs :

  $\rho ::= \sigma \mid (\text{Prod}\,\tau) \mid (\rho_1 \times \rho_2) \mid (\rho_1 \to \rho_2)$

  Types des producteurs de streams.

Les trois niveaux de types ainsi définis ne sont pas indépendants mais ils sont au contraire imbriqués selon le schéma $\rho \supseteq \sigma \supseteq \tau$: tout type fonctionnel peut être considéré comme stream ou producteur, tout type stream peut être considéré comme producteur. A l'inverse l'appartenance d'un élément du type producteur au type fonctionnel sera conditionnelle (cf. T.9 page 128). Les constructeurs de type $\to$ et $\times$ sont les mêmes au trois niveaux.

Nous manipulerons des environnements constitués d'assertions $x:\pi$, "la variable x est de type π". L'environnement $\Gamma + \Gamma'$ représente une concaténation et $\Gamma, x:\pi$ un enrichissement. Les environnements ne sont pas des ensembles, si plusieurs assertions $x:\pi$ sont faites sur une même variable seule la dernière doit être prise en compte. Ainsi dans $\Gamma + \Gamma'$ le type d'une variable ne sera cherché dans Γ que s'il n'est pas précisé par Γ', de la même façon le type de x dans $\Gamma, x:\pi$ est π indépendamment de Γ.

Le type des opérateurs primitifs est inchangé, un schéma de type tel que $\forall x \cdot T$ devant simplement être compris "quel que soit x de type producteur" et non plus "quel que soit x de type fonctionnel".

Si nous prenons l'exemple du IF, son type reste $\text{IF}: \forall \pi \cdot \text{Bool} \to \pi \to \pi \to \pi$, mais la variable π peut être instanciée avec tout type producteur (ce qui comprend les types streams et fonctionnels).

De la même façon que pour les types mutables et observeurs vus en 3.4.3, les seules expressions évaluables dans notre modèle restent les expressions du niveau fonctionnel. Une expression de type producteur ne pourra donc qu'être une sous-expression d'une expression fonctionnelle.

4.4.6 Typage des streams

4.4.6.a Règles d'Inférence

T.1 $\Gamma \vdash k : \tau$; si k objet primitif de type fonctionnel τ

T.2 $\Gamma, x : \pi \vdash x : \pi$

T.3 $\dfrac{\Gamma \vdash x : \pi}{\Gamma, y : \pi' \vdash x : \pi}$; (si $x \neq y$)

T.4 $\dfrac{\Gamma, x : \pi_1 \vdash e : \pi_2}{\Gamma \vdash (\lambda x : \pi_1 . e) : (\pi_1 \to \pi_2)}$

T.5 $\dfrac{(\Gamma \vdash f : (\pi_1 \to \pi_2)) \qquad (\Gamma \vdash e : \pi_1)}{\Gamma \vdash (f\ e) : \pi_2}$

T.6 $\dfrac{(\Gamma \vdash e_1 : \pi_1) \qquad (\Gamma \vdash e_2 : \pi_2)}{\Gamma \vdash (e_1, e_2) : \pi_1 \times \pi_2}$

T.7 $\dfrac{\Gamma \vdash e : \pi_1 \times \pi_2}{\Gamma \vdash e.i : \pi_i}$; (où $i \in \{1;2\}$)

$e.i$ représente la i^{ème} projection.

T.8 $\dfrac{\Gamma \vdash t : \tau}{\Gamma \vdash t : \text{Prod}\,\tau}$

T.9 $\dfrac{\Gamma \vdash t : \text{Prod}\,\tau}{\Gamma \vdash t : \tau}$; si Γ ne contient que des types fonctionnels

T.10 $\dfrac{(\Gamma, v_1 : \text{Stream}\,\tau_1, \ldots, v_n : \text{Stream}\,\tau_n \vdash p : \text{Prod}\,\tau) \qquad (\Gamma, v_1 : \text{List}\,\tau_1, \ldots, v_n : \text{List}\,\tau_n \vdash c : \text{Prod}\,\tau')}{\Gamma \vdash (\text{make-stream}\ (v_1 \ldots v_n)\ p\ c) : \text{Prod}\,\tau'}$

Remarquons que le type du producteur p est indifférent, seul compte le fait qu'il soit bien typé. De plus le constructeur List est le constructeur classique du niveau fonctionnel.

T.11 $\dfrac{(\Gamma \vdash s : \text{Stream}\,\tau_1) \qquad (\Gamma \vdash e : \tau_1) \qquad (\Gamma \vdash k : \text{Prod}\,\tau_2)}{\Gamma \vdash (\text{add-stream}\ s\ e\ k) : \text{Prod}\,\tau_2}$

Cette règle s'applique identiquement aux trois versions de l'opérateur.

$$\textbf{T.12}\quad \frac{(\Gamma \vdash s\colon \mathrm{Stream}\,\tau_1) \qquad (\Gamma \vdash e\colon \mathrm{List}\,\tau_1) \qquad (\Gamma, s\colon \mathrm{Closed} \vdash k\colon \mathrm{Prod}\,\tau_2)}{\Gamma \vdash (\text{close-stream } s\ e\ k)\colon \mathrm{Prod}\,\tau_2}$$

Cette règle s'applique identiquement aux trois versions de l'opérateur.

Dans le typage de l'expression `k`, le stream `s` est de type Closed, ce qui interdit de lui appliquer quelque opérateur que ce soit (il n'existe aucune primitive applicable à un argument de type Closed).

4.4.6.b Commentaires

Les règles T.1 à T.7 servent à typer les objets primitifs du niveau fonctionnel, les fonctions et les produits de façon tout à fait classique.

Les règles T.8 et T.9 permettent la coercition entre types fonctionnels et types producteurs. Dans le cas où la règle de coercition T.9 ne peut s'appliquer nous parlerons de type producteur dur. La règle T.8 indique simplement que toute expression fonctionnelle peut jouer le rôle d'un producteur, la règle T.9 indique que dans un contexte de typage fonctionnel une expression de type producteur est coercible en un type fonctionnel.

Les règles T.10 et suivantes permettent de typer nos nouveaux opérateurs.

La règle T.8 permet d'inférer un deuxième type pour toute expression de type fonctionnel. Toutefois cela ne pose pas de problème car la phase de typage à pour but d'inférer un type fonctionnel pour toute forme (expression de premier niveau autres que définition) du programme et cette règle n'est qu'un moyen pour y arriver : partout où une expression de type producteur est désirée une expression de type fonctionnel peut convenir.

Le lecteur attentif aura peut-être remarqué deux faits qui semblent incompatibles :

- Notre système de réduction de graphe ne permet d'évaluer que des expressions de type fonctionnel, donc pas de type producteur.
- La règle T.10 indique qu'une expression `(make-stream ...)` est de type producteur.

L'absence de contradiction provient du fait que toute expression `(make-stream ...)` de plus haut niveau, c'est à dire non incluse dans une autre, admet un typage dans un contexte ou toutes les assertions sont sur des types du niveau fonctionnel. Il en découle que la règle T.9 s'applique à cette expression pour lui permettre de recevoir un type fonctionnel.

4.4.7 Déterminisme

L'utilisation que nous faisons des streams soulève une interrogation : est-il possible de lancer un deuxième producteur pour un même stream?

Cette possibilité conduirait à la possibilité de réaliser un `merge` non déterministe, ce que nous voulons éviter.

Notre système de type interdit tout parallélisme sur une expression de type producteur dur (c'est à dire de type producteur non coercible en un type fonctionnel par la règle T.9).

De plus, dans le contexte de typage qui nous intéresse, il y a au moins un objet non fonctionnel : le stream à produire. Il en découle que dans ce contexte, tout objet producteur d'un stream quel qu'il soit (qui est donc de type producteur) est de type producteur dur puisque l'existence du stream dans le contexte de typage interdit d'utiliser la règle de coercition pour affecter au stream un type fonctionnel.

Nous en concluons donc que dans le contexte qui nous intéresse, les seules expressions de type fonctionnel, seules expressions sur lesquelles nous pouvons appliquer `processus`, sont celles qui ne contiennent aucune sous-expression de type producteur, ce qui nous préserve du non déterminisme évoqué plus haut.

4.4.8 Persistance des streams

Les règles d'inférence de type que nous venons de présenter permettent de garantir que tout opérateur est appliqué à des données de type approprié. Cependant, un problème reste en suspens : rien ne garantit qu'un producteur applique `close-stream` à chaque stream introduit par le `make-stream` auquel il appartient.

Considérons l'expression suivante :

```
(make-stream (s) (add-stream s 1 t) (sigma s))
```

Où `sigma` effectue la somme d'une liste de nombres. A l'aide de nos règles de typage, nous pouvons affecter les types suivants :

$$\left(\begin{array}{c} \textbf{make-stream} \quad (s) \\ (\textbf{add-stream}\ s\text{: (Stream Num)}\ 1\text{:Num}\ t\text{: (Prod Bool))}\text{: (Prod Bool)} \\ (\textbf{sigma}\text{: (List Num} \rightarrow \text{Num)}\ s\text{: (List Num))}\text{: (Prod Num)} \end{array}\right) \text{:Prod Num}$$

Cette expression est donc correctement typée, de type producteur. Ce type est coercible en Num par la règle T.9 si le contexte de typage Γ ne contient que des assertions sur des types fonctionnels (ce qui est par exemple vrai si cette expression est une forme du premier niveau).

Toutefois il est clair que cette expression ne produira jamais la moindre valeur car le producteur termine en laissant la liste lue par le consommateur à la valeur $(1 \cdot \perp)$ et que sigma $(1 \cdot \perp) = \perp$.

Ce type de non terminaison est particulier car il ne correspond pas à un calcul infini mais à une absence de calcul.

Pour détecter de telles situations, nous allons exprimer de nouvelles contraintes définissant la notion de *variation* d'une expression et permettant d'assurer qu'une expression (`make-stream` ...) termine si elle ne comporte pas une séquence de calcul infinie.

Pour permettre une vérification de cette propriété, nous allons donner un ensemble de règles d'inférence qui établissent la variation en terme de communications établies résultant de l'évaluation dans un certain contexte d'une expression bien typée.

Une variation négative correspondra alors à fermer des streams hérités de l'environnement, ce qui n'est possible que pour une expression de type producteur, une variation nulle correspondra pour une expression de type fonctionnel à refermer tous les streams ouverts, et une variation positive correspondra au fait d'avoir laissé ouvert un certain nombre de streams.

Le typage interdisant l'application de `close-stream` à un stream de type Closed, le fait qu'une expression ferme n streams dans un contexte qui en contient n nous permettra d'affirmer que tous les streams ont bien été refermés.

Par suite les seuls calculs infinis seront ceux qui comportent une séquence de calcul infini. De plus, et comme le mode d'évaluation de notre modèle est l'évaluation par nécessité, nous pourrons affirmer que tout calcul admettant une forme normale termine effectivement.

4.4.8.a Règles d'inférence de la variation

Dans les règles d'inférence qui suivent, nous interpréterons l'assertion $\Gamma \vdash e \gg \eta$ comme étant "la variation de e dans le contexte Γ est égale à η" ou encore "l'évaluation de e dans le contexte Γ entraîne une variation du nombre de streams égale à η".

Si nos règles d'inférence permettent d'établir pour une expression fonctionnelle $e:\tau$ que $e \gg 0$, alors l'expression e referme tous les streams qu'elle ouvre. Dans ce cas nous pouvons affirmer qu'aucun processus consommateur ne peut rester indéfiniment en attente d'un producteur qui a terminé sans refermer son stream.

V.1 $\Gamma \vdash f \gg \forall \nu_1 \ldots \nu_n \cdot \Phi(\{\nu_1 \ldots \nu_n\}, \tau)$; si f primitive de type fonctionnel $(\forall\ \nu_1 \ldots \nu_n \cdot \tau)$

où :

$$\Phi(\{\nu_1 \ldots \nu_n\}, \tau_1 \to \tau_2) = \Phi(\{\nu_1 \ldots \nu_n\}, \tau_1) \to \Phi(\{\nu_1 \ldots \nu_n\}, \tau_2)$$

$$\Phi(\{\nu_1 \ldots \nu_n\}, \tau_1 \times \tau_2) = \Phi(\{\nu_1 \ldots \nu_n\}, \tau_1) \times \Phi(\{\nu_1 \ldots \nu_n\}, \tau_2)$$

$$\Phi(\{\nu_1 \ldots \nu_n\}, \beta) = 0$$

$$\Phi(\{\nu_1 \ldots \nu_n\}, \nu) = \nu ; (\text{si } \nu \in \{\nu_1 \ldots \nu_n\})$$

V.2 $\Gamma \vdash \nu \gg 0$; si ν de type (Stream τ)

V.3 $\Gamma \vdash \nu \gg 0$; si ν de type Closed

V.4 $\Gamma, x \gg \eta \vdash x \gg \eta$

V.5 $\dfrac{\Gamma \vdash x \gg \eta}{\Gamma, y \gg \eta' \vdash x \gg \eta}$; (si $x \neq y$)

V.6 $\dfrac{\Gamma, x \gg \eta_1 \vdash e \gg \eta_2}{\Gamma \vdash (\lambda x \gg \eta_1 . e) \gg \eta_1 \to \eta_2}$

V.7 $\dfrac{(\Gamma \vdash f \gg \eta_1 \to \eta_2) \qquad (\Gamma \vdash e \gg \eta_1)}{\Gamma \vdash (f\ e) \gg \eta_2}$

V.8 $\dfrac{(\Gamma \vdash e_1 \gg \eta_1) \qquad (\Gamma \vdash e_2 \gg \eta_2)}{\Gamma \vdash (e_1, e_2) \gg \eta_1 \times \eta_2}$

V.9 $\dfrac{\Gamma \vdash e \gg \eta_1 \times \eta_2}{\Gamma \vdash e.i \gg \eta_i}$; (où $i \in \{1;2\}$)

$e.i$ représente la i$^{\text{ème}}$ projection.

V.10 $\dfrac{(\Gamma \vdash p \gg \eta_1) \qquad (\Gamma \vdash c \gg \eta_2)}{\Gamma \vdash (\text{make-stream } (v_1 \ldots v_n) p\ c) \gg \eta_1 + \eta_2 + n}$

V.11 $\dfrac{\Gamma \vdash k \gg \eta}{\Gamma \vdash (\text{add-stream } s\ e\ k) \gg \eta}$

Cette règle s'applique identiquement aux trois versions de l'opérateur.

V.12 $\dfrac{\Gamma \vdash k \gg \eta}{\Gamma \vdash (\text{close-stream } s\ e\ k) \gg \eta - 1}$

Cette règle s'applique identiquement aux trois versions de l'opérateur.

4.4.8.b Commentaires

La règle V.1 permet d'affecter une signature à toute constante de type fonctionnel. A l'aide de cette règle nous pouvons par exemple montrer qu'une expression de type β ne contenant que des opérateurs et constantes de types fonctionnels ne modifie pas le nombre de streams. Appliquée au cas de la primitive IF: $\forall \pi \cdot \text{Bool} \to \pi \to \pi \to \pi$ la règle V.1 permet d'inférer sa variation IF $\gg \forall \pi \cdot 0 \to \pi \to \pi \to \pi$. Il en découle le corollaire : "les deux branches d'un IF doivent avoir la même variation", ce qui est très raisonnable pour pouvoir inférer la variation totale des expressions.

Les règles V.2 et V.3 caractérisent les streams.

Les règles V.4 et V.5 correspondent à une recherche dans l'environnement.

La règle V.6 indique que si à partir des assertions de Γ en supposant que l'évaluation de x entraîne une variation η_1 il est possible déduire que l'évaluation de e entraîne une variation η_2, alors de Γ il est possible de déduire que la variation attachée à $(\lambda x.e)$ est $\eta_1 \rightarrow \eta_2$.

La règle V.7 caractérise les applications.

Les règles V.8 et V.9 traitent le cas des types produits.

La règle T.10 rend compte de l'introduction de nouveaux streams effectuée par une expression `make-stream`.

La règle V.11 indique que l'exécution d'un `add-stream` ne modifie pas le nombre de streams.

Enfin, la règle V.12 traduit la fermeture d'un stream consécutive à l'exécution d'un `close-stream`.

Reprenons maintenant notre exemple précédent et vérifions que nos règles nous permettent d'inférer une variation égale à 1, ce qui indique un blocage potentiel.

$$\left(\begin{array}{c} \textbf{make-stream} \quad (s) \\ (\textbf{add-stream}\ s \gg 0\ \ 1 \gg 0\ \ t \gg 0) \gg 0 \\ (\textbf{sigma} \gg (0 \rightarrow 0)\ \ s \gg 0) \gg 0 \end{array}\right) \gg 1$$

4.4.9 Propriétés

A l'intérieur d'une expression de type fonctionnel dont la variation est nulle, nos opérateurs garantissent la séquentialité des `add-stream` et l'application terminale de `close-stream`. Toute expression bien typée de variation nulle utilisant nos opérateurs sur les streams aura donc une sémantique clairement établie et conforme à l'intuition d'ajouts successifs à une liste.

Par exemple, les expressions suivantes sont mal typées :

a. `(close s nil (add-stream s e k))`

Dans `(add-stream s e k)`, `s` est de type Closed, ce qui interdit d'y appliquer `add-stream`.

b. `(+ (add-stream s a 1) (add-stream s b 3))`

Les deux expressions `(add-stream s a 1)` et `(add-stream s b 3)` sont de type Prodτ, ce qui interdit de les utiliser comme argument de + qui est de type Num $\rightarrow$ Num $\rightarrow$ Num

De plus nous pouvons détecter qu'une expression fonctionnelle ne referme pas tous les streams qu'elle ouvre ce qui constitue une possibilité de blocage d'un consommateur.

Comme nous l'avions établi pour les types mutables et observeurs (cf. 3.4.5.a page 98), l'application de toute construction pouvant amener une

modification de l'ordre d'évaluation doit être limité aux expressions fonctionnelles.

Cela comprend l'emploi des directives `eager` et `processus` (qui provoquent respectivement un appel par valeur séquentiel et parallèle et donc une anticipation de l'évaluation) et le déclenchement d'un processus à l'aide d'une primitive parallèle (effet identique à celui de `processus`). La limitation de ces directives et opérateurs aux expressions de type fonctionnel est garantie par leur typage.

De plus, la déclaration de stricticité (faite pour palier à l'absence d'analyse automatisée) qui est habituellement utilisée pour signaler au compilateur des opportunités d'appel par valeur n'a pas lieu d'être pour les variables de type non fonctionnel.

4.4.10 Réalisation

Nous allons maintenant voir comment la réduction de graphe et la synchronisation des accès à la mémoire (cf. 1.6 page 37) permettent une réalisation particulièrement simple, élégante et efficace de nos opérateurs.

Nous utiliserons des combinateurs déjà introduits pour les objets mutables et les observeurs (cf. 3.4.1.e page 86), nous utiliserons également `new-er` pour créer un nouveau noeud dans l'état *en réduction*. `add-stream` et `close-stream` qui ne peuvent s'exprimer simplement à l'aide d'autres opérateurs sont également des combinateurs de base de la machine.

`(new-er k)` crée un noeud dans l'état *en réduction*, et fait l'application entre `k` et ce noeud. Toute lecture de ce noeud est bloquante (cf. 1.6.2.a page 37), seul un accès par une primitive qui ne fait que récrire le noeud peut se faire sans blocage.

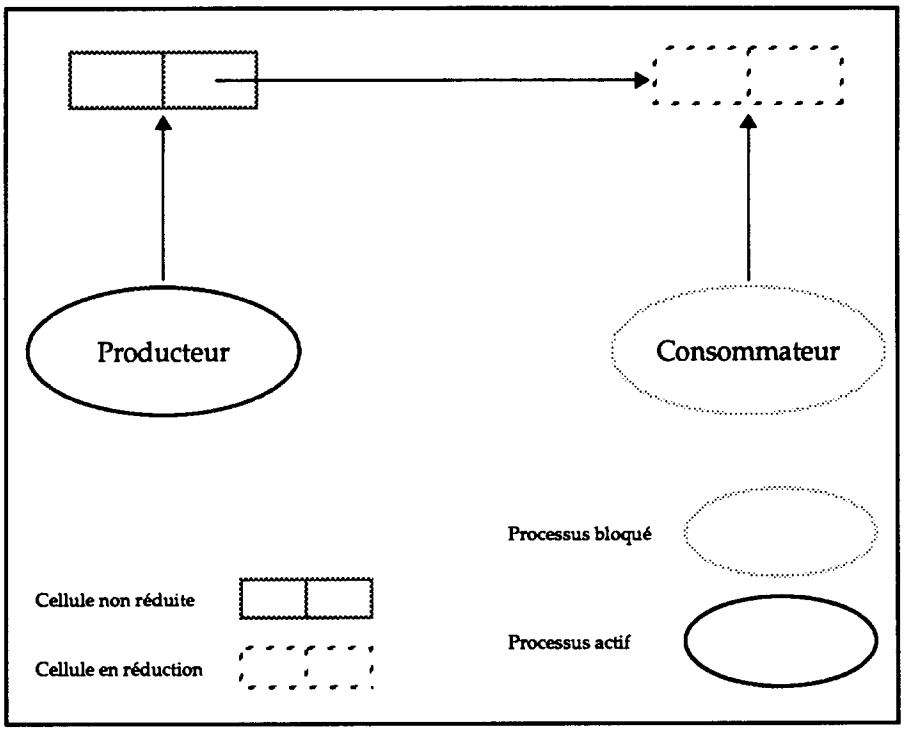
Le consommateur qui ne voit du stream qu'une simple liste, comme le garantit le typage de `make-stream` (cf. T.10 page 128), et y accède par `first` et `rest` se retrouve bloqué chaque fois qu'il accède à un noeud crée par `new-er` non encore récrit par le producteur.

Le producteur ne voit pas la liste qui le relie au consommateur mais voit le stream auquel elle est associée. Ce stream est tout simplement une référence au dernier `rest` (qui se trouve *en réduction*) du flot, cette référence permettant de récrire le `rest` en un nouveau `cons` (dont le `rest` sera également *en réduction*) et ainsi de rajouter un élément au flot.

Ainsi le producteur se contentera de conserver une référence au noeud `en réduction` à récrire, chaque récriture modifiant la location pointée par cette référence.

La synchronisation entre producteur et consommateur lors d'un accès par ce dernier à la fin du flot (cf. Figure 6) se fait grâce au mécanisme standard de synchronisation sur les noeuds de la mémoire (cf. 1.6.2.b page 38).

Figure 6 Consommateur bloqué en attente du producteur



Si le producteur et le consommateur accèdent à des cellules distinctes, ils n'interagissent pas et peuvent être actifs en parallèle (cf. Figure 7).

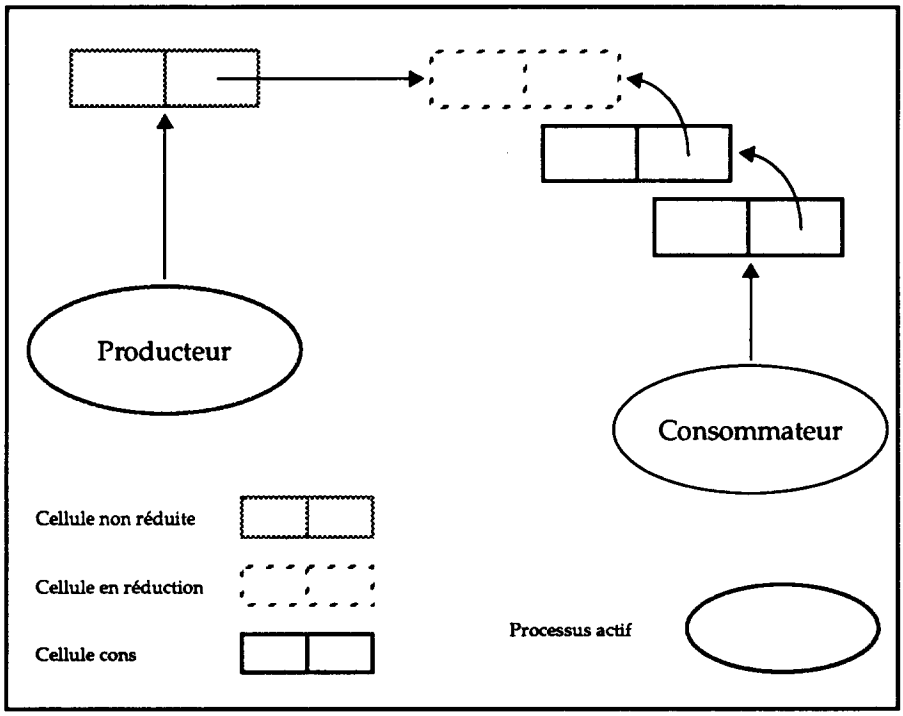
4.4.10.a Réalisation de make-stream

La macro `make-stream` introduit de nouveaux streams qui relient le producteur (qui est un nouveau processus) au consommateur.

Rappelons que d'après le typage de `make-stream` (cf. T.10 page 128), chaque communication que `make-stream` va établir est appréhendée par le producteur à travers un stream (élément d'un type Stream τ c'est à dire un stream dont les éléments sont du type fonctionnel τ), alors qu'elle est appréhendée par le consommateur à travers la liste associée au stream (qui est de type List τ c'est à dire une liste dont les éléments sont du même type τ).

Pour chaque communication, `make-stream` crée deux nouveaux noeuds. Le premier est créé par `new-er` et est donc *en-réduction*, il représente la liste $\perp$ qui est une première approximation du flot que voit le consommateur. Le second est créé par `new-app` et est donc *non-réduit*, il constitue le stream et

Figure 7 Producteur et consommateur actifs



référence l'adresse de fin provisoire de la liste que le producteur va manipuler pour la compléter. Si le consommateur accède à la fin provisoire de la liste (un noeud *en réduction* que le producteur n'a pas encore récrit), le mécanisme classique de synchronisation sur les noeuds du graphe (cf. 1.6.2.a page 37) va provoquer son blocage jusqu'à la récriture du noeud (par le producteur).

Lorsque toutes les communications sont établies, `make-stream` lance en parallèle le producteur avec le consommateur.

```
(make-stream (v1 v2 … vn) p c)
= (new-er (lambda (w1)
    (new-app !I w1
             (lambda (w2)
               (make-stream (v2 … vn) p[w2/v1] c[w1/v1])))))

(make-stream () p c)
= (pe-prog2 p c))
```

Où la notation `E[e/x]` représente classiquement l'expression `E` dans laquelle toute occurrence libre de `x` à été remplacée par `e`.

4.4.10.b Réalisation de add-stream

Les primitives de la famille de `add-stream` complètent la fin du flot associé à un stream et poursuivent en évaluant une expression qui représente la suite des calculs.

Dans l'expression `(add-stream s e k)`, `s` qui est le stream contient l'adresse de la fin provisoire de la liste, `e` est l'expression qui sera rajoutée à la suite dans la liste et `k` est l'expression qui sera évaluée après le rajout de `e`.

Le rajout de `e` à la suite de la liste se passe en trois étapes (cf. Figure 8). D'abord un nouveau noeud *en réduction* est créé qui sera la nouvelle fin provisoire de liste après le rajout, l'adresse de ce nouveau noeud remplacera alors le contenu de `s`. Ensuite la fin provisoire de la liste (dont l'adresse est dans `s`) et qui est un simple noeud *en réduction* est récrit à l'état `cons` avec `e` comme partie `first` et le noeud précédemment créé comme partie `rest`. Enfin la mise à jour de `s` est effectuée.

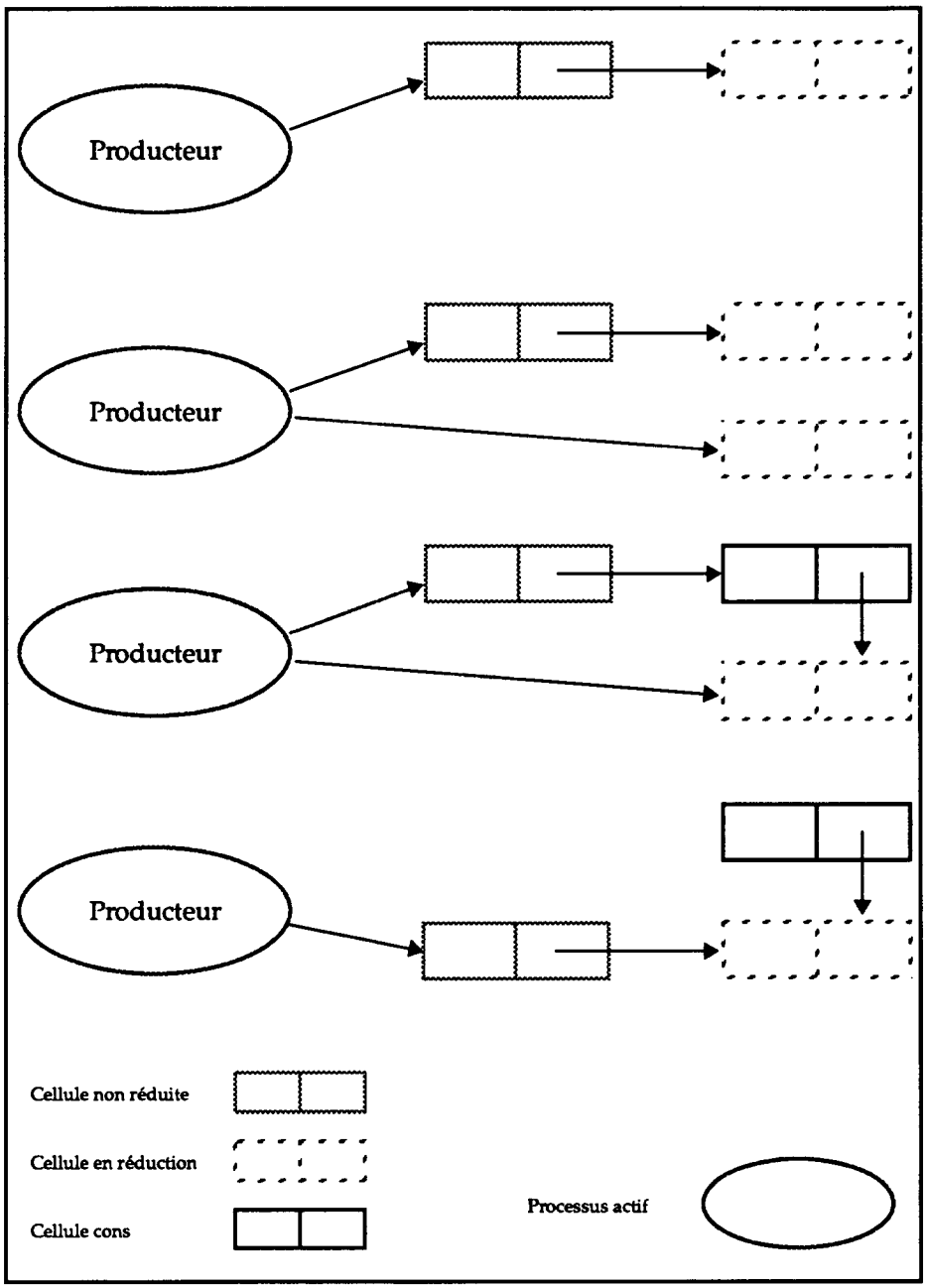
Si un consommateur était bloqué sur la fin provisoire de liste, la récriture de celle-ci en tant qu'un `cons` entre `e` et un nouveau noeud *en réduction* le débloque, son reblocage intervenant éventuellement dès la première fois où il touche une autre fin provisoire de liste.

4.4.10.c Réalisation de close-stream

Les primitives de la famille `close-stream` terminent la liste associée à un stream.

L'expression `(close-stream s e k)` clôture une communication, c'est à dire récrit la fin de la liste associée au stream `s` en `e`, puis évalue `k`. Le rajout de `e` en fin de liste est effectué en récrivant le noeud `(!I e)` comme fin de liste, `!I` étant le combinateur identité. L'expression `k` ne doit pas utiliser le stream car il est clôturé, cette propriété est garantie par le typage de `close-stream` (cf. T.12 page 128).

Figure 8 Les trois étapes du producteur



4.4.11 Exemple d'utilisation des streams : le crible d'Eratosthène

Comme exemple de traitement pipe-line, nous allons considérer le crible d'Eratosthène. Cet algorithme calcule la liste des nombres premiers en éli-

minant successivement de la liste des nombres entiers tous ceux qui sont les multiples de 2, de 3, de 5, ... et de tous les nombres premiers jusqu'à la racine carrée du nombre maximum considéré.

L'effet pipe-line est mis en œuvre à plusieurs niveaux, d'abord entre la production de la liste initiale des entiers et le filtrage des nombres composés, ensuite à l'intérieur même du filtrage qui est réalisé par autant d'instances du processus de filtrage qu'il y a de nombre premiers à considérer.

Malgré le nombre important de traitements parallèles, le potentiel de parallélisme reste assez faible car les processus successifs du pipe-line travaillent sur des listes de nombres de longueur décroissante. Ainsi la production des entiers et le filtrage des multiples de 2 se font sur n nombres, le filtrage des multiples de 3 sur n/2 nombres, le filtrage des multiples de 5 sur n/3, ... ce qui fait que les processus en bout de chaîne passent la majorité de leur temps à attendre qu'un nombre ait franchi les filtres précédents pour pouvoir le traiter.

Cet algorithme est par contre très intéressant pour notre comparaison car il met en œuvre de nombreux étages de pipe-line.

Suivent deux versions du programme, la première est la version parallèle optimisée extraite de *[MARRE 92]* qui utilise des listes entre étages du pipeline, la seconde en est directement dérivée en utilisant une communication par des streams.

4.4.11.a Le programme à base de listes

Nous reproduisons Figure 9 le listing du programme et Figure 10 les principaux résultats de simulation sur une configuration à 32 processus de `(primes 10000 100)` qui calcule la liste des nombres premiers inférieurs à 10000.

Comme attendu le taux d'attente est assez élevé et le taux de parallélisme limité. La granularité des différents processus est très faible puisque 43908 processus ont été nécessaires à l'évaluation des 338873 primitives exécutées, ce qui fait en moyenne moins de 8 primitives par processus.

4.4.11.b Le programme à base de streams

Nous reproduisons Figure 11 le listing du programme et Figure 12 les principaux résultats de simulation sur une configuration à 32 processus de `(primes 10000 100)` qui calcule la liste des nombres premiers inférieurs à 10000.

Bien que restant assez défavorable, le rapport entre processus actifs et processus en attente s'est largement amélioré, de plus chaque processus a exécuté en moyenne plus de 15700 primitives ce qui est une granularité très raisonnable.

4.4.11.c Analyse

L'effet principal attendu est bien présent, à savoir une très nette diminution du nombre de processus mis en œuvre (28 au lieu de 43908), ainsi et malgré un nombre de primitives exécutées supérieur de 30% et grâce à une considé-

Figure 9 Le crible d'Eratosthène à base de listes

```
(defun primes (n racn)
  (sieve (processus (entiers-entre 2 n)) racn))

(defun entiers-entre (m n)
  (if (> m n)
      nil
      (ep-cons m (entiers-entre (1+ m) n))))

(defun sieve (Lentiers racn)
  (if (null Lentiers)
      nil
      (if (> (first Lentiers) racn)
          Lentiers
          (ep-cons (first Lentiers)
                   (sieve (filter (rest Lentiers)
                                  (first Lentiers))
                          racn)))))

(defun filter (Lentiers p)
  (reduce (lambda (n filtre-suite)
            (if (= 0 (rem n p))
                filtre-suite
                (ep-cons n filtre-suite)))
          Lentiers
          nil))
```

rable augmentation de la granularité, le temps de calcul diminue de 508888 à 280210 cycles, soit un gain de 45%.

Les 28 processus mis en œuvre dans la version avec streams correspondent à 1 processus initial + 2 processus lancés par la fonction primes + 25 processus de filtrage correspondants au 25 nombres premiers inférieurs à 100.

4.4.12 Mise en œuvre d'un multi pipe-line

Un des objectifs affichés pour les streams était de permettre à un traitement d'avoir directement plus d'un flot en sortie. Nous allons donc voir comment réaliser cela sur un exemple très simple, la partition d'un flot de nombres en deux flots par rapport à un pivot. Ce traitement sera mis en œuvre dans l'algorithme *quicksort*.

4.4.12.a Le programme

Nous donnons Figure 13 la version complète du programme due à *[HALSTEAD 85]* transposée en MaRS_Lisp. La fonction `partition` y réalise la partition de `liste` en deux listes : celle des nombres inférieurs à `pivot` et celle des nombres supérieurs à `pivot`. La simulation des deux flots en sortie y est

Figure 10 Résultats d'exécution du crible d'Eratosthène à base de listes

```
(primes 10000 100)
=> (2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 ...

43908 processus, 508888 cycles, parallelisme = 2.54,
inactivite = 17.76, 338873 primitives executees.

     0 ---------------@@@              |
 16384 -------------------@@@          |
 32768 -------------------@@@          |
 49152 -------------------@@@          |
 65536 -------------------@@@          |
 81920 -------------------@@@          |
 98304 -------------------@@@          |
114688 -------------------@@@          |
131072 -------------------@@@          |
147456 -------------------@@           |
163840 -------------------@@@          |
180224 -------------------@@@          |
196608 -------------------@@@          |
212992 -------------------@@           |
229376 -------------------@@@          |
245760 -------------------@@@          |
262144 -------------------@@           |
278528 -------------------@@@          |
294912 -------------------@@@          |
311296 -------------------@@@          |
327680 -------------------@@           |
344064 -------------------@@@          |
360448 -------------------@@           |
376832 -------------------@@@          |
393216 -------------------@@           |
409600 -------------------@@@          |
425984 -------------------@@@          |
442368 -------------------@@@          |
458752 -------------------@@@          |
475136 -------------------@@@          |
491520 -------------------@@@          |
```

réalisée par la technique de la *structure de flots* et la structure de `partition` nous y place dans le cas `Un` (cf. 4.1.3.b page 118).

La production industrieuse des deux flots `inf` et `sup` est assurée par l'utilisation de la primitive parallèle `p-reduce` à l'intérieur de la fonction `partition`. L'autre source de parallélisme du programme est l'utilisation de `p-multiple-value-bind` à l'intérieur de `qs`, ce qui à pour effet de lancer en parallèle le producteur et le consommateur de la valeur multiple.

Figure 11 Le crible d'Eratosthène à base de streams

```
(defun primes (n racn)
  (make-stream (l-primes)
    (make-stream (entiers)
      (entiers-entre 2 n entiers)
      (sieve entiers racn l-primes))
    l-primes))

(defun entiers-entre (n fin s-entiers)
  (if (> n fin)
      (close-stream s-entiers nil t)
      (e-add-stream s-entiers n
                    (entiers-entre (1+ n) fin s-entiers))))

(defun sieve (Lentiers racn s-primes)
  (if (null Lentiers)
      (close-stream s-primes nil t)
      (if (> (first Lentiers) racn)
          (close-stream s-primes Lentiers t)
          (e-add-stream s-primes (first Lentiers)
                        (make-stream (entiers-filtres)
                          (filter (rest Lentiers) (first Lentiers)
                                  entiers-filtres)
                          (sieve entiers-filtres racn
                                 s-primes))))))

(defun filter (Lentiers p s-entiers-filtres)
  (reduce (lambda (n filtre-suite)
            (if (= 0 (rem n p))
                filtre-suite
                (e-add-stream s-entiers-filtres n filtre-suite)))
          Lentiers
          (close-stream s-entiers-filtres nil t)))
```

Nous donnons Figure 14 les définitions équivalentes basées sur les streams, nous pouvons remarquer la construction directe des deux listes `inf` et `sup` par la fonction `partition`.

La production industrieuse des deux listes est due à l'utilisation de streams à l'intérieur de la fonction `partition`. L'autre source de parallélisme du programme est l'utilisation de `make-stream` à l'intérieur de `qs`, ce qui à pour effet de lancer en parallèle le producteur et le consommateur des deux listes associées à `inf` et `sup`.

4.4.12.b Résultats de simulation

Nous avons simulé les deux versions du programme sur une liste de nombres compris entre `0` et `199` générée aléatoirement et de longueur `1000`.

Figure 12 Résultats d'exécution du crible d'Eratosthène à base de streams

```
(primes 10000 100)
=> (2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 …

28 processus, 280210 cycles, parallelisme = 5.72,
inactivite = 19.40, 440340 primitives executees.

     0 --------------@@@@@@           |
  8192 --------------------@@@@@@     |
 16384 --------------------@@@@@@     |
 24576 --------------------@@@@@@     |
 32768 --------------------@@@@@@     |
 40960 --------------------@@@@@@     |
 49152 --------------------@@@@@@     |
 57344 --------------------@@@@@@     |
 65536 --------------------@@@@@@     |
 73728 --------------------@@@@@@     |
 81920 --------------------@@@@@@     |
 90112 --------------------@@@@@@     |
 98304 --------------------@@@@@@     |
106496 --------------------@@@@@@     |
114688 --------------------@@@@@@     |
122880 --------------------@@@@@@     |
131072 --------------------@@@@@@     |
139264 --------------------@@@@@@     |
147456 --------------------@@@@@@     |
155648 --------------------@@@@@@     |
163840 --------------------@@@@@@     |
172032 ----------------------@@@@@     |
180224 --------------------@@@@@@     |
188416 --------------------@@@@@@     |
196608 --------------------@@@@@@     |
204800 --------------------@@@@@@     |
212992 --------------------@@@@@@     |
221184 --------------------@@@@@@     |
229376 --------------------@@@@@@     |
237568 --------------------@@@@@@     |
245760 --------------------@@@@@@     |
253952 --------------------@@@@@@     |
262144 --------------------@@@@@@     |
270336 @@@@@@@                        |
```

les résultats de simulation sont reproduits ci-dessous et n'appellent aucun commentaire particulier.

Version à base de listes :

```
5125 processus, 88940 cycles, parallelisme = 9.35,
inactivite = 22.16, 226101 primitives executees.
```

Figure 13 Le quicksort à base de listes

```
(defun qsort (liste) (qs liste nil))

(defun qs (liste fin)
  (if (null liste)
      fin
      (let ((pivot (first liste))
            (liste-a-partitionner (rest liste)))
        (p-multiple-value-bind (inf sup)
          (partition pivot liste-a-partitionner)
          (qs inf (processus (ee-cons pivot (qs sup fin))))))))

(defun partition (pivot liste)
  (p-reduce (lambda (nb partition-suite)
              (multiple-value-bind (inf sup)
                partition-suite
                (if (> nb pivot)
                    (values inf (cons nb sup))
                    (values (cons nb inf) sup)))
            liste
            (values nil nil)))
```

Figure 14 Le quicksort à base de streams

```
(defun qsort (liste) (qs liste nil))

(defun qs (x reste)
  (if (null x) reste
      (make-stream (inf sup)
        (partition (car x) (cdr x) inf sup)
        (qs inf (processus (ee-cons (car x) (qs sup reste)))))))

(defun partition (pivot liste inf sup)
  (reduce (lambda (nb partition-suite)
            (if (gt nb pivot)
                (e-add-stream sup nb partition-suite)
                (e-add-stream inf nb partition-suite)))
          liste
          (close-stream inf nil (close-stream sup nil t))))
```

Version à base de streams :

```
2200 processus, 32329 cycles, parallelisme = 14.25,
inactivite = 44.38, 121535 primitives executees.
```

Comme attendu la version à base de listes met en œuvre un nombre plus important de processus, exécute un plus grand nombre d'instruction (à cause de la simulation des deux flots en sortie de la fonction `partition`), et a finalement une durée 2,75 fois supérieure.

4.4.12.c Conclusion

Les résultats obtenus confirment le très grand intérêt des streams dans le cas de fonctions à plusieurs flots en sortie.

Non seulement ils permettent de réaliser la construction des flots de manière plus efficace car un seul processus est nécessaire à leur construction industrieuse, mais ils permettent une écriture directe qui une fois familière est bien plus claire qu'une écriture compliquée par la nécessité de simuler les multiples flots.

Ces résultats sont déjà excellents dans notre modèle qui, du fait du très faible coût de gestion des processus, est un des meilleurs candidats au parallélisme pipe-line classique, mais le gain serait encore supérieur dans un modèle à processus plus coûteux, comme en Multilisp par exemple.

4.5 Conclusion

Après avoir identifié le frein principal à une utilisation efficace du paradigme pipe-line, à savoir la faible granularité des très nombreux processus engendrés, nous avons pu proposer un enrichissement du système de types et des opérateurs permettant sa mise en œuvre à l'aide d'un petit nombre de processus à gros grain.

Notre solution, tout en garantissant un véritable effet pipe-line par production du flot élément par élément (et non par paquets), permet ainsi de s'affranchir du coût principal associé : la trop faible granularité et le trop grand nombre de processus générés.

Le système de type associé aux nouveaux opérateurs introduits et l'absence de blocage d'un consommateur par non clôture de stream garantie pour toute expression de type fonctionnel et de variation nulle garantit une sémantique claire correspondant à l'intuition qu'un programmeur peut avoir d'ajouts d'éléments à une liste.

Enfin, l'utilisation algorithmique du paradigme pipe-line n'est pas modifiée par notre approche qui se contente d'en augmenter le rendement parallèle, et donc son champ d'application, mais reste conforme au principe de base : un producteur et un consommateur qui travaillent en parallèle.

Notre mécanisme de streams pour le parallélisme pipe-line industrieux associé aux techniques de suppression des structures intermédiaires préconisées par *[WADLER 84]* pour les flots paresseux permet ainsi une utilisation optimale d'un des paradigmes de parallélisation les plus aisé à mettre en œuvre au niveau algorithmique.

Conclusion

Cette thèse fait suite à de nombreux autres travaux relatifs au projet MaRS, lequel a aboutit en 1991 à la réalisation d'un prototype opérationnel de machine parallèle. Elle se situe dans la perspective d'un nouveau projet bénéficiant des acquis du projet MaRS et dont les principes doivent être établis par une critique constructive. Ce nouveau projet doit déboucher sur une machine dont le champ d'application serait étendu à certains secteurs jusqu'à présent inaccessibles à l'algorithmique fonctionnelle, tels que le lancer de rayons ou la simulation d'écoulements pour lesquels la machine MaRS est déjà pleine de promesses.

Dans ce but, nous avons exposé les principes essentiels de la programmation fonctionnelle parallèle et de la réduction de graphe. Nous avons ensuite détaillé plus particulièrement notre modèle et ses particularités (processus à faible coût, processus non préemptible, processus en nombre limité, gestion mémoire en parallèle, ...) dont nous avons dégagé les préoccupations spécifiques du *bien paralléliser.*

En particulier nous avons fait ressortir la faible importance de ce qui est dans bien des modèles fonctionnels parallèles la préoccupation essentielle : la granularité des processus. A l'inverse de ces modèles, la limitation première de notre modèle en regard de la parallélisation s'avère être le taux d'activité des processus générés, ce qui nous a donc amené à isoler les causes principales d'inactivité des processus et à proposer de nouveaux mécanismes pour y remédier.

La première source de faible activité que nous envisageons est l'accès aux structures de données qui peut limiter fortement le potentiel parallèle des algorithmes les plus favorables. Pour remédier à la pauvreté initiale de notre langage en structures de données nous montrons alors comment inté-

grer les valeurs multiples, les structures et les vecteurs dans le cadre du lambda-calcul et nous en analysons la représentation combinatoire.

Nous analysons ensuite l'inactivité liée aux synchronisations inter-processus : certaines occurrences de processus qui ne peuvent être négligées pour atteindre un parallélisme important sont source d'inactivité lors des phases de calcul à fort parallélisme. Cela nous amène à proposer la notion de processus complémentaire dont le déclenchement conditionné à une faible occupation des ressources permet de simplifier la gestion de ces occurrences lors de la parallélisation.

Nous envisageons ensuite le cas plus particulier de processus coopérant à la construction de résultats et dont les contributions respectives commutent. Nous montrons alors que notre modèle impose des synchronisations inter-processus coûteuses et inutiles et proposons une nouvelle construction prenant ce parallélisme en compte mais affranchissant totalement les processus de synchronisation. Les gains très importants apportés par cette constructions sont alors montrés sur un exemple essentiel : l'algorithme de recherche idA*.

Nous étudions enfin un des paradigmes majeurs de l'algorithmique fonctionnelle parallèle : le paradigme pipe-line. Si son expressivité ne peut être contestée, nous montrons par contre sa faible efficacité, particulièrement pour des modèles dans lesquels les processus sont coûteux, mais également dans notre modèle où ce coût est aussi faible que possible. Nous en concluons à la nécessité d'envisager ce paradigme autrement et montrons comment une nouvelle approche de la synchronisation entre producteur et consommateur permet de s'affranchir du problème de granularité lié à ce paradigme, ce que nous illustrons sur divers exemples.

Nous avons donc exhibé diverses limitations de notre modèle de réduction de graphe, la plupart liées aux synchronisations implicites entre les processus. Ces limitations peuvent être levées à l'aide de nouveaux mécanismes de synchronisation implicite qui, sans remettre en cause le style de programmation initial, ouvrent de nouvelles perspectives dans la parallélisation effective et efficace d'algorithmes précédemment peu adaptés à une programmation fonctionnelle.

Nous pouvons, au vu de la critique du modèle initial que constitue cette thèse et des enrichissements qu'elle tente de lui apporter, justifier rétrospectivement tout l'intérêt porté au projet MaRS. En particulier, nous pensons que dans l'état actuel des réflexions en algorithmique parallèle, le concept MaRS, de par les solutions originales qu'il apporte à de nombreux problèmes couramment rencontrés, est un très bon candidat à la maîtrise algorithmique du nombre croissant de processeurs que la technologie rend désormais accessible. Le prochain défi à relever pour le concept MaRS serait alors d'étudier une topologie massivement parallèle, telle qu'une organisation multi-modules, technologiquement accessible et conservant toutes les

facilités de maîtrise algorithmique déjà établies pour une nombre plus faible de processeurs.

Annexe A

Compilation des structures

A.1 Rappel des définitions

Les opérateurs que nous devons compiler à l'aide de combinateurs ad-hoc sont les suivants :

(make-type $valeur_1$..$valeur_n$)
= (lambda (f) (f $valeur_1$..$valeur_n$))

(type-composante$_i$ instance)
= (instance (lambda (x_1..x_n) x_i))

(set-type-composante$_i$ instance valeur)
= (instance (lambda (x_1..x_n f) (f x_1..x_{i-1} valeur x_{i+1}..x_n)))

La compilation de ces définitions en expressions combinatoires doit intégrer certaines contraintes, comme la possibilité de définir des types sans restriction du nombre de composantes, la paresse ou l'efficacité maximum. Pour obtenir une absence de limite dans la taille des structures, nous les compilerons sous forme d'une arborescence n-aire dont la largeur sera bornée par *Lmax*, limite liée à l'indice maximum *Imax* de notre jeux de combinateurs, et dont les feuilles seront les composantes.

A.2 Compilation

A.2.1 Forme générale de l'arborescence

L'arborescence sera constituée de sous-arborescences structurellement identiques, mais dont la largeur pourra différer (tout en restant inférieure à *Lmax*). La distance entre les composantes (feuilles de l'arborescence) et la racine est une constante du type et ne dépend que de son nombre k de composantes et de *Lmax*. Une composante sera entièrement définie par la seule donnée de son chemin d'accès depuis la racine. Il faut en outre noter que quelle que soit la profondeur à laquelle nous nous plaçons dans la structure, tous les sous-arbres d'un arbre sauf éventuellement le dernier, sont de

largeur *Lmax* (on termine de remplir un sous-arbre avant d'en ajouter un autre).

A.2.2 Profondeur de l'arborescence

Une arborescence de profondeur p et dont les noeuds ont une arité maximum de *Lmax* a au maximum $Lmax^p$ feuilles en attribuant à une feuille la profondeur 0. Il en découle qu'une arborescence de profondeur p contenant k feuilles vérifie la relation $(Lmax)^{p-1} < k \leq (Lmax)^p$.

La profondeur nécessaire pour une arborescence à k feuilles et de largeur bornée par *Lmax* sera donc $p = \lceil \log_{Lmax} k \rceil$.

A.2.3 Chemin entre la racine et la i^{ième} feuille de l'arbre

Rappelons pour commencer qu'il existe entre la racine de l'arborescence et une quelconque de ses feuilles (correspondant à la i^{ième} composante de la structure) un chemin unique. Ce chemin est la séquence de sélection correspondant à tous les numéros de sous-arbre à sélectionner depuis la racine jusqu'à la feuille correspondant à la i^{ième} composante de la structure à k composantes et il sera noté $\{i_1 ... i_p\}$. Sa longueur est bien sûr égale à la profondeur p de l'arbre.

La première sélection a lieu à la profondeur p. A ce niveau, le premier sous-arbre (de profondeur $p-1$) contient les $Lmax^{p-1}$ premières feuilles, le second contient les feuilles $1 + (Lmax^{p-1})$ à $(Lmax^{p-1}) \times 2$, etc ...

Le numéro du sous-arbre de profondeur $p-1$ (à sélectionner à la profondeur p) contenant la i^{ième} feuille sera donc $i_1 = \lceil i / (Lmax)^{p-1} \rceil$

A chaque étape de sélection, choisir le x^{ième} sous-arbre de profondeur n (c'est à dire déterminer i_{p+1-n}) revient à abandonner les $x-1$ premiers sous-arbres qui contiennent $(x-1) \times (Lmax)^{n-1}$ feuilles. Ainsi la position de la i^{ième} feuille que nous cherchons à sélectionner se trouve décalée de $(x-1) \times (Lmax)^{n-1}$ positions par rapport à la première feuille de l'arbre restant. Il en résulte :

$$i_n = \left\lceil \left(i - \sum_{j=1}^{n-1} \left((i_j - 1) \times (Lmax)^{p-j} \right) \right) / (Lmax)^{p-n} \right\rceil ; (1 < n \leq p)$$

A.2.4 Nombre d'arbres d'une profondeur donnée

A la profondeur 0 il y a k feuilles (il y a donc k arbres de profondeur 0), à la profondeur n il y a $F^n(k)$ arbres de profondeur n rangés dans les arbres de profondeur $n+1$ avec les définitions $F^0(x) = x$ et $F^{n+1}(x) = F^n(\lceil x / (Lmax) \rceil)$. Cette formule découle directement du fait qu'il est possible de ranger jusqu'à *Lmax* arbres de profondeur n dans un arbre de profondeur $n+1$.

A.2.5 Feuilles contenues par un arbre donné

A la profondeur p (racine de la structure) l'arbre contient toutes les feuilles numérotée de 1 à k, la structure complète sera ainsi notée $A[p,1,k]$.

A la profondeur n; $(1 \le n \le p)$ il y a $F^n(k)$ sous-arbres qui sont dans l'ordre $A[n, 1+0\times(Lmax)^n, 1\times(Lmax)^n]$, $A[n, 1+1\times(Lmax)^n, 2\times(Lmax)^n]$, $A[n, 1+2\times(Lmax)^n, 3\times(Lmax)^n]$, ..., $A[n, 1+(F^n(k)-1)\times(Lmax)^n, k]$. Nous vérifions que quelle que soit la profondeur considérée, tous les arbres sont complètement remplis ($Lmax$ sous-arbres) sauf éventuellement le dernier.

A.2.6 Nombre de sous-arbres d'un arbre donné

Nous savons que chaque sous-arbre d'un arbre $A[n, a, b]$ est de niveau $n-1$ et qu'il contient donc $(Lmax)^n$ feuilles (sauf éventuellement le dernier). Donc le nombre de sous-arbres contenus par $A[n, a, b]$ qui contient $1+b-a$ feuilles est $\eta = \lceil (1+b-a)/(Lmax)^n \rceil$.

A.2.7 Construction de `(make-type valeur1…valeurk)`

La valeur de cette expression doit être l'arborescence à k composantes dont les k feuilles sont $\texttt{valeur}_1\ldots\texttt{valeur}_k$. Elle peut tout simplement être construite par macrogénération.

Un arbre de profondeur 1 contenant les feuilles f_x à f_y $(y-x \le Lmax)$ sera représenté en utilisant notre jeux de combinateurs par :

$!MS_i\ x_1\ldots x_i = x_i\ x_1\ x_2\ x_3\ldots x_{i-1}$

$!R_i\ x_1\ldots x_i = x_1$

$A[1,x,y] = (!MS_{y-x+4}\ !R_{y-x+2}\ !MS_{y-x+4}\ f_x\ldots f_y)$

Remarquons que notre arbre étant une $j^{ième}$ tranche d'au plus $Lmax$ feuilles de la structure, x et y vont vérifier :

$x = j\times Lmax+1$ et $y = min((j+1)\times Lmax, k)$; $(j \ge 0)$.

Un arbre $A[n, a, b]$ qui contient $\eta = \lceil (1+b-a)/(Lmax)^n \rceil$ sous-arbres, sera représenté de la même façon par :

$A[n,a,b] = (!MS_{\eta+3}\ !R_{\eta+1}\ !MS_{\eta+3}$
$A[n-1, a+0*Lmax^{n-1}, a-1+1*Lmax^{n-1}]$
$A[n-1, a+1*Lmax^{n-1}, a-1+2*Lmax^{n-1}]$ ...
$A[n-1, a+(\eta-1)*Lmax^{n-1}, b])$

A.2.8 Expression de `type-composante`$_i$

L'opérateur $\texttt{type-composante}_i$ doit sélectionner la valeur de la $i^{ième}$ composante d'une instance de type qui a k composantes. Il va donc directement

dépendre de la séquence de sélection $\{i_1 \ldots i_p\}$ telle que nous pouvons la déterminer seulement à partir de i, de k et de $Lmax$.

A partir de l'instance complète $A[p-1,1,k]$ la sélection va s'opérer en p étapes. A la première étape nous sélectionnerons le sous-arbre de rang i_1 pour obtenir $A[p-2, 1+(i_1-1)\times(Lmax)^{p-1}, min(k, i_1\times(Lmax)^{p-1})]$ sur lequel nous appliquons la séquence restante $\{i_2 \ldots i_p\}$ et ainsi de suite jusqu'à arriver à l'arbre de profondeur 0 qui contient la $i^{ième}$ composante cherchée $A[0, 1+(j-1)\times Lmax, min(k, j\times Lmax)]$, où j est le numéro de l'arbre de profondeur 0 qui contient la $i^{ième}$ feuille, soit $j = 1+(\lfloor (i-1)/(Lmax) \rfloor)$.

Nous prendrons la définition type-composantei = $(!MS_{1+p}!M_{2+i1}\ldots!M_{2+ip})$ qui réalise effectivement la sélection désirée comme nous pouvons le vérifier à partir de la définition des combinateurs en déroulant une étape de sélection après avoir rappelé les règles de réduction des combinateurs nécessaires :

$!MS_i\ x_1..x_i = x_i\ x_1..x_{i-1}$

$!M_i\ x_1..x_i = x_1\ x_i\ x_2..x_{i-1}$

$!R_i\ x_1..x_i = x_1$

Une étape de la séquence de réduction :

$\texttt{type-composante}_i\ \texttt{(make-type valeur}_1..\texttt{valeur}_k\texttt{)}$

$= ((!MS_{1+p}\ !M_{2+i_1}..!M_{2+i_p})\ A[p-1,1,k])$

$= (A[p-1,1,k]\ !M_{2+i_1}..!M_{2+i_p})$

$= (!MS_{\eta+3}\ !R_{\eta+1}\ !MS_{\eta+3}\ A[p-2,1,Lmax^{p-1}]\ ..$

$A[p-2,1+(\eta-1)*Lmax^{p-1},k])\ !M_{2+i_1}..!M_{2+i_p}\ ;\ \eta=k/Lmax^{p-1}$

$= !M_{2+i_1}\ !R_{\eta+1}\ !MS_{\eta+3}\ A[p-2,1,Lmax^{p-1}]\ ..$

$A[p-2,1+(\eta-1)*Lmax^{p-1},k]\ !M_{2+i_2}..!M_{2+i_p}\ ;\ \eta=k/Lmax^{p-1}$

$= !R_{\eta+1}\ A[p-2,1+(i_1-1)*Lmax^{p-1},i_1*Lmax^{p-1}]\ !MS_{\eta+3}$

$A[p-2,1,Lmax^{p-1}]\ ..\ A[p-2,1+(i_1-2)*Lmax^{p-1},(i_1-1)*Lmax^{p-1}]$

$A[p-2,1+i_1*Lmax^{p-1},\ (i_1+1)*Lmax^{p-1}]\ ..$

$A[p-2,1+(\eta-1)*Lmax^{p-1},k]\ !M_{2+i_2}..!M_{2+i_p}\ ;\ \eta=k/Lmax^{p-1}$

$= A[p-2,1+(i_1-1)*Lmax^{p-1},i_1*Lmax^{p-1}]\ !M_{2+i_2}..!M_{2+i_p}$

Nous venons de voir une étape de sélection (3 combinateurs exécutés) qui nous conduit de $(A[p-1,1,k]\ !M_{2+i_1}..!M_{2+i_p})$ à $(A[p-2,1+(i_1-1)*Lmax^{p-1},i_1*Lmax^{p-1}]\ !M_{2+i_2}..!M_{2+i_p})$ ce qui signifie que nous sommes passés de la structure complète de profondeur $p-1$ appliquée aux p sélections à la sous structure de profondeur $p-2$ qui contient $\texttt{composante}_i$ appliquée aux $p-1$

sélections restantes. L'exécution des p étapes nous permet donc de passer de la structure complète à la feuille désirée qui contient $\texttt{valeur}_i$.

$$\text{type-composante}_i = (!MS_{1+p} !M_{2+i_1} \ldots !M_{2+i_p})$$

A.2.9 Expression de $\texttt{set-type-composante}_i$

L'opérateur $\texttt{set-type-composante}_i$ doit modifier[1] la valeur de la i^{ième} composante d'une instance de type qui a k composantes. Tout comme $\texttt{type-composante}_i$, il va donc directement dépendre de la séquence de sélection $\{i_1 \ldots i_p\}$ telle que nous pouvons la déterminer seulement à partir de i, de k et de $Lmax$.

A partir de l'instance complète $A\,[p-1,1,k]$ la modification va s'opérer en p étapes. A la première étape nous modifierons la racine en prenant comme nouvelle valeur pour le sous-arbre de rang i_1 le résultat issu de la suite des modifications qui vont s'opérer sur ce sous-arbre de manière identique. La dernière étape de modification concernera l'arbre de profondeur 0 qui contient la i^{ième} composante cherchée $A\,[1, 1 + (j-1) \times Lmax, min(k, j \times Lmax)]$, et constituera à prendre pour nouvelle valeur de la feuille de rang i_p la valeur spécifiée.

L'expression sous forme de lambda expressions des étapes de modification telles que nous venons de les décrire est :

$$\text{set-type-composante}_i = \lambda\ o\ v \bullet (o\,(SET\,\{i_1 \ldots i_p\}\ v))$$

$$SET\,\{i_j \ldots i_p\} = \lambda\ v\ r\ m\ O_1 \ldots O_{i_j} \bullet (m\ r\ m\ O_1 \ldots O_{i_j - 1}\ (O_{i_j}\ (SET\,\{i_{j+1} \ldots i_p\}\ v)))$$

$$SET\,\{i_p\} = \lambda\ v\ r\ m\ O_1 \ldots O_{i_p} \bullet (m\ r\ m\ O_1 \ldots O_{i_p - 1}\ v)$$

Rappelons la définition nécessaire, puis déroulons une étape de cette modification :

$$!MS_i\ x_1 \ldots x_i = x_i\ x_1 \ldots x_{i-1}$$

1. Dans notre cadre fonctionnel, la modification correspond à la construction d'une nouvelle structure qui partage avec la structure d'origine ses constituants non modifiés.

```
(set-type-composante_i A[p-1,1,k] valeur)
= ((lambda (o v) (o (SET{i_1..i_n} v))) A[p-1,1,k] valeur)
= (A[p-1,1,k] (SET{i_1..i_n} valeur))
= ((!MS_η+3 !R_η+1 !MS_η+3 A[p-2,1,Lmax^p-1] ..
   A[p-2,1+(η-1)*Lmax^p-1,k]) (SET{i_1..i_n} valeur)) ; η=k/Lmax^p-1
= ((SET{i_1..i_n} valeur) !R_η+1 !MS_η+3 A[p-2,1,Lmax^p-1] ..
    A[p-2,1+(η-1)*Lmax^p-1,k]) ; η=k/Lmax^p-1
= ((lambda (v r m O_1..O_i_1)
     (m r m O_1..O_i_1-1 (O_i_1 (SET{i_2..i_n} v))))
   valeur !R_η+1 !MS_η+3 A[p-2,1,Lmax^p-1] .. A[p-2,1+(η-1)*Lmax^p-1,k])
; η=k/Lmax^p-1
= (!MS_η+3 !R_η+1 !MS_η+3 A[p-1,1,Lmax^p-1] ..
          A[p-2,1+(i_1-2)*Lmax^p-1,(i_1-1)*Lmax^p-1]
          (A[p-2,1+(i_1-1)*Lmax^p-1,i_1*Lmax^p-1] (SET{i_2..i_n} valeur))
          A[p-2,1+i_1*Lmax^p-1,(i_1+1)*Lmax^p-1] ..
          A[p-2,1+(η-1)*Lmax^p-1,k]) ; η=k/Lmax^p-1
```

Nous venons de voir la première étape de modification qui nous a conduit de `(A[p-1,1,k] (SET{i_1..i_n} valeur))` à `(!MS_η+3 !R_η+1 !MS_η+3 A[p-2,1,Lmax^p-1] .. A[p-2,1+(i_1-2)*Lmax^p-1,(i_1-1)*Lmax^p-1] (A[p-2,1+(i_1-1)*Lmax^p-1,i_1*Lmax^p-1] (SET{i_2..i_n} valeur)) A[p-2,1+i_1*Lmax^p-1,(i_1+1)*Lmax^p-1] .. A[p-2,1+(η-1)*Lmax^p-1,k])` qui ne diffèrent que par l'arborescence de rang i_1, laquelle se voit appliquer les modifications restantes $(SET\{i_2 \ldots i_p\}\ valeur)$.

Déroulons maintenant la dernière étape de modifications (qui est différente) entre l'arbre de profondeur 0 qui contient la composante à modifier, soit $A[0, 1+(j-1)\times Lmax, min(k, j\times Lmax)]$ où $j = 1+(\lfloor (i-1)/(Lmax) \rfloor)$, et

$(SET\{i_p\}\ valeur)$:

(A[0,1+(j-1)*Lmax,min(k,j*Lmax)] (SET{i_p} valeur))

= ((!MS_{x+3} !R_{x+1} !MS_{x+3} $valeur_{1+(j-1)\times Lmax}$...$valeur_x$) (SET{$i_p$} valeur))

; $j = 1 + (\lfloor (i-1)/(Lmax) \rfloor)$ $x = min(k, j \times Lmax)$

= (SET{i_p} $valeur$!R_{x+1} !MS_{x+3} $valeur_{1+(j-1)\times Lmax}$...$valeur_x$)

; $j = 1 + (\lfloor (i-1)/(Lmax) \rfloor)$ $x = min(k, j \times Lmax)$

= ((lambda (v r m O_1...O_{i_p}) (m r m O_1...O_{i_p-1} v))

$valeur$!R_{x+1} !MS_{x+3} $valeur_{1+(j-1)\times Lmax}$...$valeur_x$)

; $j = 1 + (\lfloor (i-1)/(Lmax) \rfloor)$ $x = min(k, j \times Lmax)$

= (!MS_{x+3} !R_{x+1} !MS_{x+3} $valeur_{1+(j-1)\times Lmax}$ ... $valeur_{i_p-1+(j-1)\times Lmax}$

$valeur$ $valeur_{i_p+1+(j-1)\times Lmax}$...$valeur_x$)

; $j = 1 + (\lfloor (i-1)/(Lmax) \rfloor)$ $x = min(k, j \times Lmax)$

Cette dernière étape a donc bien le fonctionnement désiré puisqu'elle modifie la $i_p^{ième}$ valeur de son arbre argument.

Nous traduisons ces lambda expressions en combinateurs indicés en utilisant les six combinateurs suivants :

!NN4+MS2 x_1 x_2 x_3 = x_2 (x_1 x_3)

!Set_a_i x_1..x_i = x_1 x_i x_3..x_{i-1} x_2

!Set_b x_1 x_2 x_3 = x_3 x_2 x_3

!Set_c x_1 x_2 x_3 x_4 = x_2 x_3 (x_1 x_4)

!Set_e_i x_1..x_i = x_1 x_3..x_{i-1} (x_i x_2)

!W_4 x_1 x_2 x_3 x_4 = x_1 x_4 x_2 x_3 x_4

set-type-composante$_i$ = (!NN4+MS2 ($SET\{i_1...i_p\}$))

$SET\{i_j...i_p\}$ = (!Set_c ($SET\{i_{j+1}...i_p\}$) !W_4!Set_e_{4+i_j})

$SET\{i_p\}$ = (!Set_a_{4+i_p}!Set_b)

Cette définition réalise effectivement la modification désirée telle que nous l'avons exprimée en lambda expressions, comme nous le vérifions pour les trois opérateurs :

- Set-type-composante$_i$

!NN4+MS2 SET{i_1..i_n} o v

= o (SET{i_1..i_n} v)

Donc (lambda (o v) (o (SET{i_1..i_n} v))) = !NN4+MS2 SET{i_1..i_n}.

- SET{i_1..i_n}

!Set_c SET{i_{j+1}..i_p} !W_4 !Set_e_{4+i_j} v r m O_1..O_{i_j}

= !W_4 !Set_e_{4+i_j} (SET{i_{j+1}..i_p} v) r m O_1..O_{i_j}

= !Set_e_{4+i_j} m (SET{i_{j+1}..i_p} v) r m O_1..O_{i_j}

= m r m O_1..O_{i_j-1} (O_{i_j} (SET{i_{j+1}..i_p} v))

Donc (lambda (v r m O_1..O_{i_j}) (m r m O_1..O_{i_j-1} (O_{i_j} (SET{i_{j+1}..i_p} v))))
= !Set_c SET{i_{j+1}..i_p} !W_4 !Set_e_{4+i_j}.

En reportant ceci dans l'étape de modification complète ((set-type-composante$_i$ A[p,1,k] valeur) qui entraîne l'exécution du combinateur !$MS_{\eta+3}$), il en résulte que celle-ci s'effectue en quatre combinateurs.

- SET{i_p}

!Set_e_{4+i_p} !Set_b v r m O_1..O_{i_p}

= !Set_b O_{i_p} r m O_1..O_{i_p-1} v

= m r m O_1..O_{i_p-1} v

Donc (lambda (v r m O_1..O_{i_p}) (m r m O_1..O_{i_j-1} v)) = !Set_e_{4+i_p} !Set_b.

En reportant ceci dans la dernière étape de modification ((A[1,1+(j-1)*Lmax,min(k,j*Lmax)] (SET{i_p} valeur)) qui entraîne l'exécution du combinateur !MS_{x+3}), il en résulte que celle-ci s'effectue en trois combinateurs.

A.3 Synthèse

Nous présentons dans cette partie le récapitulatif des divers codes compilés des différents opérateurs relatifs aux structures, ainsi que les définitions des combinateurs utilisés.

- (make-type valeur$_1$..valeur$_k$) = A[p,1,k]

où :

$p = \lceil \log_{Lmax} k \rceil$

A[1,x,y] = (!MS_{y-x+4} !R_{y-x+2} !MS_{y-x+4} valeur$_x$ … valeur$_y$)

et

$$A[n,a,b] = (!MS_{\eta+3}\ !R_{\eta+1}\ !MS_{\eta+3}\ A[n-1,a+0*Lmax^{n-1},a-1+1*Lmax^{n-1}]\ A[n-1,a+1*Lmax^{n-1},a-1+2*Lmax^{n-1}]\ \ldots\ A[n-1,a+(h-1)*Lmax^{n-1},b])$$

où $\eta = \lceil (1+b-a)/(Lmax)^n \rceil$

- type-composante$_i$ = $(!MS_{1+p}!M_{2+i1}\ldots!M_{2+ip})$

où :

$$p = \lceil \log_{Lmax} k \rceil$$

$$i_n = \left\lceil \left(i - \sum_{j=1}^{n-1} ((i_j - 1) \times (Lmax)^{p-j}) \right) / (Lmax)^{p-n} \right\rceil ; (1 < n \leq p)$$

$$i_1 = \lceil i/(Lmax)^{p-1} \rceil$$

- set-type-composante$_i$ = $(!NN4+MS2\ (SET\{i_1 \ldots i_p\}))$

où :

$$p = \lceil \log_{Lmax} k \rceil$$

$$i_n = \left\lceil \left(i - \sum_{j=1}^{n-1} ((i_j - 1) \times (Lmax)^{p-j}) \right) / (Lmax)^{p-n} \right\rceil ; (1 < n \leq p)$$

$$i_1 = \lceil i/(Lmax)^{p-1} \rceil$$

$$SET\{i_j \ldots i_p\} = (!Set_c\ (SET\{i_{j+1} \ldots i_p\})\ !W_4\ !Set_e_{4+i_j})$$

$$SET\{i_p\} = (!Set_a_{4+i_p}\ !Set_b)$$

Toutes ces définitions utilisent les combinateurs suivants :

$!MS_n\ e_1 \ldots e_n = e_n\ e_1 \ldots e_{n-1}$

$!R_n\ e_1 \ldots e_n = e_1$

$!M_n\ e_1 \ldots e_n = e_1\ e_n\ e_2 \ldots e_{n-1}$

$!NN4+MS2\ x_1\ x_2\ x_3 = x_2\ (x_1\ x_3)$

$!Set_a_i\ x_1..x_i = x_1\ x_i\ x_3..x_{i-1}\ x_2$

$!Set_b\ x_1\ x_2\ x_3 = x_3\ x_2\ x_3$

$!Set_c\ x_1\ x_2\ x_3\ x_4 = x_2\ x_3\ (x_1\ x_4)$

$!Set_e_i\ x_1..x_i = x_1\ x_3..x_{i-1}\ (x_i\ x_2)$

$!W_4\ x_1\ x_2\ x_3\ x_4 = x_1\ x_4\ x_2\ x_3\ x_4$

Bibliographie

[ABDALI 76] S.K. ABDALI.

An abstraction or combinatory logic.

The Journal of Symbolic Logic. Vol. 41 n°1.

[ABELSON 85] H. ABELSON, G. J. SUSSMAN.

Structure and Interpretation of Computer Programs.

MIT Press, Mc Graw Hill.

Structure et interprétation des programmes informatiques.

Iia, InterEditions 1989.

[BACKUS 78] J. BACKUS.

Can programming be liberated from the Von Neuman style? A functional style and its algebra of programs.

Communication of the ACM, Vol. 21-8.

[BIRD 88] R. BIRD, P. WADLER.

Introduction to functional programming.

Prentice Hall International Series in Computer Science.

[BURTON 84] F. W. BURTON.

Annotations control parallelism and reduction order in the distributed evaluation of functional programs.

ACM Trans. on Prog. Languages and Systems, Vol. 6, n°2, pp 159-174.

[BURTON 85] **F. W. BURTON.**

Speculative computation, parallelism and functional programming.
IEEE Trans. on Computers, Vol. C-34, n°12.

[BURTON 86] **F. W. BURTON.**

Controlling reduction partial order in functional parallel programs.
LNCS 279, pp 240-251.

[BURTON 87] **F. W. BURTON.**

Functional programming for concurrent and distributed computing.
The computer Journal, Vol. 30, n°5, pp 437-450.

[BURTON 91] **F. W. BURTON.**

Encapsulating non-determinacy in an abstract data type with determinate semantics.
Journal of Functional Programming Vol. 1(1) pp 3-20.

[CARCENAC 90] **Manuel CARCENAC.**

Structures de données irrégulières et infinies appliquées à la synthèse d'images.
Rapport de DEA, ENSAE, Toulouse.

[CARCENAC 91] **Manuel CARCENAC.**

Evaluation d'applications de traitement symbolique sur la machine MaRS.
Rapport final STPA n° 90-95004-10, 1/3381/RI.

[CASTAN 88] **Michel CUBERO-CASTAN.**

Vers une définition méthodique d'architecture de calculateur pour l'exécution parallèle des langages fonctionnels.
Thèse de Doctorat d'Etat-Sciences. Université Paul Sabatier.

[CHURCH 41] **Allonso CHURCH.**

The calculi of lambda conversion.
Princeton University Press.

[COUSIN 88] **Eric COUSIN.**

Etude semi-statique du compilateur de la machine MaRS.
Rapport interne ONERA-CERT-DERI.

[COUSIN 91] Eric COUSIN.

Compilation optimisée d'un langage fonctionnel pour une machine parallèle à réduction de graphe.

Thèse de Docteur Ingénieur, ENSAE.

[COUSINEAU 85] G. COUSINEAU, P. L. CURIEN, M. MAUNY.

The Categorical Abstract Machine.

Symposium on Functional Programming Languages and Computer Architecture. LNCS 201, pp 50-64.

[COUSINEAU 86] G. COUSINEAU, P. L. CURIEN.

La machine abstraite catégorique (CAM).

Bigre + Globule n° 50, pp 150-163.

[COUSINEAU 89] Guy COUSINEAU, Gérard HUET.

The CAML Primer. Version 2.6.

INRIA-ENS.

[COUSTET 90] Christophe COUSTET.

Structures de données en logique combinatoire.

Bigre n° 69, pp 134-141.

[CURRY 58] H.B. CURRY, R. FEYS, W CRAIG.

Combinatory Logic.

North Holland Publishing.

[DAMAS 82] Luis DAMAS, Robin MILNER.

Principal type-schemes for functional programs.

Proc. 9th ACM Symp. on Principles of Programming Languages pp 207-212.

[DENNIS 74] J.B. DENNIS.

First Version of a Data Flow Procedure Language.

MIT Lab. for Computer Science, Memo 93-1.

[DIJKSTRA 65] E. W. DIJKSTRA.

Solution of a problem in concurrent programming control.

Communication of the ACM, Vol. 8(9) p 569.

[DURAND 86] M.H. DURAND.

Etude et évaluation du parallélisme dans les langages fonctionnels. Une approche de la réduction de graphe par les combinateurs.

Thèse de Docteur-ingénieur ENSAE.

[GABRIEL 84] Richard P. GABRIEL, John Mc CARTHY.
Queue-based Multi-processing Lisp.
ACM Symp. on Lisp and Functional Programming, pp 25-43.

[GABRIEL 88] Richard P. GABRIEL, John Mc CARTHY.
QLISP.
Parallel Computation and Computers for AI. Kluwer Academic Publishers.

[GOLDBERG 88] B. GOLDBERG.
Multiprocessor execution of functional programs.
Ph. D. Thesis, YALE University.

[GOLDMAN 88] R. GOLDMAN, R. P. GABRIEL.
Qlisp: Experience and New Directions.
ACM Conf. on Lisp and Functional Programming, pp 143-152.

[GOLDMAN 89] R. GOLDMAN, R. P. GABRIEL,C. SEXTON.
QLISP: An interim Report.
LNCS 441, pp 161-181.

[GRAY 86] Sharon L. GRAY.
Using futures to exploit parallelism in Lisp.
S.M. Thesis, MIT EECS Dept.

[GRUE 89] K.E. GRUE.
Arrays in pure functional programming languages.
Lisp and symbolic computation, Vol. 2, Num 2, June 89, pp 105-114.

[HALSTEAD 85] R. H. HALSTEAD Jr.
Multilisp: A language for Concurrent Symbolic Computation.
Trans. of Programming Languages and Systems, Vol. 7 n°4, pp 501-538.

[HALSTEAD 86] R. H. HALSTEAD Jr.
An assessment of Multilisp: Lessons from experience.
J. of Parallel Programming, Vol. 15 n° 6 pp 1-43.

[HALSTEAD 87] R. H. HALSTEAD Jr.
Parallel computing using Multilisp.
Kowalik Ed. Parallel Computation and Computers for A.I., Kluwer Academic Publishers.

[HARRISON 89a] W. L. HARRISON.

The interprocedural analysis and automatic parallelization of scheme programs.

Lisp and Symbolic Computation: An international Journal, Vol. 2, n°3.

[HARRISON 89b] W. L. HARRISON, Z. HAMMARGUELLAT.

The Design of Automatic Parallelizers for Symbolic and Numeric Programs.

LNCS 441, pp 235-253.

[HENDERSON 80] Peter HENDERSON.

Functional Programming, Application and implementation.

Prentice-Hall International.

[HILLIS 85] W.D. HILLIS.

The Connection Machine.

MIT Press.

[HILLIS 86] W.D. HILLIS, C.L. STEELE.

Data Parallel algorithms

Communication of the ACM, Vol. 29 n° 12, pp 1170-1183.

[HUDAK 85] P. HUDAK, B. GOLDBERG.

Efficient Distributed Evaluation of Functionals Programs Using Serial Combinators.

IEEE Transactions on Computer, Vol. C-34, n° 10, pp 831-839.

[HUDAK 86] P. HUDAK, L. SMITH.

Para-Functional Programming: A paradigm for Programming Multiprocessors Systems.

ACM Symp. on Principles of Prog. Languages, pp 243-254.

[HUGHES 82] J. HUGHES.

Super-Combinators. A new implementation method for applicative languages.

ACM Symp. on Lisp and Functional Programming.

[HUGHES 90] J. HUGHES.

Why functional programming matters.

Research Topics in Functional Programming, Univ. of Texas at Austin Year of Programming Series, chapter 2, pp 17-42. Addison-Wesley.

[KAHN 87] **Gilles KAHN.**

Natural Semantics.

Symp. on Theoretical Aspects of Computer Science, Passau, Germany.

LNCS 247, pp 22-39. Rapport Inria RR-601.

[KATZ 89]

Para Tran: A transparent, transaction based runtime mechanism for parallel execution of Scheme.

TR-454, Lab. for Computer Science, MIT.

[KELLY 89] **Paul KELLY.**

Functional Programming for Loosely-coupled Multiprocessors.

MIT Press, Cambridge, Massachusetts.

[KORF 85] **Richard E. KORF.**

Depth-First an Optimal Admissible Tree Search.

Artificial Intelligence, Vol. 27 pp 97-109.

[KUMAR 90] **V. KUMAR, V. N. RAO.**

Scalable parallel Formulation of Depth-First Search.

Symbolic Computation, Parallel Algorithms for Machine Intelligence and Vision, Springer-Verlag pp 1-41.

[LAU 80] **P.BERGE, D. COMTE, N. HIFDI, B. PELOIS, J.C. SYRE.**

TEAU 18: Mise en oeuvre d'un prototype de multiprocesseur en assignation unique. Rapport final (deux volumes).

Rapport n° 3/3117/ONERA-CERT-DERI.

[LEMAITRE 90] **Michel LEMAITRE, Guy ZANON.**

Une technique de transformation de programmes applicatifs: remplacer des données par du contrôle.

Bigre n° 69, pp 118-122.

[LIVEZEY 89]

ASPEN: A Stream Processing Environment.

PARLE 89, LNCS 366.

[MANNA 91] **Zohar MANNA, Amir PNUELI.**

The Temporal Logic of Reactive and Concurrent Systems. •Specification•

Springer-Verlag

[Mc CARTHY 60] John Mac CARTHY.

Recursive functions of symbolic expressions and their computation by machine.

Communications of the ACM, Vol. 3 n° 4, pp 184-195.

[MARRE 92] Daniel MARRE.

La programmation fonctionnelle parallèle: Application à MaRS_Lisp.

Thèse de troisième cycle.

[MaRS 86a] M. CASTAN, M. H. DURAND, G. DURRIEU, B. LECUSSAN, M. LEMAITRE.

MaRS: a Multiprocessor Machine for Parallel Graph Reduction.

Proc. 19th Hawaii Inter. Conf. on System Sciences, pp 152- 159.

[MaRS 86b] M. CASTAN, A. CONTESSA, E. COUSIN, G. DURRIEU, B. LECUSSAN, M. LEMAITRE, P. NG.

Toward the design of a parallel graph reduction machine. The MaRS project.

Graph Reduction Workshop, Santa Fe. LNCS 279, pp 160-180.

[MaRS 86c] M. CASTAN, A. CONTESSA, E. COUSIN, G. DURRIEU, B. LECUSSAN, M. LEMAITRE, P. NG.

Etude sur le parallélisme non vectoriel dans les applications aéronautiques; Spécifications de la machine MaRS.

Rapport final n° 1/3267/DERI, ONERA-CERT.

[MaRS 87a] M. CASTAN, A. CONTESSA, E. COUSIN, G. DURRIEU, B. LECUSSAN, M. LEMAITRE.

Le processeur de réduction de MaRS, Machine à Réduction Symbolique.

Journées C^3 Architectures parallèles, BIGRE+GLOBULE n° 56, pp 18-28.

[MaRS 87b] M. CASTAN, M. H. DURAND, M. LEMAITRE.

A Set of Combinators for abstraction in linear space.

Information Processing Letter n° 24, pp 183-188.

[MaRS 88a] M. CASTAN, E. COUSIN, C. COUSTET, G. DURRIEU, B. LECUSSAN, M. LEMAITRE, P. NG, D. VISCAINO.

Etudes sur le parallélisme non vectoriel dans les applications aéronautiques, Vol 1: Le projet MaRS (édition révisée, Juillet 1988).

Rapport Final n° 1/3296/DERI, ONERA-CERT.

[MaRS 89a] **Michel LEMAITRE, Daniel MARRE.**

Machines informatiques symboliques. Vol. 2: algorithmique parallèle en langage fonctionnel.

Rapport final n° 1/3319/DERI, ONERA-CERT.

[MaRS 89b] **E. COUSIN, C. COUSTET, M. CUBERO-CASTAN, G. DURRIEU, B. LECUSSAN, M. LEMAITRE, D. MARRE , P. NG.**

Algorithmique parallèle en langage fonctionnel: application à MaRS_Lisp.

Journées AFCET-GROPLAN, Bigre n° 66, pp147-162.

[MaRS 90a] **M. CASTAN, A. CONTESSA, E. COUSIN, C. COUSTET, G. DURRIEU, B. LECUSSAN, M. LEMAITRE, P. NG.**

Etudes sur le parallélisme non vectoriel dans les applications aéronautiques. Le projet MaRS.

Rapport Final n° 1/3351.00/DERI, ONERA-CERT.

[MaRS 90b] **C. COUSTET, M. LEMAITRE.**

Evaluation de MaRS sur une application de traitement symbolique.

Actes du deuxième symposium sur les architectures nouvelles de machines. PRC-ANM Toulouse.

[MaRS 91b] **E.COUSIN, C. COUSTET.**

MaRS_Lisp: manuel utilisateur V1.0.

Document interne ONERA-CERT-DERI.

[MAUNY 86] **M. MAUNY, A. SUAREZ.**

De la Machine Abstraite Catégorique à l'implémentation de langages fonctionnels.

Bigre + Globule n° 50, pp 164-175.

[MILLER 87] **J. S. MILLER.**

MultiScheme, A parallel Processing System Based on MIT Scheme.

PhD Thesis, Dept. of Electrical Engineering an d Computer Science, MIT.

[MITCHEL 90] **John C. MITCHEL.**

Type Systems for Programming Languages.

Handbook of theoretical computer science. Elsevier Science Publishers.

[MOHR 90] E. MOHR, D.A. KRANZ, R.H. HALSTEAD Jr.

Lazy task creation: a technique for increasing the granularity of parallel programs.
ACM Conf on Lisp and Functional Programming, pp185-197.

[PAULSON 91] L.C. PAULSON.

ML for the Working Programmer.
Cambridge University Press.

[PEYTON-JONES 87a] S. L. PEYTON JONES.

Finding Fixpoints in abstract interpretation.
Abstract Interpretation of Declaratives Languages. Ellis Horwood.

[PEYTON-JONES 87b] S. L. PEYTON JONES.

The implementation of functional programming languages.
Prentice-Hall International.

[PLOTKIN 81] G. D. PLOTKIN.

A structural approach to operational semantics.
Proc. First Symp. Logic in Computer Science, pp 105-117.

[RAMOND 89] Philippe RAMOND.

Recherche de motifs consensus dans un ensemble de séquences d'ADN.
Mémoire d'ingénieur CNAM, Toulouse.

[ROE 91] Paul ROE.

Parallel programming using functional languages.
Ph. D Thesis, University of Glasgow.

[STEELE 78] G.L. STEELE, S.E. FAHLMAN, R.P. GABRIEL, D.A. MOON, D.L. WEINREB.

Common Lisp: the language.
Digital Press.

[STEELE 90] Guy L. STEELE Jr.

Making Asynchronous Parallelism Safe for the World.
17th Annual ACM Symp. on Principles of Prog. Languages, pp 218-231.

[SWARUP 91] V. SWARUP, U. S. REDDY, E. IRELAND.

Assignments for applicative languages.
LNCS 523, pp 192-214.

[TURNER 85a] D. A. TURNER.

Functional programs as executable specifications.

Math. Logic and Prog. Languages. Hoare and Shepherdson, Prentice Hall.

[TURNER 85b] D. A. TURNER.

Miranda: A non-strict functional language with polymorphic types.

Functional Prog. Languages and Computer Arch. Vol. 201, Springer-Verlag.

[WADLER 84] Philip WADLER.

Listlessness is Better than Laziness.

ACM Symp. on Lisp and Functional Programming, pp 45-52.

[WADLER 87a] Philip WADLER.

The concatenate vanishes.

Note distributed to FP electronic mailing list.

[WADLER 87b] Philip WADLER.

Projections for Strictness Analysis.

LNCS 274, pp 385-407. Springer-Verlag.

[WADLER 87c] Philip WADLER.

Strictness analysis on non-flat domains by abstract interpretation over finite domains.

Abstract Interpretation of Declaratives Languages. Ellis-Horwood.

[WAND 80] M. WAND.

Continuation Based Program Transformation Strategies.

Journal of the A.C.M., Vol. 27, pp 164-180.