USTL
FLANDRES ARTOIS

LIFL
U.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 473

THÈSE

Nouveau Régime

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Nathalie DEVESA

PROPOSITION D'UN SCHEMA D'EVALUATION PARALLELE DU LANGAGE FONCTIONNEL FP SUR UN RESEAU DE PROCESSUS

Thèse soutenue le 16 Janvier 1990 devant la commission d'Examen

Membres du jury

Président	V. CORDONNIER
Rapporteurs	M . RAYNAL
	E. SAINT-JAMES
Directeur de thèse	B. TOURSEL
Examinateurs	M.P. LECOUFFE
	D. LE METAYER
	R. PINO PEREZ

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Je tiens à remercier

Monsieur Vincent Cordonnier d'avoir accepté de présider le jury de cette thèse,

Monsieur Michel Raynal, qui m'a fait l'honneur de bien vouloir en être rapporteur,

Monsieur Emmanuel Saint-James d'avoir également accepté la tâche de rapporteur. Je le remercie pour ses commentaires, critiques et observations qui m'ont permis de corriger ce document et pour ses encouragements qui renforcent ma motivation,

Monsieur Daniel Le Metayer, d'avoir accepté de faire partie du jury et de m'avoir fait part de ses commentaires et critiques concernant mon travail.

Je remercie tout particulièrement Monsieur Bernard Toursel, qui a dirigé ces travaux. Son expérience, ses critiques toujours constructives et ses conseils m'ont permis non seulement de mener à bien cette étude, mais également de renforcer mon goût pour la recherche.

Je remercie tout autant Madame Marie-Paule Lecouffe qui a suivi de très près ces travaux. Sa disponibilité, ses encouragements et ses conseils m'ont apporté une aide considérable.

Je remercie très sincèrement Monsieur Ramon Pino Perez pour s'être intéressé à mes travaux. Ses observations et ses critiques suite à une lecture particulièrement attentive ont permis d'ores et déjà d'améliorer ce document, ses suggestions et encouragements me seront précieux dans l'avenir.

Mes remerciements s'adressent également

aux membres de l'équipe N-ARCH,

à tous mes "collègues de bureau" qui m'ont permis d'effectuer ces travaux dans une ambiance sympathique,

aux collègues "du bureau 322" avec lesquels j'ai vécu des repas et des pauses particulièrement animées,

à l'équipe des enseignants de l'EUDIL qui a permis l'aménagement de mon emploi du temps afin que je termine cette thèse dans les meilleures conditions,

et de façon générale, à tous mes amis qui m'ont supportée, toujours avec humour et compréhension malgré mes sautes d'humeur.

Madame Colette Laverdisse a réalisé la majeure partie de la frappe de cette thèse. Je la remercie pour le temps précieux qu'elle m'a permis "d'économiser".

Je tiens particulièrement à remercier Monsieur Henri Glanc qui a imprimé cette thèse, pour son sérieux et la qualité de son travail.

Enfin, je ne saurais terminer sans avoir remercié ma famille et tout particulièrement mes parents. Leur soutien et leurs continuels encouragements m'ont permis d'effectuer toutes mes études dans les meilleures conditions. C'est également grâce à eux que ce travail a pu être mené à bien. Qu'ils trouvent ici tous les "mercis" que j'aurais dû leur dire auparavant.

À mes parents,

pour tout ce qu'ils m'ont apporté.

PLAN GENERAL DE LA THESE

DEUXIEME PARTIE :
LE MODELE FORMEL D'EXECUTION PARALLELE DE FP

TROISIEME PARTIE : REPRESENTATION DU MODELE SOUS LA FORME D'UN RESEAU DE PROCESSUS ET EXEMPLE D'EXECUTION D'UN PROGRAMME FP

I : INTRODUCTION GENERALE

I : INTRODUCTION GENERALE

Nous présentons tout d'abord le cadre dans lequel s'est effectué ce travail, puis nous donnerons une introduction générale au contenu de la thèse.

I.1 : Le cadre du travail : le projet N-ARCH.

Le projet N-ARCH [GONCAL88] a comme objectif, la conception d'une machine parallèle à haut degré de parallélisme, adaptée à l'exécution de programmes déclaratifs. Cette machine consiste en un réseau de cellules interconnectées de telle façon que chaque cellule du réseau puisse être la racine d'un arbre de recouvrement du réseau. Les communications entre les différentes cellules sont réalisées par échanges de messages.

Une distribution uniforme des objets dans le réseau est assurée grâce à une technique de hachage à deux niveaux : une fonction de hachage globale détermine, en fonction du nom de l'objet à stocker, une cellule du réseau où l'objet doit être rangé. En cas de collision, une fonction de hachage locale permet de sélectionner, dans l'arbre de recouvrement du réseau, une cellule parmi l'ensemble des cellules filles de la cellule saturée. Si cette nouvelle cellule est également saturée, le même mécanisme est appliqué. La recherche d'un objet dans le réseau utilise également ces mécanismes de hachage sur le nom de l'objet à rechercher.

Cette méthode originale de distribution et recherche d'un objet dans le réseau introduit des contraintes au niveau de l'arbre de recouvrement qui doit être, si possible, régulier et complet. Dans [ARNAUD89], P. ARNAUD présente deux topologies de réseaux dont la simulation a montré qu'elles répondaient aux besoins de N-ARCH.

Dans N-ARCH, chaque cellule contient une mémoire associative permettant les requêtes simultanées [GONCAL88]. L'étude préliminaire d'une réalisation VLSI de cette mémoire a été réalisée avec l'aide du

laboratoire de micro-électronique de l'ISEN (Institut Supérieur d'Electronique du Nord).

Un émulateur du réseau N-ARCH a été réalisé par S. NIAR, sur un réseau de Transputers, qui a permis toute une série de tests et de mesures de performances présentées dans [NIAR89].

Parallèlement, J.C. MARTI [MARTI89] a simulé le fonctionnement au niveau macroscopique, du réseau N-ARCH, par la méthode de simulation par évènements significatifs [TARBOU70].

Diverses applications ont été développées dans le cadre du projet N-ARCH :

J.C. NICOLAS s'est intéressé aux bases de données. L'importance de la répartition des données dans le réseau est un point capital dans un contexte bases de données. Dans [NICOLA89a] est décrite une méthode permettant de répartir des tuples de relations en fonction des attributs de jointure et dans [NICOLAS89b] est décrite une autre méthode permettant la répartition des données non nécessairement sous première forme normale. Il s'intéresse aux modèles de données plus généraux que les bases de données relationnelles et utilise comme support, le modèle de données lié au langage FAD développé au MCC Austin Texas.

I. HANNEQUIN s'est intéressée au langage PROLOG et a développé un nouveau modèle d'évaluation parallèle de ce langage [HANNEQ89]. Ce modèle utilise un arbre ET/OU de réduction, élaboré dynamiquement. Le parallélisme OU y est partiellement exploité, tandis que le parallélisme ET est exploité sous forme de pipeline. Ceci a donné naissance au projet LogArch ayant pour objectif, la conception d'une machine Prolog parallèle utilisant ce modèle.

Les travaux effectués dans le cadre de cette thèse constituent une autre application développée au sein de N-ARCH. La machine N-ARCH ayant pour objectif, d'être adaptée à l'exécution de programmes déclaratifs, nous nous sommes intéressés à l'exécution de programmes fonctionnels.

I.2 : Introduction générale au contenu de la thèse

Les langages impératifs ont été conçus pour des machines von Neumann et de ce fait, reposent sur la notion d'instruction permettant de changer l'état de la machine. Cette dépendance vis-à-vis d'un modèle de machine rend les langages impératifs mal adaptés à des architectures multi-processeurs. Les langages fonctionnels sont, à l'opposé, issus de théories mathématiques, qui les rendent indépendants de toute machine. On peut distinguer, parmi les langages fonctionnels, les lambda-langages des langages sans variable. Les lambda langages trouvent leur support mathématique dans la théorie du lambda-calcul, tandis que les langages fonctionnels sans variable trouvent leur origine dans la logique combinatoire.

La complexité de la théorie du λ-calcul se répercute sur les lambda-langages. A l'opposé, les langages sans variable sont définis très simplement. Seulement, le programmeur doit réussir à se passer de variables et s'il y parvient, le programme résultant souffre parfois de quelques problèmes de lisibilité.

Malgré cela, nous avons un intérêt réel pour ce type de programmation tout à fait particulier, qui encourage le programmeur à concevoir des programmes implicitement parallèles.

La critique des langages impératifs et la présentation de ces deux familles de langages impératifs sera l'objet du chapitre II.

Le développement d'architectures parallèles est en grande partie, la cause de l'intérêt croissant pour les langages fonctionnels car il constitue une issue au problème de l'inefficacité dont souffrent les langages fonctionnels en général. L'exécution parallèle d'un langage fonctionnel nécessite le choix d'un mode d'évaluation (ou mode de contrôle) et d'un modèle d'évaluation. Il existe deux principaux modèles qui sont le modèle data-flow et le modèle de réduction. Ces deux modèles proposent des stratégies d'évaluation parallèles.

Quel que soit le modèle choisi, l'exécution parallèle de programmes n'est pas triviale puisque le concepteur est toujours confronté au

problème du choix de la granularité du parallélisme et le problème du contrôle du parallélisme. Dans le chapitre III, nous présenterons donc les différents modes et modèles d'évaluation existants et nous exposerons les problèmes dus au parallélisme.

Dans le chapitre IV, nous présentons un cas particulier d'exécution de langages fonctionnels : la réduction par combinateurs. La réduction par combinateurs permet de résoudre les problèmes d'implantation des lambda-langages en transformant une expression du lambda-langage en une expression ne contenant que des opérateurs définis indépendamment de tout contexte par des règles de réduction, et des symboles de constantes. Ces opérateurs sont appelés combinateurs. Nous présenterons donc dans ce chapitre, quelques familles de combinateurs existantes.

Les chapitres II, III, et IV constituent la première partie de ce document.

Le chapitre V représente à la fois une conclusion à cette première partie et une introduction à la deuxième partie : il nous permet de dégager les principaux points des chapitres précédents et d'arriver à la conclusion que même si le modèle de réduction de graphe est mieux adapté que le modèle data-flow à l'exécution de langages fonctionnels, il présente un certain nombre d'inconvénients en particulier dans le cas de l'évaluation de programmes sans variable. Ces inconvénients justifient les travaux effectués dans le cadre de cette thèse, ayant pour objectif la conception d'un nouveau modèle d'évaluation parallèle, présenté dans la deuxième partie de ce document.

Dans ce modèle, on a d'une part la représentation du programme en un ensemble d'arborescences et d'autre part, la représentation de l'argument du programme (la séquence argument). Le modèle est basé sur une exploration parallèle des arborescences, permettant d'acheminer à destination de la séquence argument, les opérations à réaliser, ce qui génèrera des réductions sur la séquence argument. L'exploration des arborescences est donc en quelques sorte, un "distributeur de tâches". De plus, comme elle est effectuée en parallèle, la distribution de "taches" est

également effectuée en parallèle, ce qui génère les réductions en parallèle.

Le chapitre VI nous permet de définir formellement les arborescences fonctionnelles et leur exploration, indépendamment de toute implantation et de toute machine-cible : l'exploration sera uniquement décrite par un graphe de transition d'état des noeuds de l'arborescence. Le graphe de transition d'état sera ensuite optimisé.

Dans le chapitre VII, nous ferons un premier pas vers une mise en oeuvre en concrétisant l'exploration des arborescences fonctionnelles par envois et réceptions de messages. Cette exploration par envoi de messages génère à destination de la séquence argument, des messages demandant des réductions dont nous donnerons la sémantique. Enfin, nous verrons que les réductions sur la séquence argument doivent être effectuées en respectant l'ordre d'émission des messages de demande de réduction, ce qui nous conduira à déterminer un mécanisme d'ordonnancement des messages émis par les arborescences à destination de la séquence argument.

Le chapitre VIII nous permettra de conclure cette deuxième partie en montrant pourquoi le modèle défini est validé.
Tout au long de cette deuxième partie, nous essaierons de rester au maximum indépendants de toute machine cible et de toute implantation.

Le but de la troisième partie est de décrire le modèle sous la forme d'un réseau de processus et de montrer comment un exemple de programme FP peut être appliqué à une séquence argument selon le modèle. Pour cela, nous serons obligés de faire certains choix quant à la représentation des objets et les stratégies d'exécution, ces choix seront décrits dans le chapitre IX.

Le chapitre X nous permettra alors de décrire les objets et les messages manipulés dans le modèle.

Dans le chapitre XI, nous donnerons un schéma fonctionnel du modèle et nous définirons les différents processus intervenant.

Ces trois derniers chapitres nous permettront de prendre un exemple de programme FP et de l'exécuter selon le modèle : ceci sera réalisé dans le chapitre XII

Enfin, la conclusion de cette thèse nous permettra de dégager les caractéristiques de ce nouveau modèle, de montrer dans quelle mesure nous avons répondu à nos objectifs initiaux et de dégager les perspectives envisageables.

PREMIERE PARTIE :

ANALYSE DE L'EXISTANT

II : LA PROGRAMMATION ET LES LANGAGES FONCTIONNELS

II : LA PROGRAMMATION ET LES LANGAGES FONCTIONNELS

II.1 : Introduction

Les langages de programmation peuvent être classifiés comme le montre la figure II.1.

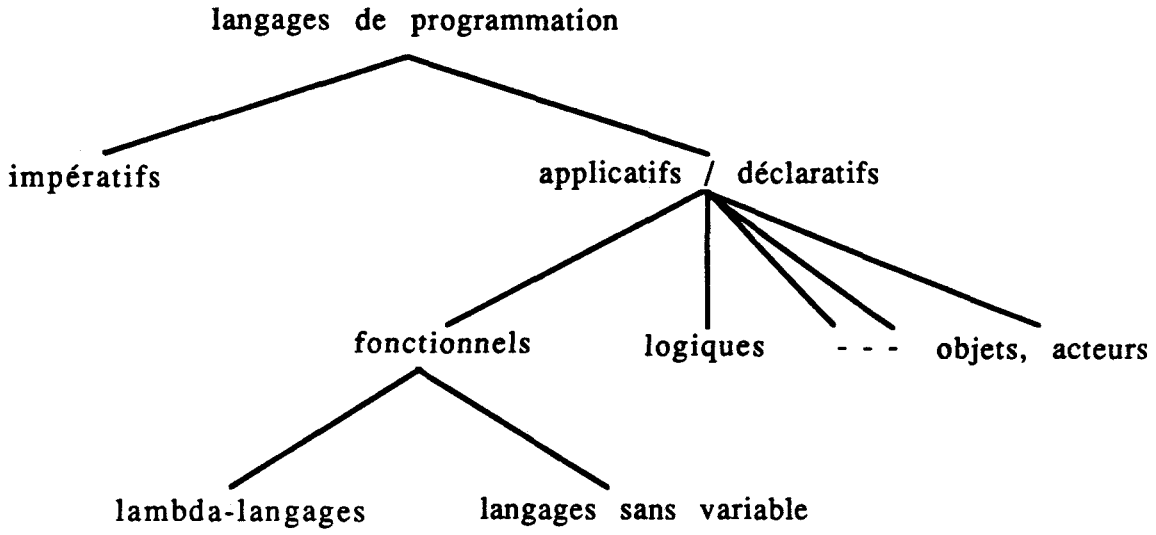


Figure II.1 : Classification des langages de programmation

La classe des langages impératifs comprend tous les langages basés sur la notion de commandement. Dans ces langages, un programme consiste en une suite d'instructions destinées à commander la machine, c'est-à-dire lui dicter toutes les opérations qu'elle doit effectuer. Parmi ces langages, citons FORTRAN, PASCAL, C..... etc. La deuxième partie de ce chapitre sera consacrée à la critique des langages impératifs.

Par opposition, la classe des langages applicatifs ou déclaratifs regroupe les langages permettant d'exprimer une solution à un problème posé, indépendamment de toute machine. Cette classe comprend les langages de programmation logique tels que PROLOG, les

langages d'acteurs et les langages orientés objet tels que SMALLTALK, les langages fonctionnels, qui nous intéressent plus particulièrement.

On peut noter que de nombreux travaux ont pour objectif de définir une nouvelle classe de langages non impératifs : les langages logico-fonctionnels dont le but est d'intégrer dans un même langage, les outils de la programmation fonctionnelle et de la programmation logique. Dans [BELLIA86], M.BELLIA et G.LEVI donnent un aperçu des techniques permettant cette intégration et des exemples sont donnés dans [FURBAC86] et [JACQUE88].

Les langages fonctionnels, comme l'indique leur nom, sont basés sur la notion de fonction. Cette notion, bien définie en mathématique, fournit une assise théorique solide aux langages fonctionnels qui est à l'origine d'un grand nombre de "bonnes propriétés", décrites dans la troisième partie, rendant très attractif ce style de programmation.

Seulement, l'étude relativement récente des langages fonctionnels fait apparaître quelques problèmes de crédibilité et d'efficacité qui sont à l'origine d'un certain scepticisme chez le programmeur non familier avec ce style de programmation et qui ternissent quelque peu l'image des langages fonctionnels. Malgré cela, les défenseurs de la programmation fonctionnelle restent optimistes : les langages fonctionnels sont inefficaces s'ils sont implantés sur des machines von Neumann mais les facilités qu'ils offrent aux mises en oeuvre parallèles "redorent leur blason". Ceci sera présenté dans la quatrième partie de ce chapitre.

La classe des langages fonctionnels peut elle-même être divisée en deux sous-classes : les lambda-langages (ou λ-langages) et les langages sans variable.

Les lambda-langages, présentés dans la cinquième partie, trouvent leur origine dans la théorie du lambda-calcul (ou λ-calcul). Parmi les lambda-langages, LISP est le plus connu mais les versions les plus élaborées de LISP sont plus proches des langages impératifs que des langages fonctionnels. On peut citer également MIRANDA, SASL, KRC ainsi que HOPE et ML qui sont actuellement les λ-langages les plus utilisés.

Les langages sans variable, présentés dans la sixième partie, trouvent leur origine dans la logique combinatoire, encore appelée théorie des combinateurs. Le programmeur est tellement habitué à utiliser des variables que ce style de programmation, qui peut paraître déconcertant au premier abord, n'a pas eu de réel succès avant que Backus, en 1978, ne définisse les systèmes FP. A partir de là, d'autres langages sans variable ont été définis. Citons FFP, le successeur de FP, JYM et son successeur GRAAL, FL etc...

Du fait de leur différence d'origine, les lambda-langages et les langages sans variable conduisent à deux styles de programmation bien différents. Dans la dernière partie de ce chapitre, nous comparerons donc ces deux sous-classes de langages fonctionnels en montrant la complexité de la lambda-expression par rapport à la simplicité des combinateurs.

II.2 : Critique des langages impératifs

Les langages impératifs sont l'objet de nombreuses critiques ([McCATH60], [BACKUS78], [BELLOT86], [GLASER87]) qui ont motivé la définition de langages applicatifs ou déclaratifs :

Le concept primordial dans les langages impératifs est celui d'instruction. Une instruction est un ordre transmis à la machine. Un programme impératif est une liste d'instructions à exécuter dans un certain ordre et donc, une suite d'opérations élémentaires que la machine doit réaliser. Parmi ces instructions, la plus fréquente est l'affectation. Elle permet de modifier la valeur d'une variable c'est-à-dire qu'elle modifie le contenu d'un emplacement mémoire. Le programmeur est ainsi amené à gérer la mémoire de la machine. D'autres instructions telles que les instructions de saut imposent un ordre d'exécution et le programmeur doit ainsi contrôler l'exécution. Le programmeur est donc amené à se concentrer davantage sur des détails de mise en oeuvre d'un algorithme plutôt que sur l'algorithme lui-même. Quand on ajoute à cela les effets pervers introduits dans certains

programmes, tels que les effets de bord, on est amené à penser qu'écrire un programme dans un langage impératif correspond à écrire une suite d'instructions, en espérant qu'une fois qu'elles seront exécutées, le résultat soit effectivement celui recherché.

Cette dernière remarque, volontairement provocatrice, se justifie en grande partie par le manque de "bonnes propriétés mathématiques" des langages impératifs. Faire une preuve de programme est réalisable sur un exemple d'école mais non envisageable sur un programme conséquent. Les études de sémantique et des propriétés des langages impératifs se révèlent très complexes du fait même de la complexité des langages.

J.W. Backus souligne de plus la rigidité des langages impératifs : une extension du langage, de ses structures ou de ses types ne peut être envisagée qu'au prix d'une refonte complète du langage.

L'étude des architectures multi-processeurs constitue à notre sens, l'avenir de l'informatique .A ce sujet, P. BELLOT ([BELLOT86]) pense que "les langages impératifs sont des versions évoluées du modèle de machine de von Neumann", et sa conclusion est la suivante : "il n'y a pas de raison apparente pour que les langages impératifs restent adaptés dans l'avenir".

Les langages applicatifs ou déclaratifs, introduisant d'autres concepts et d'autres styles de programmation, apparaissent donc comme une issue, et parmi eux, les langages fonctionnels, du fait de leurs caractéristiques ont retenu toute notre attention.

II.3 : Principes, avantages, inconvénients de la programmation fonctionnelle.

La programmation fonctionnelle repose sur l'utilisation de fonctions. La notion d'instruction disparait dans la mesure où l'opération de base dans un langage fonctionnel est l'application d'une fonction à ses arguments.

Définir une fonction revient à déterminer l'effet qu'elle doit produire lorsqu'elle est appliquée à ses arguments. La notion de commandement disparait dans la mesure où une fonction décrit des traitements sans se préoccuper de la mise en oeuvre de ces traitements.

Pour illustrer ces notions, reprenons l'exemple donné dans [GLASER87] : le but du problème est de spécifier un objet physique, en l'occurrence une remise.

De façon impérative, on peut décrire comment construire la remise :

1) mettre en place les fondations
2) bâtir les murs
3) mettre en place le plancher
4) placer le toit au-dessus

De façon fonctionnelle, on décrit la constitution de la remise :
une remise consiste en :

1) des murs portés par des fondations
2) un plancher porté par des fondations
3) un toit porté par des murs.

Cet exemple caractérise de façon intuitive, l'approche fonctionnelle. Un programme exprime un résultat (ou plus exactement une solution au problème) et non une suite d'opérations qui, lorsqu'elles sont exécutées, conduisent au résultat. Un problème peut ainsi être résolu fonctionnellement, indépendamment de toute machine et sa mise en oeuvre n'est plus la préoccupation majeure.

Il n'y a pas de contrôle explicite dans un programme fonctionnel : l'ordre dans lequel doit se dérouler l'exécution n'est pas explicitement spécifié dans le programme mais les dépendances entre les différentes actions devant être finalement réalisées peuvent être déduites : si un ordonnancement est nécessaire, il est implicite.

Il existe deux principales méthodes de programmation fonctionnelle : la **récursivité** et **l'utilisation de fonctionnelles.**

"La récursivité permet de transformer une définition inductive de fonction en un processus de calcul" [BELLOT86]. L'expression "processus

de calcul" ne signifie pas commandement ou instruction transmise à une machine mais décrit fonctionnellement la façon de mener ces calculs.

Les fonctionnelles permettent de combiner des fonctions afin d'en obtenir de nouvelles, conduisant à un mode de programmation par combinaisons de programmes : des fonctions complexes sont construites à partir de fonctions plus simples. L'analyse fonctionnelle d'un problème est donc typiquement descendante.

Remarque :

Ces deux modes de programmation peuvent être combinés dans un même programme.

Les avantages de la programmation fonctionnelle ont déjà été amplement décrits [BACKUS78], [DARLIN82], [HENDER80]. La plupart de ces avantages provient du support théorique des langages fonctionnels.

La fonction est un objet mathématiquement plus simple que l'instruction ce qui fait des langages fonctionnels, des langages mathématiquement plus simples que les langages impératifs. Le passage de la spécification formelle d'un problème au programme fonctionnel s'en trouve ainsi facilité. Les propriétés mathématiques du langage permettent de vérifier la correction des programmes.

Dans [BACKUS78], J.W. BACKUS définit les systèmes FP. Un système FP est fondé sur un ensemble fixé de fonctionnelles qui servent d'opérations à une algèbre associée au système, dont les éléments sont les programmes FP. Cette algèbre fournit les moyens de raisonner formellement sur les programmes FP. Les lois algébriques peuvent être utilisées de façon assez mécanique pour transformer les programmes et surtout prouver formellement la correction d'un programme

Dans les langages fonctionnels purs, l'affectation est absente. Si le langage autorise la manipulation de variables, elles ne peuvent être utilisées que comme argument de fonction ou comme identificateur à affectation unique. En conséquence, il n'y a pas d'effets de bord dans les langages fonctionnels purs. L'absence d'effets de bord contribue également à rendre les preuves de programmes plus aisées et assure que les mécanismes de passage par valeur et passage par nom des

arguments d'une fonction (cf §.III.1) ont la même sémantique [BERKLI75] (ceci, à condition bien sûr, que l'évaluation de la fonction selon ces modes de passage des arguments conduise effectivement à un résultat).

Backus ajoute encore que du fait de la notation compacte, les programmes décrits dans un langage fonctionnel sont plus courts que des programmes écrits dans un langage impératif. De ce fait, le programmeur augmente sa productivité et la maintenance des programmes est réduite.

Seulement, une notation compacte restreint la lisibilité d'un programme, ce qui représente à notre avis un inconvénient des langages fonctionnels.

A cela s'ajoute des problèmes de crédibilité et d'efficacité. Les détracteurs des langages fonctionnels pensent qu'ils ne permettent de résoudre qu'une certaine classe de problèmes (les problèmes pouvant être résolus par la méthode "diviser pour régner") de façon efficace, mais qu'ils sont mal adaptés aux autres classes de problèmes, surtout les problèmes comportant un grand nombre d'opérations d'entrées/sorties.

Certains pensent que si les langages fonctionnels sont inefficaces c'est qu'ils sont souvent interprétés et non compilés. Dans ce but, des techniques de compilation ont été développées. Elles sont en grande partie décrites dans [DILLER88] et [PEYTON87a].

Parallèlement, des études sur les techniques de transformation et d'optimisation de programmes fonctionnels ont été menées [BELLEG85], [KIEBUR81], [WADLER81], visant à minimiser le nombre de structures de listes intermédiaires lors de l'exécution d'un programme fonctionnel.

Ces techniques d'optimisation et de compilation viennent s'ajouter aux possibilités d'évaluation parallèle offertes par les langages fonctionnels. La facilité qu'offre les langages fonctionnels à la programmation parallèle est une issue majeure au problème de l'inefficacité, nous montrons cela dans la section suivante.

II.4 : Les langages fonctionnels facilitent la programmation parallèle

Dans les langages impératifs, le parallélisme est souvent obtenu par le partitionnement d'un problème en tâches. Ce partitionnement est fixe et statique : une tâche est conçue comme une unité relativement grande pouvant elle-même contenir des traitements parallèles. Les tâches ne peuvent généralement pas être créées ou détruites dynamiquement. Le programmeur est ainsi amené à gérer le parallélisme avec toutes les difficultés que cela implique. En effet, il est relativement difficile de raisonner sur un programme multi-tâches : même si le comportement du programme devrait être le même quel que soit l'ordonnancement des tâches, le programmeur doit s'assurer que c'est effectivement le cas et doit pour cela envisager toutes les possibilités pour l'ordre d'exécution des tâches. De plus, les tâches communiquent entre elles par messages ou grâce à des sous-programmes. Le programmeur a donc la charge d'élaborer des protocoles de synchronisation et de communication entre les tâches.

Nous avons déjà signalé dans la section précédente, que l'ordre d'exécution dans un programme fonctionnel n'est pas prédéfini mais si un ordre doit être respecté, il est implicite. Le programmeur est donc libéré des problèmes de partitionnement, de synchronisation et de communication. Un programme fonctionnel est implicitement parallèle et ce n'est pas le programmeur qui gère ce parallélisme mais les stratégies d'évaluation qui l'exploitent. Lorsqu'un ordre implicite n'est pas imposé, le théorème de Church-Rosser [CURRY58] montre, qu'étant données les propriétés des langages fonctionnels (en particulier l'absence d'effets de bord), quel que soit l'ordre dans lequel sont exécutées les opérations, le résultat final sera le même. De plus, le style de programmation fonctionnelle encourage la programmation parallèle grâce à des fonctionnelles qui permettent par exemple, d'appliquer une fonction à chacun des éléments d'une structure. Le programmeur est ainsi amené à travailler sur des structures plutôt que sur des éléments de cette structure.

Le parallélisme n'étant pas géré par le programmeur, il peut être exploité massivement ; les arguments d'une fonction, les éléments d'une structure créée dynamiquement peuvent tous être évalués en parallèle et indépendamment les uns des autres.

Malgré cela, M.H. DURAND montre dans [DURAND86] que le programmeur a encore une grande responsabilité : un programme peut être écrit selon un algorithme plus ou moins parallèle et le choix d'un algorithme peut avoir une grande influence sur le degré de parallélisme obtenu. Cette notion sera illustrée sur un exemple dans la section II.7.1.

Dans l'introduction de ce chapitre, nous avons distingué deux sous-classes de langages fonctionnels. Ayant présenté les caractéristiques communes à ces deux sous-classes, nous nous intéressons dans la section suivante à la classe des lambda-langages.

II.5 : Les Lambda-langages

La classe des lambda-langages regroupe les langages fonctionnels issus de la théorie du lambda-calcul. Après avoir rappelé quelques définitions de la théorie du λ-calcul, nous présentons très brièvement quelques lambda-langages.

II.5.1 : l'origine des lambda-langages : le lambda-calcul

Le λ-calcul est un modèle de calculabilité élaboré par Church vers 1930 [CHURCH41]. Dans ce modèle, les fonctions sont construites par abstraction d'une variable dans une expression, permettant de formaliser le passage argument-valeur. On donne alors à cette fonction le nom de λ-expression. En 1936, KLEENE montra que le λ-calcul était suffisamment puissant pour décrire toutes les fonctions calculables. Intuitivement, une fonction est calculable si elle est programmable. Ceci donne à la notion de fonction une approche différente de celle des

mathématiques classiques qui définissent une fonction par son domaine (l'espace des données) et son co-domaine (l'espace des résultats).

La fonction est représentée dans le λ-calcul par une expression qui, concaténée à une liste d'arguments, est soumise à un algorithme universel de réduction qui la transforme en la valeur de la fonction au point considéré.

Remarque :

Les définitions qui suivent sont extraites de [COMYN79] et de [GLASER87].

Définition :

Le λ -calcul peut être vu comme un système de réécriture de λ-expressions, où les λ-expressions sont des mots reconnus par le langage défini ci-dessous :

- Soit V = {x, y,...} un ensemble infini dénombrable de variables
- Soit C = {a, b,...} un ensemble de constantes
- Soit W = V ∪ C

Si *S* est l'axiome du langage, on définit les règles de production suivantes :

(1) $S \rightarrow \alpha$ $\alpha \in W$
(2) $S \rightarrow (SS)$ règle dite d'application
(3) $S \rightarrow (\lambda v.\ S)\ v \in V$ règle dite d'abstraction.

Simplifications :

u z t w simplifie (((u z) t) w)

$\lambda x_1 x_2 ... x_n.$ Y simplifie $(\lambda x_1. (\lambda x_2. (... (\lambda x_n.\ Y) ...)))$

Définition :

Si X et Y sont deux λ-expressions quelconques, on peut définir inductivement les variables libres (*Varlib*) et les variables liées (*Varliée*) dans ces expressions par :

$Varlib(x) = \{x\}$
$Varlib(X\ Y) = Varlib(X) \cup Varlib(Y)$
$Varlib(\lambda x.\ X) = Varlib(X) - \{x\}$

$Varliée(x) = \emptyset$
$Varliée(X\ Y) = Varliée(X) \cup Varliée(Y)$
$Varlieé(\lambda x.\ X) = Varliée(X) \cup \{x\}$

La notion de variable libre peut être assimilée à la notion de variable globale tandis que la notion de variable liée peut être assimilée à la notion de variable locale ou formelle.

Remarque :

Une expression M pour laquelle $Varlib(M) = \emptyset$, est dite close.

La substitution :

C'est le seul mécanisme de calcul dans les λ-expressions. $[N/x]\ M$ dénote l'expression obtenue lorsque l'on remplace toutes les occurrences libres de x dans M par N. Avec cette notation, la substitution est définie par :

$-[N/x]\ x = N$
$[N/x]\ a = a$ si $a \neq x$ et $a \in W$

$-[N/x]\ (M_1\ M_2) = ([N/x]\ M_1)\ ([N/x]\ M_2)$

$-[N/x]\ (\lambda y.\ X) = \lambda z.\ ([N/x]\ ([z/y]\ X))$
z est alors une nouvelle variable : $z \notin Varlib(N) \cup Varlib(X)$.

La dernière règle qui peut paraître a priori complexe s'illustre simplement sur l'exemple suivant :

$[y|x]\ (\lambda y.\ x\ y)$.

On désire remplacer dans la λ-expression la variable x par la variable y. Cette variable x est libre dans l'expression. Si on effectue directement la substitution, la variable y remplaçant x est capturée. Pour éviter cette capture, on change tout d'abord le nom de la variable y, puis on effectue la substitution. On obtient ainsi :

$[y|x]\ (\lambda y.\ x\ y)$
$= \lambda z.\ ([y|x]\ ([z|y]\ (x\ y))$
$= \lambda z.\ ([y|x]\ (x\ z))$
$= \lambda z.\ y\ z$

Règle de l'α-conversion

Définition :

On appelle α-radical, toute expression de la forme $\lambda y.\ X$ dans laquelle y n'est pas liée dans X.

La règle de l'α-conversion consiste à remplacer cet α-radical par $\lambda v.\ ([v/y]\ X)$ avec $v \notin Varlib(X)$.

On note : $\lambda y.\ X\ \ conv_\alpha\ \ \lambda v.\ ([v|y]\ X)$

Règle de β-conversion : (ou β-réduction)

Définition :

On appelle β-radical (ou rédex) toute expression de la forme $(\lambda x.\ M)\ N$.

La règle de β-réduction consiste à remplacer tout rédex de cette forme par la substitution $[N/x]\ M$

On note : $(\lambda x.\ M)\ N\ \ conv_\beta\ \ [N|x]\ M$

Définition :

une expression qui ne contient aucun rédex est dite sous forme normale.

Remarques :

1) Les règles d'α et β-conversions correspondent à des transformations syntaxiques et peuvent être considérées comme les règles de réécriture d'un système formel.

2) La règle de β–réduction peut être assimilée à l'appel de procédure en programmation. La règle

$(\lambda x.\ M)\ N\ conv_\beta\ [N|x]\ M$

se lit alors : Remplacer le paramètre formel x dans M par l'argument effectif N.

Règle d'extension (ou η-réduction)

Elle consiste à confondre X et Y dès lors que $X\ V = Y\ V$ pour toute expression V. Elle se traduit par la conversion suivante :

$\lambda x.\ F\ x\ \ conv_\eta\ \ F$ si x n'est pas libre dans F.

Ces trois règles de conversion permettent de simplifier les expressions. Le λ-calcul possède la propriété appelée "transparence du référent" : quand une sous-expression a été simplifiée, toute autre occurrence de la même sous-expression dans une expression globale peut être remplacée par sa forme simplifiée.

Pour la suite de notre présentation, nous nous limiterons à ces quelques définitions. Le lecteur pourra trouver [BAREND84] et [CHURCH41] une description formelle et détaillée du lambda-calcul.

II.5.2 : Exemples de lambda-langages

Etant données la puissance et la simplicité apparente du lambda-calcul, de nombreux langages fonctionnels basés sur cette théorie ont été définis dont LISP, MIRANDA, SASL, KRC, HOPE, ML, ...etc. Nous donnons dans la suite de cette section, une brève description des langages LISP, KRC, HOPE, et ML.

II.5.2.1 : LISP

Le langage LISP a été conçu par John Mc CARTHY [McCART62] au début des années 60. Le but initial était la conception d'un langage dont les entités de base sont des entités mathématiques, de manière à pouvoir y inclure la théorie des fonctions récurrentes développée pendant les 20 ou 30 années précédentes. Mc CARTHY était particulièrement intéressé par la possibilité d'effectuer des traitements puissants sur les données, et les applications possibles à l'intelligence artificielle.

LISP se caractérise par le fait que les données et le programme ne sont pas différenciés dans l'écriture. Les seules structures étant l'atome ou la liste, les définitions et évaluations de fonctions s'écrivent sous forme de listes. Un programme est alors une suite de définitions de fonctions (éventuellement récursives) et d'évaluations de ces fonctions. Les programmes et les données s'exprimant avec la même syntaxe, un programme LISP peut traiter, engendrer, ...etc, d'autres programmes LISP. Ainsi, des opérateurs tels que MAPCAR et APPLY permettent de manipuler des fonctions d'ordre supérieur (i.e. : les fonctions peuvent être utilisées comme paramètres ou résultats d'autres fonctions).

A l'origine, LISP était basé sur cinq fonctions primitives : CONS, CAR, CDR, EQ et ATOM. Puis, les versions successives de LISP ont peu à peu été enrichies et ce faisant, le langage a perdu ses propriétés fonctionnelles. Il y a donc en réalité deux LISP : le coeur fonctionnel et le

langage étendu, comprenant toutes sortes d'instructions de type impératif telles que l'affectation, et même des techniques de saut, etc...

II.5.2.2 : KRC

Le langage KRC, développé par D. TURNER [TURNER82], est basé (tout comme le langage SASL developpé également par TURNER [TURNER76]) sur une représentation des fonctions comme un ensemble d'équations ; l'écriture d'une fonction est proche de l'écriture des clauses en PROLOG.

KRC offre au programmeur, la possibilité d'utiliser des ensembles de valeurs (au sens mathématique du terme), même si ceux-ci sont en fait représentés par des listes. Ainsi, on peut définir en KRC l'ensemble

$$\{(x,y) / x \leftarrow [0..2] ; y \leftarrow [-2..2] ; x < y \}$$

qui s'évalue en : {(0 1) (0 2) (1 2)}

Grâce à l'évaluation paresseuse (voir section III.1.3), il est possible de manipuler des listes ou intervalles infinis. Par exemple, [0..] définit l'ensemble des entiers naturels.

Enfin, KRC permet une manipulation aisée des fonctions d'ordre supérieur.

On peut reprocher au langage KRC l'absence de définitions purement locales ce qui peut forcer l'utilisateur à définir des fonctions qui seront globales inutilement, introduisant un risque de confusion dans les différents niveaux de détails.

Notons enfin qu'en 1985, TURNER a défini le langage MIRANDA, en ajoutant à KRC le concept de typage des données. MIRANDA est ainsi un langage proche des langages HOPE et ML décrits ci-dessous.

II.5.2.3 : HOPE et ML

HOPE et ML sont deux langages fonctionnels développés à l'université d'Edimbourg en Ecosse.

HOPE a été conçu par R. BURSTALL & Al [BURSTA80].

ML a été initialement développé en 1974 [GORDON79] et servait de méta-langage pour le projet LCF (Logic for computable functions) à Edimbourg, destiné à la construction de preuves formelles de fonctions récursives. Très vite, ML a été utilisé indépendamment de LCF et le langage a été redéfini en reprenant les concepts principaux de HOPE, aboutissant à une nouvelle version de ML appelée "standard ML" [HARPER86]. Dans la suite, lorsque nous parlerons de ML, c'est en fait à "Standard ML" que nous ferons référence.

Propriétés communes aux deux langages : [WIRSIN87]

- Un programme s'écrit sous la forme d'une séquence de déclarations.

- Ce sont deux langages strictement typés. De plus, le concept de type polymorphe (ou générique) permet des définitions schématiques de types de données et de fonctions.

- Les fonctions sont définies inductivement par des équations récursives sur les constructeurs des types de données.

- Les deux langages permettent la manipulation de fonctions d'ordre supérieur.

Le langage HOPE

Le type d'une fonction est donné par le nom de la fonction et par sa fonctionnalité : le type des paramètres et du résultat.

Exemple :

dec max : num ≠ num → num

La fonction max a deux paramètres entiers et son résultat est un entier.

Il existe 5 types de données prédéfinis : *num*, *truval* (booléens), *char*, *list* (séquences finies), *set* (ensembles finis). Il est possible de définir de nouveaux types de données en indiquant leur constructeur.

exemple :

couleur = = blanc + + noir + + rouge + + vert.

Les constructeurs du type *couleur* sont les quatre constantes *blanc*, ...,*vert*.

Etant donné un nouveau type de données, une nouvelle fonction peut être définie en associant à chaque constructeur, (au moins) une nouvelle équation, l'ordre des équations ne jouant aucun rôle.

Un programme est ainsi constitué d'une expression et d'une séquence de déclarations de fonctions et de types de données, servant à spécifier les symboles fonctionnels utilisés dans l'expression. HOPE offre la possibilité de grouper les déclarations dans des modules, permettant ainsi de cacher des déclarations auxiliaires : seules les déclarations nécessaires à l'évaluation de l'expression sont ainsi visibles de l'extérieur.

Par rapport à KRC, HOPE possède l'avantage d'être fortement typé mais ne permet pas la manipulation d'ensembles.

Le langage ML

Le langage ML est une extension de HOPE. Dans ML, le concept de modularisation a été développé. Il repose sur trois notions : les structures, les signatures et les foncteurs.

Une **structure** permet d'encapsuler des types de données et des fonctions dans une unité de programmes.

Une **signature** définit la fonctionnalité (ou le type) d'une structure. Elle est définie par tous les noms de types de données et de fonctions de la structure qui sont visibles à l'extérieur. Elle décrit ainsi l'interface entre une unité de programme et les autres unités de programme. Il est

possible de construire des structures à partir de structures existantes, on obtient alors des structures hiérarchiques.

Un **foncteur** définit une structure paramétrée. Le paramètre d'un foncteur consiste en un ou plusieurs noms de structures munis de signatures.

A la différence des langages KRC et HOPE, ML n'est pas "purement" fonctionnel : des concepts impératifs tels que l'affectation, ont été introduits. Les concepteurs du langage font observer que ces concepts ne sont pas centraux dans ML : l'affectation ne peut être utilisée que pour modifier la valeur d'un pointeur.

On peut en revanche retenir, à l'avantage de ML, les mécanismes puissants de modularisation et le fait que ML possède une sémantique formelle dont la description est donnée dans [HARPER87].

Nous nous limiterons à ces quelques langages représentant la classe des lambda-langages. Tous ces langages ont en commun l'utilisation de variables. Nous présentons dans la section suivante la classe des langages sans variable et suite à cela, nous comparerons ces deux classes de langages

II.6 : Les langages sans variable

Au sens mathématique du terme, la variable sert à marquer une place dans un contexte. Au sens informatique, une variable est la donnée d'un couple (nom, valeur). Nous avons déjà cité les inconvénients de cette variable informatique dans les langages impératifs tels que les effets de bord et les problèmes d'homonymie. La plupart des lambda-langages résolvent ce problème en n'autorisant pas l'instruction d'affectation. Malgré cela, la variable continue à poser des problèmes, en particulier pour l'évaluation des programmes fonctionnels. Nous illustrerons ce problème en section II.7.2 et nous présenterons des

solutions consistant à compiler le programme source pour se ramener à un code sans variable dans le chapitre IV.

Une autre façon de résoudre le problème est de travailler directement avec des langages sans variable. Dans cette section, nous présenterons tout d'abord l'origine de la programmation sans variable (la logique combinatoire) puis, nous donnerons quelques exemples de langages sans variable et enfin, nous comparerons l'outil du λ-calcul et l'outil de la logique combinatoire à savoir, la λ-expression et les combinateurs.

II.6.1 : Origine des langages sans variable : la logique combinatoire

Au début du siècle, SCHONFINKEL [SCHONF24] montra qu'il était possible d'éviter les variables quantifiées apparaissant dans les formules logiques grâce à l'utilisation d'un ensemble d'opérateurs appelés **combinateurs**. Ses travaux, repris et développés par CURRY et FEYS ont donné lieu en 1958 à la logique combinatoire [CURRY58]. La logique combinatoire est un système formel qui a vite été perçu comme une théorie des fonctions en intension, c'est-à-dire, des fonctions calculables. Les fonctions sont représentées par une combinaison de combinateurs et de symboles de constantes, un combinateur étant un opérateur défini par une règle indépendante des objets auxquels il s'applique.

De la logique combinatoire sont nés les langages sans variable. Les principes guidant la programmation sans variable sont les suivants :

- La programmation est basée sur l'utilisation de combinateurs appelés formes fonctionnelles par J.W. BACKUS et fonctionnelles par P. BELLOT dont la sémantique est donnée par une ou plusieurs règles de réduction. L'étude sémantique d'un tel langage est donc donnée directement par ces règles de réduction.

- Ces fonctionnelles permettent de combiner les fonctions primitives et/ou les fonctions définies par l'utilisateur afin de construire de nouvelles fonctions d'où la flexibilité et l'extensibilité des langages sans variable.

- Un programme est donc une combinaison de fonctions et non une suite d'instructions comme dans les langages impératifs.

- L'absence de variable dans ces langages permet de décharger complètement le programmeur de la gestion mémoire de la machine. Cette gestion est prise en charge par le langage lui-même ou plutôt par le schéma d'évaluation du langage.

- Le programmeur est ainsi déchargé de toute considération opérationnelle.

La puissance de la logique combinatoire est comparable à celle du λ-calcul. Malgré cela, peu de langages sans variable ont vu le jour. Le premier fut le langage CUCH défini par C. BOHM [BOHM64] mais il passa presque inaperçu. En 1978, J.W. BACKUS définit les systèmes FP [BACKUS78] et ce faisant, relança l'intérêt pour la programmation sans variable. Puis, P. BELLOT, ardent défenseur de la programmation sans variable [BELLOT88b] définit le langage JYM, en étendant les systèmes FP [BELLOT85]. En 1986, BACKUS & Al. ont présenté dans [BACKUS86] FL, un nouveau langage intégrant les fonctionnelles de JYM. Parallèlement, dans sa thèse d'état [BELLOT86], P. BELLOT définissait le langage GRAAL.

Dans les sections suivantes, nous présentons de façon détaillée le langage FP, qui sert de langage support à notre modèle d'évaluation parallèle présenté dans la deuxième partie de cette thèse, puis nous donnerons une brève description du langage GRAAL.

II.6.2 : Le langage FP

En 1978, BACKUS pose la question suivante : "La programmation peut-elle échapper au style von Neumann ?" A cette question, il répond par l'affirmative en définissant le langage FP. Dans FP, les variables ne sont utilisées que pour donner des noms à des fonctions définies par l'utilisateur. On peut donc dire que le langage FP fait partie de la classe des langages sans variable. D'après BACKUS, le λ -calcul est un formalisme trop complexe pour décrire les fonctions calculables dans la mesure où le programmeur n'a besoin que d'objets atomiques ou de vecteurs sur ces objets, de fonctions primitives et un ensemble de formes fonctionnelles permettant de construire d'autres fonctions.

Pour définir ce langage, BACKUS a repris quelques notions déjà définies dans le langage APL, défini par Kenneth Iverson (ce langage est décrit dans "A programming language" publié en 1962). Ainsi, on retrouve dans FP la possibilité d'appliquer un opérateur à une liste d'éléments, des opérations primitives de manipulation de listes, etc... Seulement, FP est défini de façon plus pure que APL : il existe dans APL des structures de contrôle très proches du langage BASIC ; on trouve ainsi dans APL, les notions de label, de branchement conditionnel, etc...

A l'opposé, le langage FP n'est basé que sur un ensemble d'objets, un ensemble fixé de fonctions primitives et un ensemble fixé de formes fonctionnelles. Tout programme FP est alors construit à partir de fonctions simples combinées. Le seul usage des variables est de donner des noms aux fonctions définies par l'utilisateur.

Au langage FP est associé une algèbre de programmes dont nous avons déjà parlé en section II.3, permettant de raisonner formellement sur les programmes FP.

Bien que BACKUS considère qu'un ensemble de formes fonctionnelles bien choisies est suffisant pour décrire toute sorte d'applications, il définit, toujours dans [BACKUS78], les systèmes FFP. Un système FFP est un système FP enrichi d'une règle de méta-composition

permettant la définition de nouvelles formes fonctionnelles dans le système.

Etant donné que le langage FP est le langage support de notre modèle d'évaluation parallèle, dans la suite de cette section, nous définirons, dans un premier temps, le langage FP et nous donnerons quelques exemples de programmes FP, afin de familiariser le lecteur avec ce style de programmation (si ce n'est déjà fait).

Dans un deuxième temps, nous définirons deux nouvelles formes fonctionnelles permettant essentiellement d'accroître le potentiel de parallélisme dans l'exécution de programmes FP.

II.6.2.1 : description du langage FP

Le langage FP comprend :

1- un ensemble *O* d'objets,
2- une opération (l'application),
3- un ensemble *F* de fonctions comprenant
   3.1- un ensemble *P* de fonctions primitives,
   3.2- un ensemble *FF* de formes fonctionnelles,
   3.3- un ensemble *D* de définitions.

1. L'ensemble O des objets :

Soit *A* un ensemble d'atomes (A contient les atomes T, F, $\perp$, dont la signification est donnée ci-dessous, ainsi que les nombres et l'ensemble des identificateurs différents des mots réservés du langage). Un élément *X* appartient à l'ensemble *O* si l'une des propositions suivantes est vérifiée.

- $X = \perp$; *X* est alors l'objet "indéfini".
- $X \in A$; *X* est alors un atome.
- $X = \langle x_1, ..., x_n \rangle$ tel que $\forall i \geq 1, x_i \in O$; *X* est alors une séquence.
- $X = \emptyset$; *X* est alors la séquence vide.
- X = T ou X = F ; *T* et *F* sont des atomes désignant les booléens "vrai" et "faux".

Propriété :

$< x_1,...,x_n > = \perp$ si $\exists\, i \geq 1$ tel que $x_i = \perp$.

2. L'opération : l'application

Il n'existe qu'une opération dans FP : l'application d'une fonction à un objet. Soit f une fonction et x un objet, l'application de f à x est notée $f : x$ et $f : x$ dénote l'objet résultant de l'application de f à l'objet x. Toutes les fonctions de FP sont donc des fonctions unaires. Elles s'appliquent à un objet et fournissent en résultat un seul objet.

3. L'ensemble F des fonctions.

Définition :

$F = P \cup FF \cup D$ où

P est l'ensemble des fonctions primitives,

FF est l'ensemble des formes fonctionnelles,

D est l'ensemble des définitions.

Propriété :

$\forall f \in F,\ f : \perp = \perp$.

3.1. L'ensemble P des fonctions primitives

Nous décrivons ici, les fonctions primitives définies par BACKUS dans [BACKUS78]. La notation utilisée est une variante de la notation de Mc CARTHY. Ainsi, au lieu de la notation

$$(P_1 \rightarrow e_1 ; ... ; P_n \rightarrow e_n ; T \rightarrow e_{n+1}),$$

nous écrirons

$$P_1 \rightarrow e_1 ; ... ; P_n \rightarrow e_n ; e_{n+1}$$

ayant la signification suivante :

Si P_1 alors e_1 sinon
Si P_2 alors e_2 sinon
.
.
Si P_n alors e_n sinon
e_{n+1}.

Dans les définitions suivantes, les symboles x, y, z et x_i, y_i, z_i désignent des objets. De plus, afin de différencier un symbole de fonction ou d'expression, de la fonction ou expression qu'il désigne, nous adopterons la convention suivante : un symbole de fonction sera toujours noté avec la première lettre majuscule tandis que la fonction sera toujours notée en minuscules.

Les fonctions Sélecteur

$$\forall\ s \geq 1, s : x \equiv \begin{cases} x = \langle x_1, ..., x_n \rangle\ \&\ n \geq s \rightarrow x_s ; \\ \bot \end{cases}$$

La fonction Tail

$$tl : x \equiv \begin{cases} x = \langle x_1 \rangle \rightarrow \emptyset ; \\ x = \langle x_1, ..., x_n \rangle\ \&\ n \geq 2 \rightarrow \langle x_2, ..., x_n \rangle ; \\ \bot \end{cases}$$

La fonction Identité

$$id : x \equiv x$$

La fonction Atom

$$atom : x \equiv \begin{cases} x \in A \rightarrow T ; \\ x \neq \bot \rightarrow F ; \\ \bot \end{cases}$$

La fonction Equal

$$\begin{aligned} eq : x \equiv\ & x = \langle y, z \rangle \ \&\ y = z \rightarrow T ; \\ & x = \langle y, z \rangle \ \&\ y \neq z \rightarrow F ; \\ & \bot \end{aligned}$$

La fonction Null

$$\begin{aligned} null : x \equiv\ & x = \emptyset \rightarrow T ; \\ & x \neq \bot \rightarrow F ; \\ & \bot \end{aligned}$$

La fonction Reverse

$$\begin{aligned} reverse : x \equiv\ & x = \emptyset \rightarrow \emptyset ; \\ & x = \langle x_1, ..., x_n \rangle \rightarrow \langle x_n, ..., x_1 \rangle ; \\ & \bot \end{aligned}$$

Les fonctions Distribute-from-left et Distribute-from-right

$$\begin{aligned} distl : x \equiv\ & x = \langle y, \emptyset \rangle \rightarrow \emptyset ; \\ & x = \langle y, \langle z_1, ..., z_n \rangle \rangle \rightarrow \langle \langle y, z_1 \rangle, ..., \langle y, z_n \rangle \rangle ; \\ & \bot \end{aligned}$$

$$\begin{aligned} distr : x \equiv\ & x = \langle \emptyset, y \rangle \rightarrow \emptyset ; \\ & x = \langle \langle y_1, ..., y_n \rangle, z \rangle \rightarrow \langle \langle y_1, z \rangle, ..., \langle y_n, z \rangle \rangle ; \\ & \bot \end{aligned}$$

La fonction length

$$\begin{aligned} length : x \equiv\ & x = \langle x_1, ..., x_n \rangle \rightarrow n ; \\ & x = \emptyset \rightarrow 0 ; \\ & \bot \end{aligned}$$

Les fonctions arithmétiques

$+ : x \equiv x = < y, z > \ \& \ y, z$ sont des nombres $\rightarrow y + z$;
$\bot$

$* : x \equiv x = < y, z > \ \& \ y, z$ sont des nombres $\rightarrow y * z$;
$\bot$

$- : x \equiv x = < y, z > \ \& \ y, z$ sont des nombres $\rightarrow y - z$;
$\bot$

$\div : x \equiv x = < y, z > \ \& \ y, z$ sont des nombres et $z \neq 0 \rightarrow y \div z$;
$\bot$

La fonction Transpose

Trans : $x \equiv x = < \emptyset, ..., \emptyset > \rightarrow \emptyset$;
$x = < x_1, ..., x_n > \rightarrow < y_1, ..., y_m >$;
$\bot$

où $x_i = < x_{i1}, ..., x_{im} >$ et $y_j = < x_{1j}, ..., x_{nj} >$, $1 \leq i \leq n$, $1 \leq j \leq m$.

Les fonctions booléennes : And, Or, Not

and : $x \equiv x = < T, T > \rightarrow T$;
$x = < T, F > \rightarrow F$;
$x = < F, T > \rightarrow F$;
$x = < F, T > \rightarrow F$;
$\bot$

etc...

Les fonctions Append-left et Append-right

apndl : x ≡ x = < y, Ø > → < y > ;
x = < y, < z_1, ..., z_n > > → < y, z_1 , ..., z_n > ;
⊥

apndr : x ≡ x = < Ø, z > → < z > ;
x = < < y_1, ..., y_n >, z > → < y_1, ..., y_n, z > ;
⊥

Les fonctions Right-selectors et Right-tail

1r : x ≡ x = < x_1, ..., x_n > → x_n ;
⊥

2r : x ≡ x = < x_1, ..., x_n > & n ≥ 2 → x_{n-1} ;
⊥

etc...

tlr : x ≡ x = < x_1 > → Ø ;
x = < x_1, ..., x_n > & n ≥ 2 → < x_1, ..., x_{n-1} > ;
⊥

3.2 : L'ensemble FF des formes fonctionnelles

Les formes fonctionnelles sont des expressions permettant de combiner des fonctions entre elles afin d'en constituer de nouvelles. Les fonctions combinées seront alors appelées les paramètres de la forme fonctionnelle. BACKUS définit les formes fonctionnelles suivantes:

La composition :

$(f \circ g) : x \equiv f : (g : x)$

Les fonctions f et g sont alors les paramètres de la composition.

La construction :

$[f_1, ..., f_n] : x \equiv < f_1 : x, ..., f_n : x >$

La condition :

$$(p \rightarrow f ; g) : x \equiv (p : x) = T \rightarrow f : x ;$$
$$(p : x) = F \rightarrow g : x ;$$
$$\perp$$

La forme constante :

$$\overline{x} : y \equiv y = \perp \rightarrow \perp ;$$
$$x$$

Le paramètre de cette forme fonctionnelle est donc un objet.

L'insert :

$$/f : x \equiv x = < x_1 > \rightarrow x_1 ;$$
$$x = < x_1, ..., x_n > \ \& \ n \geq 2 \rightarrow f : < x_1, /f : < x_2, ..., x_n > > ;$$
$$\perp$$

L'Apply-to-all :

$$\alpha f : x \equiv x = \emptyset \rightarrow \emptyset ;$$
$$x = \langle x_1, ..., x_n \rangle \rightarrow \langle f : x_1, ..., f : x_n \rangle ;$$
$$\perp$$

Binary-to-unary :

$$(bu\ f\ x) : y \equiv f : \langle x, y \rangle$$

La forme fonctionnelle *Bu* a donc comme paramètres une fonction et un objet.

While :

$$(while\ p\ f) : x \equiv p : x = T \rightarrow (while\ p\ f) : (f : x) ;$$
$$p : x = F \rightarrow x ;$$
$$\perp$$

3.3 : L'ensemble D des définitions

Une définition est une fonction définie par l'utilisateur. Elle est de la forme :

Def L = r où,

L est un nouveau symbole de fonction et *r* est une forme fonctionnelle pouvant dépendre de *L*.

Un programme FP est donc constitué d'un ensemble de définitions.

Exemple 1 :

Def last ≡ null o tl → 1 ; last o tl

Cette définition permet de définir une nouvelle fonction *Last* qui, appliquée à une séquence, fournit en résultat le dernier élément de cette séquence.

L'application de la fonction *Last* à une séquence $X = \langle 1, 2 \rangle$ est réalisée de la façon suivante :

last : X ≡ last : < 1, 2 >
≡ (null o tl → 1 ; last o tl) : < 1, 2 >
≡ (last o tl) : < 1, 2 > car
(null o tl) : < 1, 2 > ≡ null : < 2 > ≡ F
≡ last : < 2 >
≡ (null o tl → 1 ; last o tl) : < 2 >
≡ 1 : < 2 > car
(null o tl) : < 2 > ≡ null : Ø ≡ T
≡ 2

Exemple 2 :

Def IP ≡ (/ +) o (α *) o trans

Cette fonction permet de réaliser le produit vectoriel.
Soit X = < < 1, 2, 5 > , < 4, 3, 0 > >,

IP : X ≡ IP : < < 1, 2, 5 >, < 4, 3, 0 > >
≡ (/ +) o (α *) o trans : < < 1, 2, 5 >, < 4, 3, 0 > >
≡ (/ +) o (α *) : < < 1, 4 >, < 2, 3 >, < 5, 0 > >
≡ (/ +) : < * : < 1, 4 >, * : < 2, 3 >, * : < 5, 0 > >
≡ (/ +) : < 4, 6, 0 >
≡ + : < 4, / + : < 6, 0 > >
≡ + : < 4, + : < 6, / + : < 0 > > >
≡ + : < 4, + : < 6, 0 >>
≡ + : < 4, 6 >
≡ 10

Exemple 3 :

Def MM = (α α IP) o (α distl) o distr o [1, trans o 2]

Cette fonction définit la multiplication de deux matrices quelconques, *IP* étant la fonction définie dans l'exemple 2.

Supposons que nous voulions réaliser la multiplication des matrices *A* et *B* définies par :

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 4 \\ 1 & 0 \end{bmatrix}$$

A la matrice *A* correspond la séquence *A* = < <*1*, *2*>, <*0*, *3*> > et à la matrice *B* correspond la séquence *B* = < <*2*, *4*>, <*1*, *0*> >.

MM : < A, B >
≡ (α α IP) o (α distl) o distr o [1, trans o 2] : < A, B >
≡ (α α IP) o (α distl) o distr : < 1 : < A, B >, trans o2 : < A, B > >

1 : < A, B >
≡ 1 : < <<1, 2>, <0, 3>>, <<2, 4>, <1, 0>> >
≡ <<1, 2>, <0, 3>>

trans o 2 : < A, B >
≡ trans : B
≡ trans : <<2, 4>, <1, 0>>
≡ <<2, 1>, <4, 0>>

≡ (α α IP) o (α distl) o distr :
< <<1, 2>, <0, 3>>, <<2, 1>, <4, 0>> >
≡ (α α IP) o (α distl) :
< <<1, 2>, <<2, 1>, <4, 0>> >,
<<0, 3>, <<2, 1>, <4, 0>> > >

$$\equiv (\alpha\,\alpha\,IP) : < distl : < < 1,2 >, << 2,1 >, < 4,0 >> >,$$
$$distl : < < 0,3 >, << 2,1 >, < 4,0 >> > >$$
$$\equiv (\alpha\,\alpha\,IP) : < < << 1,2 >, < 2,1 >>, << 1,2 >, < 4,0 >> >,$$
$$<< < 0,3 >, < 2,1 >>, << 0,3 >, < 4,0 >> > >$$
$$\equiv < \alpha\,IP : << < 1,2 >, < 2,1 >>, << 1,2 >, < 4,0 >> >,$$
$$\alpha\,IP : << < 0,3 >, < 2,1 >>, << 0,3 >, < 4,0 >> > >$$
$$\equiv << IP : << 1,2 >, < 2,1 >>, IP : << 1,2 >, < 4,0 >> >,$$
$$< IP : << 0,3 >, < 2,1 >>, IP : << 0,3 >, < 4,0 >> > >$$
$$\equiv << 4,4 >, < 3,0 >>$$

Exemple 4 :

Def fact $\equiv$ eq0 $\rightarrow \overline{1}$; $*$ o [id, fact o sub1]

Def eq0 $\equiv$ eq o [id, $\overline{0}$]

Def sub1 $\equiv$ - o [id, $\overline{1}$]

Fact représente la fonction factorielle ;

fact : 3 $\equiv$ (eq0 $\rightarrow \overline{1}$; $*$ o [id, fact o sub1]) : 3

eq0 : 3 $\equiv$ eq o [id, $\overline{0}$] : 3

$\equiv$ eq : < id : 3, $\overline{0}$: 3 >

$\equiv$ eq : < 3, 0 >

$\equiv$ F

$\equiv$ $*$ o [id, fact o sub1] : 3

$\equiv$ $*$: < id : 3, fact o sub1 : 3 >

sub1 : 3 $\equiv$ - o [id, $\overline{1}$] : 3

$\equiv$ - : < id : 3, $\overline{1}$: 3 >

$\equiv$ - : < 3, 1 >

$\equiv$ 2

$\equiv$ $*$: < 3, fact : 2 >

$\equiv$ $*$: < 3, (eq0 $\rightarrow \overline{1}$; $*$ o [id, fact o sub1]) : 2 >

$\equiv$ $*$: < 3, $*$: < 2, fact : 1 > >

$\equiv$ $*$: < 3, $*$: < 2, $*$: < 1, fact : 0 > > >

$\equiv$ $*$: < 3, $*$: < 2, $*$: < 1, 1 > > >

$\equiv$ $*$: < 3, $*$: < 2, 1 > >

$\equiv$ $*$: < 3, 2 >

$\equiv$ 6

II.6.2.2 : extension du langage

Nous ajoutons au langage défini par BACKUS, deux nouvelles formes fonctionnelles qui permettront soit d'introduire du parallélisme dans les traitements, soit d'optimiser ces traitements.

La forme fonctionnelle Tree :

Cette forme fonctionnelle a été introduite par J.H. WILLIAMS dans [WILLIA82]. Elle est définie par :

$$
\begin{aligned}
&(\text{tree } f) : x \equiv\ x = \langle x_1 \rangle \rightarrow x_1 ; \\
&\qquad\qquad x = \langle x_1, \ldots, x_n \rangle \rightarrow \\
&\qquad\qquad\qquad f : \langle (\text{tree } f) : \langle x_1, \ldots, x_m \rangle, (\text{tree } f) : \langle x_{m+1}, \ldots, x_n \rangle \rangle \\
&\qquad\qquad \text{avec } m = n \text{ DIV } 2 \text{ où } DIV \text{ désigne la division entière;} \\
&\qquad\qquad \perp
\end{aligned}
$$

Si f est une fonction associative, alors les expressions *(tree f) : x* et *(insert f) : x* dénotent le même objet résultat. Malgré cela, il y a un intérêt réel à ajouter cette forme fonctionnelle à celles définies par BACKUS. Alors que la forme fonctionnelle *Insert* conduit à une exécution complètement séquentielle de l'expression, la forme fonctionnelle *Tree* permet au contraire d'évaluer certaines expressions en parallèle.

Exemple :

Les fonctions *Add1* et *Add2* définies ci-dessous permettent toutes les deux d'effectuer la somme des éléments de la séquence sur laquelle elles s'appliquent.

Def add1 ≡ (insert +)
Def add2 ≡ (tree +)

Soit X = < 1, 2, 3, 4, 5, 6, 7, 8 >. L'application de la fonction *Add1* à la séquence *X* conduit à l'évaluation de l'expression E_1 :

E1 = + : < 1, + : < 2, + : < 3, + : < 4, + : < 5, + : < 6, + : < 7, 8 > > > > > > >.

L'expression *E1* peut être représentée par l'arbre de la figure II.2-a.

L'application de la fonction *Add2* à la séquence *X* conduit à l'évaluation de l'expression *E2*.

$$E2 = + : < (\text{tree } +) : < 1, 2, 3, 4 >, (\text{tree } +) : < 5, 6, 7, 8 >>$$
$$= + : < + : < + : < 1, 2 >, + : < 3, 4 >>, + : < + : < 5, 6 >, + : < 7, 8 >>>$$

L'expression *E2* est représentée par l'arbre de la figure II.2-b.

La figure II.2-a montre que l'expression *(insert f)* appliquée à une séquence de longueur *n* nécessite *(n - 1)* "pas d'exécution" tandis que la figure II.2-b montre qu'avec la forme fonctionnelle *(tree f)*, le même traitement peut-être fait en $log_2 n$ "pas d'exécution" grâce au parallélisme introduit (les sous-arbres peuvent être traités en parallèle).

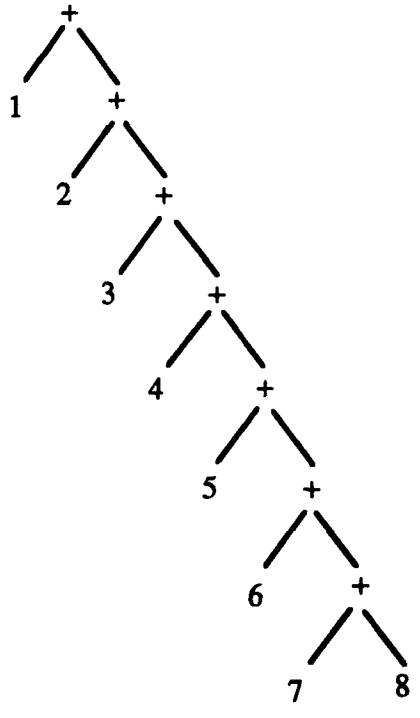


Figure II.2-a : arbre résultant de add1 : <1,2,3,4,5,6,7,8 >

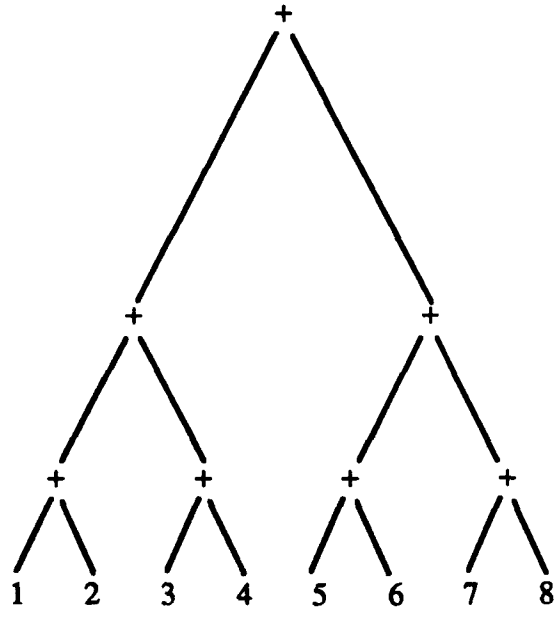


Figure II.2-b : arbre résultant de add2 : < 1,2,3,4,5,6,7,8 >

Figure II.2 : Comparaison des fonctions *(insert +)* et *(tree +)* sur un exemple

La forme fonctionnelle Distribution

Nous introduisons une nouvelle forme fonctionnelle *Distribution* définie par :

$$\{ f_1,..., f_n \} : x \equiv x = \langle x_1, ..., x_n \rangle \rightarrow \langle f_1 : x_1, ..., f_n : x_n \rangle ;$$
$$\bot$$

Cette forme fonctionnelle peut être obtenue à partir des formes fonctionnelles définies par BACKUS de la façon suivante :

$$\{ f_1,..., f_n \} \equiv [f_1 \circ 1 , ... , f_n \circ n].$$

Une construction conduit bien à une évaluation parallèle : dans la mesure où l'on dispose de *n* exemplaires de la séquence argument, les fonctions paramètres de la construction peuvent effectivement être appliquées en parallèle aux objets composant la séquence. Seulement, l'évaluation d'une construction nécessite la mise en oeuvre d'un mécanisme permettant d'obtenir *n* exemplaires de la même séquence, ce qui peut être long et coûteux. L'intérêt de cette nouvelle forme fonctionnelle permet dans ce cas de se passer de ce mécanisme tout en conservant la possibilité d'effectuer les traitements en parallèle.

II.6.3 : Le langage GRAAL

Le langage GRAAL est un langage fonctionnel, basé sur la logique combinatoire et les principes de programmation sans variable développés par BACKUS.

Dans les langages fonctionnels classiques, l'opération d'application est binaire : une fonction s'applique à un argument. Il existe deux principales méthodes pour rendre monadique, une fonction naturellement polyadique. La plupart des lambda-langages utilisent la Curryfication :

naturellement polyadique. La plupart des lambda-langages utilisent la Curryfication :

Etant donnée une fonction *f* à *n* arguments :

$f(x_1, x_2, \ldots, x_n)$,

on définit une fonction

$F = \lambda x_1. (\lambda x_2. (\ldots (\lambda x_n. f(x_1, x_2, \ldots, x_n)) \ldots))$.

On a alors la propriété suivante :

$F\ a_1\ a_2 \ldots an = ((\ldots ((F\ a_1)\ a_2) \ldots)\ a_n) = f(a_1, a_2, \ldots, a_n)$

L'inconvénient de cette méthode est que lors de l'évaluation, on est amené à gérer de multiples applications inutiles (pour passer de $F\ a_1\ a_2 \ldots a_n$ à $f(a_1, a_2, \ldots, a_n)$).

La deuxième méthode est celle utilisée en particulier par les langages LISP et FP, qui utilise la structuration des arguments. Par exemple, la fonction *addition* qui s'applique à deux arguments est écrite en FP sous la forme de la fonction *plus* appliquée à une séquence constituée des deux objets argument (LISP utilise la structure de liste au lieu de la structure de séquence). P. BELLOT dans [BELLOT88b] et [BELLOT86] reproche à cette solution le fait qu'elle nécessite la création d'une structure (par exemple, la création d'une séquence en FP et d'une liste en LISP), et ce, à chaque application.

La principale différence entre GRAAL et les autres langages fonctionnels est l'utilisation d'une application réellement polyadique : l'application d'une fonction *f* à *n* arguments s'écrit $f : a_1 \ldots a_n$ et rien n'est supposé quant à la structuration des arguments $a_1 \ldots a_n$. C'est à l'implanteur du langage que revient la tâche de représenter ces arguments.

On peut noter à ce sujet que, bien que FP propose une structuration des arguments en séquence, BACKUS ne fait aucune hypothèse quant à la représentation interne de ces séquences (à la différence de LISP dans lequel une liste est définie par son premier élément suivi "du reste de la liste"). L'implanteur du langage est donc également libre d'adopter la représentation interne qu'il désire. GRAAL se différencie également de FP par le fait que de nouvelles formes fonctionnelles peuvent être

peut devenir le nom d'une forme fonctionnelle définie, il est alors lui-même un combinateur défini.

Signalons enfin que le langage GRAAL a servi de base à la définition du langage GREL défini par R. LEGRAND [LEGRAN88]. GREL résulte de l'insertion des relations dans GRAAL ; il est basé sur la notion de relation binaire. GREL définit les relations à partir des fonctions de GRAAL, à l'aide d'une forme *Union*. Alors qu'une fonction ne rend qu'un seul résultat, une relation calcule la famille des résultats correspondant à chacun des choix effectués. GREL permet donc de généraliser les fonctions en y incorporant l'indéterminisme.

II.7 : Comparaison entre la λ-expression et les combinateurs

La lambda-expression et les combinateurs se différencient tout d'abord par le style de programmation qu'ils engendrent : il n'y a pas un style de programmation fonctionnelle, au contraire, le choix d'un langage influence le style de programmation. Nous illustrerons ceci dans la première partie de cette section.

La différence majeure entre ces deux outils se situe au niveau de la complexité de leur interprétation : les combinateurs sont beaucoup plus simples à interpréter que les lambda-expressions, ceci étant essentiellement dû au fait que les lambda-expressions sont plus "riches". Dans la deuxième partie, nous exposerons donc les problèmes inhérents à la lambda-expression tandis que dans la dernière partie, nous montrerons comment les combinateurs résolvent en partie ces problèmes.

II.7.1 : Deux styles de programmation différents

Cette section fait référence à l'article de J.H. WILLIAMS [WILLIA82], dans lequel il compare ce qu'il appelle le "style lambda" (c'est-à-dire le style de programmation induit par les lambda-langages) et le "style FP" (c'est-à-dire le style de programmation induit par un langage sans variable tel que FP).

Dans les lambda-langages, une fonction s'applique à un objet et fournit en résultat un autre objet.

Prenons l'exemple d'une fonction *LENGTH* permettant de calculer la longueur d'une liste (ou d'une séquence). Un lambda-langage nous donnerait la définition suivante :

$$\text{LENGTH} = \lambda x.\ (\text{if null}(x) \text{ then } 0 \text{ else } 1 + \text{LENGTH}(\text{tail}(x))).$$

Ainsi, la longueur d'une liste x est obtenue en construisant l'objet résultant de l'application de la fonction *Null* à l'objet x et selon cet objet, en retournant soit *0*, soit l'objet résultant de l'addition de la constante *1* et de la longueur de la liste privée de son premier élément.

Une lambda-expression de cette forme peut directement être traduite dans un langage tel que FP en combinant des fonctions à l'aide de formes fonctionnelles pour produire de nouvelles fonctions, ce qui donne

$$\text{Def LENGTH} = \text{null} \rightarrow \bar{0}\ ;\ + \circ [\bar{1}, \text{LENGTH} \circ \text{tail}]$$

Remarque :

$[\bar{1}, \text{LENGTH} \circ \text{tail}]$ dénote une fonction (et non un objet) résultant d'une construction ayant pour argument la fonction *Constante 1* et la composition des fonctions *Tail* et *LENGTH*.

Seulement, une fonction permettant de calculer la longueur d'une liste peut également être obtenue en considérant que chaque élément de la liste vaut 1 (c'est-à-dire en appliquant la fonction *Constante 1* à

chacun des éléments de la liste) puis en additionnant ces éléments, ce qui donne :

Def LENGTH = /+ ∘ $\alpha\bar{1}$

De même, la fonction MEMBER permettant de déterminer si un élément appartient à une liste peut s'écrire dans un "style lambda" en FP de la façon suivante :

Def MEMBER = null ∘ 2 → $\bar{F}$;
eq ∘ [1, 1 ∘ 2] → $\bar{T}$;
MEMBER ∘ [1, tail ∘ 2]

Cette même fonction MEMBER peut être définie dans un "style FP" par

Def MEMBER = /OR ∘ αeq ∘ distl

Grâce à l'utilisation de formes fonctionnelles telles que *Apply-to-all*, le "style FP" permet de décrire des traitements de façon parallèle. Dans la fonction *LENGTH*, ($\alpha\bar{1}$) dénote l'application simultanée de la fonction *Constante 1* à chacun des éléments de la séquence argument, il en est de même pour la fonction *(α eq)* dans la définition de la fonction *MEMBER*. Il est à noter que le langage LISP permet de programmer dans un "style FP" grâce à l'utilisation d'opérateurs tels que MAPCAR et APPLY. Seulement, la plupart du temps, le "style lambda" est plus utilisé par les programmeurs car il est interprété de façon plus efficace.

De ces exemples, nous pouvons déduire un certain nombre de conclusions quant à la complexité de la λ-expression par rapport à la simplicité des combinateurs.

II.7.2 : La complexité de la λ-expression [WILLIA82], [BELLOT88b], [BELLOT88a]

La sémantique opérationnelle d'une lambda-expression est difficile à établir dans la mesure où une lambda-expression est l'expression d'un calcul et non un moyen de calculer le résultat (i.e. : un processus de calcul). Ceci est illustré par la fonction *LENGTH* définie dans le "style lambda" : cette fonction est définie comme une équation que le programme doit satisfaire tandis que la même fonction écrite dans un "style FP" donne ce que J.H. WILLIAMS appelle une représentation directe du programme.

Le seul moyen de calculer un résultat dans une λ-expression est d'effectuer des substitutions. P. BELLOT pense que "c'est l'une des raisons de l'inefficacité toute relative des langages fonctionnels actuels" car "le fossé est large entre la spécification donnée par la lambda-expression et le processus de calcul".

Lorsque l'on raisonne sur une λ-expression et que l'on veut effectuer une substitution, on fait un parcours de la λ-expression et on reconstruit une nouvelle λ-expression.

Exemple :

Etant donnée une expression $((\lambda x.\ E)\ y)$, l'application est réalisée par substitution. On veut ainsi obtenir $[x/y]E$, ce qui revient à parcourir l'expression E pour substituer à chaque occurrence de la variable liée x, la variable y.

Ceci n'est pas envisageable dans la pratique, ce qui fait que dans les implantations de lambda-langages, c'est une simulation de la β-réduction qui est effectuée : dans l'exemple précédent, la λ-expression $(\lambda x.\ E)$ est évaluée dans un contexte contenant l'information "x à la valeur y". Les contextes sont gérés différemment selon les implantations : on peut soit utiliser une pile de liaisons (dans le cas de liaison profonde) soit une simple variable (dans le cas de liaison superficielle), soit une information sur l'état du calcul (c'est le cas de l'implantation de ML sur la machine

CAM) [COUSIN86], [MAUNY86]. Cette notion de contexte pose plusieurs problèmes : tout d'abord les problèmes classiques, présents dans toute gestion d'environnement (cohérence, mise à jour...etc). De plus, le résultat d'une application peut être une λ-expression (i.e. : un résultat fonctionnel).

Exemple :

$(\lambda x.\ (\lambda y.\ (+\ x\ y\ ))\ 3) = (\lambda y.\ (+\ 3\ y))$

Dans une implantation, le résultat de cette λ-expression serait donc la λ-expression $(\lambda y.\ (+\ x\ y))$ accompagnée du contexte "x vaut 3". Ainsi, un résultat fonctionnel n'est pas évalué. Ceci s'appelle une fermeture.

Les fermetures permettent ainsi la manipulation de fonctions d'ordre supérieur, ce qui contribue à rendre LISP plus puissant que FP. Ce gain de puissance s'accompagne d'une difficulté d'implantation qui a obligé les premiers programmeurs LISP à demander explicitement les calculs de fermeture. Dans les versions les plus récentes de LISP ou de dialectes de LISP tels que SCHEME [STEELE78], ainsi que dans les machines SECD (décrite par HENDERSON dans [HENDER 80]) et CAM (citée ci-dessus), les fermetures sont calculées automatiquement.

II.7.3 : La simplicité des combinateurs

Dans les langages sans variable, la construction d'une fonction est réalisée par combinaison de fonctions à l'aide de combinateurs (ou fonctionnelles ou formes fonctionnelles). La sémantique opérationnelle d'une combinaison est directe et simple. Par exemple, la composition de fonctions $(f \circ g)$ signifie appliquer g puis f et ce, indépendamment de tout contexte (du fait de l'absence de variables). Les combinateurs jouent ainsi le rôle de structures de contrôle fonctionnelles. Ils permettent de structurer les programmes (dans les λ-langages, la seule structure existante est l'application) et de concevoir ces programmes comme une combinaison de programmes jusqu'à atteindre des programmes élémentaires (i.e. : les fonctions primitives). La sémantique

opérationnelle d'une fonction est alors donnée par un ensemble de règles de réduction très simples (les règles de réduction des combinateurs) au lieu d'une règle complexe de β-réduction dans les lambda-langages. La mise en oeuvre des langages sans variable s'en trouve donc simplifiée : ils sont directement exécutables.

De plus, une bonne utilisation des combinateurs conduit à des programmes concis et efficaces, ceci est illustré par les deux exemples de la section précédente.

Restons objectifs, il faut quand même avouer que les programmes sans variable sont moins lisibles que les λ-expressions, surtout pour les programmeurs non familiers avec ce style de programmation. De plus, rappelons que dans le cas particulier de FP, la simplicité est obtenue au dépend d'une perte de puissance puisque FP ne permet pas la manipulation de fonctions d'ordre supérieur.

III : PRINCIPES DE MISE EN OEUVRE

DES LANGAGES FONCTIONNELS

III : PRINCIPES DE MISE EN OEUVRE DES LANGAGES FONCTIONNELS

Les différentes techniques de mise en oeuvre des langages fonctionnels se différencient par le mode d'évaluation (ou mode de contrôle) et le schéma d'évaluation adoptés.

Le mode d'évaluation détermine la façon dont les arguments d'une fonction sont évalués. Il en existe quatre principaux qui sont le passage par valeur, le passage par nom, l'évaluation paresseuse et la substitution complète, que nous présenterons en première partie de ce chapitre.

Le schéma d'évaluation (ou modèle d'exécution) détermine une stratégie d'évaluation. Il existe deux modèles principaux qui sont le modèle data-flow et le modèle de réduction, qui se prêtent particulièrement à des implantations sur des architectures multi-processeurs. Dans la deuxième partie de ce chapitre, nous présenterons ces deux modèles en décrivant pour chacun d'eux leur stratégie d'évaluation, leurs caractéristiques et en donnant quelques exemples de machines adoptant ces schémas d'évaluation (i.e. : les machines data-flow et les machines de réduction). D'autres modèles seront présentés à la suite.

Les différentes implantations parallèles de langages fonctionnels se différencient par la "granularité" du parallélisme mis en oeuvre. Le choix de la granularité est un problème difficile dans la mesure où un faible degré provoque une perte de parallélisme tandis qu'un fort degré peut à l'inverse engendrer trop de parallélisme nécessitant alors des mécanismes de contrôle du parallélisme mis en oeuvre. Nous consacrerons les deux dernières parties de ce chapitre à ces problèmes.

Une description des différents modèles d'évaluation, et des architectures multi-processeurs dédiées à l'exécution de langages fonctionnels est donnée dans [TRELEA82] et [VEGDAH84].

III.1 : Les différents modes d'évaluation

III.1.1. Le passage par valeur

Le mécanisme de passage par valeur consiste à évaluer les paramètres effectifs d'une fonction avant que ceux-ci ne lui soient transmis. Le plus grand avantage de ce mécanisme de passage est que les arguments d'une fonction ne sont évalués qu'une seule fois.

Seulement, on peut être amené à évaluer des arguments non nécessaires à l'évaluation de la fonction et faire ainsi des calculs inutiles. Par exemple, tous les arguments d'une conditionnelle seront évalués alors que deux seulement sont nécessaires. Le plus grand inconvénient de ce mode de passage est que si l'un des arguments n'est pas défini ou s'il conduit à une évaluation infinie, aucun résultat ne pourra être obtenu.

Exemple :

$(\lambda xy.\ 3 + y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))\ 6$

Cette λ-expression dont l'évaluation devrait donner 9, ne donnera aucun résultat si elle est évaluée avec le mécanisme de passage par valeur : le paramètre effectif $((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$ sera évalué bien qu'étant inutile et son évaluation conduisant à un traitement infini, aucun résultat ne sera fourni.

Par extension, ce mécanisme ne supporte donc pas le traitement de structures de données infinies.

Etant donné ce mécanisme, l'appel par valeur parallèle consiste à évaluer simultanément toutes les fonctions les plus internes dans une expression (évaluation "innermost"). Le problème de ce mécanisme est qu'alors, trop de parallélisme peut être généré.

III.1.2 : Le passage par nom

Le mécanisme de passage par nom consiste à transmettre ses paramètres à une fonction avant que ceux-ci ne soient évalués. Ils ne sont alors calculés que lorsqu'ils interviennent dans une expression. Ainsi, l'évaluation de la λ–expression de l'exemple précédent, avec un mécanisme de passage par nom fournirait effectivement 9 comme résultat. Ce mécanisme permet donc d'éviter certains traitements infinis, en évitant l'évaluation des arguments non nécessaires.

L'inconvénient majeur de ce mécanisme est que l'évaluation des paramètres effectifs est réalisée autant de fois qu'il y a de paramètres formels associés (i.e. : quand un argument est utilisé plusieurs fois, il est réévalué). De plus, la mise en oeuvre de ce mécanisme est plus coûteuse (que la mise en oeuvre de l'appel par valeur) car elle nécessite la gestion d'un grand nombre d'environnements.

Le mécanisme d'appel par nom parallèle consiste à évaluer simultanément toutes les expressions réductibles disjointes les plus externes (évaluation "outermost"). Ce processus est correct quelle que soit la sémantique attribuée au langage.

Un mécanisme d'appel par nom optimisé, appelé l'appel par nécessité, permet d'évaluer les arguments au plus une fois. Cette technique est basée sur l'utilisation de pointeurs permettant de partager l'accès aux arguments au lieu de l'utilisation de copies de l'argument dans l'appel par nom. Il combine donc les avantages de l'appel par nom et l'appel par valeur : les paramètres sont communiqués selon le mécanisme de passage par nom mais une fois calculées, leurs valeurs sont conservées, transformant les transmissions ultérieures des mêmes paramètres en technique passage par valeur.

III.1.3 : L'évaluation paresseuse

L'évaluation paresseuse est inspirée de l'appel par nécessité mais les mécanismes de l'appel par nécessité sont appliqués de manière plus fine : lorsque le paramètre d'une fonction est une structure, seuls les éléments de la structure nécessaires à l'évaluation de la fonction sont effectivement calculés. L'évaluation paresseuse permet donc la manipulation de listes potentiellement infinies.

III.1.4 : La substitution complète

La substitution complète consiste à exécuter simultanément tout ce qui est exécutable. Elle pose de gros problèmes de mise en oeuvre dans la mesure où elle génère un parallélisme massif qui doit être freiné : le nombre de processus actifs devient vite trop important et doit alors être limité.

III.1.5 : Solutions hybrides

D'autres solutions tentent de combiner les avantages de l'évaluation paresseuse et de l'appel par valeur : l'appel par valeur est plus facile à mettre en oeuvre que l'évaluation paresseuse et de ce fait plus efficace dans le cas où les fonctions à évaluer sont strictes (i.e. : tous les arguments sont nécessaires à l'évaluation). Ainsi, l'évaluation des fonctions strictes utilise le mode d'appel par valeur et l'évaluation des autres fonctions utilise le mécanisme paresseux. Le problème consiste alors à déterminer quelles sont les fonctions strictes.

Des méthodes telles que celle proposée par MYCROFT [MYCROF81] permettent de repérer automatiquement et de façon statique les fonctions strictes (dans les cas où c'est décidable), d'autres demandent au programmeur de préciser explicitement quand il désire que l'appel par valeur soit mis en oeuvre.

III.2 : Les modèles d'évaluation de langages fonctionnels

III.2.1 : Le modèle data-flow

III.2.1.1 : Principe

Dans le modèle data-flow, les programmes sont décrits en termes de graphes dirigés illustrant le flot de données entre les opérations : les noeuds du graphe représentent les opérateurs jouant le rôle d'acteurs et les arcs illustrent la dépendance par données entre deux opérateurs : ils sont chargés de transporter des données d'un acteur à un autre acteur.

Une opération est exécutable quand tous ses arguments sont disponibles c'est-à-dire quand une marque de donnée est présente sur chaque arc d'entrée du noeud opérateur correspondant. L'exécution de cette opération consiste alors à

1- prendre les valeurs sur les arcs d'entrée,
2- traiter ces valeurs en fonction de l'opérateur contenu dans le noeud,
3- délivrer la valeur résultat sur l'arc en sortie.

Dans le modèle data-flow, c'est donc le flot de données entre les noeuds opérateurs qui constitue la séquence de contrôle lors de l'exécution. Les résultats partiels d'une évaluation ne sont pas mémorisés mais sont transmis du noeud producteur au noeud consommateur.

Dans le modèle data-flow pur, le mode d'évaluation utilisé est donc l'appel par valeur. Les avantages et inconvénients de ce mode d'évaluation se retrouvent alors dans le modèle.

Afin de supprimer certains inconvénients de l'appel par valeur, certains modèles proposent d'incorporer une sémantique "demand-driven" dans les graphes data-flow. L'exécution est toujours contrôlée

par la présence d'opérandes mais les résultats partiels nécessaires à une évaluation sont demandés. En pratique, cela signifie que des arcs de demande sont insérés dans les graphes data-flow, qui véhiculent les demandes en sens inverse des données. On obtient alors un "graphe paresseux" qui ne réalisera pas de traitements inutiles. Le problème de cette solution est que l'insertion des arcs de demande introduit un "overhead" supplémentaire dû à la propagation des marques de demande. Y.H. WEI et J.L. GAUDIOT proposent dans [WEI88] une solution permettant de supprimer, à la compilation, les propagations de demande inutiles.

D'autres modèles data-flow avec propagation de demande ont été définis par Amamiya & Al. [AMAMIY84], Ono & Al. [ONO86], Anderson & Berman [ANDERS87]. Le lecteur intéressé par les modèles et architectures data-flow en général, pourra se référer à deux articles de synthèse sur le sujet, qui sont [SRINI86] et [VEEN86] .

III.2.1.2 : Caractéristiques du modèle data-flow

Dans le modèle data-flow, l'exécution d'une instruction étant décidée par un critère local (présence des opérandes sur les arcs d'entrée d'un noeud), les opérations sont effectuées de façon tout à fait asynchrone. De plus, l'effet de chaque opération est limité à la production d'un résultat devant être consommé par un nombre donné d'autres acteurs (ceci sous-entend bien l'absence d'effets de bord). La fonctionnalité de chaque opération est ainsi clairement définie.

Le grand avantage du modèle data-flow est qu'il permet l'exploitation d'un parallélisme massif. L'inconvénient majeur étant qu'une opération peut attendre longtemps (éventuellement indéfiniment) après un argument non nécessaire à son évaluation (dans les modèles data-flow "purs").

III.2.1.3 : Exemples de machines data-flow

Nous présentons dans cette section, quelques machines data-flow, afin d'illustrer comment peut être implanté le modèle data-flow. Nous ne détaillerons pas l'architecture des différentes machines mais nous essaierons de dégager leurs particularités.

La machine du MIT (Massachusetts Institute of Technology) a été conçue par DENNIS & Al., [DENNIS79], [DENNIS84]. Dans cette machine, le graphe data-flow est réparti sur un ensemble de cellules. Lorsqu'une instruction est exécutable, elle est transmise à un processeur via un réseau appelé "réseau arbitre". Le processeur évalue cette expression et délivre un résultat qui servira à mettre à jour des cellules *instruction* via un réseau de distribution. Les cellules *instruction* mises à jour peuvent éventuellement devenir exécutables auquel cas le même processus est appliqué. Les inconvénients de cette machine sont son inadaptation à traiter des structures dans la mesure où les "marques" de données véhiculées sont de longueur fixe, et son impossibilité de traiter des appels de fonctions récursives de profondeur quelconque.

La machine de Manchester [WATSON79], [WATSON82], [GURD85a] est proche de la machine du MIT mais elle permet les appels récursifs en autorisant de multiples activations d'une même cellule *instruction*. Le problème de cette machine, qui est un problème en général commun à toutes les machines data-flow est la difficulté de contrôler le parallélisme lorsqu'il devient trop important.

Les particularités de la machine de Texas Instruments [JOHNSO80] sont sa structure en anneau et la répartition statique (à la compilation) du graphe sur les processeurs.

Toutes ces machines mettent en oeuvre le modèle data-flow pur. La machine conçue en Utah [DAVIS78], dont le réseau est structuré en arbre, est un exemple d'implantation du modèle data-flow avec une sémantique "demand-driven".

Pour conclure sur le modèle data-flow, nous pouvons remarquer que malgré le parallélisme massif qu'il permet de mettre en oeuvre, il souffre d'un manque de souplesse rendant difficile (et parfois impossible) la manipulation de fonctions récursives, de structures de données complexes et l'utilisation de schémas d'évaluation tels que l'évaluation paresseuse.

III.2.2 : Le modèle de réduction

III.2.2.1 : Principe et caractéristiques.

Dans le modèle de réduction, une expression est évaluée en la transformant, par applications successives de règles de réécriture. Une expression est réduite lorsqu'aucune règle ne peut être appliquée. Il existe deux sortes de règles : les règles de simplifications, qui permettent d'appliquer une fonction primitive à ses arguments et les règles de substitution, qui permettent de remplacer l'appel d'une fonction par sa définition.

En réduction, un programme est mathématiquement équivalent à son résultat : les opérations de base étant l'application de règles de réécriture, l'expression initiale et l'expression finale (i.e. : l'expression réduite) sont deux formes différentes de la même expression.

Le modèle de réduction se caractérise donc par les propriétés suivantes :

- Les programmes et les données sont des expressions.

- Le concept de mise à jour de variable n'existe pas.

- L'évaluation d'une fonction conduit à l'évaluation des arguments de la fonction ; l'enchaînement des traitements est donc entièrement déterminé par la demande d'évaluation des arguments.

- L'évaluation d'une expression peut retourner des arguments simples ou complexes (tels que des fonctions qui seraient argument de fonctions d'ordre supérieur).

- La réécriture des expressions peut utiliser n'importe quel mode d'évaluation (appel par valeur, par nom, évaluation paresseuse...).

- La propriété de "transparence du référent" dans les langages fonctionnels rend les réductions indépendantes de tout contexte et permet donc l'exploitation du parallélisme inhérent à ces langages.

- Le modèle de réduction permet (et facilite) l'évaluation de fonctions récursives.

Dans [TRELEA82], TRELEAVEN distingue deux modèles de réduction qui sont la réduction de chaîne et la réduction de graphe. Ces deux modèles se différencient par la façon dont sont manipulés les arguments d'une fonction.

III.2.2.2 : La réduction de chaîne

Le principe de la réduction de chaîne est que chaque expression qui accède à une définition particulière manipule une copie de la définition. Les problèmes de ce mode de réduction viennent des copies multiples devant être réalisées lors d'une évaluation :

- La copie d'une définition est une opération coûteuse.

- Si plusieurs copies d'une même définition sont nécessaires, les mêmes traitements peuvent être répétés.

- La taille d'une expression réductible varie constamment au cours du temps, ce qui pose des problèmes de mise en oeuvre, en particulier la localisation des expressions réductibles.

L'exemple le plus classique d'implantation du modèle de réduction de chaîne est la machine FFP de MAGO & Al, initialement décrite dans [MAGO79a], [MAGO79b], puis dans [MAGO84] et [MAGO85].

Cette machine, developpée à l'Université de "North Carolina" a été conçue pour l'exécution directe du langage FFP de BACKUS. L'architecture de cette machine consiste en deux réseaux de processeurs interconnectés dont l'un est un tableau linéaire de cellules identiques (les cellules L) et l'autre est un réseau structuré en arbre, de cellules également identiques (les cellules T). La figure III.1 montre une représentation de cette machine.

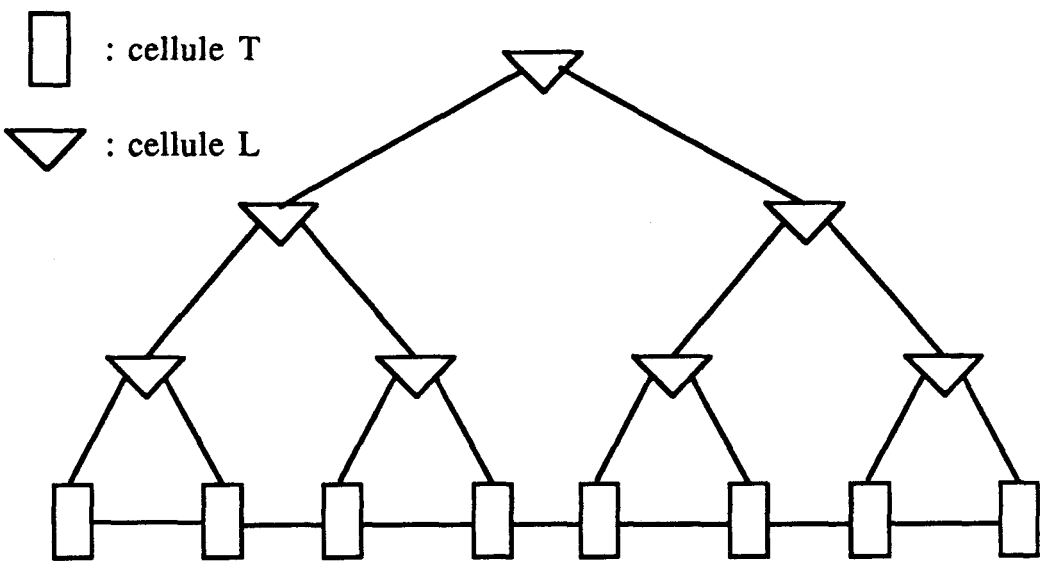


Figure III.1 : Topologie de la machine FFP

Les cellules de type T sont chargées du traitement des fonctions, du routage... etc. La cellule T racine de l'arborescence agit comme le port d'entrée/sortie de la machine. Les cellules L contiennent une représentation linéaire du programme FFP, l'ordre des symboles du programme source étant préservé. Un cycle machine est divisé en trois phases :

- La première phase consiste en un partitionnement de la machine : les cellules T sont partitionnées de façon à ce que chaque rédex (c'est-à-dire chaque expression réductible) "voie" un arbre binaire de cellules au-dessus de lui, qui traitera le rédex comme s'il était seul dans la machine.

- La deuxième phase est une phase d'exécution. Plusieurs rédex peuvent être traités simultanément générant ainsi du parallélisme.

- La troisisème phase permet de réorganiser la mémoire : elle consiste à déterminer dans un premier temps la nouvelle disposition à adopter et dans un deuxième temps, à effectuer des mouvements c'est-à-dire des décalages horizontaux pour utiliser l'espace libre.

Les caractéristiques de cette machine sont les suivantes :

- La mémoire est organisée de façon à ce que la localité soit respectée : les sous-programmes et leurs données sont groupés dans des espaces mémoires contigus. Ceci permet de réduire la distance moyenne des communications, réduisant ainsi l'encombrement moyen du réseau de communication. En conséquence, la vitesse d'exécution augmente (en théorie).

- Le programme et les données évoluant dynamiquement au cours de l'exécution, la phase de partitionnement permet d'adapter le "hardware" à chaque nouvelle configuration.

- Le parallélisme peut être exploité à plusieurs niveaux :

(i) : plusieurs programmes peuvent être exécutés simultanément.

(ii) : les opérations primitives sont exécutées en distribuant les traitements sur un groupe de processeurs.

(iii): la phase de partitionnement permet d'isoler des applications indépendantes pouvant être exécutées en parallèle.

Cette machine présente cependant un certain nombre d'inconvénients :

- Le fait que les traitements soient cycliques introduit une perte de temps dans les traitements : il faut attendre que toutes les évaluations d'expressions, lors de la phase d'exécution, soient effectivement terminées avant de pouvoir débuter la phase de réorganisation mémoire.

- Cette phase de réorganisation est coûteuse en temps d'exécution.

- Lors de la phase d'exécution, il est souvent nécessaire de "rapatrier" dans les cellules L, des segments de code utiles à une évaluation. Outre le fait que les copies de code sont coûteuses, il faut en plus que ce code soit copié dans un espace mémoire contigu ce qui rend l'opération encore plus coûteuse.

Les problèmes de la machine de MAGO illustrent de façon générale les problèmes des machines implantant le modèle de réduction de chaîne.

III.2.2.3 : La réduction de graphe

Dans le modèle de réduction de graphe, chaque expression qui accède à une définition particulière manipule, non plus des copies de la définition mais des références à cette définition.

Dans ce modèle, le programme est représenté par un graphe : l'application d'un objet M à un objet N peut être représentée par un arbre binaire dont le fils gauche est l'arbre (ou le graphe) représentant la fonction M et le fils droit est l'arbre représentant le terme N. La réduction consiste à transformer ce graphe en appliquant les règles de réduction.

Les caractéristiques du modèle de réduction de graphe sont les suivantes :

- Une réduction provoque une transformation locale du graphe. Le modèle ne présente donc pas de goulot d'étranglement tel qu'un environnement devant être consulté pour effectuer une réduction.

- A tout moment, le graphe contient des rédex indépendants qu'il est naturel de réduire simultanément ; le modèle permet donc l'exploitation du parallélisme.

- Les communications nécessaires à une évaluation se font à travers le graphe ce qui simplifie le modèle de coopération entre les différentes activités.

- L'état d'un traitement est bien défini à tout moment : il est donné par l'état du graphe.

Nous choisissons d'illustrer les mécanismes d'implantation du modèle de réduction de graphe par la description de deux machines : l'une séquentielle (SECD) et l'autre parallèle (ALICE). Ces deux machines ne sont pas les plus récentes puiqu'actuellement, la plupart des machines de réduction utilisent la représentation d'un programme sous la forme d'un graphe de combinateurs. La réduction par combinateurs sera l'objet du chapitre suivant, nous présenterons alors des machines de réduction plus récentes.

La machine SECD

La machine SECD est basée sur un automate conçu par P. LANDIN [LANDIN64] au début des années 60, pour évaluer mécaniquement des expressions mathématiques. C'est une machine séquentielle qui utilise comme mode d'évaluation, le passage par valeur : tous les paramètres sont évalués avant que ne soient effectuées les évaluations des fonctions.

Elle utilise 4 piles et à tout instant, l'état de la machine est caractérisé par les valeurs contenues dans ces 4 piles. Les piles sont nommées *Stack*, *Environment*, *Control* et *Dump*.

- La pile *Stack* stocke les résultats lors de l'évaluation d'une fonction et conserve le résultat final de l'exécution.

- La pile *Environment* est constituée d'une suite de couples donnant les valeurs des variables liées d'une expression : le premier élément de chaque couple est un identificateur et le second est la valeur associée à cet identificateur.

- La pile *Control* stocke l'expression en cours d'évaluation.

- La pile *Dump* stocke des copies des piles *Stack*, *Environment* et *Control* lors de l'évaluation de sous-expressions.

La machine SECD a servi de base à un certain nombre de travaux. Dans [HENDER80], HENDERSON montre la possibilité de modifier l'algorithme associé à la machine SECD pour permettre le mode d'évaluation paresseuse. Ceci est réalisé en introduisant la notion de suspension qui est un couple formé d'un paramètre non encore évalué et de l'environnement dans lequel il doit l'être. Dans [ABRAMS85] est présentée une extension de la machine SECD permettant d'exploiter le parallélisme. Enfin, citons la machine CAM [COUSIN85] qui fait partie des machines de réduction par combinateurs mais qui peut être considérée comme un descendant de la machine SECD.

La machine ALICE

ALICE est une machine de réduction de graphe parallèle, conçue par DARLINGTON & Al [DARLIN81] à l'Imperial College de Londres. Elle est construite à partir de Transputers INMOS accédant à une mémoire commune. Elle permet d'exécuter des programmes HOPE mais peut également supporter tout langage fonctionnel de haut niveau.

Le mode d'évaluation d'ALICE est la substitution complète (sauf pour la conditionnelle qui est exécutée séquentiellement). L'utilisateur peut cependant demander qu'une fonction soit évaluée selon le mode paresseux.

Dans ALICE, le programme est représenté par un graphe, lui-même représenté par un ensemble de paquets à réduire. Un paquet à réduire transmet aux paquets argument, une requête leur indiquant que leur réduction est demandée par tel paquet. Le paquet demandeur se met alors en attente. Lorsque les paquets demandés sont évalués, leur résultat est retourné aux paquets demandeurs. Un paquet demandeur recevant un résultat décrémente le nombre de paquets qu'il attend et lorsque ce nombre est égal à zéro, le paquet demandeur peut être réduit.

III.2.3 : Autres modèles

III.2.3.1 : La machine REDIFLOW

La machine REDIFLOW [KELLER85], developpée à l'Université d'UTAH unifie les concepts de réduction et dataflow dans une même architecture (le projet REDIFLOW est le successeur du projet AMPS : Applicative Multi-Processor System [KELLER79]). C'est une machine parallèle pour langages fonctionnels dont chaque processeur est associé à une mémoire et un processeur de communication, ces trois éléments formant un *Xputer*.

Les deux problèmes majeurs dans REDIFLOW sont la répartition des processus et la gestion du taux d'occupation de la machine. Ainsi, chaque *Xputer* contient quatre files d'attente :

- La file *LOCAL* contient les appels de fonctions ne générant pas suffisamment de parallélisme pour que ce soit "rentable" de les faire migrer vers un autre *Xputer* (le coût de communication serait trop élevé par rapport au gain dû au parallélisme).

- La file *APPLY* contient les appels de fonction générant un parallélisme moyen ou massif.

- Les files *IN* et *OUT* échangent des informations avec l'extérieur.

La régulation de la charge dans la machine se fait de la façon suivante : chaque *Xputer* connaît son propre taux d'occupation et le taux d'occupation des *Xputers* voisins. Il estime alors le taux d'occupation global de la machine à partir de ces informations. Au-delà d'un certain seuil, la saturation est détectée : à partir de ce moment là, les migrations de processus cessent et les files *APPLY* qui fonctionnaient en mode *FIFO* (évaluation "outermost") fonctionnent en mode *LIFO* (évaluation "innermost").

III.2.3.2 : Les modèles SUP et SEP

Les modèles SUP et SEP sont deux schémas d'évaluation définis par D.LE METAYER [LEMETA84] pour une mise en oeuvre parallèle du langage FFP de BACKUS. Dans ces modèles, le mode substitution complète est mis en oeuvre tant que le nombre de processeurs le permet et dès que le nombre de tâches dépasse le nombre de processeurs, une hiérarchie est introduite entre ces tâches qui permet de décider de l'attribution des processeurs.

Dans le modèle SEP (substitutions externes en parallèle), la hiérarchie est déterminée de manière heuristique : à chaque tâche est attribuée une priorité, calculée selon sa situation dans l'arbre d'exécution et la priorité de la tâche qui l'a créée. Le modèle SEP privilégie les réductions externes mais autorise d'autres réductions s'il y a suffisamment de ressources disponibles. Il peut donc être vu comme un modèle "*parallel outermost*" ajustable.

Dans le schéma SUP (Substitutions Utiles en Parallèle), la hiérarchie est déterminée par les propriétés des primitives du langage. Une occurrence réductible dans l'arbre d'exécution est dans l'un des trois états "*nécessaire a priori*", "*non nécessaire a priori*", "*inutile*". L'ensemble

tâches est séparé en deux groupes selon leur priorité et chaque groupe est contenu dans une file gérée en mode *FIFO* (une file contient les tâches nécessaires et l'autre les tâches non nécessaires). Les tâches étant gérées en mode *FIFO*, toute tâche prioritaire est exécutée au bout d'un temps fini.

Les différents modèles d'évaluation que nous venons d'évoquer permettent tous des évaluations parallèles. Ces différents modèles ont en commun un même problème : la granularité du parallélisme invoqué lors de l'évaluation.

III.3 : Le parallélisme

Quel que soit le modèle d'exécution adopté, l'implanteur d'un langage, s'il veut réaliser une évaluation parallèle, est amené à déterminer la granularité du parallélisme mis en oeuvre lors de l'exécution. Dans un premier temps, nous montrerons les différents choix et leur corrélation avec les systèmes sur lesquels ils peuvent être mis en oeuvre. Quelle que soit la granularité choisie, l'implanteur doit toujours envisager le cas où trop de parallélisme serait généré. Dans un deuxième temps, nous exposerons différentes méthodes permettant de contrôler le parallélisme.

III.3.1 : Choix de la granularité

L'implanteur d'un langage peut décider d'exploiter un parallélisme plus ou moins massif.

Parmi les solutions qui exploitent peu de parallélisme, on peut citer celles qui ne prennent en compte que le parallélisme explicitement programmé par l'utilisateur. Cette solution est adoptée dans certaines implantations du langage LISP.

Dans [KELLER79], KELLER & LINDSTROM présentent une solution n'exploitant le parallélisme qu'au niveau des appels de fonctions définies et non dans le corps de la fonction elle-même.

SLEEP & BURTON, dans [SLEEP80], adoptent une solution permettant de n'exploiter que le parallélisme inhérent aux algorithmes de type "diviser pour régner".

Toutes ces solutions présentent le même inconvénient : si peu de parallélisme est exploité, la machine peut être souvent amenée à ne pas utiliser toute sa puissance de traitement.

La plupart des implantations de langage fonctionnel exploitent un parallélisme massif : par exemple, dans la machine de réduction ALICE [DARLIN81] et la machine data-flow de DENNIS [DENNIS79] chaque noeud du graphe est représenté par un processus.

Dans ce genre de machine, un processus est donc actif pendant très peu de temps et n'effectue qu'une petite transformation très simple. Le problème lorsque l'on désire exploiter un parallélisme massif est qu'étant donné qu'un processus seul "ne fait pas grand chose", les processus ont davantage besoin de coopérer, ce qui nécessite des communications entre les processus. Or si le temps de communication devient trop important, la vitesse d'exécution diminue ce qui tend à réduire l'efficacité. Le choix de la granularité du parallélisme est donc un problème important, difficile à résoudre et qui n'est pas complètement indépendant de la machine sur laquelle on veut implanter le modèle.

En effet, on peut diviser les machines multi-processeurs en deux catégories : les architectures parallèles à "gros grain" et les architectures parallèles à "grain fin" [MAGO85].

On considère qu'une architecture parallèle est "à gros grain" si chaque processeur contient une unité arithmétique et logique (UAL) relativement puissante, un ensemble d'instructions suffisamment important et a accès à une "grande mémoire".

Par opposition, on considère qu'une architecture parallèle est "à grain fin" si chaque processeur contient une UAL peu puissante (une UAL série), un petit ensemble d'instructions et une petite mémoire locale. Ce type d'architecture est parfois appelé "logic in memory System" dans la mesure où il peut être vu comme un système dans lequel de la logique a été ajoutée à chaque mémoire locale. Ce type de multi-processeurs ne souffre donc pas de la séparation entre la puissance de traitement et la mémoire.

Si l'on considère le rapport entre la puissance de traitement et la mémoire, la puissance d'un système "à grain fin" est largement supérieure à celle d'un système "à gros grain". Un système "à grain fin" a donc les moyens d'effectuer davantage de copies et de mouvements de données qu'un système "à gros grain", en conséquence il est plus apte à répartir le programme et les données sur les processeurs dynamiquement.

Dans un système "à grain fin", si à certains moments, certains processeurs sont inactifs, cela a moins d'importance que dans un système "à gros grain" puisque qu'un processeur ne représente qu'une petite fraction de la puissance de traitement globale. Le gros problème dans les systèmes "à gros grain" est donc de faire en sorte d'utiliser au maximum tous les processeurs. Tandis que le gros problème des multi-processeurs "à grain fin" est de permettre des communications et coopérations très performantes.

Ceci illustre la corrélation entre le parallélisme exploité dans un modèle et l'architecture sur laquelle on veut implanter ce modèle : une exploitation massive du parallélisme sera implantée de façon plus efficace sur un système à grain fin tandis qu'une faible exploitation du parallélisme peut davantage être envisagée sur un système "à gros grain".

Quelle que soit sa granularité, le parallélisme doit être contrôlé : si trop de parallélisme est généré lors d'une exécution, la "consommation" d'espace mémoire peut devenir trop importante ce qui peut éventuellement conduire à un "deadlock". D'où la nécessité de mettre en

place des régulateurs. Nous présentons dans la section suivante différentes méthodes permettant de contrôler le parallélisme.

III.3.2 : Le contrôle du parallélisme

Plusieurs solutions ont déjà été utilisées dans divers projets, permettant de contrôler partiellement ou complètement le parallélisme. Des méthodes statiques permettent par exemple de détecter les fonctions strictes à la compilation et de ne mettre en oeuvre le parallélisme que sur ces fonctions. De même, dans ALICE [DARLIN81] et AMPS [KELLER79], le programmeur dit explicitement si un opérateur non strict doit être évalué en parallèle ou non. Ceci permet de limiter le parallélisme mais pas de le contrôler. Les méthodes permettant effectivement de contrôler le parallélisme sont très souvent des méthodes dynamiques : les décisions sont prises à l'exécution

Parmi ces méthodes, on peut citer le système appelé "Colonel & Sergent" décrit dans [FRIEDM78] : le parallélisme est mis en oeuvre chaque fois qu'un processeur inactif est disponible. Dans la machine ZAPP [BURTON81], avant de faire un traitement, on vérifie que l'espace mémoire disponible est suffisant. Dans REDIFLOW [KELLER84], la quantité de parallélisme généré est déterminée par la charge des processeurs.

Une méthode légèrement différente consiste à implanter en machine un code séquentiel et un code parallèle et de décider à l'exécution, selon l'activité de la machine, d'exécuter tel ou tel code. Le problème de cette méthode est d'effectuer la transformation d'un code parallèle en code séquentiel et vice-versa.

Dans [RUGGIE87], RUGGIERO & SARGEANT étudient le contrôle du parallélisme dans la machine de MANCHESTER [GURD85a], [GURD85b]. Leur idée de départ était d'implanter un mécanisme entièrement "hardware" ayant pour but de donner une priorité aux différentes marques circulant dans le graphe data-flow : les marques susceptibles

de donner des résultats nécessaires reçoivent la plus grande priorité. Pour cela, ils ont tout d'abord suggéré de transformer la traversée du graphe pour qu'elle s'effectue en profondeur d'abord, de droite à gauche (au lieu de en largeur d'abord de gauche à droite). Malheureusement, ceci donne de très mauvais résultats pour les programmes itératifs qui nécessitent une exécution de gauche à droite.

Leur deuxième idée fut d'implanter ce qu'ils appelent une "Intelligent Token Queue" (*ITQ*). Dans l'*ITQ*, un ou plusieurs processus sont considérés comme processus courants. Toutes les marques appartenant à un processus courant sont rangées dans la file pour exécution immédiate. Quand un processus courant n'a plus de marques, il est remplacé par un autre processus. Toutes les marques de ce nouveau processus sont alors rangées dans l'*ITQ*.

Le critère de sélection d'un nouveau processus n'est pas sans conséquence : si l'on choisit le processus le plus ancien, cela revient à favoriser les exécutions de gauche à droite et donc à privilégier les problèmes récursifs. Si l'on choisit le plus récent, cela revient à favoriser les exécutions de droite à gauche et donc à privilégier les problèmes itératifs. Cette solution n'était donc pas satisfaisante.

Leur troisième idée fut de contrôler le nombre de processus actifs selon les ressources disponibles dans le système. Les mesures de l'activité de la machine sont effectuées d'après la longueur de la file des marques. Quand la longueur de la file atteint une certaine limite, le régulateur (une unité "hardware") commence à suspendre des processus. Quand la longueur retombe sous une certaine limite, des processus sont débloqués.

Remarques :

- un processus est suspendu avant que son exécution ne débute.

- pour favoriser l'exécution en profondeur d'abord, le premier fils d'un processus n'est jamais suspendu.

Ceci termine notre présentation générale des modèles d'évaluation parallèles de langages fonctionnels. Nous avons présenté quelques exemples d'implantation de ces modèles et les problèmes rencontrés lorsque l'on veut mettre en oeuvre une exécution parallèle.

Dans le chapitre suivant, nous présentons un cas particulier du modèle de réduction : la réduction par combinateurs.

IV: LA REDUCTION PAR COMBINATEURS

IV : LA REDUCTION PAR COMBINATEURS

Bien que les lambda-langages (sans effets de bord) fassent une bonne utilisation de la variable, ces variables posent de réels problèmes pour l'implantation de ces langages, comme nous le montrerons dans un premier temps.

Ces constatations amenèrent TURNER à utiliser les travaux de CURRY et FEYS pour envisager une nouvelle technique d'implantation des lambda-langages, fondée sur l'élimination de toutes les variables d'un programme avant son exécution. Nous présenterons dans un deuxième temps le principe et les avantages de cette technique.

Dans la suite du chapitre, nous décrirons quelques familles de combinateurs existantes : les combinateurs S, K, I, les super-combinateurs, les combinateurs sériels et les combinateurs utilisés dans la machine MaRS. Pour chacune de ces familles, nous donnerons l'algorithme d'abstraction correspondant, c'est-à-dire l'algorithme qui permet de passer d'une lambda-expression quelconque à une expression combinatoire, et les règles de réduction permettant d'exécuter cette nouvelle expression. Nous illustrerons chaque fois ces techniques sur un exemple de fonction et nous donnerons quelques exemples de machines de réduction sur lesquelles sont implantées ces techniques.

Un certain nombre d'ouvrages ont été réalisés sur les techniques de compilation et d'implantation de langages fonctionnels par combinateurs. Le lecteur pourra se référer à [PEYTON87a], [DILLER88] et [HINDLE86].

IV.1 : Motivations

En réduction de graphe, une expression est représentée par un arbre ou un graphe. L'application d'une fonction à un argument est représentée par un arbre binaire, le noeud racine de l'arbre étant un noeud "application", son fils droit représentant la fonction et son fils

gauche, l'argument. Ainsi, l'application d'une fonction *f* à un argument *x* est représentée par la figure IV.1.

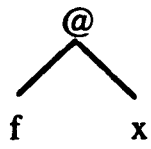


Figure IV.1 : représentation de l'expression (f x)

Les fonctions à plusieurs arguments sont traitées par curryfication. Ainsi, l'expression *(+ 4 2)* est représentée par la figure IV.2.

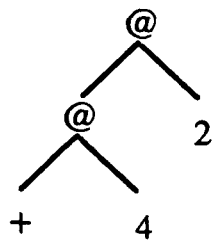


Figure IV.2 : représentation de l'expression (+ 4 2)

Dans cette représentation, le sous-arbre gauche représente l'expression *(+ 4)* qui est une fonction appliquée à l'argument *2*.

Une λ-expression de la forme *(λx. E)* est représentée comme le montre la figure IV.3.

Figure IV.3 : représentation d'une lambda-expression de la forme (λx. E)

Ainsi, l'expression ((λx. NOT x) True) est représentée par l'arbre de la figure IV.4.

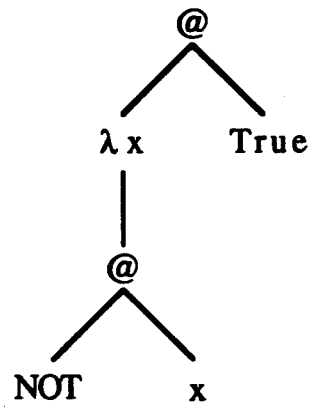


Figure IV.4 : représentation de l'expression ((λx. NOT x) True)

Cette expression est un rédex qui peut être réduit par application de la règle de β-réduction :

((λx. NOT x) True) $conv_{\beta}$ (NOT True)

Pour appliquer la β-réduction au graphe, il faut construire une instance du corps de la λ–expression, en substituant l'argument (*true*) aux occurrences libres du paramètre formel (*x*). Cette opération est appelée l'**instanciation** de la λ-expression.

Le problème est que cette instanciation est réalisée par un parcours d'arbre récursif très coûteux car

(i) lors du parcours, il faut tester le type de chaque noeud,
(ii) chaque fois que l'on rencontre une variable, il faut tester si cette variable est le paramètre formel. Un test similaire doit être effectué à chaque "noeud λ".
(iii) de nouvelles instances de sous-expressions ne contenant pas d'occurrences libres du paramètre formel peuvent être construites alors qu'elles pourraient être partagées sans aucun problème.

Pour éviter ces parcours récursifs, on pourrait envisager de compiler le programme source pour associer à chaque λ-expression une

séquence fixée d'instructions qui permettraient de construire une instance du corps de la λ-expression. De cette façon, l'instanciation serait réalisée en exécutant une séquence d'instructions au lieu d'effectuer le parcours récursif. Seulement, toutes les λ–expressions ne peuvent pas être compilées de cette façon. Nous avons déjà soulevé le problème en section 1.7.2.

Reprenons l'exemple de la λ-expression

λx. (λy. + x y)

Si on applique cette λ-expression à un argument, on obtient une nouvelle λ-expression

exemple :

(λx. (λy. + x y) 3) = (λy. + 3 y)

Dans cet exemple, on voit donc que chaque fois que l'on appliquera la λ-expression à un argument différent, on obtiendra en résultat une λ-expression différente. Il est donc impossible d'envisager de compiler cette λ-expression en une séquence fixée de code. Ceci est dû au fait que la variable *x* est libre dans la λ-expression *(λy. + x y)*.

Dans la machine SECD [LANDIN64], ce problème est résolu en donnant aux séquences de code, l'accès à un environnement contenant les valeurs de chaque variable libre. Cette technique présente plusieurs inconvénients :

- elle revêt un aspect beaucoup trop séquentiel,
- elle conduit à une sémantique d'appel par valeur, avec tous les inconvénients qu'elle comporte,
- la gestion d'un environnement introduit toujours un surcoût non négligeable.

Une autre idée pour résoudre ce problème consiste à transformer une λ-expression en une forme dans laquelle les λ-expressions sont

faciles à instancier : c'est l'approche des **combinateurs**. Nous donnons dans la section suivante le principe et les avantages de cette approche.

IV.2. Principes et avantages

Un combinateur est défini dans [BAREND84] comme une λ-expression ne contenant aucune occurrence de variable libre. Un combinateur peut donc être vu comme une fonction "pure" dans le sens où la valeur d'un combinateur appliqué à des arguments ne dépend que de la valeur des arguments et non de la valeur de variables libres.

Or, rappelons que Schönfinkel a montré dans les années 20 [SCHONF24] qu'il était possible d'éliminer toutes les variables des expressions fonctionnelles pour obtenir de nouvelles expressions constituées uniquement d'un ensemble fixé de combinateurs et de symboles de constantes. Puis CURRY et FEYS ont montré dans [CURRY58] que tout λ–langage pur pouvait être compilé en un code objet sans variable.

Les combinateurs sont définis par leurs **règles de réduction** c'est-à-dire par des règles de réécriture indiquant comment ils doivent être exécutés.

La stratégie de réduction par combinateurs consiste donc à transformer le programme en un code combinatoire puis à appliquer les règles de réduction.

L'absence de variables dans le code combinatoire supprime la nécessité d'un environnement à gérer. De plus, cette stratégie conduit à une machine de réduction très simple puisqu'elle ne doit supporter que des opérateurs et ne nécessite pas de mécanisme d'instanciation. Le code combinatoire peut jouer le rôle de langage machine et peut donc conduire à une implantation directement "hardware". De plus, le code combinatoire conserve toutes les bonnes propriétés des langages fonctionnels.

Pour toutes ces raisons, les combinateurs apparaissent comme une solution réelle au problème de l'inefficacité des lambda-langages.

IV.3 : Les combinateurs S, K, I

IV.3.1 : Règles de réduction des combinateurs S, K, I

L'application d'un objet f à un objet x est notée $(f\ x)$ ou tout simplement $f\,x$. La notation $(f\ x\ y\ z)$ désigne l'application $(((f\ x)y)z)$.

Etant données ces notations, les combinateurs S, K, I sont trois objets définis par les règles de réduction suivantes où $(E_1 \rightarrow E_2)$ signifie que l'expression E_1 se réduit en l'expression E_2.

1) S f g x → f x (g x)
2) K x y → x
3) I x → x

Remarque :

Le combinateur I est redondant dans la mesure où il peut être défini par

I = S K K

En effet :

S K K x	→ K x (K x)	par application de 1)
	→ x	par application de 2)

et

I x → x	par application de 3)

Son utilisation permet de diminuer le code combinatoire. Etant donnée cette définition des combinateurs S, K, I, toute λ-expression peut être transformée en une expression combinatoire (C-expression) ne contenant que les combinateurs S, K, I et des symboles de constantes

(dont les constantes de fonction) grâce à l'algorithme donné dans la section suivante.

IV.3.2 : Algorithme de compilation d'une λ-expression en une C-expression

Soient e, e_1, e_2 des expressions,
f, f_1, f_2 des expressions ne contenant pas de λ-expressions,
x une variable,
cv une variable ou une constante (dont les constantes de fonctions telles que +, Y (l'opérateur de point fixe), ... etc).

Soient $C[e]$ le résultat de la compilation de e en C-expression et
$Ax[f]$ l'abstraction de la variable x dans l'expression f.

$C[e_1\ e_2] = C[e_1]\ C[e_2]$
$C[\lambda x.\ e] = Ax[C[e]]$
$C[cv] = cv$

$Ax[f_1\ f_2] = S\ (Ax[f_1])\ (Ax[f_2])$
$Ax[x] = I$
$Ax[cv] = K\ cv$

De façon plus simple mais moins formelle, on peut décrire cet algorithme par l'ensemble des règles de transformation suivantes où
$(E_1 \Rightarrow E_2)$
signifie que l'expression E_1 se transforme en l'expression E_2

a) $\lambda x.\ x \Rightarrow I$
b) $\lambda x.\ c \Rightarrow K\ c \quad (c \neq x)$
c) $\lambda x.\ e_1\ e_2 \Rightarrow S\ (\lambda x.\ e_1)\ (\lambda x.\ e_2)$

Remarque :

Les règles de transformation définies ci-dessus sont les règles utilisées à la compilation tandis que les règles de réduction définies dans la section précédente sont utilisées à l'exécution.

Dans la section suivante, nous illustrons les mécanismes de compilation et d'évaluation sur un exemple.

IV.3.3 : Exemple d'application des règles de transformation et de réduction

Soit la λ-expression *(λx. OR x True)*. Appliquons à cette λ-expression, l'algorithme de compilation.

λx. OR x True
c) ⇒ S (λx. OR x) (λx. True)
c) ⇒ S (S (λx. OR) (λx. x)) (λx. True)
b) ⇒ S (S (K OR) (λx. x)) (K True)
a) ⇒S (S (K OR) I) (K True)

On obtient donc une expression combinatoire ne comportant que les combinateurs S, K, I, le symbole de fonction *OR* et la constante *True*.

L'évaluation de cette C-expression appliquée à un argument *x* est obtenue par application des règles de réduction :

S (S (K OR) I) (K True) x
1) → S (K OR) I x (K True x)
1) → K OR x (I x) (K True x)
2) → OR (I x) (K True x)
3) → OR x (K True x)
2) → OR x True

Remarque :

Les trois dernières réductions peuvent être effectuées en parallèle.

Les combinateurs S, K permettent de transformer n'importe quelle λ-expression en une C-expression. Seulement, on peut s'en apercevoir sur l'exemple précédent, le procédé de compilation peut donner de très longues chaînes de code pour des expressions initialement simples.

Afin de réduire la taille du code combinatoire, d'autres combinateurs peuvent être définis ainsi que quelques règles d'optimisation.

IV.3.4 : Autres combinateurs et règles d'optimisation

Nous n'introduisons ici qu'un sous-ensemble des combinateurs et règles d'optimisation définies par CURRY et FEYS dans [CURRY 58].

Avant même d'introduire de nouveaux combinateurs, il est possible de définir quelques règles d'optimisation telles que

1) S (K p) (K q) ⇒ K (p q)
2) S (K p) I ⇒ p

Vérifions la première règle d'optimisation :

S (K p) (K q) x
→ K p x (K q x)
→ (p q)

et K (p q) x
→ (p q)

La deuxième règle peut être vérifiée de la même façon.

B et C sont deux combinateurs définis par les règles de réductions suivantes :

B f g x → f (g x)
C f g x → f g x

Ces deux combinateurs permettent d'introduire de nouvelles règles d'optimisations telles que

3) S (K p) q ⇒ B p q
4) S p (K q) ⇒ C p q

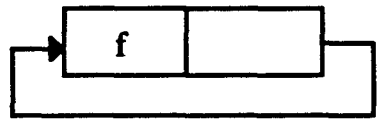
Les nouveaux combinateurs et les règles d'optimisation sont utilisés dans la plupart des machines de réduction basées sur les combinateurs S, K, I.

IV.3.5 : Exemples d'implantation

La machine SK

Les premiers travaux d'implantation utilisant les combinateurs furent effectués par D. TURNER [TURNER79a], [TURNER79b] : il définit la machine abstraite SK permettant de réduire un programme source SASL. Les programmes sont représentés sous la forme de graphes de combinateurs et identificateurs de constantes. La machine SK réduit le code objet en appliquant directement les règles de réduction définissant les combinateurs et les primitives. Ces réductions sont réalisées par des transformations de graphe.

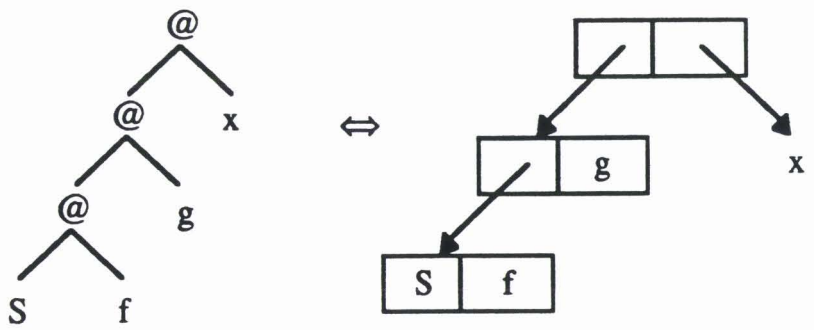
Plusieurs réductions peuvent être réalisées à chaque étape mais la machine SK est une machine séquentielle : c'est le rédex le plus à gauche et le plus externe qui est réduit à chaque étape. La machine SK permet un traitement efficace de la récursivité en implantant le combinateur Y défini par $Y\,f = f\,(Y\,f)$ de la façon suivante



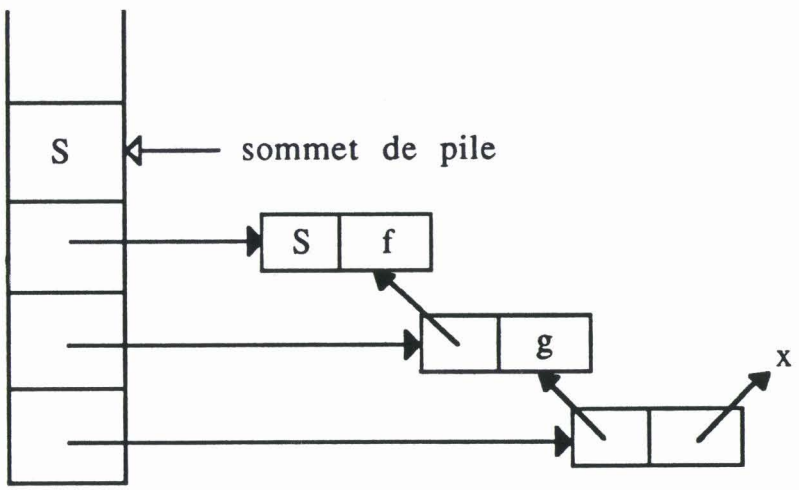
Pour éviter, à chaque étape, un parcours du graphe afin de trouver le prochain rédex, TURNER construit une pile des ancêtres gauches du graphe, le sommet de pile contenant le prochain rédex à réduire.

Exemple :

L'expression *(S f g x)* est représentée par le graphe



Au départ, la pile ne contient qu'un pointeur vers l'expression à évaluer puis la recherche du premier rédex conduit la pile dans l'état suivant :



Les travaux de TURNER furent développés par CLARKE & Al qui réalisèrent un prototype de la machine SK, appelé SKIM [CLARKE80]. L'idée de SKIM est d'implanter les combinateurs directement au niveau hardware et de les considérer comme faisant partie de l'ensemble des instructions machine. De cette façon, SKIM se révèle beaucoup plus efficace que la machine SK.

P. HUDAK et D. KRANZ proposent dans [HUDAK84], un compilateur optimisé du langage ALFL, un langage fonctionnel proche du langage SASL. Leur but est de démontrer qu'un langage ayant une sémantique

d'évaluation paresseuse peut être exécuté de façon efficace si le code compilé est optimisé : les combinateurs offrent un langage intermédiaire permettant d'effectuer des transformations de programmes et des optimisations. Ceci permet d'éliminer les sources d'inefficacité que les compilateurs conventionnels ne traitent pas. Un compilateur optimisé transforme donc tout d'abord le programme en une C-expression puis le compilateur crée ses propres procédures optimisées qui ne correspondent pas nécessairement à celles définies dans le programme source.

Tous ces travaux sont basés sur une même famille de combinateurs. Leur particularité est de travailler sur un ensemble fixé de combinateurs. Plus récemment, des travaux se sont basés sur une génération dynamique des combinateurs à la compilation. C'est le cas de la compilation en super-combinateurs, que nous présentons dans la section suivante

IV.4 : Les super-combinateurs

Les super-combinateurs ont été introduits par HUGHES dans [HUGHES82]. La technique de compilation en super-combinateurs est également décrite dans [PEYTON87a].

IV.4.1 : L'idée

Nous avons vu dans la section précédente que les combinateurs permettent de résoudre le problème de l'instanciation d'une lambda-expression. Les super-combinateurs de HUGHES adoptent une approche différente.

Reprenons l'exemple de la λ-expression qui nous avait permis d'illustrer le problème de l'instanciation en section IV.3.1.

Exemple : $(\lambda x.\ \lambda y.\ +\ x\ y)\ 3\ 4$

Pour réduire cette expression, il faut appliquer deux fois la β-réduction. Si l'on effectue ces deux β-réductions l'une après l'autre on obtient

$$(\lambda x.\ \lambda y.\ +\ x\ y)\ 3\ 4$$
$$\rightarrow (\lambda y.\ +\ 3\ y)\ 4$$
$$\rightarrow (+\ 3\ 4)$$

A la première étape, on a créé une instance de la λ-expression qui pose des problèmes.

Si maintenant on effectue les deux β-réductions simultanément, on obtient :

$$(\lambda x.\ \lambda y.\ +\ x\ y)\ 3\ 4$$
$$\rightarrow (+\ 3\ 4)$$

De cette façon, aucune structure intermédiaire n'est créée. Il n'y a donc plus de problèmes dus à l'occurrence libre de la variable *x* dans la λ-expression *(λy. + x y)*, sans que l'on ait perdu la moindre information. C'est l'approche adoptée par les super-combinateurs.

IV.4.2 : Définitions

Définition 1 :

Un super-combinateur *$S* d'arité *n* est une λ–expression de la forme

$$\lambda x_1.\ \lambda x_2.\ \ldots\ .\lambda x_n.\ E$$

où *E* n'est pas une λ-abstraction et *E* est telle que

(i) *$S* n'a pas de variable libre,

(ii) toute λ-abstraction dans *E* est un super-combinateur,

(iii) $n \geq 0$.

Définition 2 :

Un rédex consiste en l'application d'un super-combinateur d'arité *n* à *n* arguments.

Une réduction consiste alors à remplacer un rédex par une instance du corps du super-combinateur correspondant dans lequel les occurrences libres du paramètre formel ont été remplacées par les arguments correspondants.

Etant donnée la définition d'un super-combinateur, son instanciation ne pose aucun problème. En effet :

- Les super-combinateurs d'arité supérieure à 0 n'ont pas de variable libre (d'après (i)). Ils peuvent donc être compilés en une séquence fixée de code. De plus, les λ-expressions internes étant d'après (ii) des super-combinateurs, elles ne contiennent pas non plus d'occurrence de variables libres. Il est donc inutile de les copier quand le corps d'un super-combinateur est instancié.
- quant aux super-combinateurs d'arité 0, par définition, "ils n'ont pas de λ-devant", ils ne seront donc jamais instanciés.

Le but est donc de transformer un programme de manière à ce qu'il ne contienne que des super-combinateurs : le but n'est plus de supprimer les variables d'une expression mais de supprimer les occurrences libres de variables dans les λ-expressions (i.e. : les occurrences de variables qui posent des problèmes !)

IV.4.3 : Algorithme de transformation d'une λ-expression en super-combinateur

IV.4.3.1 : L'idée de l'algorithme

La stratégie de l'algorithme est de transformer une lambda-expression pour obtenir :

(i) un ensemble de définitions de super-combinateurs,
(ii) une expression à évaluer.

Notation :

Dans l'exemple de la section IV.4.1, la λ-expression *(λx.λy. + x y)* est un super-combinateur. On lui donne un nom, par exemple $XY :

$XY = λx. λy. + x y

On adopte alors la notation suivante :

$XY x y = + x y

L'expression *(λx.λy. + x y) 2 3* est alors transformée en

$XY x y = + x y

$XY 2 3

Lorsqu'une λ-expression n'est pas un super-combinateur, il est possible de la transformer. Nous donnons tout d'abord une idée intuitive de la transformation à effectuer, nous donnerons ensuite l'algorithme complet de transformation.

Soit l'expression *(λx. (λy. + y x) x) 4.*

Pour transformer la λ-expression en super-combinateurs, on réalise une abstraction de la variable libre dans la λ-expression interne. C'est-à-dire que l'on transforme

(λy. + y x) en (λx. λy. + y x) x

Cette opération, appelée la β-abstraction, est l'opération inverse de la β-réduction (si on applique la β-réduction à l'expression résultant de la transformation, on retrouve l'expression initiale). Ces deux expressions sont donc équivalentes.

Pour plus de clarté, on peut réaliser une α-conversion, c'est-à-dire transformer

(λx. λy. + x y) x en (λw. λy. + w y) x.

L'expression initiale *(λx. (λy. + x y) x) 4* s'écrit maintenant

(λx.(λw. λy. + w y) x x) 4

Dans cette expression, la λ-expression interne est un super-combinateur. On lui donne un nom, par exemple

$Y = λw. λy. + w y

et l'on note cette équation sous la forme

$Y w y = + w y

L'expression initiale devient

(λx. $Y x x) 4.

La λ-expression *(λx. $Y x x)* est un super-combinateur, donc de la même façon, on lui donne un nom :

$X = λx. $Y x x

ou $X x = $Y x x

Ce super-combinateur est appliqué à l'objet *4*, ce qui s'écrit

$X 4

Finalement, l'expression initiale est donc transformée en

$Y w y = + w y

$X x = $Y x x

$X 4.

Pour exécuter le programme, il suffit de réaliser des réductions de super-combinateurs en partant de l'expression à évaluer :

$X 4 → $Y 4 4

→ + 4 4

→ 8

On remarque dans cette méthode, que l'on est amené à générer des super-combinateurs : ils ne sont pas prédéfinis comme dans le cas des combinateurs SK.

IV.4.3.2 : L'algorithme

Définitions :

• Etant donnée une λ-expression $E = λv.\ e$, une expression libre dans E est une sous-expression de e dans laquelle v n'apparait pas libre.

• Une expression libre est maximale si elle n'est pas contenue dans une autre expression libre.

Etant donnée E une λ-expression, l'algorithme donné par HUGUES pour transformer cette λ−expression en super-combinateurs est le suivant :

1) Prendre dans E la λ-expression la plus interne et la plus à gauche. Soit $L = \lambda v.exp$ cette lambda-expression.

2) Prendre les expressions libres maximales de L.
Soient e_1, e_2, ..., e_n, ces expressions.

3) Créer un nouveau super-combinateur (appelé ici α) défini par :
$\alpha\ i_1 ... i_n\ v = exp[i_1/e_1, ..., i_n/e_n]$
tel que les paramètres formels $i_1 .... i_n$ n'apparaissent pas libres dans exp.

4) substituer $(\alpha\ e_1 ... e_n)$ à L dans E.

5) répéter les étapes 1) à 4) jusqu'à ce que l'étape 1) échoue.

IV.4.4.Exemple d'application de l'algorithme

Soit le programme ALFL permettant de calculer $n!$ par la méthode "diviser pour régner" :

```
{fac 0 == 1 ;
 ' n == pfac 1 n ;

 pfac k h == h = k → k,
             h = k + 1 → k * h,
             (pfac k (k + h) / 2) * (pfac ((k + h )/2 + 1) h) ;
 result fac n }
```

La fonction *Pfac* s'écrit sous la forme d'une lambda-expression de la façon suivante :

```
pfac = Y (λf. (λx. (λy. if (= x y )
                           x
                           (if (= y (+ x 1))
                               (* x y)
                               (* (f x (/ (+ x y) 2))
                                  (f (+ (/ (+ x y )2)1)y)
                               )
                           )
                    )
               )
          )
```

où Y est l'opérateur de point fixe défini par $Y\,f = f\,Y\,f$.

L'algorithme de HUGUES s'applique à cette λ–expression de la façon suivante :

1) L = (λy. if .....)
2) e_1 = f e_2 = (+ x 1) e_3 = (f x) e_4 = x
3) création d'un super-combinateur α défini par

```
α a b c d y =
          if (= d y )
             d
             (if (= y b)
                 (* d y)
                 (* (c (/ (+ d y) 2))
                    (a (+ (/ (+ d y) 2) 1) y)
                 )
             )
```

4) La substitution de *(α e_1...e_n)* à *L* dans *pfac* fournit :
   Pfac = Y (λf. (λx. α f (+ x 1) (f x) x))

On recommence à l'étape 1) :

1) L = λx. α f (+ x 1) (f x) x
2) e_1 = (α f) e_2 = f
3) β p q x = p (+ x 1) (q x) x
4) Pfac = Y (λf. β (αf) f)

1) L = λf. β (αf) f
2) n = 0 (pas d'expression libre)
3) γ f = β (αf) f
4) Pfac = Y γ

1) Il n'y a plus de λ-expressions dans *Pfac*, l'algorithme s'arrête.

La transformation de *Pfac* en super-combinateurs fournit donc les règles suivantes :

```
pfac = Y γ
γ f = β (α f) f
β p q x = p (+ x 1) (q x) x
α a b c d y =
        if (= d y )
           d
           (if (= y b)
              (* d y)
              (* (c (/ (+ d y) 2))
                 (a (+ (/ (+ d y) 2) 1) y)
              )
           )
```

Cet exemple montre que le code généré n'est pas beaucoup plus important que le programme source. Par rapport aux combinateurs, il est beaucoup moins important.

De plus, HUGUES a ajouté à l'algorithme quelques optimisations telles que

- la détection et l'élimination des combinateurs et variables redondants,
- l'optimisation de l'ordre des paramètres de façon à rendre les expressions libres maximales aussi grandes que possible.

Les travaux de HUGUES ont donné lieu à de nombreux travaux d'implantation de langages fonctionnels.

IV.4.5 : Exemples d'implantations par super-combinateurs

IV.4.5.1 : Implantations séquentielles

La G-machine

La G-machine, développée au "Chalmers Institute of Technology" en Suède par JOHNSSON et AUGUSTSSON [JOHNSS84], [AUGUST84], est l'une des premières réalisations séquentielles de réduction de graphe basée sur une compilation en super-combinateurs.

Une fois le programme source compilé en super-combinateurs, un algorithme de compilation en G-code prend l'ensemble des définitions de super-combinateurs *{$S1..... = ..., ..., $Sn.....= .....}* et le super-combinateur à évaluer *$PROG*, et produit un programme G-code contenant

(i) un segment de code initialisation chargé de réaliser à l'exécution les initialisations nécessaires,

(ii) un segment de G-code qui évalue le super-combinateur *$PROG* et imprime sa valeur,

(iii) un segment de G-code pour chaque définition de super-combinateur, chacun de ces segments étant identifié par un label,

(iiii) les segments de G-code correspondant à chaque fonction primitive (ils constituent la bibliothèque d'exécution puisque ce sont les mêmes pour tous les programmes).

L'état de la G-machine est donné par le triplet *<S,G,C,D>* où *S*, *G*, *C*, *D* sont les quatre composants de la machine :

- *S* représente la pile,
- *G* représente le graphe,
- *C* représente la séquence de G-code restant à exécuter,
- *D* représente le "Dump" i.e. : une pile de paires *(S,C)*.

Les opérations de la G-machine peuvent ainsi être décrites par des transitions d'état.

Les principales règles d'exécution sont les suivantes :

(i) quand l'exécution du code correspondant à un corps de super-combinateur débute, les arguments sont en sommet de pile et juste au-dessus d'eux se trouve un pointeur vers la racine du rédex.

(ii) quand l'exécution du super-combinateur est terminée, seul le pointeur vers le graphe réduit reste dans la pile. Le graphe ne peut être que partiellement réduit auquel cas la dernière instruction du super-combinateur initialise la réduction suivante.

D'autres implantations séquentielles ont été réalisées telles que la Ponder Abstract Machine [FAIRBA86], basée sur une approche similaire à la G-machine.

Bien qu'elle diverge de la réduction de graphe, une autre approche est d'utiliser un dialecte de LISP tel que SCHEME [STEELE78] ou T [BOHM64] comme intermédiaire. Cette approche, adoptée par HUDAK [HUDAK84] a l'avantage de bénéficier des efforts déjà réalisés pour obtenir des implantations performantes de Lisp.

IV.4.5.2 : Implantations parallèles

La machine GRIP [PEYTON87b]

La machine GRIP (Graph Reduction In Parallel) est une machine multi-processeurs faiblement couplée, dont les communications sont réalisées via un bus. L'un des processeurs est considéré comme le "System manager" : il communique directement avec la machine hôte et il est chargé de l'allocation des ressources globales et du contrôle. Tous les autres processeurs peuvent être considérés comme des machines de réduction de super-combinateurs et supportent des fonctionnalités d'entrée/sortie.

Le programme à exécuter, représenté sous forme de graphe est réparti sur les différentes mémoires.

L'unité de parallélisme est la tâche : elle réalise une séquence de réductions pour réduire un sous-graphe particulier. Lorsqu'une tâche a

besoin d'une valeur, elle initialise alors une sous-tâche pour évaluer le sous-graphe correspondant. Si une tâche essaie d'accéder au noeud d'un graphe en cours d'évaluation, elle est bloquée. Le mécanisme de blocage attache alors la tâche bloquée au noeud bloquant et quand l'évaluation du noeud est terminée, la tâche bloquée est libérée.

Les super-combinateurs sont représentés dans le graphe comme des types particuliers de noeuds plutôt que d'être préchargés dans les processeurs. Ceci permet à GRIP d'exécuter des programmes dont les super-combinateurs occupent plus de mémoire que les processeurs n'en ont de disponible.

Les processeurs chargent et conservent dynamiquement les parties de programmes qu'ils sont en train d'exécuter.

Les problèmes d'efficacité de la machine GRIP viennent en grande partie de l'architecture de la machine : du fait que les communications sont réalisées via un bus, elles sont relativement lentes, ce qui freine le parallélisme.

La machine FLAGSHIP [WATSON87]

L'architecture de la machine FLAGSHIP est constituée d'un ensemble de processeurs/mémoires faiblement couplés interconnectés par un réseau de communication performant.

Le graphe représentant le programme compilé est réparti sur les différentes mémoires et chaque processeur réduit les sous-graphes rangés dans sa mémoire locale.

Si un processeur doit réduire un sous-graphe qui n'est pas entièrement présent dans sa mémoire locale, une information présente à la racine de chaque sous-graphe permet à la machine de rassembler tout le sous-graphe sur le même noeud avant toute opération de réécriture, ceci étant réalisé directement par "hardware", éventuellement simultanément avec la réduction d'un autre sous-graphe.

Les super-combinateurs sont compilés en un code machine séquentiel conventionnel et à la différence de la G-machine et de GRIP, le code compilé d'un super-combinateur est rangé dans un paquet dans le graphe. L'application d'un super-combinateur à ses arguments est alors représentée par un paquet dont le premier champ représente l'adresse du paquet super-combinateur et les autres champs sont les paramètres ou leur adresse.

Le mécanisme d'exécution est "data-driven" : quand un graphe est réduit, le résultat de la réduction est envoyé à tous les autres graphes attendant le résultat.

Les premières expériences réalisées sur la machine FLAGSHIP utilisaient les combinateurs de TURNER et le λ-calcul. La réduction par super-combinateurs s'est révélée entre 10 et 100 fois plus efficace, ce qui nous amène naturellement à comparer les combinateurs et les super-combinateurs.

IV.4.6 : Comparaison entre les combinateurs et les super-combinateurs [PEYTON87a]

Les combinateurs et les super-combinateurs ont tous deux l'avantage de conduire à une machine de réduction très simple.

Avantages des combinateurs par rapport aux super-combinateurs

- un petit ensemble fixé de combinateurs peut être implanté directement au niveau "hardware". En revanche, les super-combinateurs étant générés lors de la compilation, ils ne peuvent pas être implantés directement au niveau hardware.

- un programme compilé en combinateurs s'exécute plus paresseusement qu'un programme compilé en super-combinateurs.

Exemple :

Soit le super-combinateur *$F x = IF* e_c e_t e_f où e_t et e_f sont des grandes expressions. Quand *$F* est appliqué, de nouvelles instances de e_t et e_f sont construites bien que l'une d'elle ne sera pas retenue.

Avec les combinateurs SK, la compilation de

$(\lambda x.\ \text{If}\ e_c\ e_t\ e_f)$

fournit :

$(\lambda x.\ \text{If}\ e_c\ e_t\ e_f)$

$\Rightarrow S\ (\lambda x.\ \text{If}\ e_c\ e_t)\ (\lambda x.\ e_f)$

$\Rightarrow S\ (S\ (\lambda x.\ \text{If}\ e_c)\ (\lambda x.\ e_t))\ (\lambda x.\ e_f)$

$\Rightarrow S\ (S\ (\ S(\lambda x.\ \text{If})\ (\lambda x.\ e_c))\ (\lambda x.\ e_t))\ (\lambda x.\ e_f)$

$\Rightarrow S\ (S\ (\ S\ (K\ \text{If})\ (\lambda x.\ e_c))\ (\lambda x.\ e_t))\ (\lambda x.\ e_f)$

Supposons que : $\lambda x.\ e_c \Rightarrow c_c$ (i.e. : $\lambda x.\ e_c$ se compile en c_c),

$\lambda x.\ e_t \Rightarrow c_t$,

$\lambda x.\ e_f \Rightarrow e_f$.

L'expression initiale est donc compilée en

$S\ (S\ (S\ (K\ \text{If})\ c_c)\ c_t)\ c_f$

L'application à un argument x fournit alors :

$S\ (S\ (S\ (K\ \text{If})\ c_c)\ c_t)\ c_f\ x$

$\rightarrow S\ (S\ (K\ \text{If})\ c_c)\ c_t\ x\ (c_f\ x)$

$\rightarrow S\ (K\ \text{If})\ c_c\ x\ (c_t\ x)\ (\ c_f\ x)$

$\rightarrow K\ \text{If}\ x\ (c_c\ x)\ (c_t\ x)\ (c_f\ x)$

$\rightarrow \text{If}\ (c_c\ x)\ (\ c_t\ x)\ (c_f\ x)$

Aucune instance de c_t ou c_f n'a été construite : seule la branche sélectionnée par le *If* sera évaluée plus tard.

Le problème est que ceci est réalisé au prix d'une allocation de noeuds intermédiaires pour conserver les branches partiellement instanciées du "*If*".

Inconvénients des combinateurs par rapport aux super-combinateurs :

- La traduction en combinateurs est coûteuse par rapport aux techniques super-combinateurs et le programme résultant est plus important, ce qui augmente le nombre d'accès mémoire nécessaires et conduit également à la création et à l'accès de noeuds intermédiaires.
- Le "grain" des combinateurs est trop fin : les arguments d'une fonction sont repoussés dans le corps de la fonction d'un niveau à la fois. De cette façon, beaucoup de noeuds *application* intermédiaires sont créés. Un réducteur de combinateurs consomme donc énormément de mémoire avec tous les inconvénients que cela implique, en particulier l'accroissement de la tâche du "garbage collector".
- De plus, dans une C-expression, l'accès à une valeur n'est pas réalisé en temps constant, ce qui contribue fortement à l'inefficacité des combinateurs.

P. HUDAK pense également que la granularité des combinateurs est trop fine mais il fait remarquer dans [HUDAK85] que celle des super-combinateurs est trop grande. Il définit alors les combinateurs sériels qui sont de granularité moyenne.

IV.5 : Les combinateurs sériels [HUDAK85]

IV.5.1 : motivations

D'après HUDAK, les super-combinateurs sont conçus pour des machines séquentielles : leur granularité est trop grande ; ils peuvent contenir du parallélisme interne qui est perdu. Le but des combinateurs sériels est de conserver les avantages des combinateurs et augmenter leur granularité tout en s'assurant que l'on ne perd pas de parallélisme.

Un combinateur sériel est un raffinement d'un super-combinateur dans lequel plusieurs contraintes ont été ajoutées, la plus importante étant qu'il n'ait pas de sous-structure concurrente, c'est-à-dire qu'il ne soit pas contenu dans un combinateur plus grand ayant la même propriété.

IV.5.2 : Algorithme de génération de combinateurs sériels

L'algorithme de génération de combinateurs sériels est le suivant :

1) traduire le programme source en une λ–expression,

2) convertir la λ-expression en super-combinateurs par la méthode de HUGUES [HUGHES82],

3) Convertir en combinateurs sériels en faisant, pour chaque super-combinateur :

a) Déterminer les sous-structures concurrentes (ceci revient déterminer les opérateurs primitifs qui sont stricts pour plus d'un de leurs arguments, tels que +, -, =, >, etc...)

b) En partant de l'expression la plus externe, pour chaque sous-arbre contenant plusieurs sous-expressions concurrentes, retenir l'une de ces sous-expressions dans la définition du combinateur courant et compiler chacune des autres sous-expressions dans un combinateur sériel séparé (afin de permettre leur évaluation en parallèle).

c) pour chaque combinateur sériel ainsi généré, répéter l'étape 3b) et arrêter quand aucun raffinement n'est fait

IV.5.3 : Exemple d'application de l'algorithme

Dans la section IV.4.4, nous avions pris l'exemple d'un programme ALFL qui calculait *n!* par la méthode "diviser pour régner". Nous avions alors généré l'ensemble des super-combinateurs suivant :

```
pfac = Y γ
γ f = β (α f) f
β p q x = p (+ x 1) (q x) x
α a b c d y =
        if (= d y )
           d
           (if (= y b)
              (* d y)
              (* (c (/ (+ d y) 2))
                 (a (+ (/ (+ d y) 2) 1) y)))
```

Appliquons l'algorithme de génération en combinateurs sériels :

a) les sous-structures concurrentes sont :

```
(c (/ (+ (d y ) 2))
```

et

```
(a (+ (/ (+ d y ) 2) 1) y)
```

b) On conserve la première et on transforme la deuxième en combinateur sériel. On obtient :

```
η j k m = j (+ (/ (+ k m) 2) 1) m)
α a b c d y =
    if ( = d y )
       d
       (if ( = y b)
          (* d y)
          (* (c (/ (+ (d y) 2))
             (η a d y)
          )
       )
```

c) On recommence l'algorithme pour le nouveau combinateur η mais comme celui-ci n'a pas de sous-structure concurrente, l'algorithme s'arrête.

Grâce à cet exemple, on remarque que :

- Les combinateurs sériels conservent les bonnes propriétés des combinateurs et super-combinateurs, facilitant leur utilisation dans la réduction de graphe.
- Ils s'exécutent aussi "paresseusement" que les super-combinateurs.
- Ils n'ont pas de sous-structure concurrente et donc, on ne perd pas de parallélisme.
- Il n'existe pas de plus grands objets ayant les mêmes propriétés. L'exécution n'engendre donc pas de coût supplémentaire dû à une granularité trop fine.

IV.5.4 : Exemple d'implantation

HUDAK définit dans [MYCROF81], un modèle abstrait d'implantation. Dans ce modèle, le programme (sous forme de combinateurs sériels) est représenté par un graphe de façon classique et on considère qu'il y a un espace d'adressage global.

Un noeud du graphe est dans l'un des trois états
- dormant : attendant d'être évalué
- actif : en cours d'évaluation
- terminal : complètement évalué.

L'évaluation est réalisée par échange de messages entre les différents noeuds : les messages "*get-val*" demandant le résultat de l'évaluation d'un noeud et le message "*return-val*" permettant de retourner la valeur d'un argument.

Dans le modèle multiprocesseurs, l'échange de messages définit le protocole de communication inter-processeurs. Chaque processeur est responsable d'une portion contigüe de l'espace d'adressage global. Une file de tâches sur chaque processeur contient essentiellement des messages *get-val* ou *return-val*, agissant comme des pointeurs vers des rédex disponibles.

L'exécution du programme consiste à plaquer le graphe sur l'espace mémoire et envoyer un message *get-val* au noeud racine. Au fur et à mesure de l'évaluation, des portions du graphe sont distribuées aux processeurs voisins pour accroître le parallélisme. Cette distribution est contrôlée par un mécanisme dynamique de répartition de charge, prenant en compte des facteurs tels que la charge des processeurs, l'utilisation mémoire etc...

Ce modèle a été implanté sur un hypercube iPSC d'Intel à 128 noeuds qui montre que les combinateurs sériels apparaissent plus efficaces que les combinateurs S, K, I, et les super-combinateurs de HUGHES.

Nous terminons ce chapitre sur la réduction par combinateurs en présentant une autre famille de combinateurs utilisée dans la machine MaRS developpée au CERT-DERI de Toulouse.

IV.6 : Les combinateurs de MaRS

[CASTAN86a], [CASTAN86b], [CASTAN87]

IV.6.1 : l'idée

L'idée de combinateurs indicés a été initialement developpée par CURRY et FEYS [CURRY58]. Les combinateurs S, K, I, permettent d'abstraire les variables d'une λ-expression et cette abstraction est réalisée variable après variable. Les combinateurs indicés permettent au contraire de réaliser une abstraction par rapport à n'importe quelle variable ce qui donne plus de souplesse dans le choix de l'ordre d'abstraction.

Cette idée a été reprise dans la machine MaRS : les auteurs ont défini une famille de combinateurs indicés, l'indice correspondant à un numéro d'ordre de la variable dans l'expression.

La machine MaRS (Machine multi-processeurs à Reduction Symbolique) est développée au CERT-DERI de Toulouse et est dédiée à la réduction de graphe pour langages fonctionnels. Les combinateurs utilisés dans MaRS sont basés sur les combinateurs S, K, I.

IV.6.2 : Règles de réduction des combinateurs

Un ensemble fixé de combinateurs indicés est défini par les règles de réduction suivantes :

1) Le combinateur de déplacement

$$M_n\, e_1\, e_2 ... e_n \rightarrow e_1\, e_n\, e_2 .... e_{n-1} \quad (n \geq 3)$$

2) Le combinateur de copie-déplacement

$W_n\ e_1\ e_2 ... e_n \rightarrow e_1\ e_n\ e_2 .... e_{n-1}\ e_n\ (n \geq 2)$

3) Le combinateur de création de noeud

$N_n\ e_1\ e_2 ... e_n \rightarrow e_1\ (e_2\ e_n)\ e_3 ... e_{n-1}\ (n \geq 3)$

4) Le combinateur de copie-création de noeud

$Q_n\ e_1\ e_2 .. e_n \rightarrow e_1\ (e_2\ e_n)\ e_3 ... e_n\ (n \geq 3)$

5) Le combinateur de composition

$B\ x\ y\ z \rightarrow x\ (y\ z)$

6) Le combinateur de destruction

$K_n\ e_1\ e_2 ... e_n \rightarrow e_1\ e_2 ... e_{n-1}\ (n \geq 3)$

7) Le combinateur identité

$I\ x \rightarrow x$

IV.6.3 : Algorithme d'abstraction

Notation :

$<x_1, ..., x_n> E$ désigne le résultat de l'abstraction de l'expression E contenant seulement les variables $x_1 ... x_n$ et des constantes.

Donc : si $F = < x_1, ..., x_n> E$ alors $F\ x_1 ... x_n \rightarrow E$ pour tout x_i.

L'idée de l'algorithme est de réduire par étapes successives l'expression E dans la forme $(k\ x_1 \ ...\ x_n)$ où k contient seulement des combinateurs et les constantes de E.

Après chaque étape de transformation, l'expression intermédiaire équivalente de E a donc la forme canonique suivante :

$(k\ e_1 \ ...\ e_p\ v_1 \ ...\ v_m)$, $p \geq 0$, $0 \leq m \leq n$

où k est une constante,

les e_i sont des expressions telles que e_i n'est pas une constante

$v_1 \ ...\ v_m$ est un sous-ensemble ordonné de $x_1 \ ...\ x_n$.

L'algorithme est alors donné par un ensemble de règles de transformation.

De façon générale, toute expression peut se ramener à la forme canonique en utilisant les règles de transformation suivante :

(0.1) $x_i\ e_2 \ldots e_p\ v_1 \ldots v_m \Rightarrow I\ x_i\ e_2 \ldots e_p\ v_1 \ldots v_m$

(0.2) $k_1 \ldots k_p\ e_1 \ldots e_n\ v_1 \ldots v_m \Rightarrow (k_1 \ldots k_p)\ e_1 \ldots e_n\ v_1 \ldots v_m$

Les règles de transformation d'une expression sous forme canonique sont :

(1) Si $e_1 = x_i$ et $x_i \notin (v_1, \ldots, v_m)$ alors

$k\ x_i\ e_2 \ldots e_p\ v_1 \ldots v_m \Rightarrow M_j\ k\ e_2 \ldots e_p\ u_1 \ldots u_{m+1}$

avec

- $u_1 \ldots u_{m+1} = v_1 \ldots v_m$ dans lequel x_i a été inséré à sa place
- $j = p + q$ où q est tel que $x_i = u_q$

(2) Si $e_1 = x_i$ et $x_i \in (v_1, \ldots, v_m)$ alors

$k\ x_i\ e_2 \ldots e_p\ v_1 \ldots v_m \Rightarrow W_j\ k\ e_2 \ldots e_p\ v_1 \ldots v_m$

avec

$j = p + q$ où q est tel que $x_i = v_q$

(3) Si e_1 n'est pas une variable et $e_1 = (d_1 \ldots d_r\ x_i)$, $r \geq 1$, $x_i \notin (v_1, \ldots v_m)$ alors

$k\ (d_1 \ldots d_r\ x_i)\ e_2 \ldots e_p\ v_1 \ldots v_m \Rightarrow$

$N_j\ k\ (d_1 \ldots d_r)\ e_2 \ldots e_p\ u_1 \ldots u_{m+1}$

avec

- $j = p + q + 1$
- $u_1 \ldots u_{m+1}$ est défini comme dans (1)

(4) si e_1 n'est pas une variable et $e_1 = (d_1 \ldots d_r\ x_i)$, $r \geq 1$, $x_i \in (v_1, \ldots, v_m)$ alors

$k\ (d_1 \ldots d_r\ x_i)\ e_2 \ldots e_p\ v_1 \ldots v_m \Rightarrow$

$Q_j\ k\ (d_1 \ldots d_r)\ e_2 \ldots e_p\ v_1 \ldots v_m$

avec

$j = p + q + 1$

(5) Si $e_1 = (d_1 \ldots d_r)$, $r > 1$ et $d_r \neq x_i$ alors

$k\ (d_1 \ldots d_r)\ e_2 \ldots e_p\ v_1 \ldots v_m \Rightarrow$

$B\ k\ (d_1 \ldots d_{r-1})\ d_r\ e_2 \ldots e_p\ v_1 \ldots v_m$

Après application de ces différentes règles, l'expression initiale

$k' e_1 \ldots e_p v_1 \ldots v_m'$

est transformée en la forme

$k\ v_1 \ldots v_m \quad (p = 0)$

Les règles suivantes permettent alors d'extraire les variables de l'expression

(6) si $x_n = v_m$, $n \geq 1$ alors

$\langle x_1 \ldots x_n \rangle\ (k\ v_1 \ldots v_m) \Rightarrow \langle x_1 \ldots x_{n-1} \rangle\ (k\ v_1 \ldots v_{m-1})$

(7) si $x_n \neq v_m$, $n \geq 0$ alors

$\langle x_1 \ldots x_n \rangle\ (k\ v_1 \ldots v_m) \Rightarrow \langle x_1 \ldots x_{n-1} \rangle\ (K_j\ k\ v_1 \ldots v_m)$

avec

$j = m + 2$

(8) $< > k = k$

IV.6.4 : Exemple d'application de l'algorithme

Soit l'expression suivante à transformer :

$E = (* (+ x\ 2)\ (+ y\ x))$

Nous notons en début de ligne, la règle appliquée pour la transformation :

$* (+ x\ 2)\ (+ y\ x)$

(5) $\Rightarrow B * (+ x)\ 2\ (+ y\ x)$

(3) $\Rightarrow N_5\ (B *) + 2\ (+ y\ x)\ x$

(4) $\Rightarrow Q_3\ (N_5\ (B *) + 2)\ (+ y)\ x$

(3) $\Rightarrow N_4\ (Q_3\ (N_5\ (B *) + 2)) + x\ y$

Il reste à abstraire les variables de l'expression :

$\langle x, y\rangle$ ((N4 (Q3 (N5 (B *) + 2)) +) x y)
(6) $\Rightarrow \langle x\rangle$ ((N4 (Q3 (N5 (B *) + 2)) +) x)
(6) $\Rightarrow$ (N4 (Q3 (N5 (B *) + 2)) +)

On obtient ainsi l'expression combinatoire correspondant à l'expression initiale. Appliquons cette expression aux arguments 4 et 5 par exemple, il suffit d'appliquer les règles de réduction des combinateurs :

N4 (Q3 (N5 (B *) + 2)) + 4 5
(3) $\rightarrow$ Q3 (N5 (B *) + 2) (+ 5) 4
(4) $\rightarrow$ N5 (B *) + 2 (+ 5 4) 4
(3) $\rightarrow$ B * (+ 4) 2 (+ 5 4)
(5) $\rightarrow$ * (+ 4 2) (+ 5 4)
$\rightarrow$ * 6 9
$\rightarrow$ 54.

Les combinateurs de MaRS offrent les avantages des combinateurs pour la réduction de graphe en réduisant le code combinatoire généré.

IV.6.5 : Exemple d'implantation

La réduction au moyen de cette famille de combinateurs indicés est implantée sur la machine MaRS de façon classique. Pour réaliser ces réductions, la machine MaRS contient 4 types de processeurs :

- mémorisation : ils gèrent le graphe représentant le programme et répondent aux requêtes des processeurs de réduction.
- réduction : ils appliquent les règles de réduction sur le graphe.
-communication : ils gèrent les échanges d'informations par messages.
-E/S : ils gèrent les opérations d'entrées/sorties.

Ceci termine notre chapitre sur la réduction par combinateurs. Le chapitre suivant nous permet de faire un petit bilan des travaux décrits jusqu'à maintenant afin d'introduire le travail effectué dans le cadre de cette thèse.

V : CONCLUSION SUR LES TRAVAUX EXISTANTS

V : CONCLUSION SUR LES TRAVAUX EXISTANTS

Les langages fonctionnels suppriment en grande partie les problèmes que l'on rencontre dans les langages impératifs, du fait de leurs propriétés mathématiques. Parmi les langages fonctionnels, on a distingué les lambda-langages des langages sans variable.

Les lambda-langages font une "bonne utilisation" des variables, ce qui empêche les effets de bord. Néanmoins, les variables continuent à poser des problèmes qui conduisent dans certaines implantations, à la gestion d'un environnement formé de couples (variables, valeurs). La gestion d'un environnement est très lourde, surtout dans une implantation parallèle, c'est pourquoi une autre solution pour pallier le problème des variables consiste à compiler les λ-expressions en expressions combinatoires, i.e. : des expressions formées à partir de combinateurs, c'est-à-dire des opérateurs qui sont définis, indépendamment de tout contexte, par des règles de réduction. Pour cela, les combinateurs de Turner ou les combinateurs de la machine MaRs permettent de transformer une lambda-expression en une expression combinatoire ne contenant que des combinateurs ou des constantes, tandis que les supercombinateurs de HUGHES ou les combinateurs sériels permettent de supprimer les variables libres d'une λ-expression pour obtenir une expression combinatoire sous la forme

$$\$S\ x_1 \ldots x_n = E(x_1, \ldots, x_n)$$

Les variables formelles utilisées dans cette expression n'ont pas la même signification que les variables au sens informatique du terme : elles ne désignent pas un emplacement mémoire mais servent uniquement à donner un nom à un objet ou une expression. Elles ont donc la même signification que les variables utilisées dans les systèmes de réécriture.

La transformation d'une λ-expression en une expression combinatoire supprime donc les problèmes dus aux variables libres dans les λ-expressions et de plus, elle conserve le parallélisme intrinsèque au langage source. De ce fait, elle facilite l'implantation des λ-langages sur machine parallèle.

Pour notre modèle, nous avons choisi comme langage support, FP, le langage sans variable de Backus. Ce choix a été guidé tout d'abord par notre intérêt pour les langages sans variable. Ils permettent un style de programmation tout-à-fait particulier, très différent du style de programmation des λ-langages (et a fortiori des langages impératifs). Ils incitent le programmeur à concevoir des programmes naturellement parallèles. Ces raisons nous ont incité à nous y intéresser.

Le modèle est défini pour le langage FP mais nous envisageons de l'étendre aux systèmes FFP, afin de permettre la manipulation de fonctions d'ordre supérieur. On peut considérer que les systèmes FFP représentent le "coeur" de tout langage fonctionnel sans variable. Les autres langages sans variable reprennent et développent les concepts énoncés dans FP.

Nous disposons donc d'un support simple et général à une classe de langages offrant des caractéristiques intéressantes pour l'évaluation parallèle (du fait du parallélisme intrinsèque au langage). De plus, FP offre un ensemble suffisamment riche de fonctions primitives permettant la manipulation d'objets structurés.

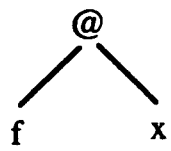
Dans le chapitre III, nous avons présenté les deux principaux modèles d'évaluation parallèle de langages fonctionnels qui sont le modèle data-flow et le modèle de réduction. Le modèle data-flow présente, comme nous l'avons vu, les inconvénients de ne pas permettre aisément le contrôle du parallélisme et la manipulation de structures de données complexes, et de mal supporter l'exécution de fonctions récursives.

On peut différencier le modèle de réduction de chaîne du modèle de réduction de graphe. Nous avons montré les inconvénients du modèle de réduction de chaîne. Le modèle de réduction de graphe nous semble mieux adapté à l'exécution des langages fonctionnels.

Nous avons vu dans le chapitre concernant la réduction par combinateurs, que les expressions combinatoires sont exécutées dans la plupart des cas, selon le modèle de réduction de graphe.

Un programme écrit dans un langage sans variable peut être assimilé à une expression combinatoire : les formes fonctionnelles, qui

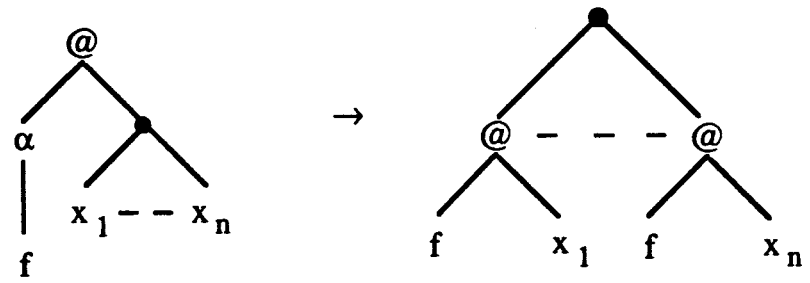
permettent de combiner les fonctions afin d'en construire de nouvelles jouent le rôle de combinateurs. Un langage sans variable peut donc être exécuté selon le même modèle de réduction, dans lequel le programme est représenté par un graphe : l'application d'une fonction f à un argument x est généralement représentée par un noeud application dont le fils gauche repère le graphe représentant la fonction f et le fils droit repère l'argument x, ce que nous représenterons par



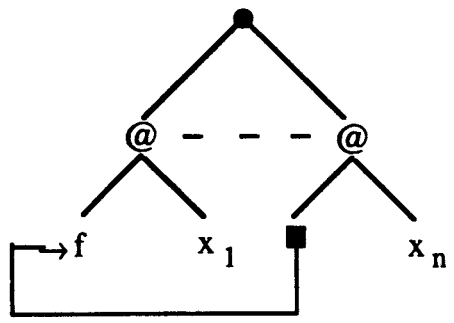
Exemple :

Si l'on représente la forme fonctionnelle αf (en FP) par

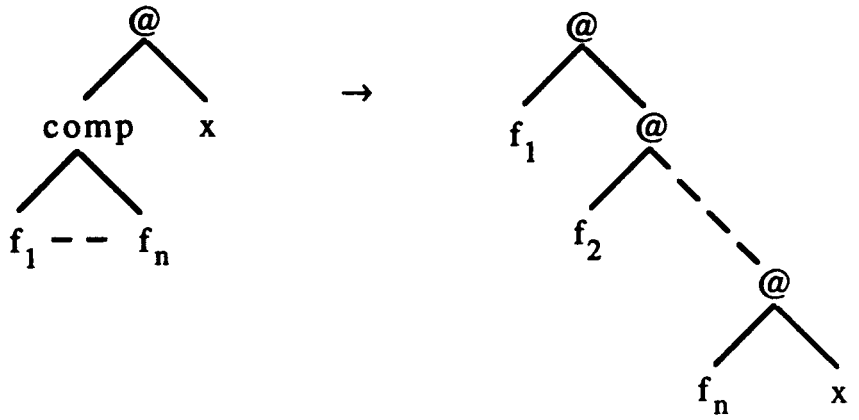
cette forme fonctionnelle est définie par la règle de réduction suivante :



ou



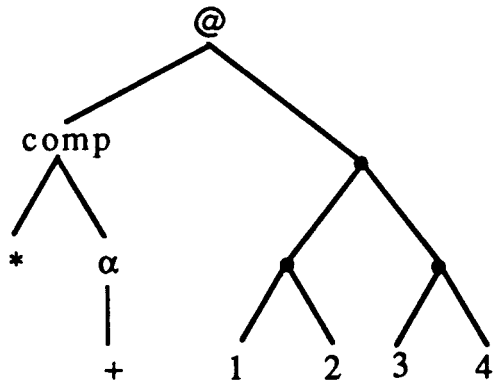
De même, la règle de réduction correspondant à la composition des fonctions $f_1 \ldots f_n$ serait définie de la façon suivante :



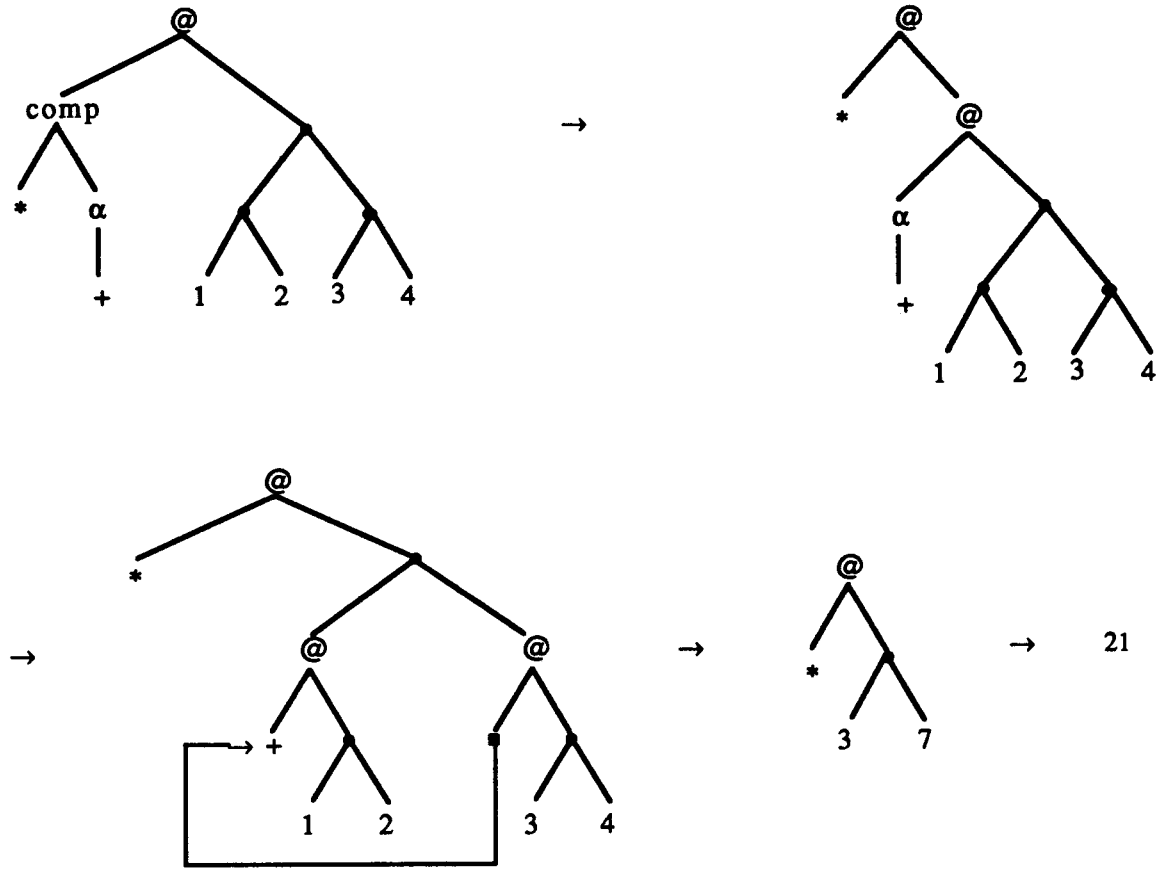
Soit $P = * \circ \alpha +$ un programme FP.

Soit $X = \langle\langle 1, 2\rangle, \langle 3, 4\rangle\rangle$, la séquence sur laquelle est appliqué le programme P.

En réduction de graphe, et étant données les notations précédemment définies, l'application de P à son argument est représentée par :



Les différentes étapes de réduction sont alors les suivantes :



Cet exemple fait apparaître les inconvénients du modèle de réduction de graphe.

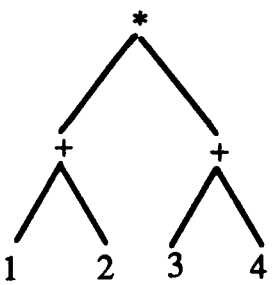
1) La recherche dans un graphe d'une expression réductible est coûteuse à mettre en oeuvre.

2) La réduction de graphe conduit à la création et donc la gestion de nombreux noeuds applications intermédiaires qui encombrent l'espace mémoire.

3) L'instanciation d'une lambda-expression consiste grossièrement à remplacer chaque occurrence d'une variable liée par l'argument (ce qui nécessite un parcours du graphe représentant la λ-expression, pour rechercher les variables). L'instanciation d'un combinateur consiste à appliquer la règle de réduction (ou règle de réécriture) correspondant au combinateur et à réécrire le graphe réductible en la partie droite de règle dans laquelle les "variables" sont remplacées. Les combinateurs permettent de simplifier l'étape d'instanciation dans la réduction, dans la mesure où il n'y a pas d'environnement à gérer mais l'instanciation

reste une opération coûteuse, comme toute opération de transformation de graphe.

Notre but était donc de définir un modèle permettant de réduire ces inconvénients. Dans le modèle, le programme *P* de l'exemple ci-dessus sera représenté par une arborescence qui, lorsqu'elle sera explorée, permettra d'envoyer les fonctions + et * directement à leurs arguments pour obtenir



sans que l'arborescence *P* ne soit modifiée.

Le fait d'avoir deux représentations distinctes pour le programme et l'argument permet :

1) de distribuer les "tâches" (les fonctions + et * dans notre exemple) à la séquence argument, qui n'a plus besoin de les rechercher.

2) de ne pas créer de noeuds applications intermédiaires : les arborescences sont statiques (i.e. : non modifiées) et les opérateurs sont envoyés directement à leur argument.

3) il n'y a plus d'opération d'instanciation proprement dite et donc, plus de transformation de graphe.

Dans la deuxième partie, nous présentons le modèle formel d'évaluation parallèle défini dans le cadre de cette thèse.

DEUXIEME PARTIE :

LE MODELE FORMEL D'EXECUTION PARALLELE DE FP

Dans cette deuxième partie, nous décrivons le modèle d'exécution parallèle de FP en essayant de nous dégager au maximum de tout problème d'implantation et de toute machine cible. Dans un premier temps, après avoir donné une idée intuitive du schéma d'exécution, nous décrirons la représentation d'un programme FP sous forme d'arborescences puis l'exploration parallèle de ces arborescences qui sert de moteur au modèle d'exécution.

Ces deux points ayant mis en évidence la nécessité d'établir une communication entre les différents objets manipulés dans le modèle, nous ferons alors un premier choix "de mise en oeuvre" qui nous orientera vers une solution basée sur une communication par messages entre les différents objets. Dans un troisième temps, nous décrirons donc l'exploration des arborescences en termes d'émission et réception de messages et nous montrerons de quelle façon cette exploration peut générer les réductions et comment ces réductions sont réalisées.

Enfin nous montrerons la nécessité de mettre en oeuvre un mécanisme d'ordonnancement des messages générés lors de l'exploration des arborescences et nous décrirons ce mécanisme.

VI : UNE PRESENTATION FORMELLE DU MODELE

VI : UNE PRESENTATION FORMELLE DU MODELE

VI.1 : concepts et parallélisme

Le but de cette description est de donner une idée intuitive du schéma d'exécution proposé et de la façon dont il sera présenté dans la suite.

Le modèle d'exécution est basé d'une part sur une représentation particulière d'un programme FP et d'autre part sur une représentation de la séquence sur laquelle s'applique le programme. La représentation du programme FP illustre la façon dont le programme peut être découpé en sous-programmes ayant certaines relations entre eux.

Par exemple, deux sous programmes peuvent être indépendants et être ainsi exécutés en parallèle ; ils peuvent au contraire être liés par une relation de dépendance exprimant que l'un devra obligatoirement être exécuté après l'autre.

Ceci conduit à représenter un programme FP sous la forme d'un ensemble d'arborescences, que nous appelons ***arborescences fonctionnelles***. Une arborescence fonctionnelle représente un sous-programme. Chaque noeud dans une arborescence est la racine d'une sous-arborescence représentant également un sous-programme. Les arcs reliant les différents noeuds sont de différentes natures et expriment les relations entre les sous-programmes.

Selon notre modèle, l'exécution d'un programme FP consiste à explorer les arborescences fonctionnelles le composant et à analyser celles-ci de façon à pouvoir acheminer les fonctions primitives vers leur séquence argument. Ainsi, à l'inverse des principaux modèles de réduction, les opérandes ne sont pas acheminés vers l'opérateur mais dans un premier temps, ce sont les fonctions qui sont amenées à leur argument. Lorsqu'une fonction ainsi envoyée arrive à destination, elle

possède son argument et des mécanismes classiques de réduction sont alors générés sur la séquence argument.

Les arborescences sont explorées selon certaines règles déterminées par la nature des différents sous-programmes et leurs relations entre eux. Ainsi, si deux sous-programmes sont indépendants, les sous-arborescences fonctionnelles correspondantes peuvent être parcourues en parallèle, générant alors l'acheminement en parallèle de plusieurs fonctions primitives vers leur séquence argument et donc des opérations de réduction en parallèle sur la séquence argument.

La figure VI.1 schématise le principe du modèle d'exécution ainsi défini.

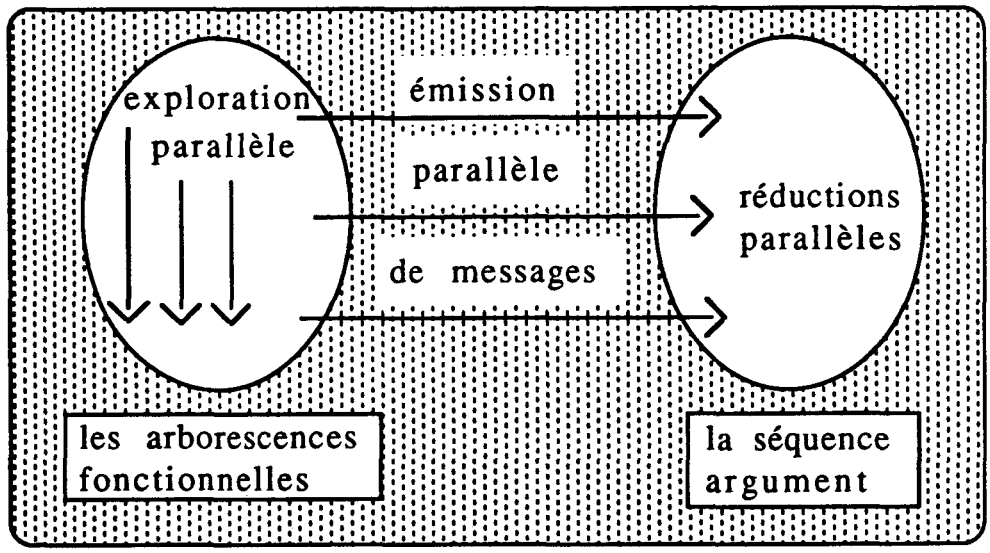


Figure VI.1 : Schéma du modèle d'exécution

Les arborescences fonctionnelles constituent donc l'organe de commande et de contrôle du modèle d'exécution parallèle d'un programme FP. Plus précisément, ce sont les règles d'exploration des arborescences fonctionnelles qui permettent d'assurer le contrôle de l'exécution. Elles permettent en effet de déterminer quelles arborescences peuvent être explorées en parallèle et plus généralement d'imposer un "ordre" d'exploration de ces arborescences.

On peut observer dès à présent que les arborescences fonctionnelles sont statiques : elles ne sont pas modifiées par l'exécution du programme FP correspondant, dans la mesure où les seuls traitements effectués sont leur exploration et l'envoi de fonctions primitives à la séquence argument. Le caractère statique des arborescences fonctionnelles permettra un partage de code dans le cas d'appels simultanés d'une même fonction, comme nous le verrons par la suite. La liste argument à l'inverse est dynamique : elle est modifiée à chaque application de fonction.

La suite de ce chapitre concerne la description formelle du modèle d'exécution avec, dans un premier temps, la description des arborescences fonctionnelles et dans un deuxième temps, l'exploration parallèle de ces arborescences.

VI.2 : Les arborescences fonctionnelles

Nous décrivons dans un premier temps de quelle façon nous découpons un programme FP en sous-programmes ce qui nous conduira, dans un deuxième temps à définir les arborescences fonctionnelles.

VI.2.1 : Découpage d'un programme FP en sous-programmes

Un programme FP est constitué d'un ensemble de définitions. Parmi ces définitions, l'une peut être considérée comme le moteur du programme, c'est-à-dire comme ***définition principale***. Par opposition, les autres définitions seront considérées comme définitions ***secondaires***. Ainsi,

$\forall$ PR tel que PR est un programme FP, PR = $\{def_1, def_2, \ldots, def_n\}$
où def_1 est la définition principale de PR
et si $n > 1$, $\forall i \in [2, n]$, def_i est une définition secondaire.

Les définitions secondaires peuvent être considérées comme des sous-programmes, ce qui constitue un découpage "grossier" du programme en sous-programmes. Ce découpage n'est pas suffisamment fin pour que l'on puisse en déduire des relations entre les sous-programmes. Pour obtenir un découpage plus fin, intéressons-nous à une définition.

Soit P l'ensemble des fonctions primitives et soit FF l'ensemble des formes fonctionnelles de FP. L'ensemble PR $\cup$ P $\cup$ FF constitue un ensemble de fonctions FP.

Chaque définition appartenant à PR peut s'écrire sous la forme d'une composition de fonctions :

$\forall i \in [1, n]$, $def_i = f_1 \circ f_2 \circ ... \circ f_m$ / $\forall j \in [1, m]$, $f_j \in PR \cup P \cup FF^-$

où FF^- est l'ensemble des formes fonctionnelles privé de la composition

i.e. : $f_i \neq f_{i1} \circ f_{i2} \circ ... \circ f_{in}$; la décomposition est maximale.

L'ensemble des fonctions $\{f_1, f_2, ..., f_m\}$ peut à nouveau être considéré comme un ensemble de sous-programmes. Etant donnée la nature des fonctions FP, on peut dire que ces sous-programmes sont "presque" indépendants. En effet, ils ne partagent aucune information mais sont liés par composition. Cela signifie qu'en pratique, ils ne pourront pas être exécutés en parallèle mais au contraire, dans un ordre strict. Heureusement, cela ne signifie pas que l'exécution d'un programme FP est séquentielle ! Pour extraire le parallélisme de FP, il faut effectuer un découpage en sous-programmes encore plus fin et s'intéresser aux fonctions.

Rappels :

- $PR = \{def_1, ..., def_n\}$
- $\forall i \in [1, n]$, $def_i = f_1 \circ f_2 \circ ... \circ f_m$
- $\forall j \in [1, m]$, $f_j \in PR \cup P \cup FF^-$

Nous allons étudier successivement les cas où f_j est élément de P, de FF^- et de PR.

1er cas : $f_j \in P$, f_j est donc une fonction primitive.

C'est une entité indivisible (dans FP) qui constitue donc le plus petit sous-programme. Néanmoins, ce sous-programme peut conduire à une exécution parallèle selon la nature de la fonction primitive.

Exemple :

La fonction *Distribute-from-left* est définie par :

$distl : x \equiv x = \langle y, \emptyset \rangle \rightarrow \emptyset$;

$x = \langle y, \langle z_1, ..., z_m \rangle \rangle \rightarrow \langle \langle y, z_1 \rangle, ..., \langle y, z_m \rangle \rangle$

Sans faire aucun a priori sur la représentation des séquences, on peut concevoir l'exécution de la fonction *Distl* comme la construction simultanée des sous-séquences $< y, z_i > , i \in [1, n]$.

En revanche, d'autres fonctions primitives se prêtent difficilement à une exécution parallèle. L'exemple le plus trivial (pour ne pas dire inintéressant) est la fonction *Identité* définie par $(id : x \equiv x)$. En termes d'exécution, c'est la fonction qui "ne fait rien". Il est donc difficile d'en concevoir une exécution parallèle.

2ème cas : $f_j \in FF^-$, f_j est donc une forme fonctionnelle.

Une forme fonctionnelle est une combinaison de fonctions et peut donc s'écrire sous la forme $ff(f_1 , ... , f_n)$ où ff est un symbole ou un nom désignant le type de la forme fonctionnelle et $\forall\, i \geq 1$, f_i désigne un paramètre de la forme fonctionnelle c'est-à-dire en général, une fonction appartenant à l'ensemble $PR \cup P \cup FF$.

Remarque :

Dans le cas où les paramètres sont des objets, ils ne peuvent être décomposés. Dans la suite, nous ne nous intéresserons donc qu'aux paramètres appartenant à l'ensemble $PR \cup P \cup FF$.

Chaque paramètre (non objet) de la forme fonctionnelle peut être considéré comme un sous-programme. Ces sous-programmes sont totalement indépendants les uns des autres. Il existe néanmoins des relations entre eux ; ils sont tous paramètres de la forme fonctionnelle, et ils sont en quelque sorte ordonnés : f_i est le $i^{ème}$ paramètre de la forme fonctionnelle. Chaque paramètre f_i, élément de l'ensemble $PR \cup P \cup FF$, s'écrit sous la forme $f_i = f_{i1} \circ ... \circ f_{in}$ et peut donc à nouveau être décomposé en sous-programmes selon les mêmes règles.

L'essentiel du parallèlisme de FP se trouve dans certaines formes fonctionnelles. En effet, la forme fonctionnelle *Construction* est définie par :

$$[f_1, ..., f_n] : x \equiv < f_1 : x, ..., f_n : x >.$$

Les éléments constituant la séquence résultat sont donc le résultat d'une application de différentes fonctions à un même objet x. Les fonctions f_i sont les paramètres de la construction et $\forall\, i, j \in [\, 1, n\,]$, tels que $i \neq j$, les applications $(f_i : x)$ et $(f_j : x)$ sont totalement indépendantes et peuvent être réalisées en parallèle à condition bien sûr de disposer de suffisamment d'exemplaires de la séquence x. Les fonctions f_i peuvent donc bien être considérées comme des sous-programmes indépendants. La forme fonctionnelle *Apply-to-all* est également intrinsèquemment parallèle bien qu'elle n'ait qu'un paramètre. En effet, elle est définie par

$$\alpha f : x \equiv \begin{array}{l} x = \emptyset \rightarrow \emptyset\,; \\ x = \langle x_1, \ldots, x_n \rangle \text{ et } n \geq 1 \rightarrow \langle f : x_1, \ldots, f : x_n \rangle\,; \\ \perp \end{array}$$

La deuxième clause de cette définition met bien en évidence le potentiel de parallèlisme que l'on peut mettre en oeuvre lors de l'exécution d'une telle expression : le paramètre f peut être appliqué simultanément à tous les objets de la séquence x et toutes les applications de f aux x_i sont indépendantes les unes des autres.

3ème cas : $f_j \in PR$, f_j est donc un nom de définition.

L'application de f_j à x consiste à appliquer la fonction représentant cette définition à la séquence x. Cette définition représente un sous-programme qui peut être décomposé selon les mêmes règles que précédemment.

VI.2.2 : Elaboration des arborescences fonctionnelles

Les arborescences fonctionnelles constituent une représentation d'un programme FP. Elles sont directement issues du découpage du programme en sous-programmes et mettent en évidence les relations entre les différents sous-programmes. Un noeud de l'arborescence est

appelé **noeud fonctionnel**. Nous avons précédemment défini un programme PR comme un ensemble de définitions contenant une définition principale et éventuellement des définitions secondaires :

$$PR = \{def_1, def_2, ..., def_n\}.$$

A chaque définition correspond dans le modèle, une arborescence fonctionnelle. Toute définition, de même que tout paramètre de forme fonctionnelle (non objet) s'écrit sous la forme d'une composition de fonctions $f_1 o ... o f_m$ telle que $\forall j \in [1, m], f_j \in PR \cup P \cup FF^{-}$. Dans une arborescence fonctionnelle, à chaque fonction f_j correspond alors un noeud fonctionnel. Pour exprimer le fait que ces fonctions sont liées par composition, quel que soit $j \in [1, m]$, le noeud fonctionnel représentant f_{j+1} est "relié" au noeud fonctionnel f_j par un symbole que nous appelons symbole de séquencement et représenté par le symbole " $\downarrow$". Ceci illustre alors le fait que la fonction f_j doit être appliquée à la séquence résultant de l'application de la fonction $f_{j+1} o ... o f_m$ à la séquence initiale. La figure VI.2 illustre la représentation d'une composition de fonctions.

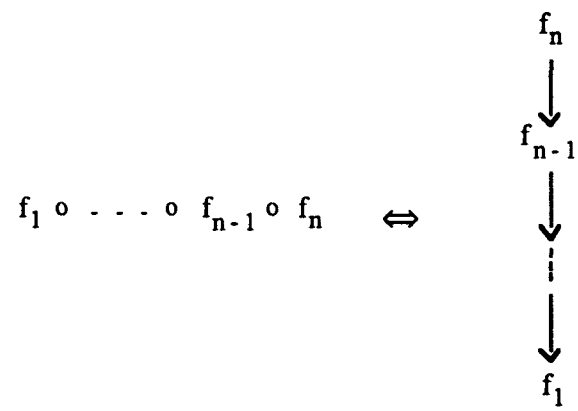


Figure VI.2 : Représentation d'une composition de fonctions dans une arborescence fonctionnelle

La représentation de chaque fonction f_j dépend de la nature de la fonction et nous allons étudier successivement les cas où il s'agit d'une fonction primitive, d'une forme fonctionnelle ou d'un nom de définition.

1er cas : La fonction f_j à représenter est une fonction primitive.

Elle constitue alors un sous-programme indivisible et donc le noeud fonctionnel représentant cette fonction primitive est suffisant. Dans les schémas qui suivent, pour représenter ce noeud fonctionnel, nous ferons figurer le nom de la fonction primitive qu'il représente.

2ème cas : La fonction f_j à représenter est une forme fonctionnelle.

Nous nous intéresserons ici à toute forme fonctionnelle autre que la composition dans la mesure où nous avons déjà donné sa représentation. Si la fonction f_j est une forme fonctionnelle, elle peut alors s'écrire sous la forme :

ff $(f_{j1}, f_{j2}, \ldots, f_{jk})$, $k \geq 1$ où

- *ff* est le symbole de forme fonctionnelle
- $\forall\, i \in [\,1, k\,]$, f_{ji} est une fonction FP ou un objet.

Si f_{ji} est un objet, un noeud fonctionnel représentera alors cet objet.

Si f_{ji} est une fonction, f_{ji} peut être écrite sous la forme d'une composition de fonctions qui peut être représentée selon les règles énoncées dans cette section. Cela signifie alors que chaque paramètre peut être représenté par une sous-arborescence fonctionnelle. Pour exprimer l'ordre des paramètres de la forme fonctionnelle, les noeuds fonctionnels racines de chacune des sous-arborescences sont reliés entre eux par un symbole que nous appellerons symbole de continuation et que nous représenterons par le symbole "$\rightarrow$". L'ensemble de ces arborescences fonctionnelles est en relation avec le noeud fonctionnel précédemment défini, représentant la fonction f_j.

Dans les schémas qui suivent, pour représenter ce noeud fonctionnel, nous ferons figurer le symbole de forme fonctionnelle correspondant. Ce noeud fonctionnel sera relié au noeud racine de la sous-arborescence représentant le premier paramètre de la forme fonctionnelle par un symbole que nous appellerons symbole de paramètre et qui sera représenté par le symbole " $\hookrightarrow$".

Avec ces conventions et étant donnés $p, f, g, f_1, \ldots, f_n$ des fonctions quelconques et x un objet, la figure VI.3 montre la représentation des différentes formes fonctionnelles dans une arborescence fonctionnelle. Dans cette figure, un triangle représente une sous-arborescence, paramètre d'une forme fonctionnelle et le paramètre représenté est indiqué dans le triangle.

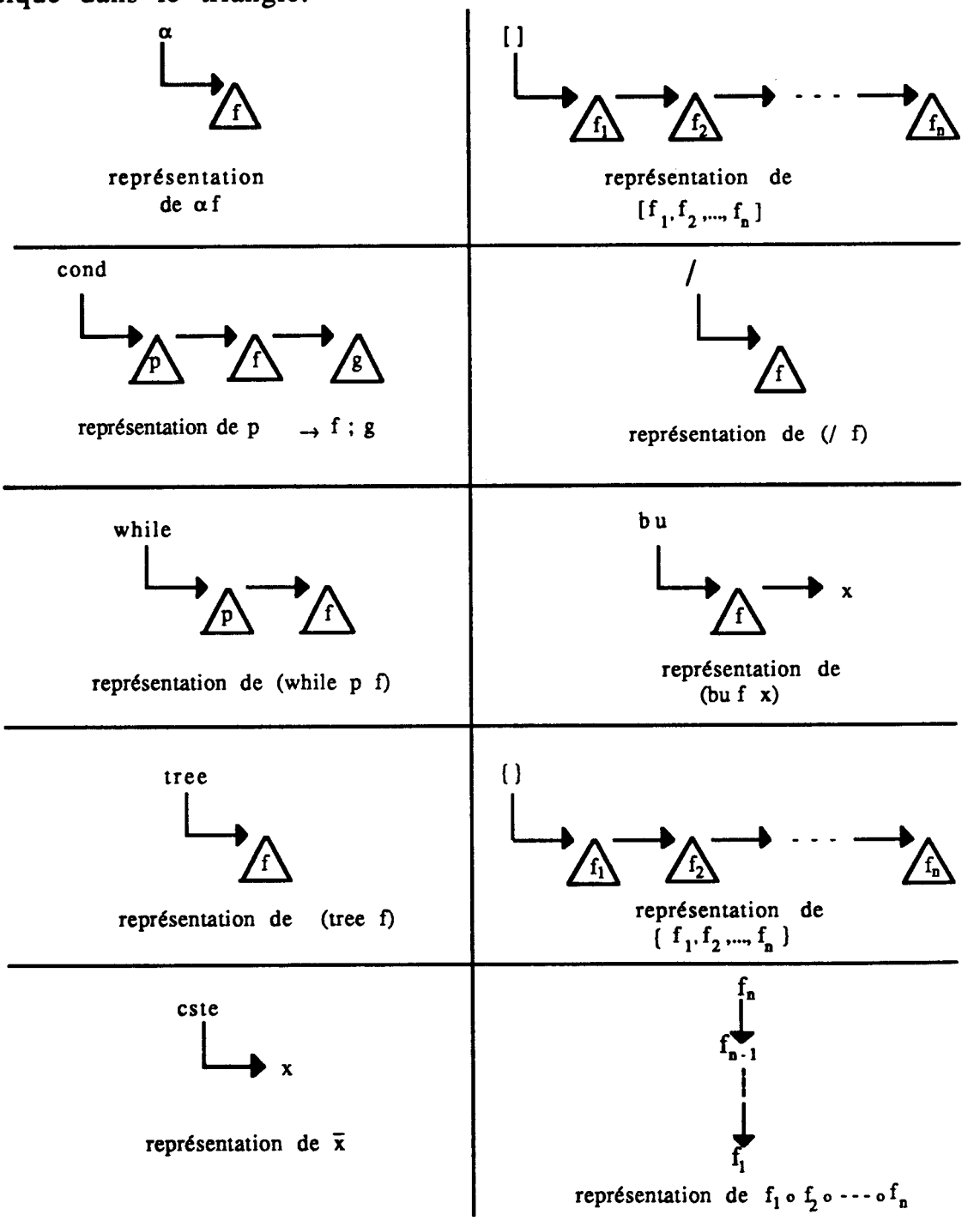


Figure VI.3 : représentation des différents formes fonctionnelles

3ème cas : La fonction f_j à représenter est un nom de définition

Une arborescence fonctionnelle représente alors cette définition. Le noeud fonctionnel représentant f_j est alors suffisant pour représenter cette information. Dans les schémas suivants, pour représenter ce noeud fonctionnel, nous ferons figurer le nom de la définition.

Ces trois règles permettent de représenter sous forme d'arborescences fonctionnelles tout programme FP.

Exemple :

Le programme de multiplication de deux matrices quelconques défini par :

Def MM = α α IP ∘ α distl ∘ distr ∘ [1, trans ∘ 2]
Def IP = (/ +) ∘ (α *) ∘ trans

est représenté par les deux arborescences fonctionnelles de la figure VI.4.

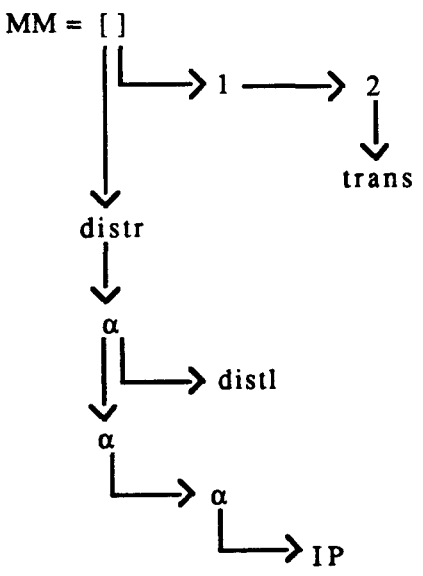


Figure VI.4.a : représentation de de la fonction MM

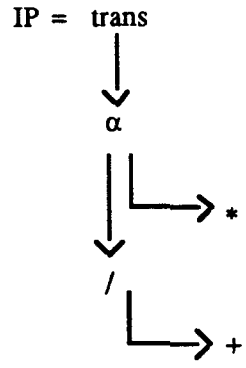


figure VI.4.b : représentation de la fonction IP

Figure VI.4 : Arborescences fonctionnelles correspondant au programme de multiplication de matrices.

La fonction MM peut également être écrite sous la forme MM' équivalente à MM, MM' étant définie par :

Def MM' = α (α IP o distl) o distr o [1, trans o 2]

La fonction MM' ainsi définie est représentée par l'arborescence fonctionnelle de la figure VI.5.

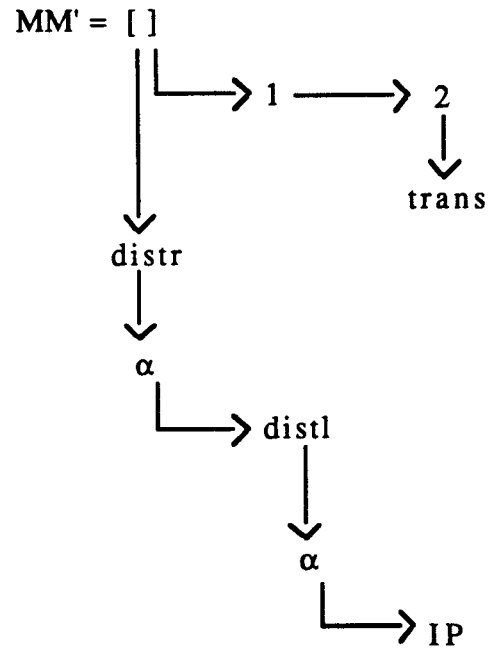


Figure VI.5 : représentation de la fonction MM'

VI.2.3 : définitions formelles

Dans cette section, nous formalisons les notions introduites précédemment de façon intuitive et nous donnons un certain nombre de définitions permettant de fixer le vocabulaire employé par la suite.

Etant donnés P l'ensemble des fonctions primitives et FF l'ensemble des formes fonctionnelles de FP, un programme FP est donné par un ensemble de définitions :

$PR = Def_1 \cup \{Def_2, ..., Def_n\}$ où

Def_1 est la définition principale,

$\forall i \in [2, n]$, Def_i est une définition secondaire,

$\forall i \in [1, n]$, $Def_i = f_1 \circ f_2 \circ ... \circ f_m / \forall j \in [1, m], f_j \in \{PR \cup P \cup FF\}$

Nous appelons ***arborescence fonctionnelle principale*** la représentation de la définition principale d'un programme FP. De même, une arborescence fonctionnelle représentant une définition secondaire sera appelée ***arborescence fonctionnelle secondaire***. L'arborescence fonctionnelle principale et les arborescences fonctionnelles secondaires sont construites de la même façon selon les règles énoncées ci-dessous.

Toute fonction FP, en particulier une définition, peut être décomposée sous la forme d'une composition de fonction :

$$\forall f \in PR \cup P \cup FF, f = f_n \circ f_{n-1} \circ ... \circ f_1 / \forall i, f_i \in PR \cup P \cup FF^-$$

• Etant donnée cette décomposition, à toute fonction f_i est associée une sous-arborescence fonctionnelle représentant la fonction f_i, notée $arb(f_i)$:

$$\forall i, f_i \mapsto arb(f_i)$$

Nous noterons $rac(arb(f_i))$, le noeud fonctionnel racine de $arb(f_i)$.

• Le noeud fonctionnel racine de $arb(f_1)$ est alors considéré comme le noeud fonctionnel racine de l'arborescence représentant la fonction f :

$$rac(arb(f_1)) = rac(arb(f))$$

• Soient n_i = rac(arb(f_i)),

n_{i+1} = rac(arb(f_{i+1})).

A tout couple (n_i, n_{i+1}) est alors associé dans l'arborescence un arc étiqueté par le symbole "↓", reliant les noeuds fonctionnels n_i et n_{i+1} ;

i.e. : (n_i, n_{i+1}) ↦ arc(n_i ↓ n_{i+1}).

On note A_1 l'ensemble des arcs d'étiquette "↓".

Les noeuds fonctionnels des arborescences sont alors obtenus de la façon suivante :

• Si f_i est un nom de définition ou de fonction primitive, $arb(f_i)$ est alors réduite à un noeud fonctionnel n_i d'étiquette f_i ;

i.e. : si $f_i \in PR \cup P$, arb(f_i) = n_i d'étiquette f_i ; n_i = rac(arb(f_i))

On appelle N_1 l'ensemble des noeuds fonctionnels d'étiquette un nom de définition ou de fonction primitive.

• Si f_i est une forme fonctionnelle autre que la composition, f_i peut alors s'écrire sous la forme $ff_i(f_{i1}, ..., f_{im})$ où

ff_i est le symbole de forme fonctionnelle et

$\forall\, j, f_{ij}$ est un paramètre de forme fonctionnelle.

Quel que soit j, f_{ij} est donc soit une fonction FP, soit un objet ;

i.e. : si $f_i \in FF^-$, $f_i = ff_i(f_{i1}, ..., f_{im}) / \forall\, j, f_{ij} \in PR \cup P \cup FF \cup O$;

Au symbole ff_i est alors associé dans l'arborescence un noeud fonctionnel n_i d'étiquette ff_i, tel que le noeud fonctionnel n_i soit la racine de $arb(f_i)$;

i.e. : ff_i ↦ n_i d'étiquette ff_i / n_i = rac(arb(f_i)).

On appelle N_2 l'ensemble des noeuds fonctionnels d'étiquette un symbole de forme fonctionnelle.

Quel que soit j, à f_{ij} est associée une sous-arborescence fonctionnelle $arb(f_{ij})$.

- Si f_{ij} est un objet, $arb(f_{ij})$ est réduite à un noeud fonctionnel n_{ij} d'étiquette f_{ij} ;

i.e. : si $f_{ij} \in O$, $arb(f_{ij}) = n_{ij}$ d'étiquette f_{ij} ; $n_{ij} = rac(arb(f_{ij}))$.

On note N_3 l'ensemble des noeuds fonctionnels d'étiquette un objet.

- Si f_{ij} est une fonction FP, f_{ij} peut alors être décomposée sous la forme d'une composition de fonctions et l'arborescence fonctionnelle représentant f_{ij} peut alors être obtenue de la même façon qu'est obtenue l'arborescence fonctionnelle représentant la fonction f initiale.

• Soit $n_{ij} = rac(arb(f_{ij}))$. Etant donnés les noeuds fonctionnels $\{n_i, n_{i1}, n_{i2}, ..., n_{im}\}$ tels qu'ils ont été définis précédemment, à tout couple (n_i, n_{i1}) est associé dans l'arborescence un arc étiqueté par le symbole "$\hookrightarrow$" reliant les noeuds n_i et n_{i1} et à tout couple $(n_{ij}, n_{i(j+1)})$ est associé dans l'arborescence un arc étiqueté par le symbole "$\rightarrow$ " reliant les noeuds n_{ij} et $n_{i(j+1)}$.

i.e. : $\forall$ i, $(n_i, n_{i1}) \mapsto arc(n_i \hookrightarrow n_{i1})$

$\forall$ i, j, $(n_{ij}, n_{i(j+1)}) \mapsto arc(n_{ij} \rightarrow n_{i(j+1)})$

On appelle N_2 l'ensemble des arcs d'étiquette "$\hookrightarrow$" et A_3 l'ensemble des arcs d'étiquette "$\rightarrow$ ".

Une arborescence fonctionnelle notée Arb est donc définie de la façon suivante :

Arb = (N, A) où

N = {N1 $\cup$ N2 $\cup$ N3},

A= {A1 $\cup$ A2 $\cup$ A3}.

Un programme FP est alors représenté par une forêt d'arborescences fonctionnelles :

$F = \{Arb_1, ..., Arb_m\}$

où $\forall$ i, $1 \leq i \leq m$, Arb_i est une arborescence fonctionnelle.

Arb_1 est l'arborescence fonctionnelle principale et $\forall\, i \geq 2$, Arb_i est une arborescence fonctionnelle secondaire.

$\forall\, n \in N$, n représente soit :

- une fonction primitive. Nous dirons alors que n est un ***noeud fonction primitive*** ou ***type(n) = fct-prim.***
- un symbole de forme fonctionnelle. Nous dirons alors que n est un ***noeud forme fonctionnelle*** ou ***type(n) = f-fonct.***
- un nom de définition. Nous dirons alors que n est un ***noeud définition*** ou ***type(n) = def.***
- un objet. Nous dirons alors que n est un ***noeud objet*** ou ***type(n) = objet.***

Etant donné un noeud fonctionnel r, nous appelons ***r-arborescence***, l'arborescence de racine r (notée ***Arb(r)***).

Si $\exists$ *arc (n1* $\downarrow$ *n2)*, le noeud fonctionnel n_2 est alors appelé **successeur** de n_1, noté ***succ(n_1)***, et l'arborescence de racine n_2 est appelée *sous-arborescence principale* de n_1, notée ***SAP(n_1)***. Si un noeud fonctionnel n n'a pas de successeur, nous noterons alors $succ(n) = \emptyset$.

De même, si $\exists$ *arc (n1* $\hookrightarrow$ *n2)*, l'arborescence de racine n_2 est alors appelée *première sous arborescence* de n_1, notée ***$SA_1(n_1)$***. Nous noterons donc ***$n_2 = rac(SA_1(n_1))$***.

Si $\exists$ *arc (n1* $\rightarrow$ *n2)* et n_1 est racine de la première sous-arborescence d'un noeud fonctionnel n, (i.e. : $SA_1(n)$) alors nous dirons que n_2 est racine de la **deuxième sous-arborescence** de n (notée ***$SA_2(n)$***). Ainsi, d'une façon générale,

si $\exists$ arc $(n_1 \rightarrow n_2)$ et $n_1 = rac(SA_i(n))$ alors $n_2 = rac(SA_{i+1}(n))$.

Remarques et rappels :

- Si il existe un arc $(n_1 \hookrightarrow n_2)$, cela signifie que n_1 est un noeud forme fonctionnelle et la sous-arborescence de racine n_2 représente alors le premier paramètre de cette forme fonctionnelle.

- Si il existe un arc $(n_1 \rightarrow n_2)$ tel que n_1 soit la racine de $SA_i(n)$ alors n est un noeud forme fonctionnelle, $SA_i(n)$ représente le $i^{ème}$ paramètre de la forme fonctionnelle et $SA_{i+1}(n)$ ayant pour racine n_2 représente le $(i+1)^{ème}$ paramètre de la forme fonctionnelle.

Soit *Arb* une arborescence fonctionnelle et *r* un noeud fonctionnel appartenant à cette arborescence. Nous appelons **chemin séquentiel** de racine *r*, noté ***CS(r)***, un chemin dans *Arb*, menant de *r* à une feuille de *Arb* et tel que

$$\forall\, n \in CS(r),\ n \neq r \Rightarrow \exists\, n' \in CS(r) \ /\ n = succ(n')$$

ce qui peut également s'écrire

$$\forall\, n \in CS(r),\ n = succ^*(r)$$

i.e.: tout arc du chemin a l'étiquette $\downarrow$.

Nous dirons qu'un chemin séquentiel de racine *r* dans une arborescence *Arb* est un **chemin séquentiel maximal** si

$$\nexists\, n \in Arb \ /\ r = succ(n).$$

A toute r-arborescence *Arb(r)*, on peut associer un chemin séquentiel de racine *r* que l'on appellera **chemin séquentiel gauche** de *Arb(r)*, noté ***CSG(Arb(r))***.

Propriété :

Etant donnée Arb(r) une r-arborescence, CSG(Arb(r)) est unique.

En effet,

$\forall n \in Arb(r),\ succ(n) \neq \emptyset \Rightarrow \exists!\ n' \in Arb(r) / n' = succ(n)$

Exemple :

La figure VI.6 représente une arborescence fonctionnelle quelconque dans laquelle les r_i, r_{ij} *et* r_{ijk} sont des noeuds fonctionnels.

- Cette arborescence a pour racine r_1, nous la désignons donc par r_1-arborescence ou $Arb(r_1)$.

- r_3 et r_{33} sont des noeuds forme fonctionnelle car on a $arc\ (r_3 \hookrightarrow r_{11})$ et $arc\ (r_{33} \hookrightarrow r_{111})$.

- $SAP(r_1)$ désigne la sous-arborescence principale de r_1, c'est-à-dire la r_2-arborescence.

- $SA_1(r_3)$ désigne la première sous-arborescence de r_3, c'est-à-dire la r_{11}-arborescence.

- $SA_2(r_3)$ désigne la deuxième sous-arborescence de r_3, c'est-à-dire la r_{21}-arborescence.

- La r_1-arborescence contient 6 chemins séquentiels maximaux qui sont $CS(r_1)$, $CS(r_{11})$, $CS(r_{21})$, $CS(r_{31})$, $CS(r_{111})$, $CS(r_{121})$ et qui sont encadrés dans la figure.

- $CS(r_1) = CSG(Arb(r_1))$ c'est-à-dire : le chemin séquentiel de racine r_1 est le chemin séquentiel gauche de la r_1-arborescence. $CS(r_1)$ est encadré en pointillés sur la figure.

- $\forall i \in [1, 3]$, $CS(r_{i1}) = CSG(SA_i(r_3))$ c'est-à-dire $CS(r_{i1})$ est le chemin séquentiel gauche de la sous-arborescence paramètre de racine r_{i1}.

- $CS(r_3)$ est un chemin séquentiel non maximal.

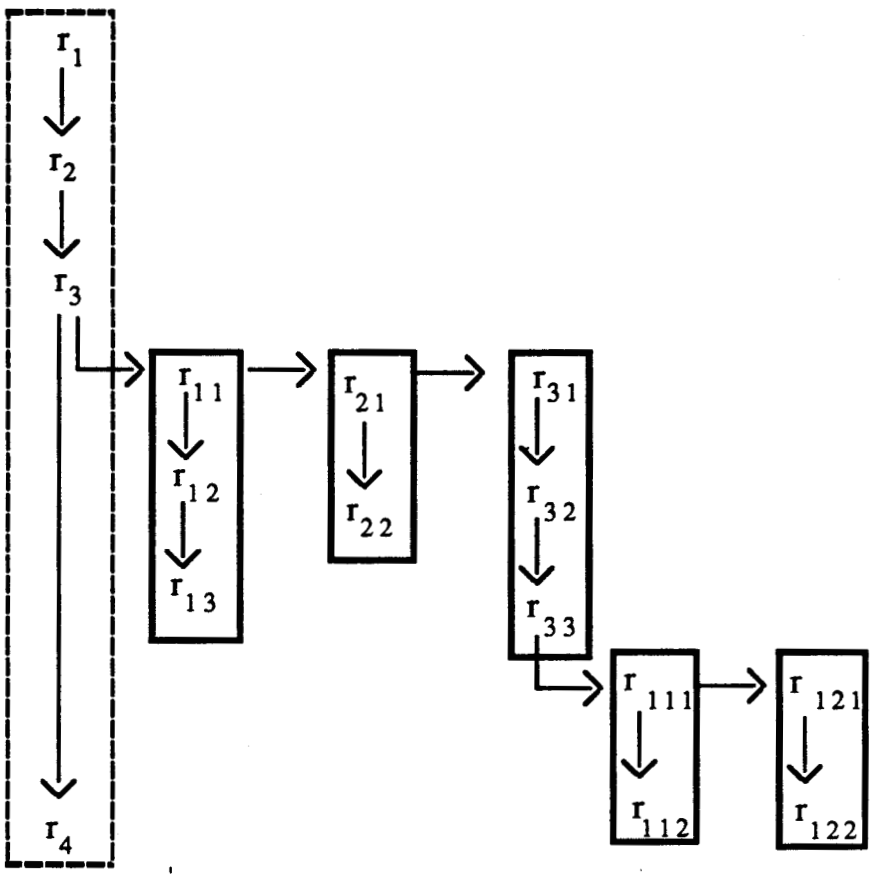


Figure VI.6 : Découpage d'une arborescence fonctionnelle en chemins séquentiels.

VI.3 : exploration des arborescences fonctionnelles

L'exploration d'une forêt d'arborescences fonctionnelles joue le rôle de moteur d'exécution du programme FP qu'elle représente. Elle a pour but d'acheminer les fonctions primitives vers leur séquence argument, générant ainsi des opérations de réduction. C'est donc l'organe de commande et de contrôle de l'exécution.

Nous avons vu précédemment que les différentes fonctions composant un programme FP ne sont pas indépendantes. Il existe certaines relations entre elles, représentées dans une arborescence fonctionnelle par différents types d'arcs entre les noeuds fonctionnels racines des arborescences fonctionnelles représentant ces fonctions. Ainsi, un arc étiqueté "↓" représente une composition de fonctions et impose donc un ordre d'exécution des fonctions liées par cet arc. Le moteur d'exécution se charge donc d'explorer les arborescences fonctionnelles en respectant l'ordre d'exécution des différentes fonctions. Ainsi, les fonctions primitives seront acheminées vers leur séquence argument en respectant l'ordre dans lequel elles doivent être exécutées. La façon dont une séquence argument respecte l'ordre d'exécution des fonctions primitives qui lui sont acheminées sera l'objet de la section VII.3.

Dans la présente section, nous ne nous intéresserons qu'à l'exploration des arborescences fonctionnelles. Dans un premier temps, nous décrirons l'exploration d'une arborescence fonctionnelle en termes de changement d'état des noeuds fonctionnels, définissant ainsi un graphe de transition d'état. Dans un deuxième temps, nous montrerons comment ce graphe de transition d'état peut être optimisé.

VI.3.1 : L'exploration en termes de changement d'état

Dans cette section, nous déterminons les règles d'exploration des arborescences fonctionnelles que nous exprimons en termes de changement d'état des noeuds fonctionnels. Nous ne nous préoccuperons pas des traitements à effectuer sur la séquence argument. Notre seul but dans cette section est de faire en sorte que l'exploration d'une arborescence fonctionnelle respecte l'ordre d'exécution des différentes fonctions.

Soit $F = \{Arb_1, ..., Arb_n\}$ une forêt d'arborescences fonctionnelles représentant un programme FP.

$\forall\ i \in [1, n]$, $Arb_i = \{N_i, A_i\}$ où N_i est l'ensemble des noeuds fonctionnels appartenant à Arb_i et A_i est l'ensemble des arcs étiquetés.

$\forall\ r \in N_i$ tel qu'il existe un chemin séquentiel de racine r, noté $CS(r)$, dans l'arborescence Arb_i, $CS(r)$ représente une composition de fonctions et peut donc s'écrire sous la forme

$f_1 \circ f_2 \circ ... \circ f_m$ où $\forall\ j \in [1, m]$, f_j est une fonction FP.

A chacune des fonctions f_j correspond un noeud fonctionnel dans $CS(r)$: à f_m correspond le noeud fonctionnel r, à f_{m-1} correspond le noeud $succ(r)$, etc...

L'ordre d'exécution est déterminé par la composition de fonctions qui impose donc que l'exploration d'un chemin séquentiel débute par la racine du chemin. Nous dirons que l'exploration d'un chemin séquentiel débute par l'activation du noeud fonctionnel racine du chemin séquentiel. La règle d'exploration d'un chemin séquentiel est donc la suivante :

Règle 1 :

$\forall$ n, r $\in N_i$ tels que n $\in$ CS(r), si $\exists$ n' = succ (n) alors n' $\in$ CS(r) et n' ne peut être activé qu'après l'activation de n.

En respectant cette règle lors de l'exploration d'une arborescence fonctionnelle, non seulement on respecte l'ordre d'activation des noeuds d'un chemin mais on s'assure également que si le dernier noeud du chemin séquentiel a été activé, tous les autres noeuds du même chemin l'ont été.

Lors de l'exploration d'une forêt d'arborescences fonctionnelles, les différents états que peut prendre un noeud fonctionnel sont :

- inactif,
- actif,
- en attente,
- en-cours-d'exploration,
- exploré.

On se définit une fonction *e* permettant d'associer à tout noeud fonctionnel un état parmi l'ensemble des états précédemment défini :

$$e : N_i \rightarrow \{\text{inactif, actif, en-attente, en-cours d'exploration, exploré}\}$$
$$n \mapsto e(n).$$

Initialement, tous les noeuds fonctionnels sont dans l'état *inactif*. En fin d'exploration de la forêt, le noeud racine de l'arborescence principale est dans l'état *exploré* et tous les autres noeuds ont retrouvé l'état *inactif*.

D'une façon générale, lors de l'exploration d'une arborescence fonctionnelle, si un noeud fonctionnel *n* est dans l'état *exploré*, cela signifie que la n-arborescence a été complètement explorée et que tout noeud fonctionnel différent de *n* et appartenant à la n-arborescence est dans l'état *inactif*.

Si un noeud fonctionnel *n* est dans l'état en cours d'exploration, cela signifie qu'il a été activé mais que la n-arborescence n'est pas encore complètement explorée. La signification des autres états sera donnée par la suite.

Un changement d'état sera représenté dans les schémas suivants comme le montre la figure VI.7 et aura la signification suivante : si un noeud fonctionnel se trouve dans l'état e_1 et que la condition c est vérifiée, alors, exécuter l'action a et mettre le noeud fonctionnel dans l'état e_2.

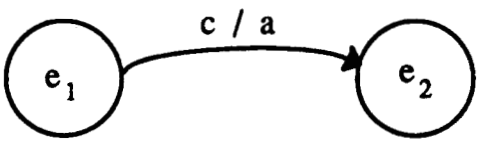


Figure VI.7 : transition d'état d'un noeud fonctionnel

Il se peut qu'une transition d'état ne soit pas accompagnée d'une action à exécuter. Dans ce cas, dans les schémas, nous utiliserons la notation "*C / Ø*".

Activation d'un noeud fonctionnel

Nous considérons qu'un noeud fonctionnel est activé au moyen d'un signal d'activation. Un noeud fonctionnel ne peut être activé que s'il se trouve dans l'état *inactif*. Si un noeud *inactif* est activé, il passe alors dans l'état *actif* sans qu'aucune autre action ne soit déclenchée. Cette transition d'état est représentée par la figure VI.8.

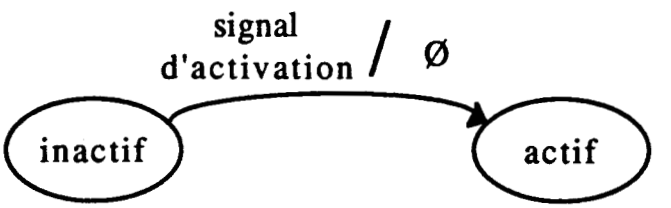


Figure VI.8 : activation d'un noeud fonctionnel

Cas d'un noeud dans l'état actif :

Lorsqu'un noeud fonctionnel est dans l'état *actif*, le passage à un autre état est conditionné par la nature du noeud fonctionnel, c'est-à-dire selon qu'il s'agit d'un noeud fonction-primitive, d'un noeud forme-fonctionnelle ou d'un noeud définition et selon que le noeud est ou non, le dernier noeud d'un chemin séquentiel.

Soient n, r $\in N_i$ tels que n $\in$ CS(r).

1er cas : type (n) = fct-prim (ou *n* est un noeud fonction-primitive).

Dans ce cas, si le noeud fonctionnel *n* est le dernier noeud du chemin séquentiel *CS(r)*, (i.e. : *succ(n)* = $\emptyset$), la n-arborescence (réduite dans ce cas au noeud *n*) est explorée, ce qui signifie que le noeud fonctionnel *n* peut passer à l'état exploré.

Si *succ(n)* existe (i.e. : *succ(n)* $\neq \emptyset$), l'action à exécuter consiste alors à activer *succ(n)* et le noeud fonctionnel *n* passe alors dans l'état en cours d'exploration. Il conservera cet état tant que la n-arborescence ne sera pas complètement explorée.

La transition de l'état actif à un autre état pour un noeud fonction-primitive peut donc être représentée par la figure VI.9.

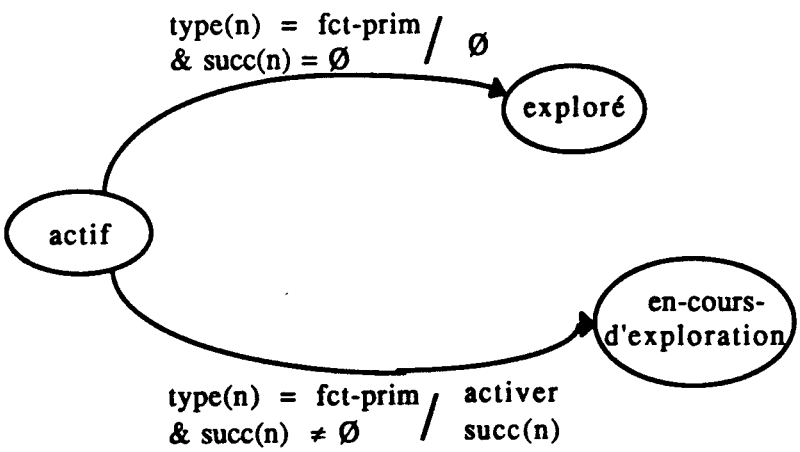


Figure VI.9 : transition de l'état actif à un autre état pour un noeud fonction-primitive

2ème cas : type(n) = f-fonct (*n* est un noeud forme-fonctionnelle).

Soit *p* le nombre de paramètres de la forme fonctionnelle. L'ensemble $\{SA_1(n), ..., SA_p(n)\}$ correspond à l'ensemble des sous-arborescences représentant les paramètres de la forme fonctionnelle et la forme fonctionnelle est donnée par l'ensemble $\{n, SA_1(n), ..., SA_p(n)\}$ Cela signifie donc que l'activation du noeud fonctionnel *n* doit conduire à l'exploration des sous-arborescences $SA_i(n)$, $\forall\, i \in [1, p]$. De plus, pour respecter l'ordre d'exécution des fonctions, le noeud fonctionnel $succ(n)$ ne pourra être activé que lorsque les $SA_i(n)$ auront toutes été explorées. Ceci nous donne donc une deuxième règle d'exploration :

Règle 2 :

$\forall$ n $\in$ N tel que type(n) = f-fonct, si $\exists$ n' = succ(n) alors $succ(n)$ ne peut être activé qu'après exploration des sous-arborescences paramètres de la forme fonctionnelle.

Ainsi, si *n* est dans l'état actif, l'action à exécuter consiste à activer les noeuds racine des $SA_i(n)$ et le noeud fonctionnel *n* passe alors dans l'état en-attente, qu'il conservera tant que les $SA_i(n)$ n'auront pas été explorées.

La transition de l'état actif à un autre état pour un noeud forme-fonctionnelle peut donc être représentée par la figure VI.10.

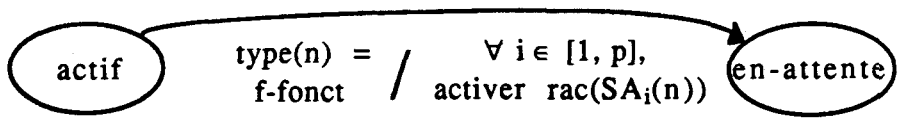


Figure VI.10 : transition de l'état actif à un autre état pour un noeud forme-fonctionnelle à p paramètres.

3ème cas : type(n) = def (ou *n* est un noeud définition).

Si *n* est un noeud définition, il existe dans la forêt d'arborescences fonctionnelles, une arborescence fonctionnelle représentant cette définition (notée Arb_{def}). Ainsi, de même que pour un noeud forme-

fonctionnelle, l'activation de ce noeud *n* doit conduire à l'exploration de cette arborescence fonctionnelle secondaire et on peut énoncer la règle suivante :

Règle 3 :

Si n ∈ N et type(n) = def alors, si ∃ n' = succ(n),
n' ne pourra être activé qu'après exploration complète de l'arborescence fonctionnelle secondaire Arb_{def}.

Ainsi le noeud *n* sera dans l'état en-attente tant que l'arborescence secondaire n'aura pas été complètement explorée. La transition de l'état actif à un autre état pour un noeud définition est donc représentée par la figure VI.11.

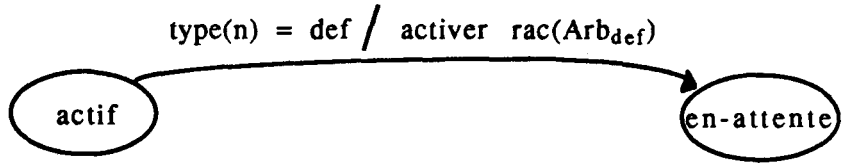


Figure VI.11 : transition de l'état actif à un autre état pour un noeud définition

Reste maintenant à compléter le graphe de transition en envisageant successivement les cas où un noeud fonctionnel est à l'état en-attente, en-cours-d'exploration et exploré.

Cas d'un noeud dans l'état en attente :

Si un noeud fonctionnel est à l'état en-attente, il s'agit alors d'un noeud forme-fonctionnelle ou d'un noeud définition.

- S'il s'agit d'un noeud forme-fonctionnelle à *p* paramètres et que $\forall\ i \in [1,\ p]$, la racine de $SA_i(n)$ est dans l'état exploré (noté $e(rac(SA_i(n))) = exploré$), alors toutes les sous-arborescences fonctionnelles représentant les paramètres de la forme fonctionnelle ont été explorées.

Si *succ(n)* existe, *succ(n)* peut alors être activé et le noeud *n* passe alors dans l'état en-cours-d'exploration.

Si *succ(n)* = $\emptyset$, alors l'arborescence de racine *n* est explorée et le noeud fonctionnel *n* peut alors passer à l'état exploré.

Dans les deux cas, les racines des sous-arborescences paramètres sont alors désactivées (au moyen d'un signal de désactivation) et retrouvent l'état inactif.

- De même, si *n* est un noeud définition et que la racine de Arb_{def} est dans l'état exploré (noté *e(rac(Arb$_{def}$))* = *exploré*) alors *n* passe dans l'état en-cours-d'exploration si *succ(n)* $\neq \emptyset$ auquel cas *succ(n)* peut être activé, et *n* passe dans l'état exploré si *succ(n)* = $\emptyset$.

Le graphe de transition de l'état en-attente à un autre état est donc représenté par la figure VI.12.

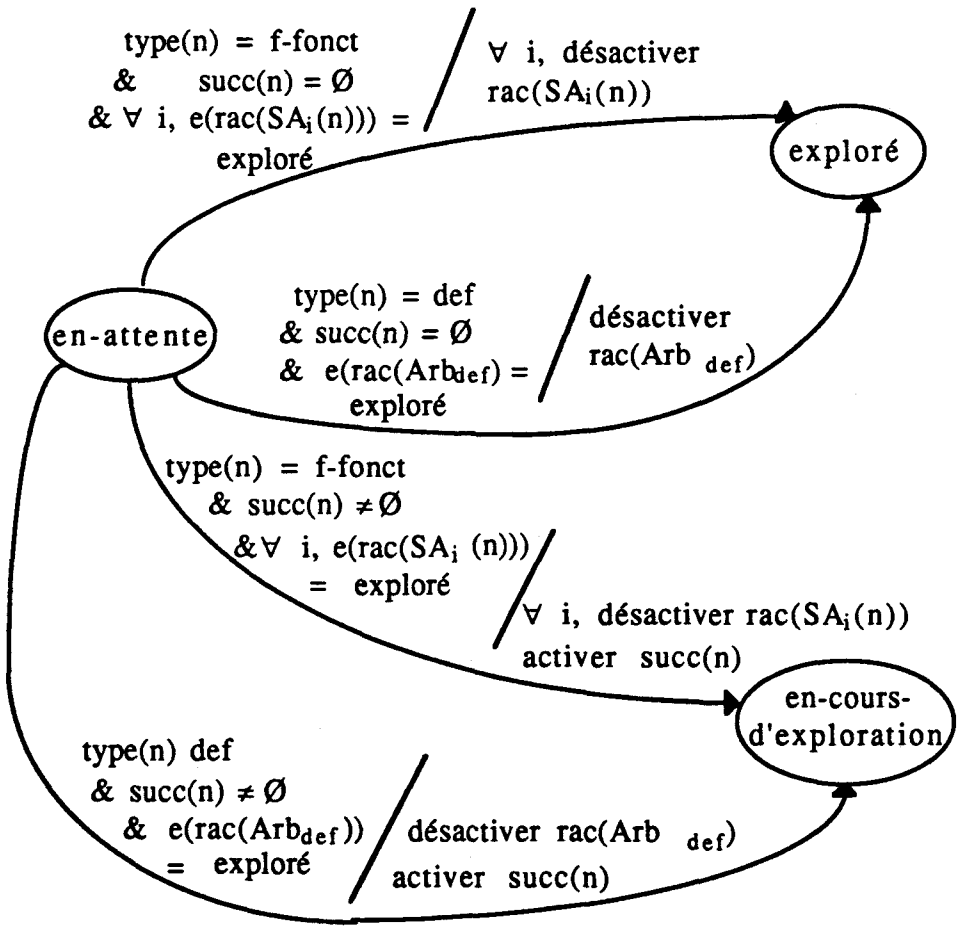


Figure VI.12 : transition de l'état en-attente à un autre état pour un noeud définition ou un noeud forme-fonctionnelle.

Cas d'un noeud dans l'état en-cours d'exploration

Si un noeud fonctionnel *n* est dans l'état en-cours-d'exploration cela signifie que *succ(n)* existe (ceci peut-être vérifié dans les graphes précédents). Dans ce cas, si *succ(n)* est dans l'état exploré, cela signifie que la n-arborescence est explorée et le noeud fonctionnel *n* peut alors passer dans l'état exploré. Le noeud *succ(n)* est alors désactivé : il retrouvera ainsi l'état inactif.

Le graphe de transition de l'état en-cours-d'exploration à un autre état est donc représenté par la figure VI.13

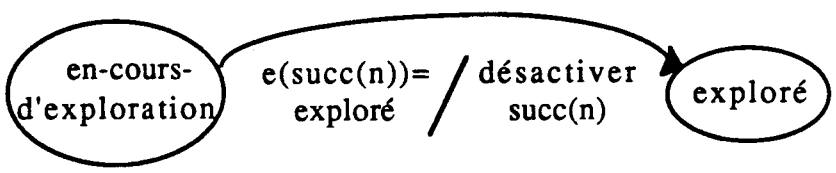


Figure VI.13 : transition de l'état en-cours-d'exploration

Cas d'un noeud dans l'état exploré :

Enfin, si un noeud fonctionnel est dans l'état exploré, nous avons vu dans le graphe précédent qu'il retrouvait l'état inactif, grâce à un signal de désactivation, ce qui est représenté par la figure VI.14.

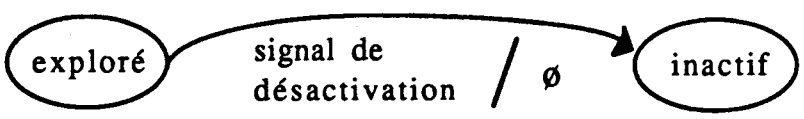


Figure VI.14 : désactivation d'un noeud dans l'état exploré

En regroupant les différentes transitions décrites précédemment, on obtient le graphe complet de transitions d'états représenté par la figure VI.15.

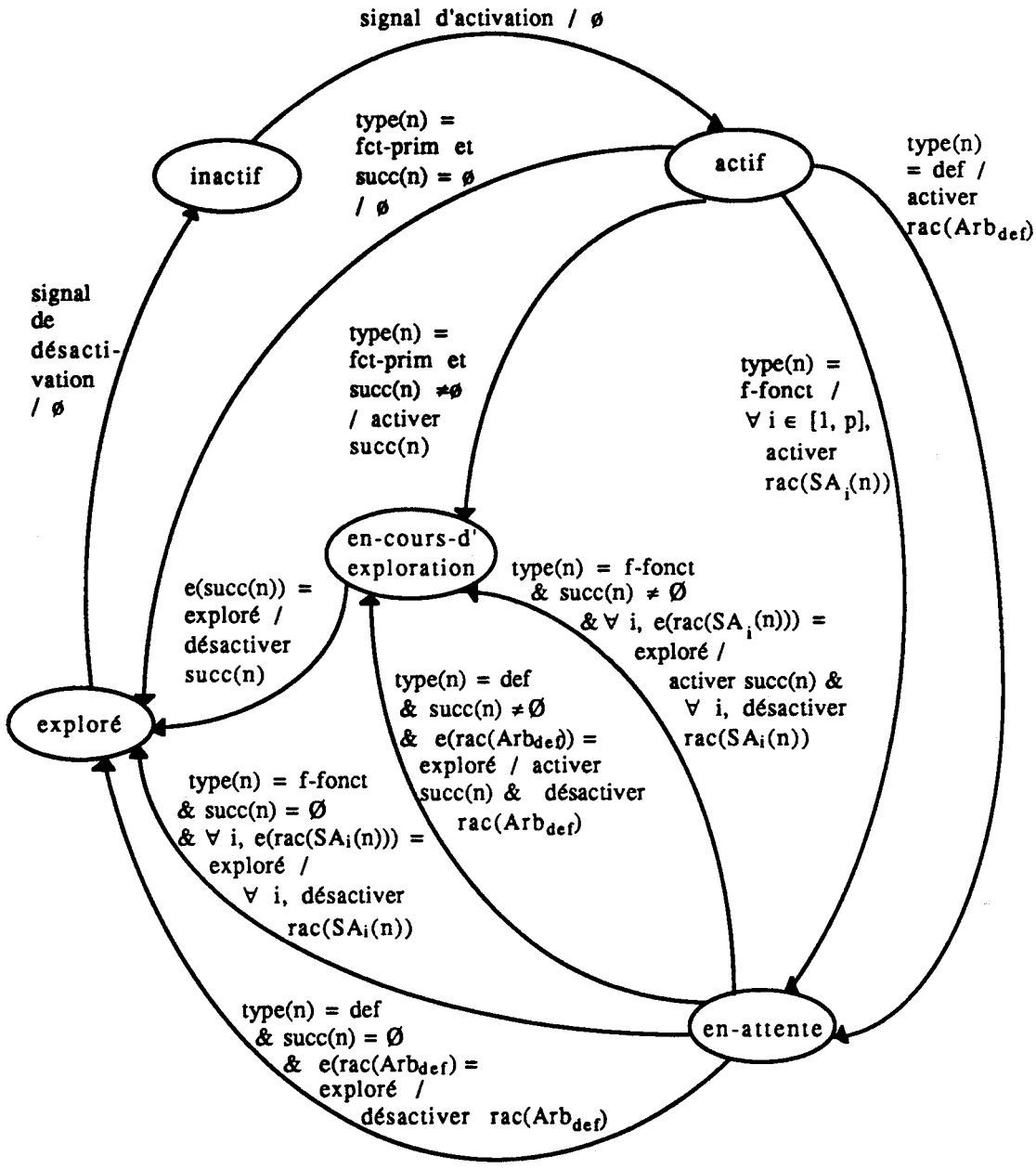


Figure VI.15 : Graphe de transition d'état d'un noeud fonctionnel

Remarque :

Dans le graphe, on peut remarquer que lorsqu'un noeud forme-fonctionnelle passe de l'état actif à l'état en attente, les noeuds racine de toutes les sous-arborescences paramètres de la forme fonctionnelle sont activés, ce qui signifie que ces sous-arborescences sont explorées en parallèle.

VI.3.2 : optimisation du graphe de transition d'état

Dans cette section, nous apportons une optimisation du graphe de transition d'état précédemment défini. Pour cela, nous montrerons tout d'abord la propriété suivante :

Etant donnée *Arb(r)* une r-arborescence,
e(r) = exploré $\Rightarrow \forall$ n $\in$ Arb(r), (n $\neq$ r), e(n) = inactif.

Par extension, nous démontrerons la propriété suivante :

Etant donnée une forêt d'arborescences fonctionnelles,
F = {Arb_0, Arb_1, ..., Arb_n } où
- Arb_0 est l'arborescence fonctionnelle principale,
- $\forall\ i > 0$, Arb_i est une arborescence secondaire,

e(rac(Arb_i)) = exploré $\Leftrightarrow$
$\forall$ i, $\forall$ n $\in$ Arb_i (n $\neq$ rac(Arb_i)), e(n) = inactif.

Nous définirons alors l'état *E* d'une arborescence fonctionnelle de la façon suivante :

E(Arb(r)) = exploré $\Leftrightarrow$ e(r) = exploré.

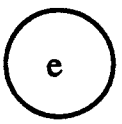
Nous définirons ensuite un nouvel état (appelé *exploré-bis*) d'une arborescence de la façon suivante :

E(Arb(r)) = exploré-bis $\Leftrightarrow$
Soit n $\in$ CS(r) tel que succ(n) = $\emptyset$, e(n) = exploré.

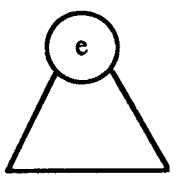
La comparaison entre l'état exploré et l'état exploré-bis d'une arborescence fonctionnelle nous permettra de définir un nouveau graphe de transition d'état "équivalent" au précédent (nous définirons alors ce que nous entendons par équivalent) et nous montrerons dans quelle mesure l'exploration d'une arborescence fonctionnelle selon ce deuxième graphe est optimisée.

VI.3.2.1 : notations

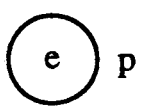
Dans le but de simplifier les démonstrations, nous décrirons le graphe de transition d'état par un système de réécriture dans lequel nous utiliserons les symboles suivants :



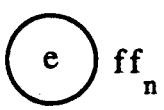
: noeud fonctionnel quelconque dans l'état *e*.



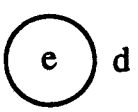
: arborescence fonctionnelle non vide dont la racine est dans l'état *e*



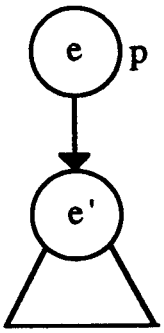
: noeud fonction primitive dans l'état *e*, n'ayant pas de successeur



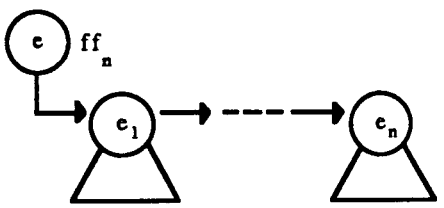
: noeud forme fonctionnelle à *n* paramètres, dans l'état *e*, n'ayant pas de successeur



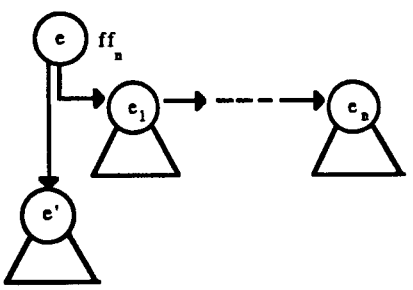
: noeud définition dans l'état *e*, n'ayant pas de successeur



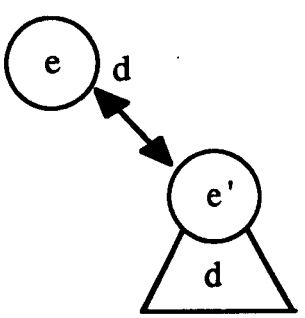
: arborescence fonctionnelle dont la racine est un noeud fonction-primitive dans l'état *e*, le successeur de la racine étant dans l'état *e'*



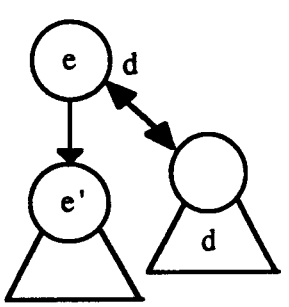
: arborescence fonctionnelle représentant une forme fonctionnelle à *n* paramètres, le noeud forme fonctionnelle n'ayant pas de successeur



: le noeud forme fonctionnelle a un successeur dans l'état *e'*



: arborescence fonctionnelle dont la racine est un noeud définition dans l'état *e*. L'arborescence reliée à ce noeud par le symbole "↔" correspond à l'arborescence fonctionnelle secondaire représentant la définition. La racine de cette arborescence secondaire est dans l'état *e'*.



: noeud définition ayant un successeur dans l'état *e'*

De plus, nous noterons:

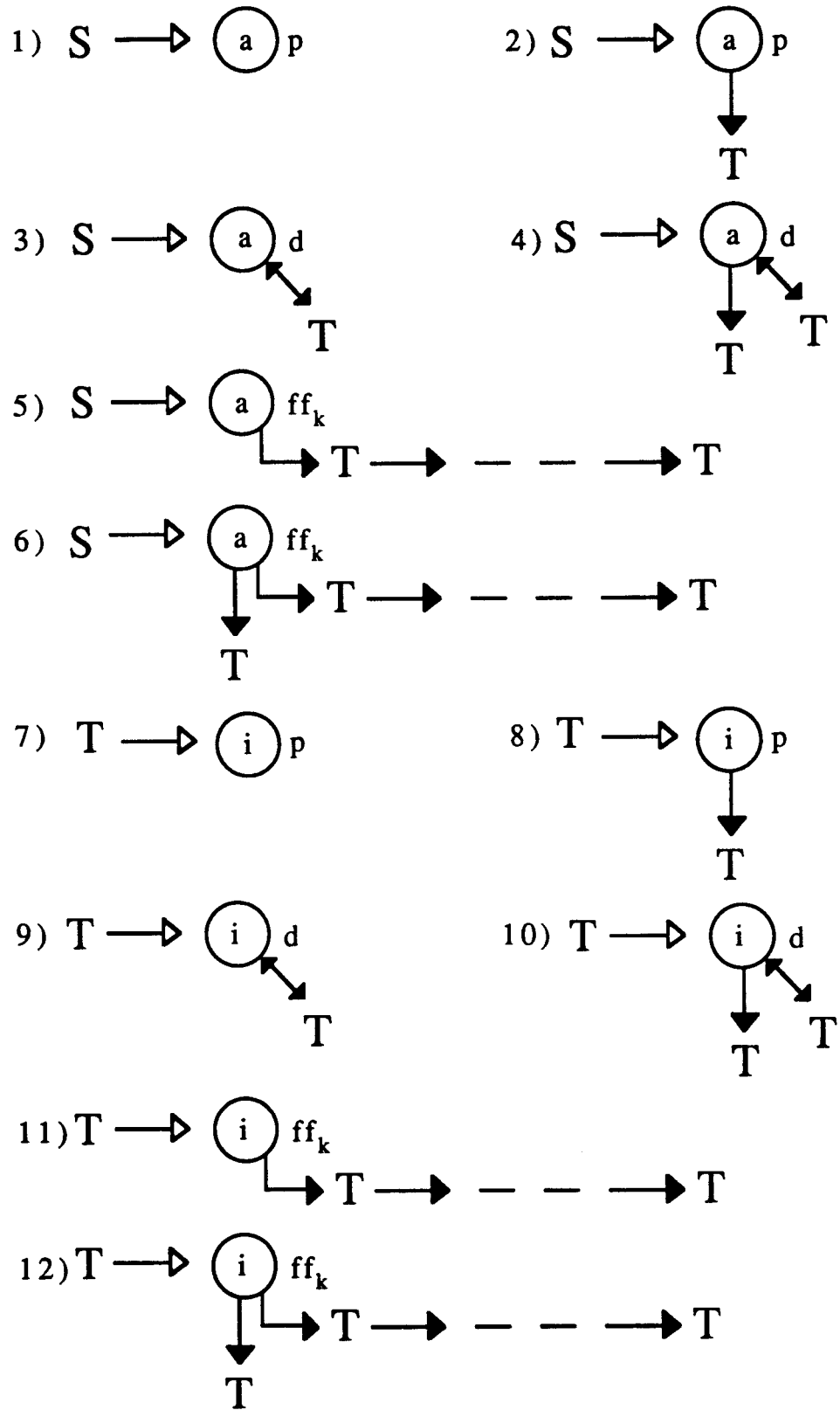
- a : l'état actif,
- i : l'état inactif,
- e : l'état exploré,
- ea : l'état en-attente,
- ec : l'état en-cours-d'exploration.

VI.3.2.2 : formalisme utilisé

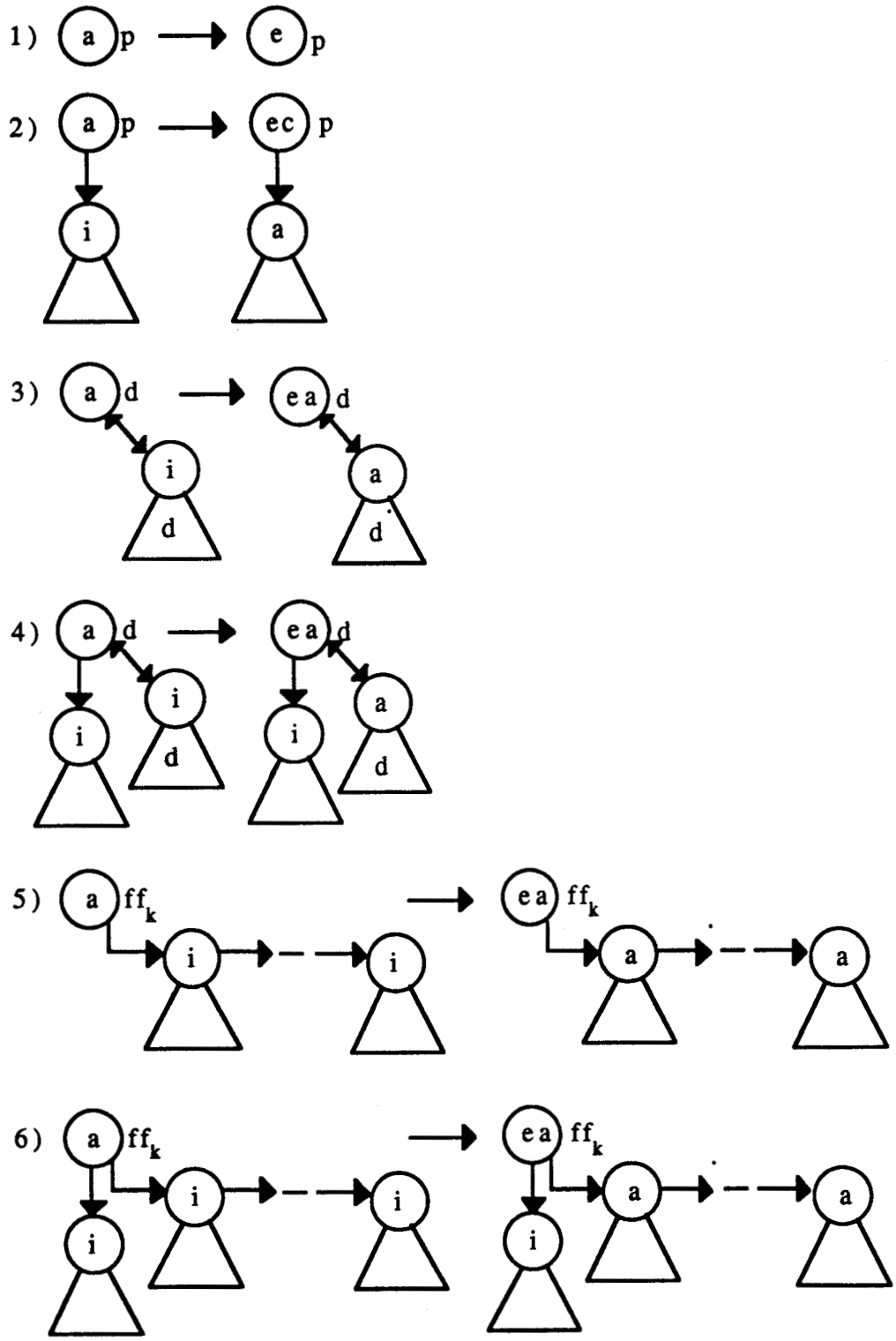
A l'aide des notations précédemment définies, nous pouvons nous définir une grammaire d'arbres régulière, engendrant une forêt reconnaissable F telle que :

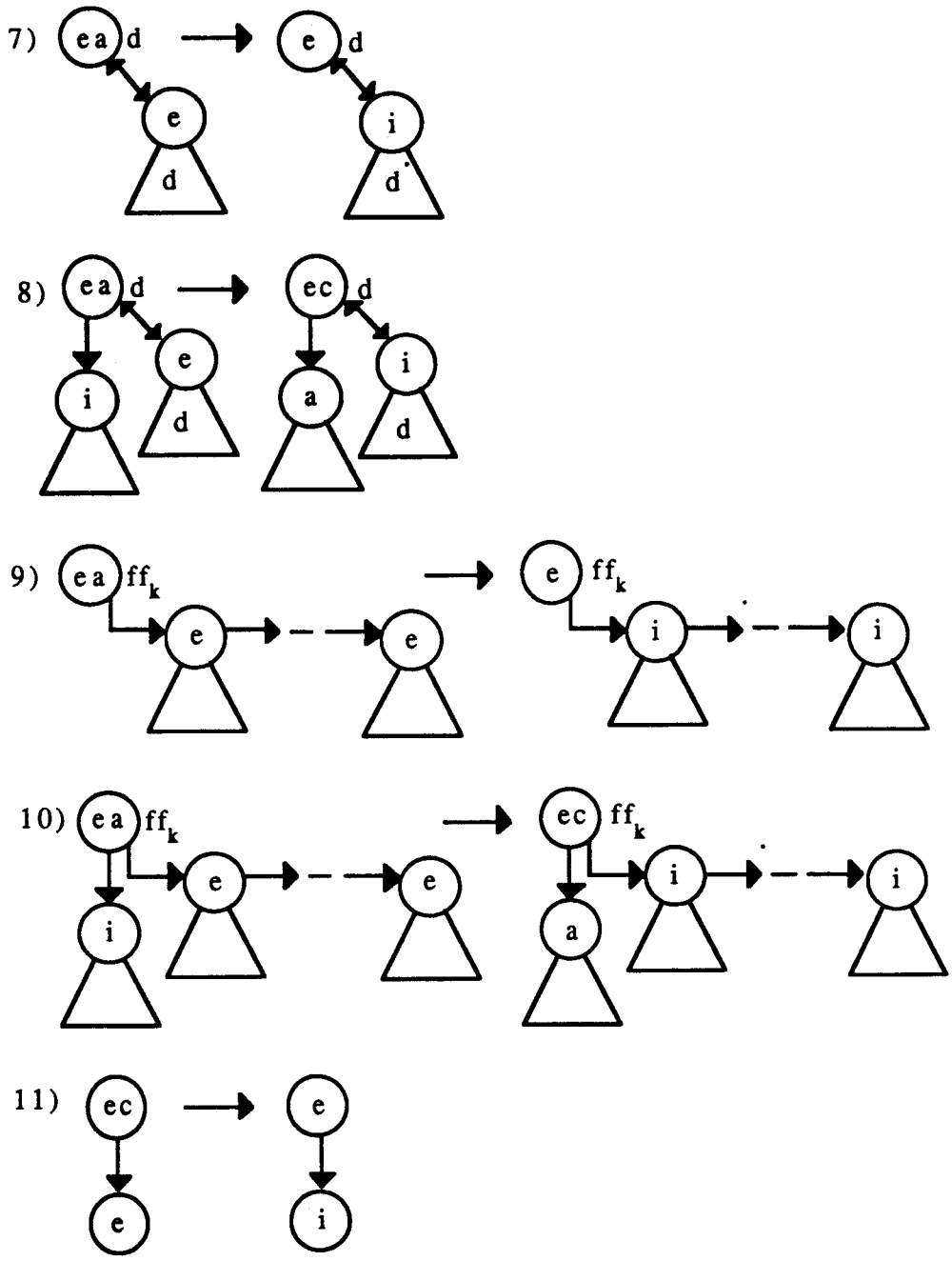
$\forall f \in F, f$ est une forêt d'arborescences fonctionnelles dont la racine de l'arborescence fonctionnelle principale est dans l'état actif et tous les autres noeuds sont dans l'état inactif.

Soit *S* l'axiome de la grammaire. Les règles sont les suivantes :



$\forall f \in F$, l'exploration de l'arborescence fonctionnelle f est décrite par un système de réécriture, obtenu d'après le graphe de transition d'état. Le système de réécriture est donné par les règles suivantes :





A l'aide de ce formalisme, on peut démontrer un certain nombre de propriétés

VI.3.2.3 : propriétés et démonstrations

Propriété 1 :

$\forall$ n $\in$ Arb(r) tel que succ(n) $\neq \emptyset$,
e(n) = exploré $\Rightarrow$ e(succ(n)) = inactif.

Démonstration :

La démonstration est immédiate si l'on s'intéresse aux règles ayant pu être appliquées pour conduire un noeud fonctionnel n'ayant pas de successeur dans l'état exploré. Les seules règles permettant de conduire un noeud quelconque dans l'état exploré sont les règles 1, 7, 9, 11. Les règles 1, 7, 9 concernent des noeuds fonctionnels n'ayant pas de successeur. Dans notre cas, seule la règle 11 a donc pu être appliquée. Le successeur du noeud dans l'état exploré est donc obligatoirement dans l'état inactif.

Propriété 2 :

$\forall$ n $\in$ Arb(r) tel que succ(n) $\neq \emptyset$,
e(n) = inactif $\Rightarrow$ e(succ(n)) = inactif.

Démonstration:

- Initialement, seule la racine de l'arborescence fonctionnelle principale est dans l'état actif et tous les autres noeuds sont dans l'état inactif. La propriété est donc vérifiée initialement.

- Il s'agit maintenant de montrer que lorsqu'un noeud passe d'un état quelconque à l'état inactif, son successeur est alors dans l'état inactif et tant qu'un noeud est dans l'état inactif, son successeur est dans l'état inactif. Les seules règles qui permettent de mener un noeud d'un état différent de l'état inactif à l'état inactif sont les règles 7, 8, 9, 10 et 11 et toutes ces règles imposent qu'un

noeud passant à l'état inactif était précédemment dans l'état exploré. De plus, la transition de l'état exploré à l'état inactif ne modifie pas l'état du successeur du noeud. Or, d'après la propriété 1, le successeur d'un noeud dans l'état exploré est dans l'état inactif, donc lorsque le noeud passe de l'état exploré à l'état inactif, son successeur est toujours dans l'état inactif. Comme aucune règle n'est applicable lorsqu'un noeud et son successeur sont dans l'état inactif, la propriété reste vraie tout au long de l'exploration.

Propriété 3 :

$$\forall\, n \in \text{Arb}(r),\ e(n) = \text{exploré} \Rightarrow \forall\, n' \in \text{CS}(n),\ e(n') = \text{inactif}.$$

Démonstration :

La démonstration est immédiate d'après les propriétés précédentes :

$$e(n) = \text{exploré} \Rightarrow e(\text{succ}(n)) = \text{inactif (propriété 1)} \Rightarrow e(\text{succ}^*(n)) = \text{inactif (propriété3)}.$$

Propriété 4 :

$$\forall\, n \in \text{Arb}(r),\ e(n) = \text{exploré} \Rightarrow \forall\, n' \in \text{Arb}(n) \,/\, n' \neq n,\ e(n') = \text{inactif}.$$

Démonstration :

D'après la propriété 3,

$$e(n) = \text{exploré} \Rightarrow \forall\, n' \in \text{CS}(n),\ e(n') = \text{inactif},$$

i.e. : la propriété est vérifiée pour le chemin séquentiel gauche de *Arb(n)*. Il suffit donc de démontrer que la propriété est vraie pour tous les autres chemins séquentiels de *Arb(n)*.

Grâce à la propriété 2, il suffit de démontrer que la racine de chaque chemin séquentiel maximal de l'arborescence est dans l'état inactif. Les deux points suivants sont donc à démontrer :

1- type(n) = f-fonct et e(n) = exploré $\Rightarrow$
$\forall$ i, e(rac(SA_i(n))) = inactif.

2- $\forall$ n' $\in$ Arb(n) / type(n') = f-fonct et e(n') = inactif $\Rightarrow$
$\forall$ i, e(rac(SA_i(n))) = inactif.

Démonstration de 1 :

1er cas : succ(n) = $\emptyset$
La seule règle permettant à un noeud forme fonctionnelle n'ayant pas de successeur de passer dans l'état exploré est la règle 9 qui impose alors que le noeud racine de chaque sous-arborescence paramètre passe dans l'état inactif (C.Q.F.D.).

2ème cas : succ(n) $\neq \emptyset$
Un noeud ayant un successeur ne peut être dans l'état exploré qu'après application de la règle 11. Avant application de cette règle, il était donc dans l'état en-cours-d'exploration. Un noeud forme-fonctionnelle ayant un successeur ne peut être dans l'état en-cours-d'exploration qu'après application de la règle 10 qui impose également que le noeud racine de chaque sous-arborescence paramètre passe dans l'état inactif (C.Q.F.D.).

Démonstration de 2 :

Un noeud ne peut se trouver dans l'état inactif qu'après avoir été dans l'état exploré (règles 7, 8, 9, 10, 11). Ainsi, un noeud forme-fonctionnelle dans l'état inactif était précédemment dans l'état exploré et d'après la démonstration du premier point, le noeud racine de chaque sous-arborescence paramètre a alors été mis dans l'état inactif (C.Q.F.D.).

Propriété 5 :

Soit $F = \{Arb_0 , ...,Arb_n\}$ une forêt d'arborescences fonctionnelles où Arb_0 est l'arborescence fonctionnelle principale et quel que soit i, Arb_i est une arborescence fonctionnelle secondaire.
Soit r le noeud racine de Arb_0.

e(r) = exploré $\Rightarrow \forall$ i, $\forall$ n $\in$ Arb_i, n $\neq$ r, e(n) = inactif.

Démonstration :

Comme pour la propriété 4, il suffit de montrer les deux points suivants :

1- type(r) = def et e(n) = exploré $\Rightarrow$ e(rac(Arb_{def})) = inactif
où Arb_{def} est l'arborescence fonctionnelle secondaire associée au noeud r.

2- $\forall$ i, $\forall$ n $\in$ Arb_i, type(n) = def et e(n) = inactif
$\Rightarrow$ e(rac(Arb_{def})) = inactif
où Arb_{def} est l'arborescence fonctionnelle secondaire associée au noeud n.

La démonstration est similaire à la démonstration de la propriété 4 :

Démonstration de 1 :

1er cas : succ(r) = $\emptyset$
Le noeud définition étant dans l'état exploré, seule la règle 7 a pu être appliquée. Le noeud racine de l'arborescence fonctionnelle secondaire associée au noeud n est donc bien dans l'état inactif.

2^{eme} cas : succ(n) $\neq \emptyset$

De même que pour la propriété 4, la règle 11 a été appliquée, le noeud définition était donc précédemment dans l'état en-cours-d'exploration suite à l'application de la règle 8. D'après cette règle, le noeud racine de l'arborescence secondaire associée au noeud définition est alors dans l'état inactif.

Démonstration de 2 :

Un noeud définition dans l'état inactif était précédemment dans l'état exploré (règles 7, 8, 9, 10, 11). D'après la démonstration du premier point, le noeud racine de l'arborescence fonctionnelle secondaire est alors dans l'état inactif.

C.Q.F.D.

VI.3.2.4 : ajout d'actions associées aux activations

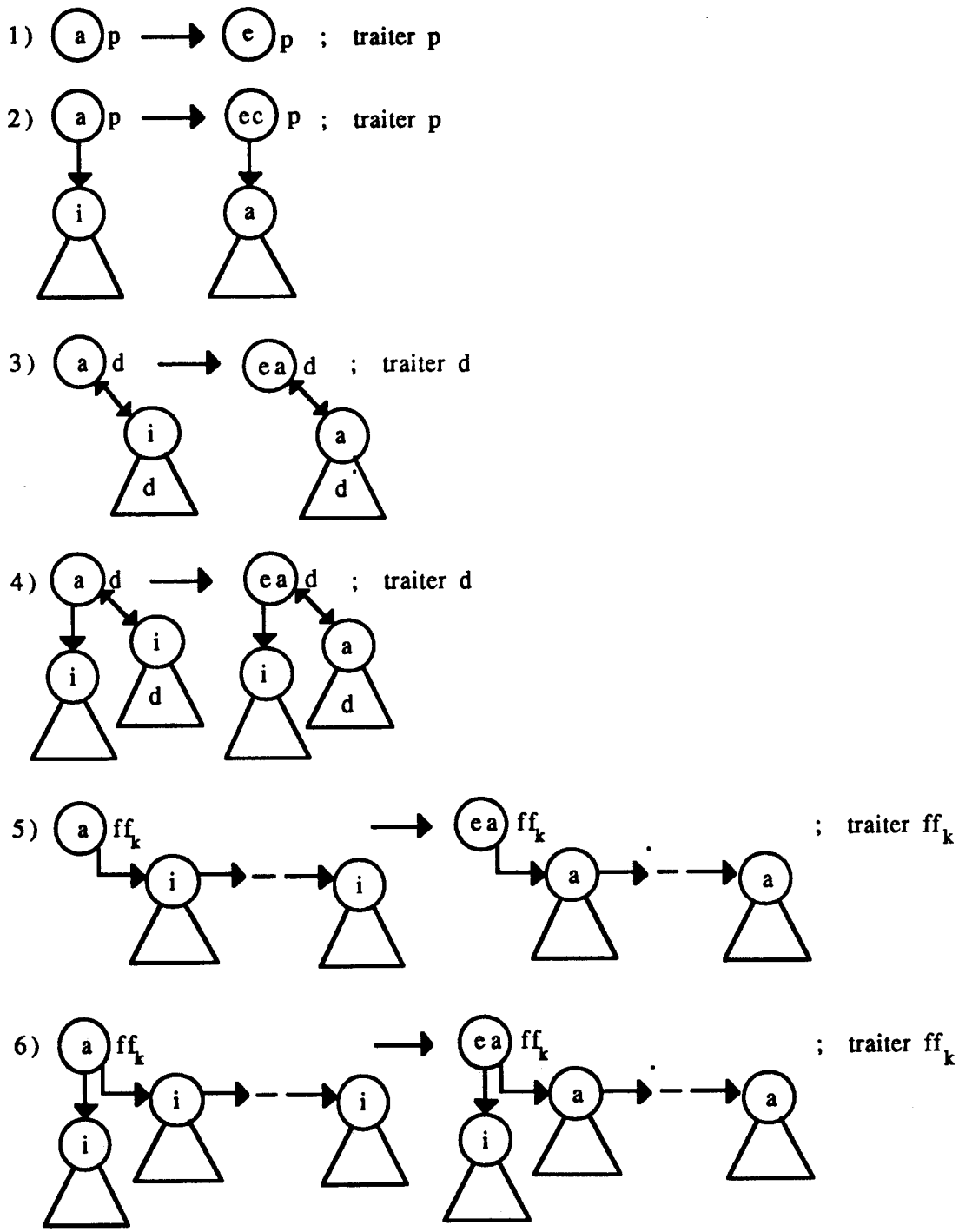
Le graphe de transition d'état décrit le principe de l'exploration d'une forêt d'arborescences fonctionnelles sans prendre en compte les traitements générés lors de l'exploration. Ces traitements sont effectués lors de l'activation d'un noeud fonctionnel : un noeud activé est amené à réaliser certaines actions. Dans le système de réécriture décrivant le graphe de transition d'état, nous pouvons donc ajouter à chaque règle, l'action associée. Ainsi, nous écrirons

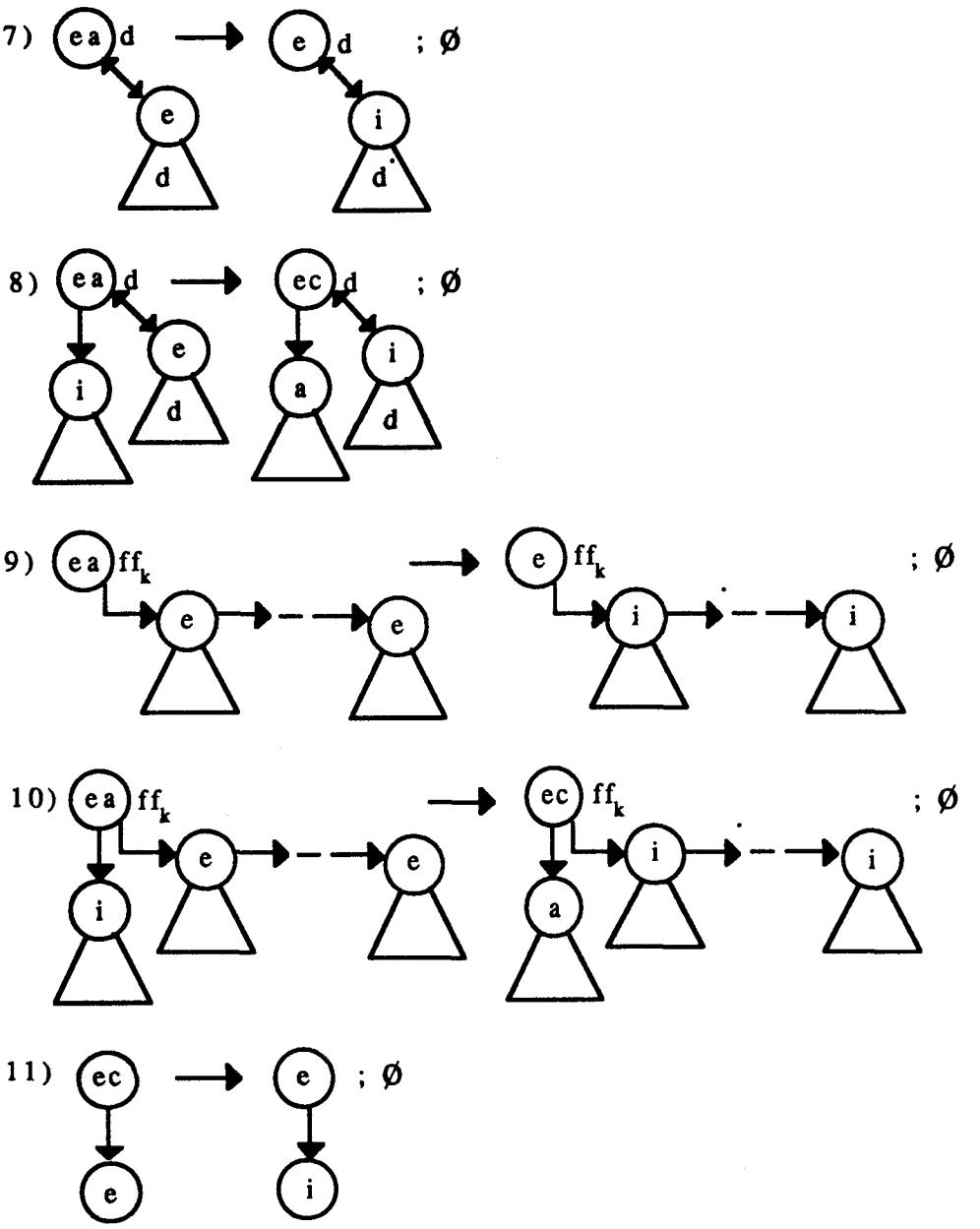
(g $\rightarrow$ d ; a)

une règle ayant la signification suivante : le noeud de la racine d'une arborescence unifiable avec "*g*" exécute l'action "*a*" et l'arborescence initiale se réécrit alors en l'arborescence "*d*". Si ($a = \emptyset$), aucune action n'est exécutée ; la règle de réécriture correspond alors à une opération de contrôle de l'exploration. Les actions étant engendrées lors de l'activation d'un noeud fonctionnel, aucune action ne sera associée aux règles 7 à 11.

Les règles 1 et 2 concernent l'activation d'un noeud fonction-primitive, nous leur associons donc l'action "traiter p". De même, aux règles 3 et 4, nous associons l'action "traiter d" et aux règles 5 et 6, nous associons l'action "traiter ff_k".

Le système de réécriture avec les actions associées aux règles est donc le suivant :





VI.3.2.5 : dernière étape !

Définitions :

Nous définissons l'état *E* d'une arborescence fonctionnelle de la façon suivante :

E(Arb(r)) = exploré ⇔ e(r) = exploré.

L'état E_F d'une forêt d'arborescences fonctionnelles $F = \{Arb_0, ..., Arb_n\}$, Arb_0 étant l'arborescence fonctionnelle principale est alors défini par

$E_F(F)$ = exploré $\Leftrightarrow$ $E(Arb_0)$ = exploré.

Conséquences :

- D'après la propriété 5,
  E(Arb(r)) = exploré $\Rightarrow \forall$ n $\in$ Arb(r) / n $\neq$ r, e(n) = inactif.

- D'après la propriété 6, si *r* est le noeud racine de l'arborescence fonctionnelle principale alors
  $E_F(F)$ = exploré $\Rightarrow$ e(r) = exploré et
  $\forall$ i, $\forall$ n $\in$ Arb(r) / n $\neq$ r, e(n) = inactif.

Définitions :

Soit $Arb(r)$ une r-arborescence et $n \in CSG(r)$, le noeud fonctionnel tel que $succ(n) = \emptyset$. Nous définissons un nouvel état "*exploré-bis*" d'une arborescence de la façon suivante :

E(Arb(r)) = exploré-bis $\Leftrightarrow$ e(n) = exploré.

Soit $F = \{Arb_0, ..., Arb_n\}$ une forêt d'arborescences fonctionnelles, Arb_0 étant l'arborescence fonctionnelle principale. L'état *exploré-bis* d'une forêt d'arborescences fonctionnelles est alors défini par

$E_F(F)$ = exploré-bis $\Leftrightarrow$ $E(Arb_0)$ = exploré-bis.

Notre but dans cette section est de montrer que les états exploré et exploré-bis d'une forêt d'arborescences fonctionnelles sont équivalents dans le sens où si une forêt est dans l'état exploré-bis, les mêmes actions

auront été générées que si elle était dans l'état exploré, ces actions étant de plus, générées dans le même ordre.

On peut remarquer qu'au cours de l'exploration d'une arborescence ou sous-arborescence fonctionnelle *Arb(r)*, les noeuds de l'arborescence sont à un moment donné dans les états suivants :

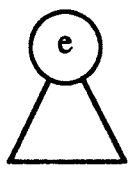
- $\forall$ n $\in$ CSG(Arb(r)),
  si succ(n) $\neq \emptyset$ alors e(n) = en-cours-d'exploration,
  si succ(n) = $\emptyset$ alors e(n) = exploré

- $\forall$ n' $\in$ Arb(r) / n' $\notin$ CSG(Arb(r)), e(n') = inactif.

La seule règle pouvant alors être appliquée est la règle 11 qui permet en quelque sorte de "remonter" l'état exploré le long du chemin séquentiel gauche de *Arb(r)* jusqu'à la racine de l'arborescence et ce, sans qu'aucune action ne soit générée et sans que l'état des noeuds n'appartenant pas à *CSG(Arb(r))* ne soit modifié. Lorsque l'état exploré est remonté à la racine, l'arborescence est alors, (selon notre première définition), explorée et l'on considère que l'exploration est terminée.

L'idée est donc de supprimer les étapes de "remontée" de l'état exploré dans l'arborescence et de considérer que l'exploration d'une arborescence fonctionnelle est terminée lorsque le dernier noeud du chemin séquentiel gauche de l'arborescence est dans l'état exploré.

Une arborescence dans l'état exploré est représentée dans le système de réécriture par le symbole



Une arborescence dans l'état exploré-bis sera représentée par le symbole

Nous pouvons alors définir un nouveau système de réécriture qui, au lieu d'amener une forêt d'arborescences fonctionnelles dans l'état exploré l'amène à l'état exploré-bis. Pour cela, nous voulons supprimer la "remontée" de l'état exploré dans les chemins séquentiels : dans le nouveau système de réécriture, la règle 11 sera donc supprimée.

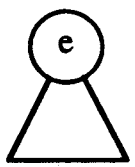
Remarque :

La suppression de la règle 11 n'engendre aucune suppression d'action.

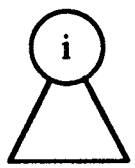
Dans le premier système, la fin d'exploration d'une arborescence quelconque est détectée lorsque le noeud racine de l'arborescence est dans l'état exploré. Une sous-arborescence ou une arborescence secondaire dans l'état exploré est ensuite désactivée (i.e. : le noeud racine retrouve l'état inactif). La fin d'exploration d'une forêt est donc détectée lorsque le noeud racine de l'arborescence fonctionnelle principale est dans l'état exploré.

Dans le nouveau système, la fin d'exploration d'une arborescence quelconque doit être détectée lorsque le dernier noeud du chemin séquentiel gauche de l'arborescence est dans l'état exploré. Cette arborescence est alors dans l'état exploré-bis et s'il s'agit d'une sous-arborescence ou d'une arborescence secondaire, elle sera ensuite désactivée de la même façon : le noeud dans l'état exploré retrouvera l'état inactif. La fin d'exploration d'une forêt sera donc détectée lorsque le dernier noeud du chemin séquentiel gauche de l'arborescence principale sera dans l'état exploré.

Pour cela, les règles permettant de réécrire des arborescences représentées par le symbole



en des arborescences représentées par le symbole



seront transformées dans le nouveau système de réécriture en des règles permettant de réécrire des arborescences représentées par le symbole

en des arborescences représentées par le symbole

Le système de réécriture obtenu reste cohérent dans la mesure où la règle 11 a été supprimée. De plus, il engendre, lors de l'exploration d'une forêt d'arborescences fonctionnelles, les mêmes actions, dans le même ordre que le système de réécriture précédent.

Nous appelons opération de désactivation d'une forêt d'arborescences fonctionnelles, l'opération qui consiste, après détection de la fin d'exploration à remettre le noeud qui est dans l'état exploré, dans l'état inactif.

Avec le premier système de réécriture, une forêt désactivée possède la propriété suivante :

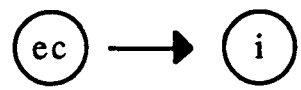
$\forall\ n \in F,\ e(n) = \text{inactif}.$

Tel que le second système est actuellement défini, une forêt désactivée dans ce second système ne possède pas cette propriété mais la propriété suivante :

$\forall n \in F, \forall n' \in F / n' \in CS(n)$,
si succ(n') $\neq \emptyset$, e(n') = en-cours-d'exploration,
si succ(n') = $\emptyset$, e(n') = inactif.

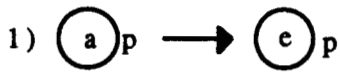
Pour que les deux systèmes puissent être considérés comme équivalents, il est nécessaire qu'ils conduisent tous les deux dans le même état final, ce qui n'est pas le cas. Ceci vient du fait que dans le premier système, l'état en-cours-d'exploration est un état intermédiaire, permettant de remonter l'état exploré le long d'un chemin séquentiel (la règle 11 est la seule règle faisant apparaître l'état en-cours-d'exploration en partie gauche de règle).

Dans le nouveau système, il suffirait alors d'ajouter la règle suivante :

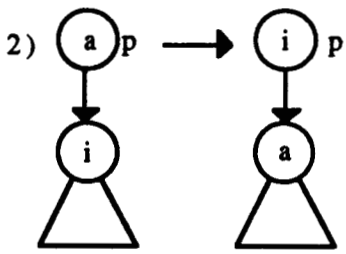


Cette règle étant la seule qui fasse apparaître l'état en-cours-d'exploration en partie gauche de règle, elle montre que l'état en-cours-d'exploration est devenu inutile : au lieu de mettre un noeud dans l'état en-cours-d'exploration, on peut le mettre directement dans l'état inactif.

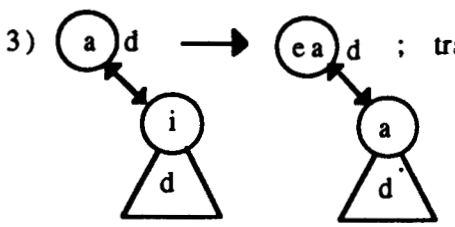
Le nouveau système de réécriture est donc le suivant :

1) 

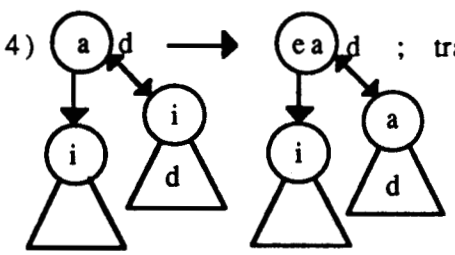
; traiter p

2) 

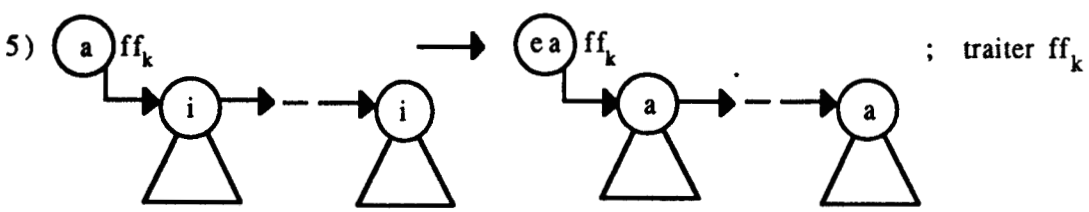
; traiter p

3) 

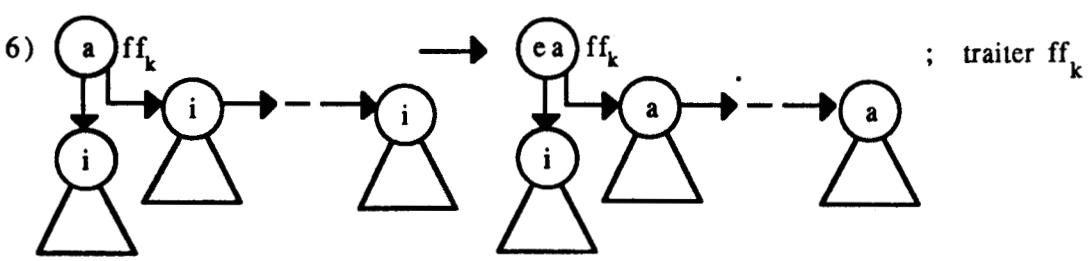
; traiter d

4) 

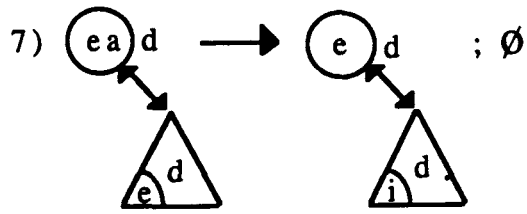
; traiter d

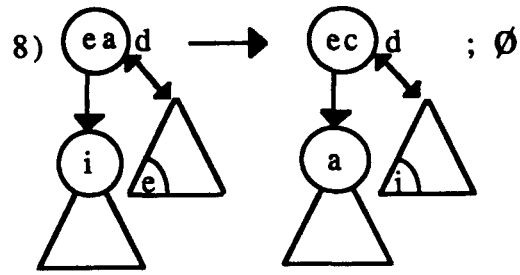
5) 

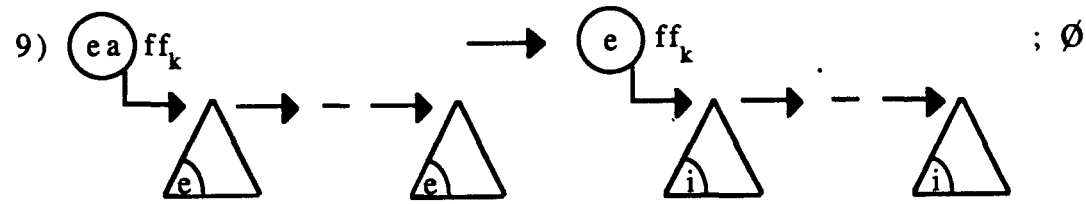
; traiter ff$_k$

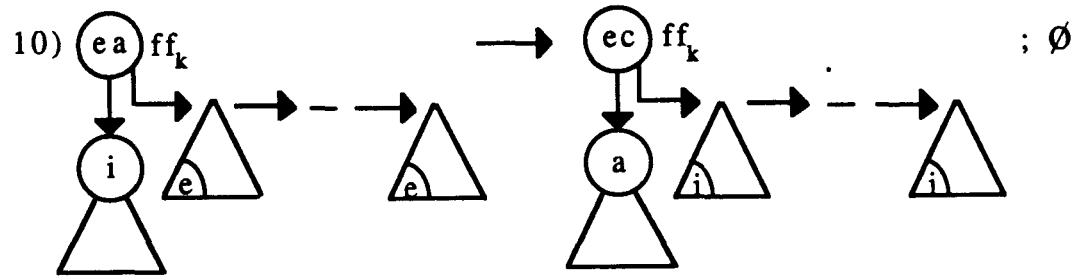
6) 

; traiter ff$_k$









Avec ce système :

- les mêmes actions sont générées qu'avec le premier système,
- les actions sont générées dans le même ordre que précédemment,
- en fin d'exploration, la désactivation d'une forêt conduit au même état final.

De plus, l'exploration décrite avec ce système est optimisée dans le sens où l'état final est obtenu en moins d'étapes qu'avec le premier système du fait de la suppression de la règle 11.

Ce nouveau système fournit le graphe de transition d'état de la figure VI.16. Ce graphe de transition d'état est donné pour un noeud fonctionnel *n*. Nous notons *extr(Arb(r))*, le dernier noeud du chemin séquentiel gauche de *Arb(r)*. Pour des raisons purement techniques, nous n'avons pas pu inscrire dans le schéma les descriptions des transitions. C'est pourquoi les transitions sont numérotées et la description correspondant à chaque "numéro" est donnée ci-dessous (les numéros donnés aux transitions correspondent au numéro des règles de réécriture du système optimisé):

(A) signal d'activation / Ø
(B) signal de désactivation / Ø

(1) type(n) = fct-prim et succ(n) = Ø / Ø

(2) type(n) = fct-prim et succ(n) ≠ Ø / activer succ(n)

(3-4) type(n) = def / activer rac(Arb_{def})

(5-6) type(n) = f-fonct / ∀ i ∈ [1, p], activer rac($SA_i(n)$)

(7) type(n) = def et succ(n) = Ø et e(extr(Arb_{def}) = exploré / désactiver extr(Arb_{def})

(8) type(n) = def et succ(n) ≠ Ø et e(extr(Arb_{def}) = inactif / désactiver extr(Arb_{def}), activer succ(n)

(9) type(n) = f-fonct et succ(n) = Ø et
∀ i ∈ [1, p], e(extr($SA_i(n)$)) = exploré /
∀ i ∈ [1, p], désactiver extr($SA_i(n)$)

(10) type(n) = f-fonct et succ(n) ≠ Ø et
∀ i ∈ [1, p], e(extr($SA_i(n)$)) = exploré /
∀ i ∈ [1, p], désactiver extr($SA_i(n)$), activer succ(n)

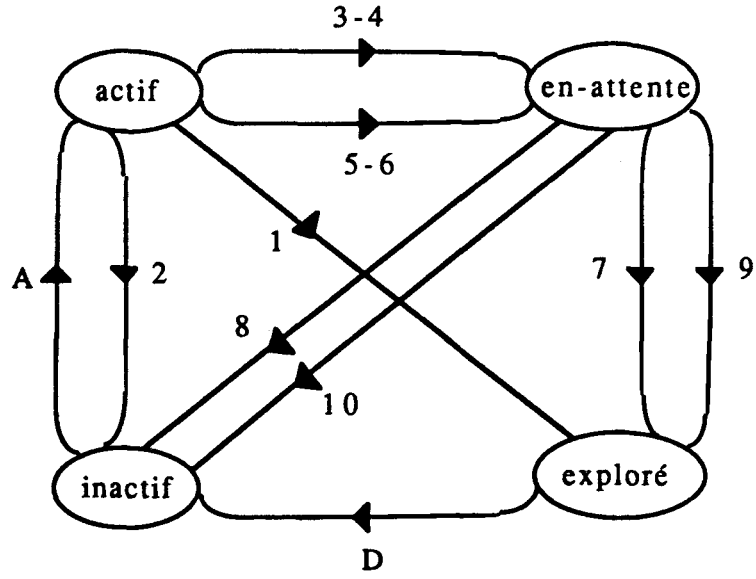


Figure VI.16 : nouveau graphe de transition d'état d'un noeud fonctionnel

VII : VERS UNE MISE EN OEUVRE : LES MESSAGES

VII : VERS UNE MISE EN OEUVRE : LES MESSAGES

Le système de réécriture optimisé, précédemment défini, décrit l'exploration d'une forêt d'arborescences fonctionnelles dans le modèle, indépendamment de toute machine et de surcroît, sans a priori sur l'implantation. Dans cette section, nous faisons un premier pas vers la réalisation : nous essaierons de rester éloigné de toute machine cible mais nous étudierons une mise en oeuvre des mécanismes précédemment décrits, basée sur une communication par messages entre les différents noeuds fonctionnels. En fait, tout le modèle sera basé sur une communication par messages : l'acheminement des fonctions primitives vers leur séquence argument sera également réalisé par envoi de messages ainsi que les mécanismes de réduction.

VII.1 : Exploration d'une forêt par messages

Dans cette section, nous ne définissons que le principe d'exploration des arborescences fonctionnelles, les messages générés à destination de la séquence argument ne seront donc pas pris en compte.

Dans un premier temps, nous montrerons que lors de l'exploration d'une forêt, une même arborescence fonctionnelle peut être explorée plusieurs fois ; dans un deuxième temps, nous définirons un algorithme d'exploration d'une forêt prenant en compte la possibilité d'exploration multiple d'une même arborescence par partage de code.

VII.1.1 : Exploration multiple d'une arborescence fonctionnelle

Dans le modèle, une même arborescence fonctionnelle peut-être explorée plusieurs fois, ce qui conduit à une activation multiple des mêmes noeuds fonctionnels. A titre d'exemple, la figure VII.1 représente une arborescence fonctionnelle dans laquelle les noeuds fonctionnels *Def*

sont des noeuds définition, *Def* étant une définition secondaire et *f* étant une fonction primitive quelconque.

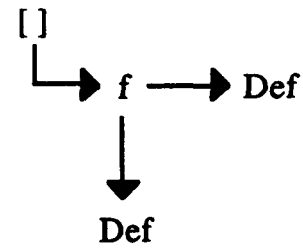


Figure VII.1 : Exemple d'exploration multiple d'une même arborescence secondaire.

L'exploration de cette arborescence fonctionnelle conduit à l'exploration des deux sous-arborescences paramètres en parallèle. Chacun des deux noeuds définition sera activé, ce qui conduira à explorer deux fois l'arborescence secondaire représentant la définition *Def*. Ainsi, les noeuds fonctionnels de l'arborescence secondaire seront activés deux fois.

L'algorithme d'exploration par message doit donc mettre en oeuvre un mécanisme permettant de prendre en compte les possibilités d'exploration multiple d'une même arborescence fonctionnelle. On peut imaginer diverses solutions permettant les explorations multiples, certaines étant basées sur une recopie des arborescences à chaque nouvelle exploration. Dans le modèle, nous ne retenons pas ces solutions dans la mesure où l'exploration d'une arborescence fonctionnelle ne modifie pas l'arborescence. En conséquence, en cas d'exploration multiple, une solution basée sur le "partage de code" est tout-à-fait envisageable et plus performante.

VII.1.2 : algorithme d'exploration par messages

Nous définissons dans cette section l'exploration d'une forêt d'arborescences fonctionnelles au moyen d'émissions et réceptions de messages.

Les mécanismes précédemment décrits en termes de changement d'état d'un noeud fonctionnel doivent donc maintenant être décrits en termes d'émission et réceptions de messages.

VII.1.2.1 : Cas d'un noeud dans l'état actif

L'activation d'un noeud fonctionnel est réalisée par l'émission d'un message que nous appellerons message d'activation, à destination de ce noeud fonctionnel. Ainsi, l'état actif correspond maintenant au fait que le noeud ait reçu un message d'activation.

L'exploration d'une arborescence débute obligatoirement par l'activation du noeud racine de l'arborescence. Afin de permettre l'exploration multiple d'une arborescence fonctionnelle, il suffit alors d'associer au noeud racine de chaque arborescence fonctionnelle, un numéro d'activation, incrémenté à chaque nouvelle activation du noeud (nous pouvons alors parler de numéro d'activation d'une arborescence fonctionnelle). Ce numéro d'activation est véhiculé dans les messages d'activation afin de permettre la mise en oeuvre des mécanismes décrits dans la suite de cette section. Nous supposons qu'initialement, le noeud racine de chaque arborescence fonctionnelle a le numéro d'activation 0.

VII.1.2.2 : Cas d'un noeud dans l'état en attente

Un noeud en-attente est un noeud forme-fonctionnelle ou un noeud définition.

- Un noeud forme-fonctionnelle est en-attente tant que le dernier noeud des chemins séquentiels gauches de chaque sous-arborescence paramètre activée n'est pas dans l'état exploré. Dans une implantation par messages, la fin d'exploration d'une sous-arborescence paramètre sera signalée au noeud forme-fonctionnelle par l'envoi d'un message que nous appellerons message de fin-d'exploration. Ainsi, si un noeud forme fonctionnelle a activé p sous-arborescences paramètres, il attend p messages de fin-d'exploration.

Ainsi, la condition qui s'écrivait dans le graphe de transition d'état

$\forall i \in [1, p]$, e(extr(CSG(SA_i(n)))) = exploré,

correspond dans l'implantation par messages au fait que le noeud *n* ait reçu *p* messages de fin d'exploration.

- de même, un noeud définition est en-attente tant que le dernier noeud du chemin séquentiel gauche de l'arborescence secondaire correspondante n'est pas dans l'état exploré. Dans l'implantation par messages, le noeud définition attendra donc un message de fin-d'exploration et la condition qui s'écrivait dans le graphe de transition d'état :

e(extr(CSG(Arb_{def}))) = exploré,

correspond dans l'implantation par messages au fait que le noeud définition ait reçu un message de fin-d'exploration.

VII.2.1.3 : Cas d'un noeud dans l'état exploré

Un noeud à l'état exploré est un noeud en fin de chemin séquentiel. La transition à l'état inactif consiste donc maintenant à émettre un message de fin-d'exploration.

Un noeud fonctionnel émetteur d'un message de fin-d'exploration doit connaître le destinataire du message. Pour résoudre ce problème, lorsqu'un noeud forme-fonctionnelle active les sous-arborescences paramètres, il transmet dans le message d'activation son identification. Cette identification est véhiculée dans les messages d'activation tout le long d'un chemin séquentiel. De cette façon, le dernier noeud du chemin séquentiel "connaît" le destinataire du message de fin d'exploration. De même, un noeud définition qui émet un message d'activation à destination du noeud racine de l'arborescence secondaire correspondante, transmet dans ce message son identification qui sera

véhiculée le long du chemin séquentiel gauche de l'arborescence secondaire. Le dernier noeud du chemin aura donc la possibilité d'émettre un message de fin d'exploration à destination du noeud définition.

Tout noeud fonctionnel en fin de chemin séquentiel est maintenant amené à émettre un message de fin-d'exploration. Nous avons dit précédemment qu'un noeud forme-fonctionnelle ou un noeud définition attend la réception de messages de fin-d'exploration. Cette "attente" a pour but de retarder l'émission d'un message d'activation à destination du successeur s'il existe et sinon, de retarder l'émission du message de fin-d'exploration que le noeud doit émettre.

Ceci peut poser des problèmes dans le cas d'exploration multiple, en particulier dans le cas où un noeud en attente de messages de fin-d'exploration reçoit un second message d'activation. Pour résoudre ces problèmes, dans le modèle, un noeud forme-fonctionnelle ou un noeud définition crée, s'il a un successeur, un message d'activation bloqué à destination de son successeur. Ce n'est donc plus le noeud fonctionnel qui est en attente de messages de fin d'exploration mais le message bloqué. Ceci permet en quelque sorte de libérer le noeud fonctionnel qui peut éventuellement traiter un autre message d'activation. De cette façon, un message d'activation ne véhicule plus les références d'un noeud en attente de messages de fin-d'exploration mais il véhicule directement les références d'un message créé bloqué. Afin de gérer les activations multiples, la référence d'un message bloqué contiendra donc le numéro d'activation du noeud fonctionnel émetteur de ce message (i.e. : le numéro d'activation que le noeud a reçu dans le message d'activation).

De même, un noeud forme-fonctionnelle ou un noeud définition en fin de chemin séquentiel, au lieu de se mettre en attente de messages de fin d'exploration, crée un message de fin-d'exploration bloqué. Ainsi, les messages de fin d'exploration des sous-arborescences paramètres ou de l'arborescence secondaire seront directement envoyés au message bloqué.

Un message créé bloqué, que ce soit un message d'activation ou un message de fin d'exploration, doit avoir une référence, cette référence servant de destinataire à de futurs messages de fin-d'exploration. Cette référence n'est utile que si le message est créé bloqué. Afin de pouvoir gérer les explorations multiples d'une même arborescence fonctionnelle, ces références devront contenir le numéro d'activation reçu dans le message d'activation.

VII.1.2.4 : Les messages d'activation et de fin-d'exploration

Etant donné ce qui a été défini prédédemment, un message d'activation contiendra les informations suivantes :

- REF-MSG : les références du message. Si le message est créé bloqué, ces références contiennent une informations sur l'identité du message et le numéro d'activation :

REF-MSG = (ID, NA).

Si le message peut être envoyé sans devoir attendre dans un premier temps des messages de fin-d'exploration, l'information *ID* est alors inutile, nous noterons $ID = \emptyset$; sinon, l'information *ID* doit être générée ce que nous noterons *générer(ID)*.

- DEST : le destinataire du message d'activation.

- NB-ATT : le nombre de messages de fin d'exploration que le message d'activation doit attendre avant d'être effectivement envoyé, i.e. : si $NB\text{-}ATT = 0$, le message est envoyé, il est bloqué sinon.

- DEST-MSG-FIN : les références d'un message en attente de message de fin-d'exploration en fin de chemin séquentiel.

Nous noterons

Activation(REF-MSG, DEST, NB-ATT, DEST-MSG-FIN)

un message d'activation de référence *REF-MSG*, à destination de *DEST*, en attente de *NB-ATT* messages de fin-d'exploration, le noeud fonctionnel récepteur de ce message (i.e. : *DEST*) devant émettre un message de fin-

d'exploration à *DEST-MSG-FIN*, s'il est en fin de chemin séquentiel (i.e. : il n'a pas de successeur).

De même, un message de fin-d'exploration contiendra les informations suivantes:

-REF-MSG = (ID, NA) : les références du message, *ID* étant l'identification du message et *NA* le numéro d'activation.

-DEST : le destinataire du message. L'information *DEST* est donc la référence d'un message précédemment créé bloqué.

- NB-ATT : le nombre de messages de fin-d'exploration que ce message doit attendre avant d'être effectivement envoyé.

Nous noterons

Fin-d'exploration(REF-MSG, DEST, NB-ATT),

un message de fin-d'exploration, de référence *REF-MSG*, en attente de *NB-ATT* messages de fin-d'exploration, émis à destination du message bloqué dont les références sont *DEST*.

VII.1.2.5 : l'algorithme d'exploration

L'exploration d'une forêt d'arborescences fonctionnelles débute par l'émission d'un message d'activation à destination du noeud racine de l'arborescence fonctionnelle principale.

Etant donnée $F = \{Arb_0, ..., Arb_n\}$ une forêt où Arb_0 est l'arborescence fonctionnelle principale, le destinataire du message d'activation est donc le noeud $rac(Arb_0)$. Le message est directement envoyé au noeud, sans attendre préalablement d'autres messages (i.e. : *NB-ATT* = *0*). Dans l'information *REF-MSG*, on a donc (*ID* = $\emptyset$). Le numéro d'activation sera donné par le noeud racine de l'arborescence. Initialement, on a donc (*NA* = *0*). En fin de chemin séquentiel gauche de l'arborescence principale, l'exploration de la forêt sera terminée. Nous noterons donc (*DEST-MSG-FIN* = $\emptyset$).

La gestion des numéros d'activation est assurée par les noeuds racine des arborescences fonctionnelles : ces noeuds racine possèdent un numéro, incrémenté à chaque réception d'un message d'activation et c'est ce numéro qui est véhiculé dans les messages d'activation lors de l'exploration d'une arborescence fonctionnelle. Mis à part le noeud racine de l'arborescence principale, un noeud racine ne peut être activé que par un noeud définition. De plus, le message d'activation émis à destination du noeud racine d'une arborescence secondaire n'est jamais créé bloqué. De cette façon, un noeud définition peut émettre à destination du noeud racine de l'arborescence correspondante, un message d'activation dont l'information *NA* sera notée "Ø", signalant au noeud récepteur, que le numéro d'activation des messages d'activation générés par ce noeud devra être égal au numéro d'activation du noeud incrémenté. Nous noterons cette information *Mise-à-jour(NA)*.

Le message d'activation émis à destination du noeud racine de l'arborescence principale en début d'exploration est donc de la forme :

Activation ((Ø, Ø), rac(Arb_0), 0, Ø).

Les traitements effectués par un noeud fonctionnel *DEST* à la réception d'un message d'activation de la forme

activation((ID, NA), DEST, NB-ATT, DEST-MSG-FIN)

sont alors décrits par l'algorithme suivant :

```
Début
    Si NA = Ø alors
        Mise-à-jour(NA)
    Fsi
    Selon les cas faire

        Cas1 : type(DEST) = fct-prim
            {règles 1 et 2 du système de réécriture optimisé}
```

```
Si succ(DEST) = Ø alors
    {un message de fin d'exploration doit être émis à
    destination du message bloqué dont les références
    sont DEST-MSG-FIN}

    envoyer
        Fin-d'exploration((Ø, NA), DEST-MSG-FIN, 0)
Sinon
    {activer le successeur de DEST}

    envoyer Activation((Ø, NA), succ(DEST), 0,
                        DEST-MSG-FIN, 0)
Fin-si
Fin-cas1

Cas2 : type(DEST) =def
    {règles 3 et 4 du système de réécriture optimisé}

    Si succ(DEST) = ø alors
        {un message de fin-d'exploration à destination du
        message dont les références sont DEST-MSG-FIN
        doit être créé bloqué en attente d'un message de
        fin-d'exploration}

        generer(ID) ;
        envoyer
            Fin-d'exploration((ID, NA), DEST-MSG-FIN, 1)

    Sinon
        {un message d'activation doit être créé bloqué à
        destination du successeur de DEST, en attente de
        un message de fin-d'exploration}

        générer(ID)
        envoyer Activation((ID, NA), succ(DEST), 1,
                            DEST-MSG-FIN)
    Fsi
```

{Dans les deux cas, le noeud racine de l'arborescence secondaire doit être activé et en fin de chemin séquentiel gauche de l'arborescence secondaire, un message de fin-d'exploration devra être retourné au message bloqué dont les références sont *(ID, NA).*}

Soit Arb_{def} l'arborescence secondaire correspondante.
envoyer Activation((Ø, Ø), rac(Arb_{def}), 0, (ID, NA))}

Fin-cas2

Cas3 : type(DEST) = f-fonct
{règles 5 et 6 du système de réécriture optimisé}

Soit *p* le nombre de paramètres de la forme fonctionnelle

Si succ(DEST) = ø alors
{un message de fin-d'exploration doit être créé bloqué à destination de *DEST-MSG-FIN*, en attente de *p* messages de fin-d'exploration}

générer(ID) ;
envoyer
Fin-d'exploration((ID, NA), DEST-MSG-FIN, p}

Sinon
{un message d'activation doit être créé bloqué en attente de *p* messages de fin-d'exploration à destination du successeur de *DEST*}

générer(ID) ;
envoyer Activation((ID, NA), succ(DEST), p,
DEST-MSG-FIN)

Fin-si

{dans les deux cas, les noeuds racine des p sous-arborescences paramètres sont activés et les derniers noeuds des chemins séquentiels gauches des sous-arborescences paramètres devront retourner un message de fin-d'exploration au message bloqué dont les références sont *(ID, NA)*}

$\forall i \in [1, p]$, envoyer
Activation((Ø, NA), rac(SA_i(DEST)), 0, (ID, NA))

Fin-cas3

Fin-selon

Fin

Les règles 1 à 6 du système de réécriture optimisé ont ainsi été décrites en termes d'émission et réception de messages. L'état en-attente d'un noeud correspond maintenant au fait qu'un message de fin-d'exploration (règles 7 et 9) ou un message d'activation (règles 8 et 10) a été créé bloqué en-attente de messages de fin d'exploration.

Un message bloqué recevant un message de fin d'exploration décrémente le nombre de messages attendus (*NB-ATT*) et quand *NB-ATT* devient égal à 0, le message est effectivement envoyé. S'il s'agit d'un message d'activation, cela traduit alors les règles 8 et 10, s'il s'agit d'un message de fin d'exploration, cela traduit alors les règles 7 et 9.

VII.1.2.5 : la file d'attente des messages d'activation

Avec cet algorithme, lorsque deux noeuds définition sont activés simultanément, si ces deux noeuds fonctionnels font référence à la même définition, deux messages d'activation seront émis à destination du noeud racine de l'arborescence secondaire représentant cette définition. Ces deux messages ne pourront pas être traités simultanément par le même noeud fonctionnel mais dès que l'un des messages sera traité, l'autre pourra l'être sans attendre l'exploration complète de l'arborescence. Le traitement d'un message d'activation par un noeud fonctionnel consiste uniquement à émettre des messages à destination d'autres noeuds fonctionnels et à destination d'un noeud

séquence. Les deux explorations de l'arborescence secondaire seront donc réalisées presque simultanément.

Afin d'interdire à un noeud fonctionnel de prendre en compte plusieurs messages simultanément, nous pouvons associer à tout noeud fonctionnel, une file d'attente des messages d'activation à traiter ainsi qu'une marque (*libre/occupé*) indiquant si le noeud fonctionnel est en train de traiter un message ou non. De cette façon, un noeud fonctionnel ne peut prendre en compte un message d'activation que si sa marque indique qu'il est "libre". Si la marque indique "occupé", un message d'activation arrivant à destination du noeud fonctionnel est rangé dans la file d'attente des messages à traiter. Quand le noeud fonctionnel a terminé les traitements concernant le message d'activation, sa "marque" est repositionnée à "libre" et le noeud fonctionnel peut alors prendre en compte un autre message d'activation. Le message pris en compte sera le premier message de la file d'attente.

Se pose alors le problème de savoir comment ranger les messages dans la file d'attente, ce qui revient à déterminer l'ordre dans lequel les noeuds fonctionnels doivent traiter les messages d'activation.

La résolution de ce problème nécessite l'étude des différents cas où des explorations simultanées d'une même arborescence fonctionnelle peuvent être demandées.

Il y a exploration multiple lorsque dans une forêt, plusieurs noeuds définition $(n_1, ..., n_m)$ font référence à la même définition et donc à la même arborescence fonctionnelle que nous noterons Arb_{def}. La réception d'un message d'activation par ces noeuds fonctionnels génère alors l'émission d'un message d'activation à destination du noeud racine de Arb_{def} ce qui génère l'exploration de l'arborescence secondaire.

Soient i, j, tels que $1 \leq i \leq m$ et $1 \leq j \leq m$ et $i \neq j$. Si les noeuds fonctionnels n_i et n_j appartiennent au même chemin séquentiel, les explorations de Arb_{def} générées ne peuvent pas être simultanées. En effet, si n_i est "avant" n_j dans le chemin séquentiel, n_i a obligatoirement un successeur (éventuellement n_j) qui n'est activé que lorsque

l'exploration de Arb_{def} est terminée, on peut donc bien parler d'explorations multiples non simultanées.

On ne peut donc générer des explorations simultanées d'une même arborescence fonctionnelle que lors de l'exploration parallèle de sous-arborescences. Si des sous-arborescences sont explorées en parallèle, ce sont des sous-arborescences paramètre d'un noeud forme-fonctionnelle ayant créé un message bloqué, ce message étant en attente de messages de fin-d'exploration des sous-arborescences paramètre. On aurait donc interêt à privilégier le message en provenance de la sous-arborescence paramètre "la plus longue à explorer". La détermination de ce critère conduirait à étudier le temps d'exploration des arborescences fonctionnelles qui ne dépend pas uniquement de la "taille" des arborescences mais également de la nature des noeuds fonctionnels la composant. Cette mesure nous fournirait alors un temps théorique devant être corrigé en prenant en compte les contraintes matérielles. A partir de là, on pourrait attribuer un poids aux noeuds fonctionnels, ce poids étant transmis dans les messages et permettant d'ordonnancer les messages. Seulement, le temps introduit par l'analyse de cette complexité a de fortes chances d'être nettement supérieur au gain de temps obtenu si l'on respecte les priorités. C'est pourquoi, nous décidons que les files d'attente associées aux noeuds fonctionnels fonctionneront en mode *F.I.F.O.* (First In First Out).

VII.2 : L'exploration avec envois de messages à la séquence argument

L'algorithme précédent permet l'exploration d'une forêt d'arborescences fonctionnelles sans prendre en compte les traitements à effectuer sur la séquence argument. Or, lors de l'exploration, les noeuds fonctionnels sont amenés à émettre des messages à destination de la séquence argument afin de générer les réductions. Ainsi, un noeud fonction-primitive devra générer à destination de la séquence argument un message contenant cette fonction primitive, un noeud forme-fonctionnelle devra envoyer un message ayant pour but de "préparer" la

séquence argument de telle façon que les paramètres de la forme fonctionnelle puissent être appliqués à la séquence. Par exemple, un noeud forme-fonctionnelle représentant une *Construction* à *n* paramètres devra générer à destination de la séquence argument un message demandant *n* copies de l'argument de la *Construction*.

Dans un premier temps, nous montrerons comment est représentée dans le modèle, la séquence argument sous la forme d'un arbre. Dans un deuxième temps, nous déterminerons à quel noeud de la séquence (appelé noeud-séquence) un noeud fonctionnel envoie un message si nécessaire. Dans un troisième temps, nous définirons la sémantique des réductions effectuées sur la séquence argument.

VII.2.1 : La séquence argument

Nous appelons séquence argument, la séquence sur laquelle s'applique le programme FP. Comme nous l'avons déjà souligné, un programme FP s'écrit sous la forme d'une composition de fonctions : $f_1 \circ f_2 \circ \ldots \circ f_n$. Ainsi, l'exécution d'un programme FP consiste à appliquer la fonction f_n à la séquence argument ; soit S la séquence résultant de cette application, la fonction f_{n-1}, est ensuite appliquée à la séquence S, produisant une autre séquence S' qui sera l'argument de la fonction f_{n-2} etc...

Ainsi, la séquence argument initiale est modifiée lors de l'exécution d'un programme FP, à chaque application de fonction, mais à tout moment de l'exécution d'un programme FP, il n'y a qu'une séquence argument. Ceci respecte le principe de FP selon lequel toute fonction s'applique à un seul argument pour fournir en résultat un seul objet. Pendant l'exécution, la séquence argument évolue de plusieurs façons : une fonction peut s'appliquer à la séquence argument ou bien à un ou plusieurs des objets composant la séquence. Mais quel que soit l'objet sur lequel elle s'applique, le résultat de cette application conduit toujours à une séquence résultat. Dans le modèle, une séquence argument sera représentée par un arbre.

Soit une séquence $x = < x_1, x_2, ..., x_n >$ où les x_i sont des atomes. La séquence x est représentée par l'arbre de la figure VII.2. Si maintenant il existe i tel que x_i est une séquence, le $i^{ème}$ fils de la racine de l'arbre sera un arbre représentant cette séquence. La figure VII.3 montre des exemples de représentation de différents objets. Ainsi, un arbre peut être réduit à un noeud dans le cas où l'objet à représenter est un atome ou la séquence vide.

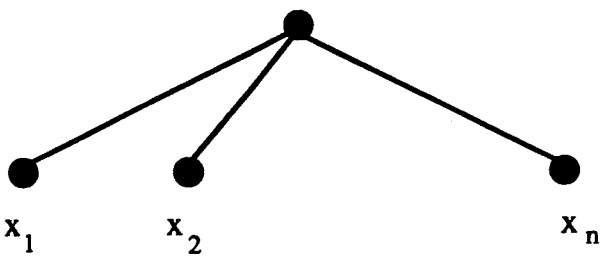


Figure VII.2 : représentation de la séquence x

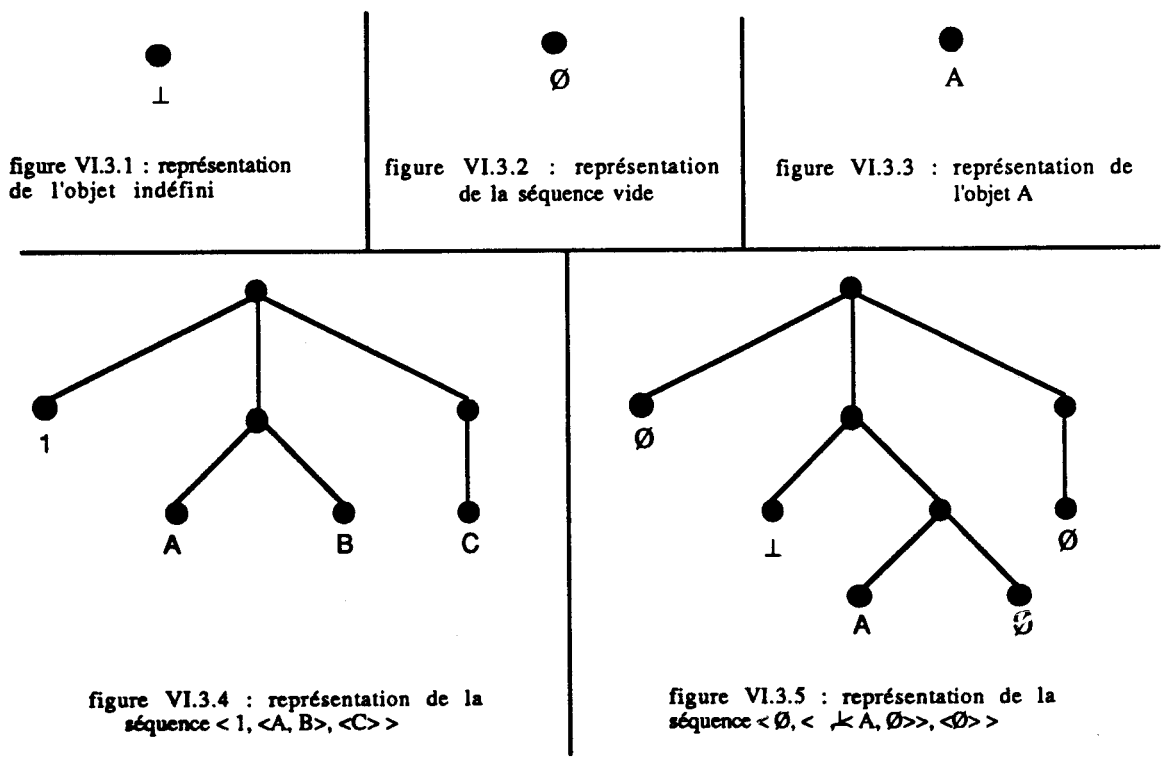


figure VI.3.1 : représentation de l'objet indéfini

figure VI.3.2 : représentation de la séquence vide

figure VI.3.3 : représentation de l'objet A

figure VI.3.4 : représentation de la séquence < 1, <A, B>, <C> >

figure VI.3.5 : représentation de la séquence < Ø, < ⊥ A, Ø>>, <Ø> >

Figure VII.3 : représentation de différents objets.

VII.2.2 : Définition d'un noeud-cible

Toute fonction FP s'applique à une séquence représentée dans le modèle par un arbre. Nous appelons **noeud-cible** d'un noeud fonction-primitive, un noeud-séquence, racine de l'arbre représentant l'argument de la fonction primitive. Nous étendons alors la notion de noeud-cible à tout noeud fonctionnel de la façon suivante : le noeud-cible d'un noeud forme-fonctionnelle ou d'un noeud définition est un noeud séquence, racine de l'arbre représentant l'argument de la forme fonctionnelle ou de la définition.

Avec ces conventions, tout message émis par un noeud fonctionnel à destination de la séquence argument est envoyé au noeud-cible du noeud fonctionnel.

Exemple :

Soit F la fonction FP définie par

$F = \alpha*$.

Cette fonction est une forme-fonctionnelle représentée par une arborescence ne comportant que deux noeuds fonctionnels. Afin de simplifier l'écriture (et la lecture !), nous appelons "noeud fonctionnel α", le noeud fonctionnel ayant pour étiquette le symbole de forme fonctionnelle α. De même, nous désignons par "noeud fonctionnel *", le noeud fonctionnel d'étiquette *.

Etant donnée S la séquence argument de la fonction F, S est l'argument de la forme fonctionnelle α. Le noeud séquence racine de S, est donc le noeud-cible du noeud fonctionnel α. La fonction primitive * s'applique à chaque sous-séquence de la séquence S. Chaque noeud-séquence, racine de ces sous-séquences est donc un noeud-cible du noeud fonctionnel *.

Le traitement d'un message émis par un noeud fonctionnel à destination d'un noeud séquence n conduit, en général, à la modification de l'arbre ayant pour racine le noeud séquence n. C'est le cas par exemple des messages contenant des fonctions primitives : une fonction primitive autre que la fonction *identité* conduit généralement à une séquence argument différente de la séquence initiale. De même, les messages émis par des noeuds forme-fonctionnelle ont pour rôle de permettre la préparation de la séquence argument à l'application des paramètres de la forme fonctionnelle. Cette "préparation" peut conduire également à des modifications de la séquence-argument, par exemple lorsque des copies sont demandées.

Soit n un noeud séquence. Nous désignons par *ARB*, l'arbre de racine n représentant une séquence. Si le noeud séquence n reçoit un message en provenance d'un noeud fonctionnel, le traitement de ce message peut donc entraîner une modification fournissant en résultat un autre arbre que nous désignons par *ARB-RES*.

Nous imposons une contrainte d'exécution qui est la suivante : si l'on accède au noeud racine de *ARB* par une certaine information *A* (cette information pouvant être, selon la mise en oeuvre et la machine cible, un nom, une adresse, etc...), on doit également accéder au noeud racine de *ARB-RES* grâce à la même information *A*. Ainsi, même si le noeud racine de *ARB-RES* est un nouveau noeud, créé lors du traitement du message, celui-ci devra avoir l'ancienne "identité" du noeud racine de *ARB*.

Pourquoi cette contrainte ? Tout d'abord, un message peut être émis à destination d'un noeud séquence racine d'un sous-arbre et non de l'arbre représentant la séquence (par exemple, lors du traitement d'une forme fonctionnelle *Apply-to-all*). Si l'on ne preserve pas l'identité des noeuds racine des sous-arbres, cela nécessite une mise à jour des noeuds pères dans l'arbre. De plus, prenons par exemple une fonction FP définie par $f \circ g$, à appliquer à une séquence donnée, f et g étant des fonctions primitives. Supposons que l'on accède au noeud racine de la séquence-argument par une certaine information n_1. Le noeud fonctionnel g a pour noeud-cible, le noeud-séquence désigné précédemment par n_1. Si l'application de la fonction g à la séquence-argument a pour résultat une

séquence de racine n_2 (n_2 étant l'information grâce à laquelle on accède au noeud), la fonction f (qui s'applique à cette séquence de racine n_2) ne pourra pas être envoyée au noeud-séquence n_2 avant que l'application de g au noeud séquence n_1 ne soit effectuée, et que le noeud fonctionnel f soit informé de l'identité de son noeud-cible. Cela supposerait alors des échanges d'informations continuels entre les arborescences fonctionnelles et la séquence-argument.

La contrainte imposée permet au contraire d'assurer que tous les noeuds fonctionnels appartenant à un même chemin séquentiel aient obligatoirement le même noeud-cible. On peut alors parler du noeud-cible d'un chemin séquentiel.

Initialement, le noeud-cible du chemin séquentiel gauche de l'arborescence fonctionnelle principale est connu dans la mesure où il s'agit du noeud racine de la séquence argument.

Lors de l'exploration d'une arborescence fonctionnelle, un noeud définition activé engendre l'activation du noeud racine du chemin séquentiel gauche de l'arborescence secondaire correspondante : le noeud-cible de ce chemin pour cette activation est alors le même que celui du noeud définition.

Il reste alors à déterminer le noeud-cible des chemins séquentiels gauche des différentes sous-arborescences paramètre d'un noeud forme-fonctionnelle. Leur noeud-cible est a priori inconnu mais lorsque le noeud forme-fonctionnelle est activé, un message est émis à destination de la séquence argument qui est alors éventuellement modifiée. Si la séquence argument n'est pas modifiée, l'information "noeud-cible" peut être obtenue immédiatement et transmise aux noeuds fonctionnels nécessitant cette information sinon, dès que cette information est disponible, elle est communiquée aux noeuds fonctionnels par envois de messages.

VII.2.3: Sémantique des réductions sur la séquence argument

Après avoir défini le formalisme utilisé, nous décrirons dans cette section, la sémantique des transformations ou réductions devant être réalisées sur une séquence argument suite à l'activation d'un noeud fonctionnel dans une arborescence.

VII.2.3.1: Description du formalisme utilisé.

Tout argument d'une fonction est représenté par un arbre, réduit à un noeud dans le cas où l'argument est la séquence vide ou un atome.

Dans le but de rester le plus possible indépendant de toute machine et de toute implantation, le formalisme adopté sera le suivant :

Nous désignerons par

- *arbre vide*, un arbre représentant une séquence vide,
- *arbre atome*, un arbre représentant un atome,
- *arbre séquence*, un arbre représentant une séquence non vide.

L'appellation "*arbre quelconque*" désignera alors soit un arbre vide, soit un arbre atome, soit un arbre séquence. Les différents symboles que nous utiliserons dans les schémas sont représentés en figure VII.4.

Figure VII.4 : symboles utilisés pour la représentations des arbres

Un arbre séquence est alors représenté par un noeud racine ayant au moins un fils, chaque fils étant un arbre quelconque, comme le montre la figure VII.5, dans laquelle $Arb_1, \ldots, Arb_n$ sont des noms donnés aux arbres.

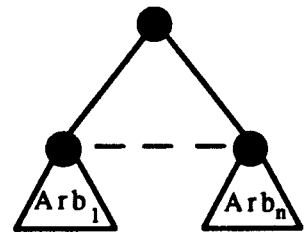


Figure VII.5 : représentation d'un arbre séquence

Remarque :

Deux arbres ayant le même nom représentent le même arbre. Deux arbres n'ayant pas le même nom représentent des arbres éventuellement différents.

L'activation d'un noeud fonctionnel engendre généralement l'envoi d'un message à destination de son ou ses noeuds-cible, dont le but est de demander que certains traitements soient effectués sur l'arbre ayant pour racine ce noeud-cible. Nous considérons dans la présente section, qu'un noeud fonctionnel étiqueté *f* envoie à son noeud-cible un message de type *f*. Les traitements devant être effectués sur la séquence argument sont alors définis par un système de réécriture dont chaque règle est de la forme

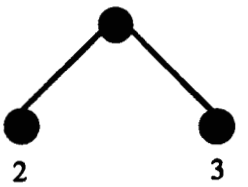
$$(f : arb_1 \rightarrow arb_2).$$

Un noeud-séquence recevant un message doit alors rechercher dans le système, la règle à appliquer.

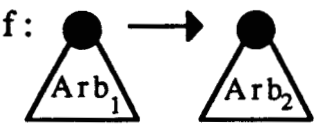
Une règle peut être appliquée si le type du message reçu et l'arbre ayant pour racine, le noeud récepteur du message, peuvent être unifiés avec la partie gauche de règle.

Exemple :

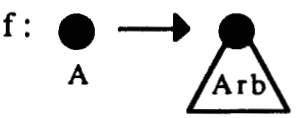
Le noeud racine de l'arbre représenté par



recevant un message de type *f* peut appliquer une règle de la forme



En revanche, il ne peut pas appliquer une règle de la forme



Remarque :

D'après les contraintes d'exécution imposées, une règle de la forme

$$(f : arb_1 \rightarrow arb_2)$$

implique que l'information servant à accéder au noeud racine de l'arbre d'étiquette arb_2 soit rigoureusement identique à l'information permettant d'accéder au noeud racine de l'arbre d'étiquette arb_1.

Les règles doivent être examinées en respectant l'ordre dans lequel elles sont énoncées. Une seule de ces règles doit être appliquée : il s'agit de la première règle dont l'unification est un succès. De cette façon, le système de réécriture est déterministe.

A l'aide de ce formalisme, nous pouvons donc maintenant définir les règles de réécriture.

VII.2.3.2 : règles de réécriture

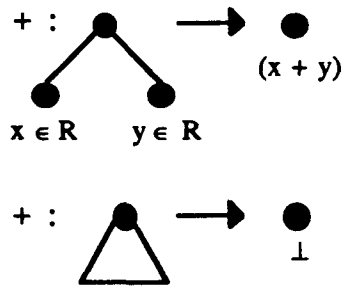
Tout noeud fonctionnel emet un message à destination de son noeud cible. Les traitements à effectuer à la réception de ces messages sont alors décrits par des règles de réécriture.

a : message émis par un noeud fonction primitive

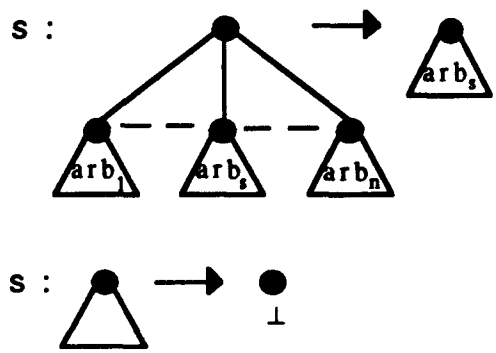
Nous décrivons dans cette section, quelques règles de réécriture à appliquer selon le type du message, c'est-à-dire selon la fonction primitive.

Les fonctions arithmétiques :

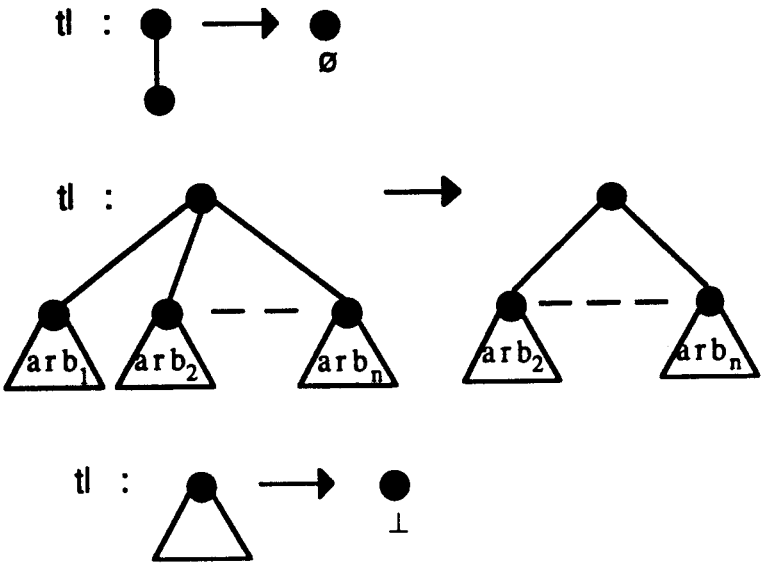
Nous donnons les règles pour l'addition, les règles correspondant aux autres fonctions arithmétiques peuvent être obtenues de la même façon.



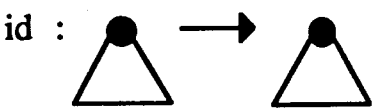
Les fonctions Sélecteur



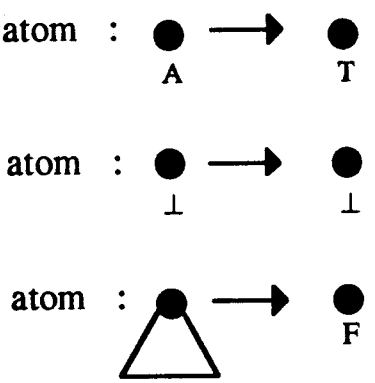
La fonction Tail



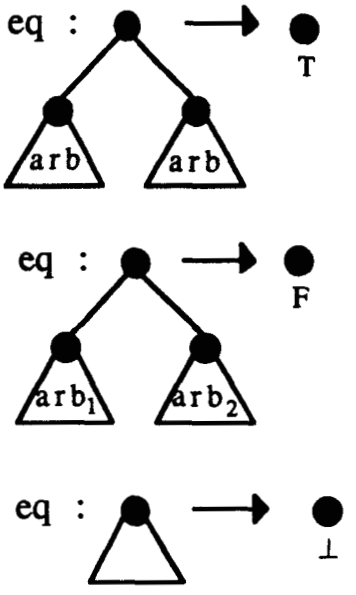
La fonction Identité



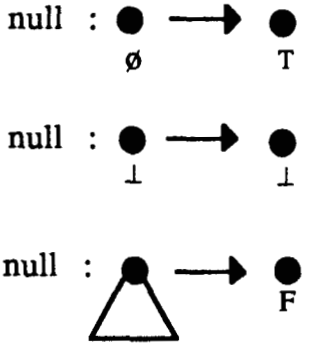
La fonction Atom



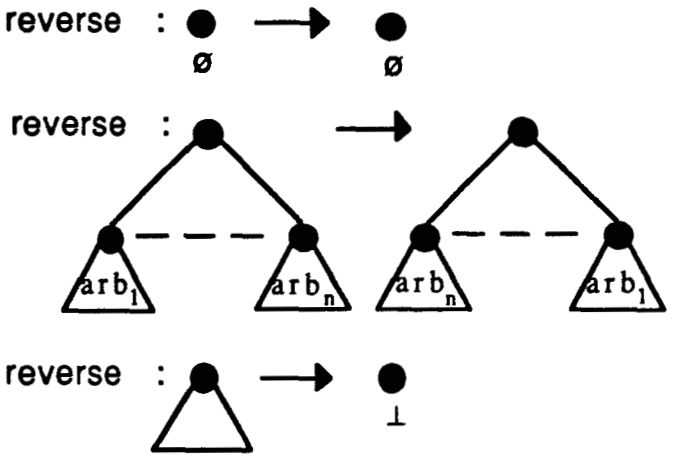
La fonction Equal



La fonction Null



La fonction Reverse



Les règles de réécriture correspondant aux autres fonctions primitives peuvent être décrites de la même façon.

b : message émis par un noeud forme-fonctionnelle

Le traitement d'un message émis par un noeud forme fonctionnelle a pour but de préparer la séquence argument à l'application des paramètres de la forme fonctionnelle.

Cette "préparation" peut éventuellement conduire à une modification de la séquence argument, ceci pouvant être décrit en termes de réécriture de la même façon que sont décrites les modifications apportées à une séquence suite à l'application d'une fonction primitive.

Seulement, les formes fonctionnelles jouent le rôle de structures de contrôle ; des opérations de réécriture ne sont donc pas suffisantes. Dans la plupart des cas, les opérations de contrôle se résumeront, dans le modèle, à la détermination des noeuds cible des paramètres de la forme fonctionnelle (ou plus exactement, des noeuds cible des chemins séquentiels gauches de chaque sous-arborescence paramètre). Des cas particuliers nous conduiront à préciser l'algorithme général d'exploration défini dans le chapitre précédent.

Ainsi, dans les règles de réécriture suivantes, un "rond blanc" représentera un noeud séquence qui est le noeud cible d'un paramètre de forme fonctionnelle.

Si la forme fonctionnelle n'a qu'un paramètre, chaque "rond blanc" en partie droite de règle sera alors un noeud cible du paramètre. Si la forme fonctionnelle a plusieurs paramètres, un numéro dans le "rond blanc" indiquera de quel paramètre le noeud séquence est le noeud cible.

Etant donné un symbole de forme fonctionnelle *ff*, nous appellerons "*noeud fonctionnel ff*", le noeud forme fonctionnelle ayant pour étiquette le symbole de la forme fonctionnelle *ff*.

b.1 : message émis par un noeud fonctionnel α

Dans FP, la forme fonctionnelle *Apply-to-all* est définie par

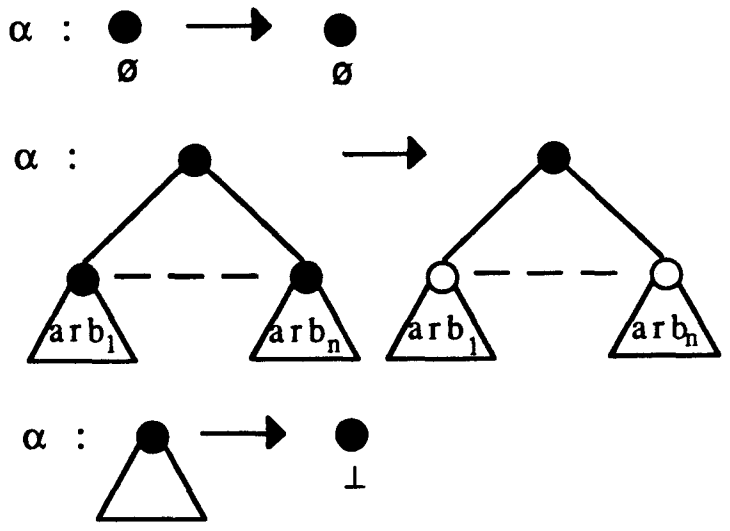
$\alpha f : x \equiv x = \emptyset \rightarrow \emptyset$; (1)

$x = \langle x_1, ..., x_n \rangle \rightarrow \langle f : x_1, ..., f : x_n \rangle$; (2)

$\perp$ (3)

La règle (2) montre que si l'objet argument est une séquence composée de n objets, la fonction paramètre de la forme fonctionnelle est appliquée à chacun de ces objets. Ces applications peuvent être effectuées sans qu'aucune modification ne soit préalablement apportée à la séquence argument. Le rôle d'un message émis par un noeud fonctionnel α se limite à la détermination des noeuds cible du paramètre de la forme fonctionnelle. La fonction paramètre devant être appliquée à chaque objet composant la séquence argument, chaque noeud fils du noeud cible du noeud fonctionnel α est un noeud cible de la fonction paramètre. Cette information devra être communiquée au noeud racine de la sous arborescence paramètre, mettant ainsi en évidence la nécessité d'un retour d'information.

La règle (1) montre que si l'objet argument de la forme fonctionnelle est la séquence vide, le résultat de l'application de la forme fonctionnelle à son argument est la séquence vide. Autrement dit, le résultat est déterminé sans que la fonction paramètre ne soit appliquée à un objet. Cela signifie donc que dans le modèle, l'exploration de la sous-arborescence paramètre est inutile : le message émis par le noeud fonctionnel α sera suffisant pour obtenir le résultat de l'application de la forme fonctionnelle à son argument ; dans la règle de réécriture, aucun "rond blanc" n'apparaîtra dans la partie droite de règle, le retour d'information devra alors signaler au noeud fonctionnel α qu'il doit considérer que la sous-arborescence paramètre est explorée. Dans tous les autres cas, la règle (3) est appliquée et le même principe que celui énoncé dans la règle (2) est mis en oeuvre. Les règles de réécriture correspondant au traitement d'un message émis par un noeud fonctionnel α sont donc les suivantes :



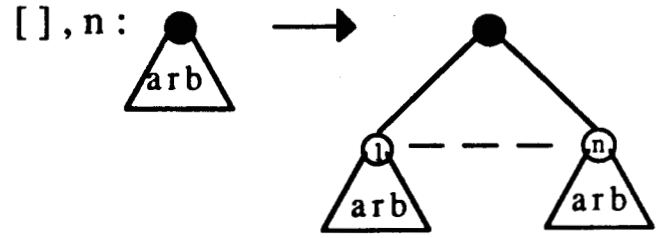
Dans la suite, chaque fois qu'aucun rond blanc n'apparaîtra en partie droite de règle, cela signifiera qu'il est inutile d'explorer les sous-arborescences paramètres, le résultat de l'application de la forme fonctionnelle à la séquence argument étant directement obtenu.

b.2. Message émis par un noeud fonctionnel [ ]

La forme fonctionnelle *Construction* est définie par

$$[f_1, \ldots, f_n] : x = \langle f_1 : x, \ldots, f_n : x \rangle$$

Quel que soit l'objet x, le résultat de l'application d'une *Construction* à l'objet x est donc une séquence, les objets de cette séquence étant le résultat de l'application des paramètres de la *Construction* à l'objet x. Comme nous l'avons déjà signalé, l'application à une séquence, d'une *Construction* ayant n paramètres, nécessite la création de n exemplaires de cette séquence. La règle de réécriture correspondant au traitement que doit effectuer un noeud séquence à la réception d'un message émis par un noeud fonctionnel *[ ]* à n paramètres (noté *[ ], n*) consiste donc à réécrire la séquence argument en une séquence constituée de n objets, ces n objets étant des copies de la séquence argument initiale. De plus, le noeud racine de la $i^{ème}$ copie est le noeud cible du $i^{ème}$ paramètre de la *Construction*. La règle de réécriture est donc la suivante :

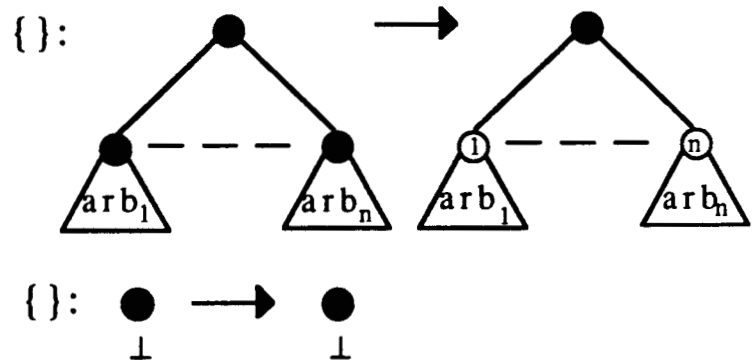


b.3. Message émis par un noeud fonctionnel { }

La forme fonctionnelle *Distribution* introduite en section I.6.2.2 est définie par

$$\{f_1, \ldots, f_n\} : x \equiv \underset{\perp}{x} = \langle x_1, \ldots, x_n \rangle \rightarrow \langle f_1 : x_1, \ldots, f_n : x_n \rangle ;$$

Les règles de réécriture peuvent être obtenues de la même façon qu'ont été obtenues les règles pour l'*Apply-to-all* et la *Construction* :



Remarque :

Les règles correspondant aux formes fonctionnelles Apply-to-all, *Construction* et *Distribution* mettent en évidence le parallélisme pouvant être mis en oeuvre lors des réductions : à un même niveau des arborescences apparaissent plusieurs "ronds blancs". Les noeuds séquence ainsi représentés sont les noeuds-cible de paramètres de forme fonctionnelle. Dans le cas de l'*Apply-to-all*, le message émis à destination de la séquence argument par tout noeud fonctionnel appartenant au chemin séquentiel gauche de la sous-arborescence

paramètre sera diffusé à chacun des noeuds-cible, générant ainsi des réductions en parallèle. Dans le cas de la *Construction* et la *Distribution*, les sous-arborescences paramètres sont explorées simultanément générant ainsi des envois de messages en parallèle aux noeuds-cible.

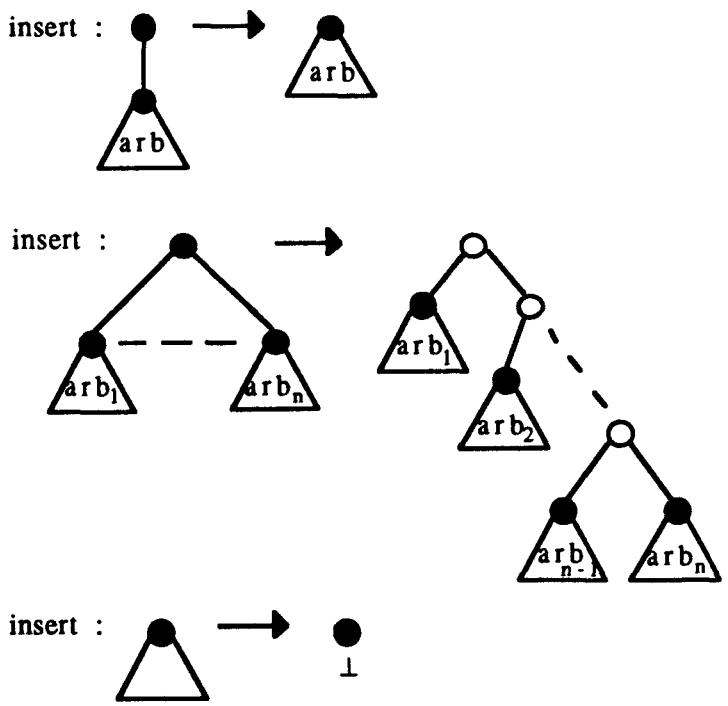
Les règles de réécriture montrent que le parallélisme est effectif dans la mesure où chaque noeud-cible est racine d'un sous-arbre et les réductions effectuées sur les différents sous-arbres sont indépendantes les unes des autres.

b.4. Message émis par un noeud fonctionnel /

La forme fonctionnelle *Insert* est définie par :

$/f : x \equiv x = \langle x_1 \rangle \rightarrow x_1 ;$
$x = \langle x_1, \ldots, x_n \rangle \ \& \ n \geq 2 \rightarrow f : \langle x_1, /f : \langle x_2, \ldots, x_n \rangle \rangle ;$
$\perp$

Les règles de réécriture correspondant au traitement d'un message émis par un noeud fonctionnel / sont alors les suivantes :



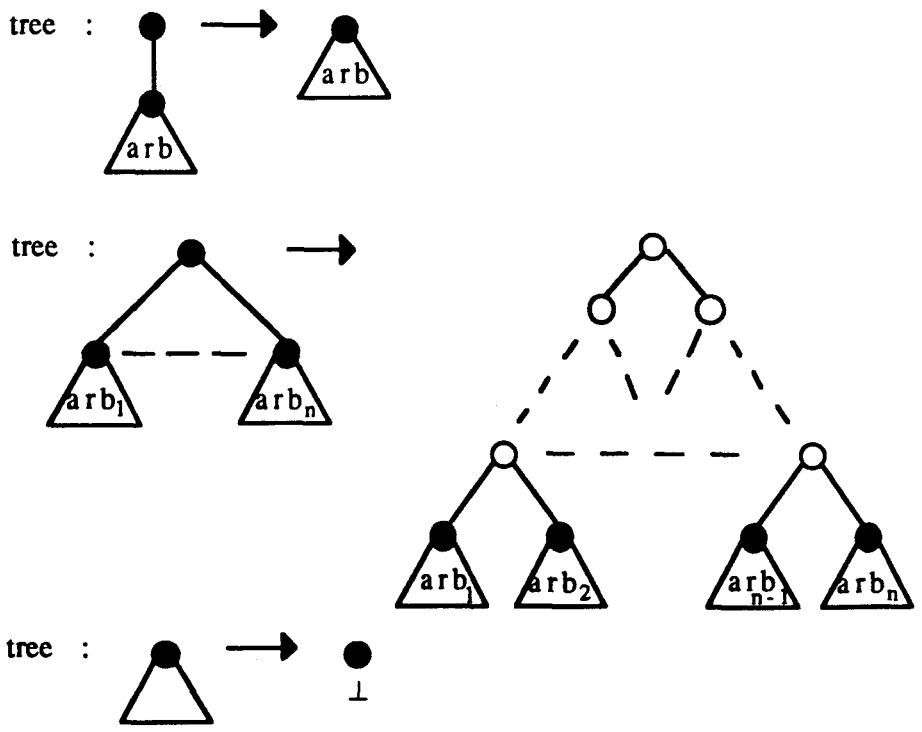
La deuxième règle fait apparaître qu'une transformation de la séquence initiale est nécessaire pour l'application des paramètres de la forme fonctionnelle. De plus, elle met en évidence l'aspect séquentiel de cette forme fonctionnelle qui avait déjà été signalé en section II.6.6.2 ; justifiant l'utilisation de la forme fonctionnelle *Tree* chaque fois qu'elle est possible.

b.5. Message émis par un noeud fonctionnel *Tree*

La forme fonctionnelle *Tree* introduite en section II.6.2.2 est définie par :

(Tree f) : $x \equiv x = \langle x_1 \rangle \rightarrow x_1$;
$x = \langle x_1, \ldots, x_n \rangle \rightarrow$
f : < (tree f) : $\langle x_1,\ldots,x_m \rangle$, (tree f) : $\langle x_{m+1},\ldots, x_n \rangle$ >>
avec m = n DIV 2 ;
⊥

Les règles de réécriture sont les suivantes :



Le parallélisme introduit par cette forme fonctionnelle est mis en évidence dans la deuxième règle où plusieurs "ronds blancs" apparaissent à un même niveau de l'arbre, permettant ainsi des réductions en parallèle.

b.6. Message émis par un noeud fonctionnel *Cste*

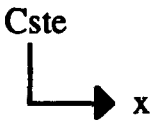
La forme fonctionnelle *Constante* est définie par :

$$\overline{x} : b \equiv y = \bot \rightarrow \underset{x}{\bot} ;$$

Le paramètre x est un objet quelconque : un atome ou une séquence. Quel que soit cet objet, si la forme fonctionnelle $(\overline{x})$ est appliquée à un objet y, le résultat de l'application est l'objet x, i.e. : le paramètre de la forme fonctionnelle.

Si l'objet x est un atome, le noeud fonctionnel représentant le paramètre de $(\overline{x})$ contiendra l'atome sinon, le paramètre x sera représenté de la même façon qu'est représentée une séquence argument et le noeud fonctionnel représentant le paramètre de $(\overline{x})$ contiendra l'identification du noeud racine de l'arbre représentant l'objet x.

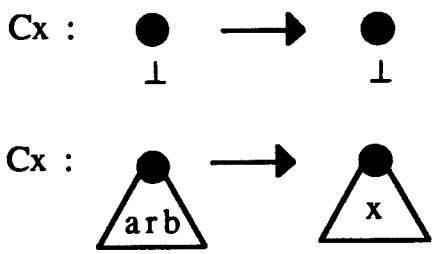
Dans la section VI.2.2., nous avions donné la représentation suivante de $(\overline{x})$:



Avec cette représentation, lorsque le noeud *Cste* reçoit un message d'activation, il n'a aucun traitement particulier à effectuer, si ce n'est activer son paramètre. En effet, aucune modification préalable de la séquence argument n'est nécessaire et le noeud cible du paramètre est le même que le noeud cible du noeud *Constante*. Cela signifie que le noeud *Cste* est inutile : un seul noeud fonctionnel peut représenter la forme fonctionnelle *Cste*. La forme fonctionnelle $(\overline{x})$ sera donc représentée par un noeud fonctionnel étiqueté par Cx où x est soit un

atome soit l'identification du noeud séquence, racine de l'arbre représentant l'objet x.

De cette façon, les règles de réécriture correspondant au traitement d'un message émis par un noeud fonctionnel Cx sont les suivantes :



où

désigne l'arbre représentant l'objet x

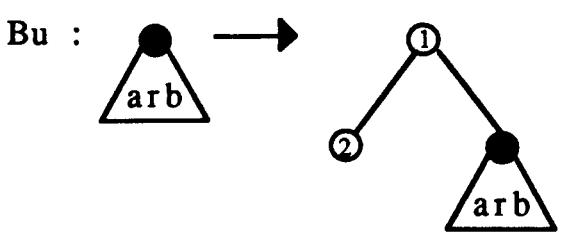
b.7. Message émis par un noeud fonctionnel *B u*

La forme fonctionnelle *Binary-to-Unary* est définie par :

(Bu f x) : y ≡ f : < x, y >

Le paramètre x de cette forme fonctionnelle étant un objet, il est représenté de la même façon qu'est représenté le paramètre de la forme fonctionnelle $(\overline{x})$.

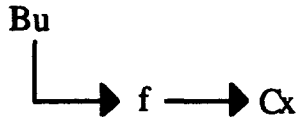
La règle de réécriture est donc la suivante :



Le noeud racine de l'arbre en partie droite de règle est le noeud cible du premier paramètre, c'est-à-dire la fonction f, et son fils gauche

est le noeud cible du deuxième paramètre, c'est-à-dire le noeud représentant l'objet x.

Cela signifie que la forme fonctionnelle *(Bu f x)* peut être représentée par l'arborescence



où l'étiquette *Cx* a la même signification que cette utilisée dans la représentation de la forme fonctionnelle $(\overline{x})$.

b.8. Message émis par un noeud fonctionnel *Cond*

La forme fonctionnelle *Condition* est définie par :

$$(p \rightarrow f ; g) : x \equiv (p : x) = T \rightarrow f : x ;$$
$$(p : x) = F \rightarrow g : x ;$$
$$\perp$$

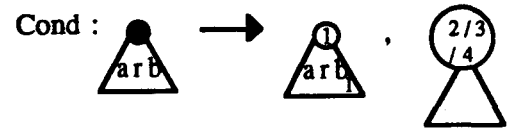
Ainsi, si l'application de la fonction p à l'objet x produit l'atome *True*, la fonction f doit être appliquée à l'objet x. Si l'application de la fonction p à l'objet x produit l'atome *False*, c'est la fonction g qui doit être appliquée à l'objet x. Sinon, le résultat de l'application de la *Condition* à l'objet x est l'objet *indéfini*, c'est-à-dire que la forme fonctionnelle *Constante* ayant pour paramètre l'objet *indéfini* (notée $\overline{\perp}$), doit être appliquée à l'objet initial. Dans les arborescences fonctionnelles, nous ajouterons donc un quatrième paramètre à la forme fonctionnelle *Condition*, ce quatrième paramètre étant la représentation de la fonction $\overline{\perp}$. Si l'application de la fonction prédicat ne produit ni l'atome *True* ni l'atome *False*, c'est ce quatrième paramètre qui devra être appliqué.

Ainsi, deux fonctions différentes sont appliquées au même objet : la fonction prédicat et l'une des autres fonctions paramètre. L'objet argument d'une *Condition* doit donc être dupliqué. Le traitement d'un message émis par un noeud fonctionnel *Cond* à destination de son noeud

cible doit donc générer dans un premier temps, la duplication de l'arbre ayant pour racine le noeud cible.

Le noeud racine de la copie est alors le noeud cible de la fonction prédicat (notée p dans la définition) et le noeud racine de l'original est le noeud cible, soit de la fonction f, soit de la fonction g, soit de la fonction $\bar{\perp}$ selon le résultat de l'application de la fonction p à la copie.

Ainsi, la règle de réécriture correspondant au traitement d'un message émis par un noeud fonctionnel *Cond* est la suivante :



dans laquelle (2/3/4) signifie que le noeud séquence est le noeud cible du 2ème, 3ème ou 4ème paramètre.

L'application de la fonction prédicat à la copie de l'objet initial produit un objet résultat permettant de déterminer quelle fonction doit être appliquée à l'objet initial. Une fois la décision effectuée, l'objet résultant de l'application de p doit être détruit.

Pour cela, dans les arborescences fonctionnelles, nous ajouterons un noeud fonctionnel en fin de chemin séquentiel gauche de la sous-arborescence représentant la fonction prédicat. Ce noeud fonctionnel aura pour étiquette "*Fcond*" et contiendra comme information, les identifications des noeuds fonctionnels racine des sous-arborescences paramètres représentant les fonctions f, g et $\bar{\perp}$.

Une forme fonctionnelle *Cond* sera donc représentée dans les arborescences comme le montre la figure V.7.

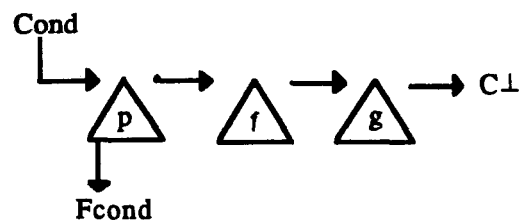


Figure V.7 : représentation d'une forme fonctionnelle Condition dans une arborescence fonctionnelle

De cette façon, lors de l'exploration d'une arborescence fonctionnelle, l'activation d'un noeud fonctionnel d'étiquette "*Fcond*" signifie la fin d'exploration d'une sous-arborescence prédicat. Un message sera alors émis par le noeud *Fcond* à destination de son noeud cible (i.e. : le noeud racine de l'arbre résultant de l'application de la fonction prédicat), ce message contenant les identifications des noeuds fonctionnels racines des sous-arborescences paramètres représentant les fonctions f, g, et $\bar{\perp}$.

Si le noeud séquence recevant ce message représente l'atome *True*, un message d'activation devra alors être envoyé au noeud racine de la sous-arborescence paramètre représentant la fonction f.

Si le noeud séquence représente l'atome *False*, un message d'activation devra alors être envoyé au noeud racine de la sous-arborescence paramètre représentant la fonction g.

Sinon, le message d'activation devra être envoyé au noeud racine de la sous-arborescence paramètre représentant la fonction $\bar{\perp}$.

Les messages d'activation peuvent ainsi être émis par des noeuds séquence. Lors de l'exploration, le noeud fonctionnel *Cond* n'activera donc que la sous-arborescence paramètre représentant la fonction prédicat, l'une des autres sous-arborescences sera activée par un noeud séquence. Le message créé bloqué par le noeud fonctionnel *Cond* (message d'activation à destination du successeur s'il existe ou message de fin d'exploration sinon) sera donc en attente de 2 messages de fin d'exploration.

Il ne reste alors qu'un problème de noeud cible. Tel que le principe est défini, le noeud séquence qui active l'une des sous-arborescences paramètre (i.e. : qui envoie un message d'activation au noeud racine de cette sous-arborescence) ne connait pas le noeud cible de ce noeud fonctionnel. Ce noeud cible est, comme le montre la règle de réécriture, le noeud cible du noeud fonctionnel *Cond*. Plusieurs solutions peuvent

alors être envisagées. Nous donnons ici, celle qui nous semble la plus aisée à mettre en oeuvre

Les arborescences fonctionnelles sont statiques, i.e. : elles sont construites à la compilation et ne sont plus modifiées par la suite. Un noeud fonctionnel *Cond* peut donc "connaître" l'identification du noeud fonctionnel *Fcond* : il suffit de lui donner cette information à la compilation. A partir de là, un noeud fonctionnel *Cond*, lorsqu'il est activé, peut envoyer au noeud fonctionnel *Fcond*, un message contenant l'identification de son noeud cible et son numéro d'activation. Le noeud *Fcond* mémorise ces informations. Si le noeud *Cond* est activé plusieurs fois, le noeud *Fcond* possédera alors une liste de couples (n° d'activation, noeud-cible). Lorsque le noeud *Fcond* reçoit un message d'activation dont le numéro d'activation est *n*, il recherche dans cette liste le couple dont le numéro d'activation est *n* et le message qu'il transmet à son noeud cible contient, en plus de l'identification des noeuds racines des sous-arborescences paramètre, le noeud cible de ces noeuds fonctionnels (i.e. : le noeud-cible présent dans le couple sélectionné). Le noeud séquence récepteur peut alors émettre un message d'activation à destination de ces noeuds fonctionnels.

b.9. Message émis par un noeud fonctionnel *While*

La forme fonctionelle *While* est définie par

$$\begin{aligned}(\text{while } p\ f) : x \equiv\ & p : x = T \rightarrow (\text{while } p\ f) : (f : x)\ ; \\ & p : x = F \rightarrow x\ ; \\ & \perp\end{aligned}$$

De même que pour la forme fonctionnelle *Cond*, nous apportons quelques modifications à la représentation d'une forme fonctionnelle *While* dans une arborescence.

La "partie droite" de la définition de la forme fonctionnelle *While* peut être écrite sous la forme d'une forme fonctionnelle *Condition* :

$$(\text{While } p\ f) : x \equiv p \rightarrow (\text{while } p\ f) : (f : x) : x$$

ce qui peut également s'écrire

$$(\text{while } p\ f) \equiv p \rightarrow (\text{while } p\ f) \circ f\ ;\ id$$

cela signifie qu'une fonction de la forme

$$\text{Def } P = f_1 \circ \ldots \circ f_{i-1} \circ (\text{while } p\ f_i) \circ f_{i+1} \circ \ldots \circ f_n$$

peut être transformée en

$$\text{Def } P = f_1 \circ \ldots f_{i-1} \circ w_i \circ f_{i+1} \circ \ldots \circ f_n$$

avec

$$\text{Def } w_i = p \rightarrow w_i \circ f_i\ ;\ id$$

De cette façon, la fonction *P* est représentée par les deux arborescences de la figure V.8.

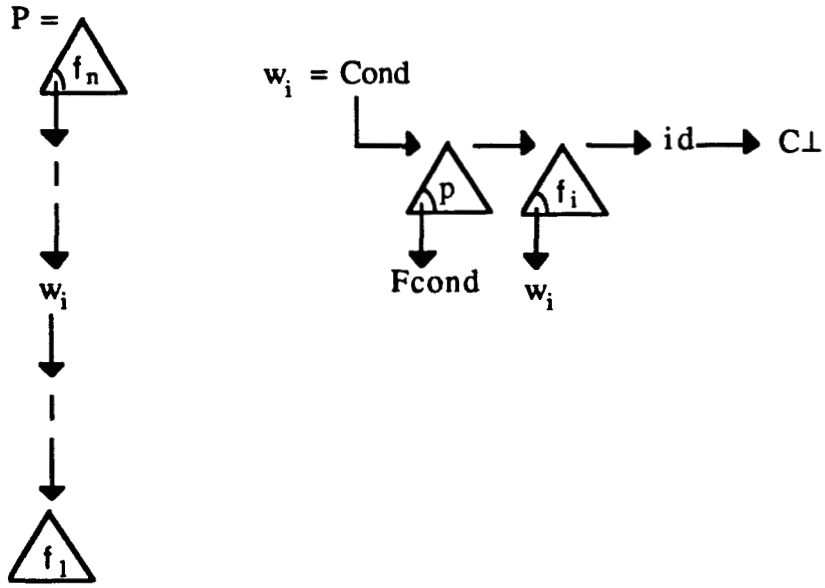


Figure V.8 : représentation de la forme fonctionnelle *While*

Ainsi, nous n'avons pas besoin de règle de réduction pour la forme fonctionnelle *While* dans la mesure où l'on se ramène à une forme fonctionnelle *Cond* .

c : autres messages

Dans les sections b6 et b7, nous avons décrit les traitements engendrés lors de la réception d'un message émis par un noeud objet. Il reste donc à traiter les messages émis les noeuds définition. L'activation d'un noeud définition conduit à l'activation du noeud racine de l'arborescence secondaire correspondante sans qu'aucune réduction ou transformation de la séquence argument ne soit nécessaire. A ce stade, il ne semble donc pas utile qu'un noeud définition émette un message à destination de son noeud-cible. Malgré cela, nous considérons qu'un noeud définition activé, envoie à son noeud-cible un message de type *Def*. De cette façon, tout noeud fonctionnel activé emet un message à destination de son ou ses noeuds-cible.

Les messages de type *Def* joueront un role important dans l'ordonnancement des messages à destination de la séquence argument, comme nous le verrons dans la section suivante.

Ceci termine la définition des règles de réécriture devant être appliquées lorsqu'un noeud séquence traite un message en provenance d'un noeud fonctionnel.

L'exploration des arborescences décrit l'envoi de messages à la séquence argument, les règles de réécriture décrivent le traitement de ces messages, il ne reste plus qu'à traiter l'ordonnancement des messages émis par les noeuds fonctionnels à destination des noeuds séquence.

VI.3. L'ordonnancement des messages

Si l'algorithme d'exploration des arborescences fonctionnelles permet d'émettre les messages à destination de la séquence-argument dans l'ordre où ils devront être traités, rien ne permet en revanche, d'assurer que ces messages arrivent à destination dans l'ordre de leur émission. Ainsi, tel que l'algorithme est défini, lorsqu'un message est reçu par un noeud-séquence, rien ne permet au noeud-séquence de décider s'il peut ou non traiter ce message ou s'il doit attendre un message précédemment envoyé et non encore reçu. Nous proposons un mécanisme d'ordonnancement basé sur un nouveau découpage des arborescences fonctionnelles en branches séquentielles.

VI.3.1. Les trois niveaux d'ordonnancement

Le problème de l'ordonnancement des messages à destination des noeuds-séquence se retrouve à trois niveaux :

1) Les noeuds fonctionnels appartenant à un même chemin séquentiel, ayant le même noeud-cible, tout message émis par ces noeuds est à destination du même noeud-séquence. A un premier niveau, un noeud-séquence doit donc pouvoir ordonner les messages en provenance d'un même chemin séquentiel

2) Plusieurs chemins séquentiels peuvent avoir le même noeud cible.

exemple 1 :

Soit *F1* une fonction FP définie par

$$\text{Def F1} = f_1 \circ \ldots \circ f_i \circ \text{Def} \circ f_{i+1} \circ \ldots \circ f_n$$

où les f_i sont des fonctions quelconques, *Def* étant le nom d'une définition secondaire(notée Arb_{def}) représentée par une arborescence fonctionnelle. Nous notons Arb_{F1} l'arborescence fonctionnelle représentant la fonction *F1*.

Le chemin séquentiel gauche de Arb_{def} ($CSG(Arb_{def})$) a le même noeud-cible que le chemin séquentiel gauche de Arb_{F1} et les messages émis par les noeuds fonctionnels appartenant à $CSG(Arb_{def})$ doivent être traités avant ceux émis par les noeuds appartenant à $CS(f_i)$ (i.e. : le chemin séquentiel ayant pour racine le noeud fonctionnel étiqueté f_i)

exemple 2 :

Soit *F2* une fonction FP définie par

$$F_2 = \alpha\, f_1 \circ f_2 \circ \alpha\, f_3$$

où les f_i sont des fonctions primitives.

Dans la figure VI.9 qui représente cette fonction, les "α" sont indicés de façon à pouvoir nommer les noeuds fonctionnels sans ambiguïté.

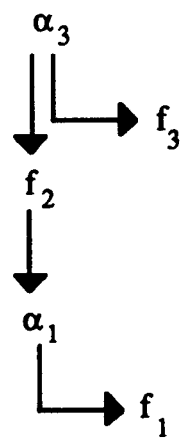


Figure 18 : Arborescence fonctionnelle représentant la fonction $F_2 = \alpha\, f_1 \circ f_2 \circ \alpha\, f_3$

Soit n, le noeud racine de la séquence argument et $\{n_1, \ldots, n_m\}$, l'ensemble de ses fils. Les noeuds fonctionnels α_3, f_2, et α_1 ont pour noeud-cible, le noeud séquence n. Le noeud fonctionnel α_3 a pour noeuds-cible l'ensemble des noeuds-séquence $\{n_1,\ldots,n_m\}$. Quelle que soit la fonction f_3, l'information d'accès à ces noeuds ne sera pas modifiée par l'application de la fonction f_3. Si l'application de la fonction f_2 à la séquence argument ne modifie pas les informations d'accès aux fils du

noeud n, le noeud fonctionnel f_1 aura donc également pour noeuds cible, les noeuds-séquence $\{n_1,...,n_m\}$. Ces deux exemples montrent la nécessité de mettre en place un mécanisme permettant à un noeud séquence d'ordonner à un deuxième niveau les messages en provenance de différents chemins séquentiels.

3) Les deux premiers cas concernent les messages à destination d'un même noeud séquence. A un troisième niveau, le mécanisme d'ordonnancement doit également permettre d'ordonner les messages à destination de noeuds séquence différents, si leurs traitements ne peuvent pas être effectués en parallèle.

Exemple :

Si un message contenant une fonction primitive f arrive à destination d'un noeud séquence n et que simultanément, un message contenant une fonction primitive g arrive à destination d'un noeud séquence n', n' étant un fils de n, on doit pouvoir déterminer si la fonction f doit d'abord être appliquée au noeud n, puis la fonction g au noeud n' ou inversement.

Dans un premier temps, nous proposons une solution permettant l'ordonnancement aux deux premiers niveaux cités ci-dessus (i.e. : l'ordonnancement des messages à destination d'un même noeud-séquence).

Dans un deuxième temps, nous montrerons que l'ordonnancement au troisième niveau ne nécessite aucun mécanisme particulier : il est implicite.

VI.3.2. L'ordonnancement des messages à destination d'un même noeud séquence

Ce problème est résolu en introduisant un nouveau découpage des chemins séquentiels maximaux d'une arborescence, en ce que nous appellerons des branches séquentielles.

De façon intuitive, une branche séquentielle BS est un sous-chemin séquentiel tel que si n_1 et n_2 sont deux noeuds fonctionnels appartenant à BS tels que $n_2 = succ(n_1)$ et NDC est leur noeud-cible, le message émis à destination de NDC par le noeud fonctionnel n_2 devra être traité immédiatement après le message émis par le noeud fonctionnel n_1.

Soit CS, un chemin séquentiel dans une arborescence fonctionnelle
Soient $n_1, n_2 \in CS$ tels que $n_2 = succ(n_1)$, et $m1, m2$, les messages émis respectivement par n_1 et n_2 à destination de NDC .

• Si n_1 est un noeud fonction primitive ou un noeud objet étiqueté par Cx, il est clair que NDC devra traiter $m2$ immédiatement après $m1$, et donc :

si $n_1 \in$ BR et type(n_1) = f-prim ou type(n) = objet
alors $n_2 \in$ BR

• Si n_1 est un noeud forme-fonctionnelle, NDC doit traiter $m2$ immédiatement après $m1$ seulement si NDC n'est pas le noeud cible de l'un des paramètres de la forme fonctionnelle, auquel cas, les messages émis par le paramètre devraient être traités avant $m2$. Les règles de réécriture, définies en section VI.2.3.2 nous montrent quelles sont les formes fonctionnelles dont un des paramètres a le même noeud-cible que le noeud forme fonctionnelle : il s'agit des formes fonctionnelles dont les règles de réduction montrent en partie droite de règle "un rond blanc" à la racine de l'argument et donc les formes fonctionnelles *Insert, Tree, Binary-to-unary, Condition.*

Si n_1 est un noeud forme fonctionnelle étiquetté par *Tree*, /, *Bu* ou *Cond*, le message $m2$ ne sera donc pas obligatoirement traité immédiatement après le message $m1$ et donc :

```
Si n1 ∈ BR et type(n1) = f- fonct alors
|   si étiquette(n1) ∈ {|, Tree, Bu, Cond} alors
|   |   n2 ∉ BR
|   sinon
|   |   n2 ∈ BR
|   Fsi
Fsi
```

• Si n_1 est un noeud définition et que Arb_{def} est l'arborescence secondaire représentant cette définition, $\forall\, n \in CSG(Arb_{def})$, le noeud cible de n est également NDC et le message émis par n devra être traité par NDC avant $m2$ et donc :

Si $n_1 \in$ BR et type(n_1) = def alors $n2 \notin$ BR

A chaque noeud, nous associons un couple (i, n) déterminant à quelle branche séquentielle le noeud appartient.

- Soit $Arb = \{N, A\}$ une arborescence fonctionnelle
- Soient $r_1, ..., r_n \in N$ tels que

$CS(r_1),...,CS(r_n)$ soient les chemins séquentiels maximaux de Arb.

$Succ^j(r_i)$ désigne le j^{ième} successeur de r_i ($succ^0(r_i) = r_i$).

L'affectation d'un couple (i, n) à un noeud fonctionnel $succ^j(r_i)$ est notée

nbr(succ^j(r_i)) ← (i, n).

Avec ces conventions, l'algorithme permettant d'associer un couple (i, n) à chaque noeud fonctionnel de Arb est alors le suivant :

```
Pour chaque r_i faire
|   j ← 0
|   n ← 1
|   TQ succ^j(n_i) ≠ Ø faire
|   |   nbr(succ^j(r_i)) ← (i, n)
|   |   Si type(succ^j(r_i)) = def ou
|   |   |   (type(succ^j(r_i)) = f.fonct et
|   |   |       étiquette(succ^j(r_i)) ∈ {|, tree, Bu, Cond})
|   |   alors
|   |   |
|   |   |   n ← n + 1
|   |   |
|   |   Fsi
|   |   j ← j + 1
|   FTQ
F pour
```

De cette façon, le couple *(i, n)* désigne un nom de branche séquentielle. Nous désignerons dans la suite par *N_BS*, le nom de la branche séquentielle associée à un noeud fonctionnel.

Remarque :

L'algorithme est défini de telle façon que deux branches séquentielles appartenant à la même arborescence n'ont pas le même nom.

A chaque noeud fonctionnel est également associé le nom de l'arborescence à laquelle il appartient, noté *N_ARB*. De plus, les noeuds fonctionnels appartenant à une même branche séquentielle sont numérotés par ordre croissant à partir de 1 : si un noeud a le numéro *n* et que son successeur appartient à la même branche séquentielle, le successeur aura le numéro (*n* + 1). Le numéro d'ordre d'un noeud fonctionnel dans une branche séquentielle sera noté *N_ORD*.

Enfin, tout noeud fonctionnel possède un indicateur de fin de branche (noté *IFB*) positionné à vrai ou faux selon que le noeud fonctionnel est, ou n'est pas, le dernier noeud de la branche. Si

l'indicateur de fin de branche d'un noeud fonctionnel est positonné à vrai, cela signifie donc que le message émis par le noeud fonctionnel à destination de son noeud cible est le dernier message en provenance de cette branche, que le noeud cible devra traiter.

Ainsi, les références d'un noeud fonctionnel seront constituées de :

- *N_ARB* : le nom (ou numéro) de l'arborescence fonctionnelle à laquelle il appartient,

- *N_BS* : le nom de la branche séquentielle à laquelle il appartient,

- *N_ORD* : son numéro d'ordre dans la branche,

- *IFB* : l'indicateur de fin de branche positionné à vrai ou faux.

Les arborescences fonctionnelles étant statiques, les informations *N_ARB*, *N_BR* et *N_ORD* peuvent être attribuées aux noeuds fonctionnels lors de la construction des arborescences.

L'information contenue dans les messages à destination des noeuds séquence concerne les traitements à effectuer sur la séquence argument. Pour chaque noeud fonctionnel, on peut faire en sorte de transmettre toutes les informations necessaires dans un seul message, ce qui évite de devoir ordonner les messages en provenance d'un même noeud fonctionnel. Nous considèrerons donc que tout noeud fonctionnel envoie un message à son noeud-cible.

De cette façon, si un noeud fonctionnel envoyant un message à son noeud-cible transmet dans le message ses références, le noeud-cible a maintenant la possibilité d'ordonner les messages en provenance des noeuds fonctionnels d'une même branche séquentielle : un noeud-cible ayant traité un message dont les références émetteur sont

(N_ARB, N_BS, N_ORD, IFB = faux)

devra traiter immédiatement après, le message ayant pour référence

(N_ARB, N_BS, N_ORD + 1, IFB = faux/vrai)

Il reste maintenant à mettre en place un mécanisme permettant d'ordonner les branches séquentielles. Si un noeud séquence a traité un message ayant pour référence émetteur

(N_ARB, N_BS1, N_ORD, IFB = vrai),

et qu'il reçoit un message ayant pour référence émetteur

(N_ARB, N_BS2, N_ORD = 1, IFB = faux ou vrai),

rien ne permet d'affirmer qu'il n'y ait pas un ou plusieurs messages en provenance d'une branche *BS3* non encore reçus, mais devant être pris en compte avant.

Nous voulons qu'un noeud-séquence puisse déterminer quelle branche séquentielle il doit traiter, après avoir traité un message en provenance du dernier noeud d'une branche séquentielle. Pour cela, il suffit de respecter la condition suivante :

(C) : Tout message émis par un noeud fonctionnel en fin de branche séquentielle doit être effectivement reçu par son noeud-cible avant que tout autre message d'une autre branche séquentielle (devant être traitée ensuite) soit émis à destination de ce même noeud-séquence.

Cela sous-entend que nous allons introduire des points de blocage dans l'exploration des arborescences fonctionnelles. Nous allons donc être amenés à préciser l'algorithme d'exploration de façon à respecter la condition *(C)* énoncée précédemment.

D'après l'algorithme de découpage en branches séquentielles, un noeud en fin de branche est soit :

(1) - un noeud forme fonctionnelle dont l'étiquette appartient à l'ensemble *{/, Tree, Bu, Cond}*,

(2) - un noeud définition,

(3) - un noeud fonction-primitive n'ayant pas de successeur,

(4) - un noeud objet représenté par *Cx* n'ayant pas de successeur,

(5) - un noeud forme-fonctionnelle dont l'étiquette n'appartient pas à l'ensemble *{/, Tree, Bu, Cond}*, n'ayant pas de successeur,

(6) - un noeud *Fcond* (cf : représentation de la forme fonctionnelle *Cond* en section VI.2.3.2).

Soit n le dernier noeud d'une branche séquentielle BS, et NDC son noeud-cible (et donc, un noeud séquence). Etudions les différents cas selon le type de n.

1er cas : type(n) = f-fonct et étiquette(n) ∈ {/, Tree, Bu, Cond}

Rappel : $Rac(SA_i(n))$ désigne le noeud racine de la sous-arborescence représentant le ième paramètre de la forme fonctionnelle.

La condition *(C)* se traduit dans ce cas par :

(C1) : ∀ i, $Rac(SA_i(n))$ ne peut être activé qu'après que NDC ait reçu le message émis par n.

Remarques :

- les formes fonctionnelles *Insert* et *Tree* n'ont qu'un paramètre,
- la forme fonctionnelle *Bu* a deux paramètres mais seul le noeud désigné par $Rac(SA_i(n))$ a le même noeud-cible que le noeud n.
- la forme fonctionnelle *Cond* a quatre paramètres (dans notre représentation). Le premier s'applique à une copie de l'argument de la forme fonctionnelle et l'un des autres paramètres s'applique à l'arbre ayant pour racine NDC,

i.e. : ∃! i > 1 tel que NDC est le noeud-cible de $Rac(SA_i(n))$.

Ces remarques nous permettent d'énoncer la propriété suivante:

∀ n / type(n) = f-fonct et étiquette(n) ∈ {/, Tree, Bu, Cond},
si NDC est le noeud-cible de n alors
∃! i tel que NDC soit également le noeud-cible de $Rac(SA_i(n))$.

Afin de respecter la condition *(C1)*, lorsque le noeud n est activé, le message d'activation destiné à $Rac(SA_i(n))$ doit être créé bloqué, en attente d'un accusé de réception. Le message émis par n à NDC doit alors contenir les références du message d'activation créé bloqué. Lorsque

NDC reçoit ce message, il transmet alors au message d'activation bloqué, un accusé de réception permettant de débloquer ce message.

2ème cas : type(n) = def

Soit Arb_{def} l'arborescence secondaire représentant cette définition. $Rac(Arb_{def})$ désigne le noeud racine de Arb_{def} et si *NDC* est le noeud-cible de *n*, *NDC* est également le noeud-cible de $Rac(Arb_{def})$.

La condition *(C)* se traduit donc dans ce cas par

(C2) : $Rac(Arb_{def})$ ne peut être activé qu'après que *NDC* ait reçu le message émis par *n*.

Le même principe que précédemment peut alors être mis en oeuvre pour respecter la condition *(C2)* : lorsque le noeud *n* est activé, il crée un message d'activation bloqué à destination de $Rac(Arb_{def})$ et transmet à *NDC* un message contenant les références du message d'activation créé bloqué. A la réception de ce message, *NDC* retourne au message d'activation créé bloqué, un accusé de réception permettant de débloquer ce message.

3ème cas : (type(n) = fct-prim et succ(n) = Ø)
ou (type(n) = objet et succ(n) = Ø)
ou (type(n) = f-fonct et étiquette(n) ∈ {/, Tree, Bu, Cond}
et succ(n) = Ø)
ou étiquette(n) = Fcond

D'après l'algorithme d'exploration, lorsque *n* est activé, il doit retourner un message de fin-d'exploration à un message précédemment créé bloqué. Ce message de fin-d'exploration a pour but de débloquer le message créé bloqué et donc, de permettre la poursuite de l'exploration. Afin de respecter la condition *(C)*, il suffit de retarder l'émission de ce message jusqu'à ce que *NDC* ait reçu le message en provenance de *n*, i.e. : le message de fin d'exploration est créé bloqué en attente d'un

accusé de réception devant être transmis par NDC à la réception du message en provenance de n.

Ainsi, d'une façon générale, tout noeud fonctionnel en fin de branche séquentielle transmet à son noeud-cible ses références et les références d'un message bloqué. A la réception de ce message, le noeud-cible devra retourner un accusé de réception au message bloqué.

De cette façon, nous pouvons assurer qu'un noeud séquence ne peut pas recevoir deux messages en provenance de deux branches séquentielles différentes sans avoir reçu un message en provenance du dernier noeud de l'une des branches séquentielles. De plus, lorsqu'un noeud-liste a reçu un message en provenance du dernier noeud d'une branche séquentielle BR, s'il reçoit un message en provenance d'une autre branche BR', cela signifie que cette branche BR' doit être appliquée au noeud-cible immédiatement après la branche BR.

Rangement des messages en file d'attente

Puisqu'un noeud fonctionnel peut recevoir plusieurs messages en provenance des arborescences fonctionnelles, une file d'attente est associée à chaque noeud-séquence, permettant de ranger ces messages. Les messages sont insérés dans la file en respectant leur ordre de traitement (i.e. : leur ordre d'émission) de telle sorte qu'un message devant être traité immédiatement par un noeud-séquence soit rangé en tête de file.

Soit m un message à ranger dans la file. Ce message contient les références du noeud émetteur. Nous désignons par

- $m.N\_ARB$, le nom de l'arborescence fonctionnelle à laquelle appartient le noeud fonctionnel émetteur,
- $m.N\_BS$, le nom de sa branche séquentielle,
- $m.N\_ORD$, son numéro d'ordre dans la branche,
- $m.IFB$, son indicateur de fin de branche.

Si *m.IFB = vrai*, le message contient alors les références d'un message en attente d'un accusé de réception que nous désignerons par *m.DEST_ACK*.

Nous désignerons par *FA[i]*, le $i^{ème}$ message rangé en file d'attente. Donc, *FA[i].N_ARB* désigne le nom de l'arborescence à laquelle appartient le noeud fonctionnel émetteur du $i^{ème}$ message rangé en file d'attente. De même, nous désignerons par *FA[i].N_BS*, *FA[i].N_ORD*, *FA[i].IFB* les autres informations.

Etant données ces notations, lorsqu'un noeud séquence reçoit un message en provenance d'un noeud fonctionnel, il range ce message dans la file d'attente lui étant associée selon l'algorithme suivant :

```
I ← 1
SI FA[1] = Ø ALORS
  | {la file d'attente est vide, ranger m en tête de file}
  |
  |   FA[I] ← m
  |
SINON
  | {rechercher la place de m dans FA}
  |
  |   SI m.IFB = vrai ALORS
  |     | {tout message devant être traité après m n'a pas encore été
  |     | envoyé}
  |     |
  |     |   ranger m en fin de file
```

```
SINON
  {rechercher les messages en provenance de la même branche
  séquentielle}

    TANT-QUE    non (fin de file)
                et (FA[I].N_ARB ≠ m.N_ARB ou FA[I].N_BS ≠ m.N_BS)
                FAIRE
        {avancer dans la file}

        I ← I + 1

    FIN-TANT-QUE

    SI non (fin de file) ALORS
      {FA[I].N_ARB = m.N_ARB et FA[I].N_BS = m.N_BS}

        TANT-QUE    non (fin de file) et
                    FA[I].N_ARB = m.N_ARB et
                    FA[I].N_BS = M.N_BS et
                    FA[I].N_ORD < m.N_ORD FAIRE

                I ← I + 1

        FIN-TANT-QUE

    FIN-SI
```

```
        SI fin de file ALORS
          {ranger le message en fin de file}
            FA[I] ← m
        SINON
          {FA[I].N_ORD > m.N_ORD > FA[I-1].N_ORD}
            insérer m entre FA[I] et FA[I-1]
        FIN-SI
    FIN-SI
FIN-SI
```

Avec cet algorithme de rangement dans la file, un noeud-séquence peut donc traiter le message en tête de file si l'une de conditions suivantes est vérifiée :

(1) il n'a pas encore traité de message et *FA[1].N_ORD = 1*,

(2) le dernier message traité par le noeud séquence a les références
(N_ARB = a, N_BS = b, N_ORD = o, IFB = faux)
et le message en tête de file a les références
(N_ARB = a, N_BS = b, N_ORD = o + 1, IFB = faux ou vrai),

(3) le dernier message traité par le noeud-séquence a les références
(N_ARB = a, N_BS = b, N_ORD = o, IFB = vrai)
et le message en tête de file a les référence
(N_ARB = a', N_BS = b', N_ORD = 1, IFB = vrai ou faux)
avec soit a' ≠ a et b' ≠ b,
soit a' = a et b' ≠ b,
soit a'≠ a et b' ≠ b.

Remarque :

Les points de blocage introduits dans l'exploration, permettant de mettre en oeuvre le mécanisme d'ordonnancement de messages ne freinent pratiquement pas l'exécution. Seuls les messages émis par les noeuds fonctionnels en fin de branche séquentielle nécessitent un accusé de réception et les accusés de réception sont retournés à la réception d'un message et non après traitement de ce message. Ainsi, si le message nécessitant un accusé de réception est rangé en tête de file, la transmission de l'accusé de réception aura lieu simultanément avec le traitement de ce message par le noeud séquence et s'il n'est pas rangé en tête de file, l'accusé de réception sera retourné avant que tous les messages précédents dans la file d'attente ne soient traités. L'exécution est donc freinée uniquement dans le cas où le message est rangé en tête de file d'attente (i.e. : c'est le prochain message que le noeud séquence traitera) et que le traitement de ce message nécessite moins de temps que le retour de l'accusé de réception et l'envoi du message suivant au noeud-séquence. Or, on peut espérer que d'une façon générale, les temps de communication soient inférieurs aux temps de traitement !

Le mécanisme mis en oeuvre dans cette section permet d'ordonner les messages à destination d'un même noeud séquence. Dans la section suivante, nous abordons le problème de l'ordonnancement des messages à destination de noeuds-séquence différents.

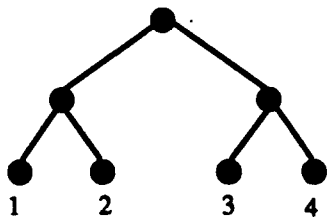
VI.3.3 : L'ordonnancement de messages à destination de noeuds séquences différents

Rappel du problème :

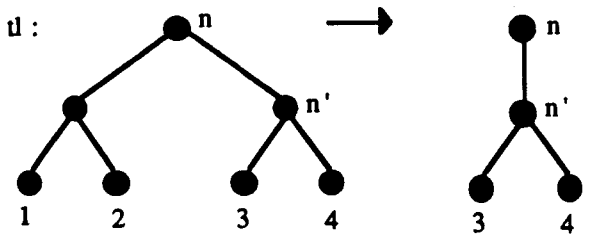
Soit n un noeud séquence ayant pour fils un noeud séquence n'. Soit m un message en tête de la file d'attente de n tel que m soit le prochain message que n doit traiter. Soit m' un message en tête de file d'attente de n' tel que m' soit le prochain message que n' doit traiter.

Le problème est de savoir si m doit être appliqué après ou avant l'application de m' à n', s'ils ne peuvent pas être appliqués simultanément.

Le message m peut ne pas modifier le sous-arbre de racine n'. Par exemple, soit n le noeud racine de l'arbre représenté ci-dessous et n' son deuxième fils.



Si m est un message contenant la fonction primitive $Tail$, l'application de la règle de réécriture correspondante produit :



Ainsi, quel que soit le message m' à destination de n', m' peut être appliqué à n' avant, après, ou simultanément. Aucun ordonnancement n'est nécessaire.

Dans le cas où le message m modifie le sous-arbre de racine n', on peut montrer également qu'aucun mécanisme d'ordonnancement particulier n'est nécessaire : le message m' doit obligatoirement être appliqué à l'arbre de racine n' avant que le message m ne soit appliqué à l'arbre de racine n.

En effet, nous allons démontrer que :

(P1) le message m' aura obligatoirement été émis avant le message m.

Soient f le noeud fonctionnel émetteur du message m,
r un noeud fonctionnel tel que $f \in CS(r)$,
g le noeud fonctionnel émetteur de m'.

La propriété (P1) s'écrit donc

(P2) le noeud fonctionnel *g* a été activé avant le noeud fonctionnel *f*

Démonstration :

On peut tout d'abord remarquer que $g \notin CS(r)$, sinon, *g* et *f* auraient le même noeud cible. C'est donc que *g* appartient à une sous-arborescence de *Arb(r)* ou à une arborescence secondaire explorée lors de l'exploration de *Arb(r)*.

1er cas : *g* appartient à une sous-arborescence de *Arb(r)*.

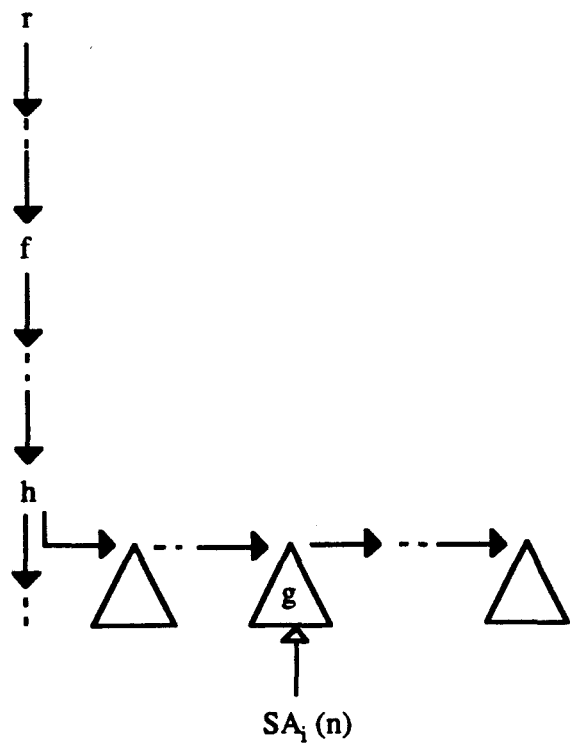
Dans ce cas,

$\exists$ h $\in$ Arb(r) et i $\geq$ 1 tels que type(h) = f-fonct et g $\in$ SA_i(h).

Considérons tout d'abord le cas où le noeud fonctionnel *h* appartient à *CS(r)*. La propriété (P2) s'écrit dans ce cas

(P3) le noeud fonctionnel *h* n'est pas le successeur de *f*.

Supposons que *h* soit un successeur de *f*. L'arborescence *Arb(r)* peut alors être représentée par



Tous les noeuds appartenant à $CS(r)$ ont pour noeud-cible le noeud séquence n. Lorsque h est activé, il envoie un message à son noeud cible. La règle de réécriture correspondant à h doit alors être appliquée à l'arbre ayant comme racine le noeud séquence n. L'étiquette h étant un symbole de forme fonctionnelle, cette règle va permettre d'effectuer éventuellement une transformation de l'arbre et surtout, de déterminer le noeud-cible des paramètres de la forme fonctionnelle.

Or, tout message d'activation d'un noeud fonctionnel doit impérativement contenir le noeud-cible du noeud fonctionnel destinataire. Cela signifie qu'en pratique, un message d'activation à destination d'une arborescence paramètre ne pourra être envoyé que lorsque le message en provenance du noeud forme fonctionnelle correspondant aura été pris en compte par son noeud cible.

Donc, dans notre exemple, lorsque le noeud g a été activé, cela signifie que le noeud séquence n a déjà pris en compte le message en provenance du noeud h. D'après le mécanisme d'ordonnancement précédemment décrit, le noeud n traite les messages dans l'ordre de leur émisssion, et donc, le message m' émis par f aura déjà été consommé. Il ne peut donc plus être dans la file d'attente, ce qui est en contradiction avec nos hypothèses. Donc, h ne peut pas être un successeur de f.

On a considéré ci-dessus que h appartenait à $CS(r)$. On peut étendre le raisonnement afin de montrer que :

(P4) : $h \in Arb(r)$ et $h \notin CS(r) \Rightarrow$ h est activé avant f.

En effet, de la même façon que précédemment,

$h \in Arb(r)$ et $h \notin CS(r) \Rightarrow$
$\exists\ k \in CS(r)$ et $i \geq 1$ tels que type(h) = f-fonct et $h \in SA_i(k)$.

On a montré que si le noeud g été activé, cela signifiait que le message émis par le noeud h avait été pris en compte par son noeud-cible et donc, que le noeud h avait été activé. Donc, de la même façon, si

le noeud h a été activé, c'est que le message émis par le noeud k a été pris en compte par son noeud-cible. Comme k appartient à $CS(r)$, on se ramène au cas précédent.

La propriété (P4) est donc également vérifiée.

2ème cas : le noeud g appartient à une arborescence secondaire.

Soit Arb_{def} cette arborescence. Si les noeuds f et g n'ont pas le même noeud cible, alors g n'appartient pas à $CSG(Arb_{def})$ et donc,

$\exists$ h $\in$ CSG(Arb_{def}) et i$\geq$ 1 tels que type(h) = f-fonct et g $\in$ SA_i(h)

On se ramène là encore au cas précédent et donc, là encore, si g est activé, c'est que h a été activé, et si h est activé après f alors le message m émis par f a été consommé et donc m n'est plus dans la file d'attente. Ceci étant en contradiction avec nos hypothèses, on en déduit que h a été activé avant f et donc, que g a été activé avant f, et donc, que m' doit être traité avant m. C.Q.F.D.

Remarque : Dans notre exemple initial, nous avons considéré que le noeud n' était un fils du noeud n. Le raisonnement est également valable si n' est un noeud quelconque dans l'arbre de racine n.

VIII : VALIDATION

VIII : VALIDATION

Cette deuxième partie décrit le modèle d'exécution parallèle que nous avons défini pour le langage fonctionnel FP. En réalité, la description est faite de telle manière que le modèle se trouve automatiquement validé, la validation découlant de la façon dont le modèle est construit.

Dans le chapitre VI, nous avons présenté l'exploration des arborescences fonctionnelles. Cette exploration, décrite par un graphe de transition d'état, est construite de telle façon que les noeuds fonctionnels sont activés en respectant l'ordre d'exécution des fonctions composant le programme.

Le chapitre VII a montré comment ce graphe de transition d'état pouvait être traduit en termes d'émission et réception de messages. Dans le même chapitre, nous décrivons l'envoi de messages à la séquence argument suite à l'activation des noeuds fonctionnels. Ces noeuds étant activés en respectant l'ordre d'exécution des fonctions, les messages émis à destination de la séquence argument sont donc émis dans l'ordre où ils doivent être traités. Or, nous avons défini un mécanisme d'ordonnancement permettant à la séquence argument de retrouver l'ordre d'émission des messages. Il ne reste plus qu'à montrer que le traitement de ces messages respecte le sémantique du langage FP : les règles de réécriture devant être appliquées lors du traitement de ces messages ont été définies dans ce but.

Il est clair que les règles de réécriture correspondant au traitement d'un message émis par un noeud *fonction-primitive* sont directement issues de la définition de la fonction primitive correspondante en FP, moyennant la transformation d'une séquence en un arbre. La sémantique d'application d'une fonction primitive à une séquence argument est donc bien respectée.

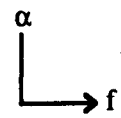
La sémantique d'une composition de fonctions $(f \circ g)$ est donnée par la règle "appliquer g puis appliquer f". L'exploration des arborescences fonctionnelles et le mécanisme d'ordonnancement de messages

permettent de respecter cette règle. Quant à la sémantique des formes fonctionnelles, elle est également implicitement respectée par les règles de réécriture.

Prenons l'exemple de la forme fonctionnelle α. Elle est définie en FP par :

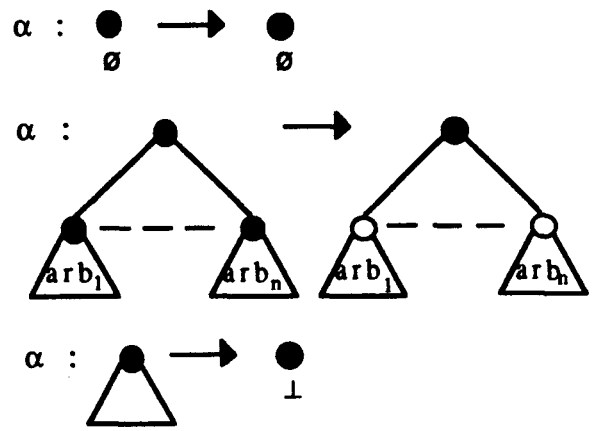
$$\alpha\, f : x \equiv x = \emptyset \rightarrow \emptyset\ ;$$
$$x = \langle x_1, \ldots, x_n \rangle \rightarrow \langle f : x_1, \ldots, f : x_n \rangle$$
$$\bot.$$

Cette forme fonctionnelle est représentée par l'arborescence fonctionnelle suivante :



et l'argument x par l'arbre

Lors de l'exploration, un message émis par le noeud fonctionnel α sera émis à destination du noeud racine de la séquence argument et les règles de réécriture correspondant au traitement de ce message sont les suivantes :



La première et la dernière règle traduisent directement les premiers et derniers cas de la définition en FP de cette forme fonctionnelle. Quant à la deuxième règle, elle permet de "dire" que le paramètre de l'*Apply-to-all* doit s'appliquer à chacun des sous-arbres de la séquence argument : la sémantique de l'*Apply-to-all* est ainsi respectée.

Nous avons pris ici l'exemple de la forme fonctionnelle α mais nous pourrions montrer de la même façon que quelle que soit la forme fonctionnelle, l'application de la règle de réécriture correspondante suivie de l'application des règles de réécriture correspondant aux paramètres de la forme fonctionnelle permettent de "reconstruire" la définition FP des formes fonctionnelles. Nous pouvons ainsi affirmer que la sémantique du modèle respecte la sémantique de FP.

TROISIEME PARTIE :

REPRESENTATION DU MODELE SOUS LA FORME D'UN RESEAU DE PROCESSUS ET EXEMPLE D'EXECUTION D'UN PROGRAMME FP

TROISIEME PARTIE :
REPRESENTATION DU MODELE
SOUS LA FORME D'UN RESEAU DE PROCESSUS
ET EXEMPLE D'EXECUTION D'UN PROGRAMME FP

Dans la deuxième partie, nous avons décrit :

- l'algorithme d'exploration par messages, générant l'émission de messages à destination de la séquence argument,
- les règles de réécriture décrivant la sémantique des traitements effectués sur la séquence argument,
- l'ordonnancement des messages à destination de la séquence argument.

Le but de cette troisième partie est de faire le lien entre ces différents points afin de pouvoir décrire concrètement et précisément le déroulement de l'exécution d'un programme FP selon le modèle défini. Nous serons donc amenés dans cette troisième partie, à faire des choix, aussi bien pour la représentation des objets que pour la façon de mener les traitements.

Les choix seront effectués, non pas d'après des critères d'efficacité mais plutôt selon des critères de simplicité et de lisibilité.

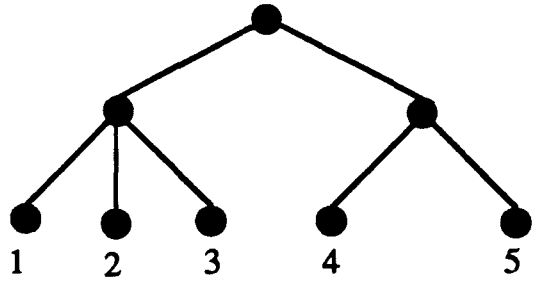
Dans un premier temps, nous présenterons donc les choix effectués. Dans un deuxième temps, nous préciserons les objets et les messages manipulés. Puis, nous décrirons un schéma fonctionnel permettant de représenter le modèle sous la forme d'un réseau de processus communiquant par messages. La description de ces processus nous permettra de terminer en illustrant le "fonctionnement" du modèle : nous prendrons un exemple de programme FP et nous montrerons comment il peut être appliqué à une séquence argument selon les traitements décrits dans cette troisième partie.

IX : LES CHOIX EFFECTUES

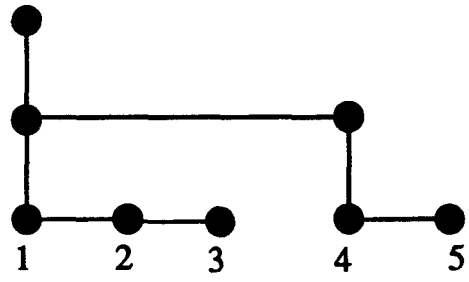
IX : LES CHOIX EFFECTUES

IX.1 : choix d'une représentation de la séquence argument

Nous choisissons de représenter la séquence argument par un arbre binaire, chaque noeud de l'arbre étant un noeud séquence, ce qui permet de manipuler les informations selon un format fixe. Ainsi, la séquence *<<1, 2, 3>, <4, 5>>*, représentée dans les parties précédentes par :



sera représentée dans la suite par



C'est-à-dire que chaque noeud de l'arbre connait son fils et son frère dans l'arbre.

IX.2 : les réductions

Dans la section VII.2.3 nous avons vu que tout noeud fonctionnel étiqueté par un symbole *f* émet à destination de son noeud cible, un message de type *f*. Le traitement de ce message consiste à appliquer la règle de réduction correspondante, les règles de réductions ont été définies en section VII.2.3.2. La représentation de la séquence argument a une influence sur la façon dont les règles de réduction sont mises en oeuvre. Il est tout d'abord nécessaire de modifier les règles de façon à ce qu'elles puissent être appliquées à des arbres binaires. Pour cela, il suffit simplement de transformer dans les règles, les arbres n-aires en arbres binaires. Il faut ensuite déterminer une stratégie permettant de mettre en oeuvre ces réductions. Nous ne détaillerons pas la façon de mettre en oeuvre toutes les règles de réduction. Nous donnerons juste le minimum d'informations qui nous permettra par la suite d'illustrer le fonctionnement du modèle sur quelques exemples.

IX.2.1 : contraintes de réduction

Nous considérons qu'une stratégie a été définie telle que, lors d'une réduction, les noeuds séquence à modifier passent par une succession d'états que nous ne détaillerons pas. Nous considèrerons seulement qu'initialement, tous les noeuds de la séquence argument sont dans l'état *réduit*. Le traitement d'un message par un noeud séquence fait passer le noeud séquence dans l'état *en-cours-de-réduction* et le noeud ne retrouve l'état *réduit* que lorsque les informations ne seront plus modifiées par le traitement de ce message et, si le noeud séquence a un fils (selon la représentation donnée ci-dessus), ce fils existe effectivement. De plus, si le traitement du message donne lieu à l'envoi d'un message au fils, ce fils doit avoir effectivement reçu le message (ce qui peut être connu par l'envoi d'un accusé de réception). Etant donnée cette règle, une condition nécessaire pour qu'un noeud séquence puisse traiter un message est qu'il se trouve dans l'état *réduit*.

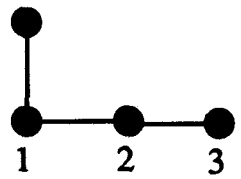
IX.2.2 : les messages émis par un noeud fonction primitive

Appelons ce type de message, *message fp*. Les réductions engendrées par un *message fp* peuvent être implantées par des mécanismes de réduction classiques. Il faudrait ainsi, pour chaque fonction primitive, détailler la stratégie à mettre en oeuvre pour appliquer la règle de réduction correspondante. Ceci ne présente pas de difficultés particulières et nous ne préciserons pas ici ces mécanismes classiques.

Nous pouvons simplement remarquer que la contrainte de réduction énoncée dans la section précédente permet de traiter un message sans devoir obligatoirement attendre la fin du traitement du message précédent.

Exemple :

Soit la séquence



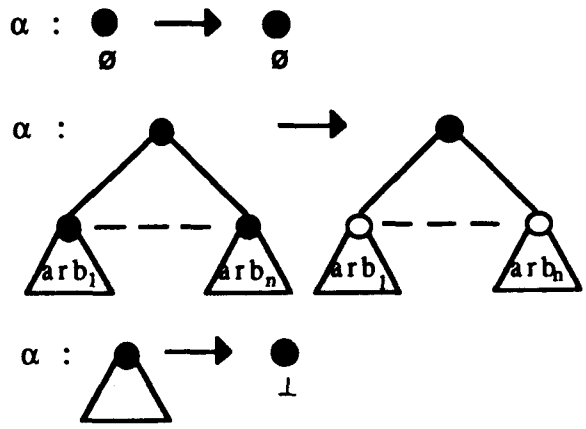
Supposons que le noeud racine de cette séquence doive traiter un message *"Reverse"* puis un message *"1"* (i.e. : un message émis par un noeud étiqueté par la fonction *Sélecteur 1*).

Le début du traitement de la fonction *Reverse* fait passer le noeud racine dans l'état *en-cours-d'exploration*. L'application de la fonction *Reverse* ne modifie pas la structure de la séquence argument. Si on imagine une stratégie permettant d'appliquer cette règle en ne faisant que des permutations de valeurs (et non des permutations de noeuds), l'application de la fonction *Reverse* n'entrainera aucune modification du noeud racine de la séquence argument. Ainsi, dès que les messages émis aux fils seront effectivement reçus, le noeud racine pourra alors retrouver l'état *réduit* et le message *"1"* pourra ainsi être pris en compte et son traitement pourra débuter.

IX.2.3 : les messages émis par un noeud forme fonctionnelle

De même, nous ne détaillerons pas les différentes stratégies à mettre en oeuvre pour chaque forme fonctionnelle. Nous illustrerons cet aspect en prenant l'exemple de la forme fonctionnelle α.

Lorsqu'un noeud forme fonctionnelle, étiqueté par le symbole α est activé, il envoie un message (que nous appellerons message α) au noeud séquence, message dont les règles de réduction sont



Le traitement d'un message α par un noeud séquence doit donc conduire à l'application de l'une de ces trois règles.

La deuxième règle s'applique lorsque le noeud racine de la séquence argument a au moins un fils. La mise en oeuvre de cette règle doit donc permettre l'application du paramètre à chacun des noeuds fils du noeud racine de l'arbre.

Pour cela, nous choisissons la stratégie suivante : les messages émis par les noeuds fonctionnels appartenant à la sous-arborescence paramètre ($SA_1(\alpha)$) seront émis à destination de leur noeud cible avec la mention *"faire-suivre"*. C'est-à-dire que lorsque le noeud cible recevra le message, il le transmettra à son frère qui fera de même. Un pipeline sera ainsi créé qui permettra effectivement d'appliquer la fonction paramètre en parallèle.

Comment mettre en oeuvre cette stratégie ?

1ere solution :

Le noeud fonctionnel α demande à son noeud cible l'identification de son fils, attend la réponse, et envoie un message d'activation à $rac(SA_1(\alpha))$. Le message d'activation envoyé comporte la mention *"générer des messages à faire suivre"* : c'est-à-dire que le message émis par $rac(SA_1(\alpha))$ à son noeud cible comportera la mention *"faire suivre"*.

2ème solution :

Le noeud fonctionnel α crée un message d'activation bloqué (avec l'information noeud cible manquante) à destination de $rac(SA_1(\alpha))$ et demande à son noeud cible l'identification de son fils, cette information étant directement retournée au message en attente. Cette deuxième solution a l'avantage de "libérer" le noeud fonctionnel α qui peut alors traiter d'autres messages.

3ème solution :

Le noeud fonctionnel α envoie à son noeud cible un message comportant toutes les informations nécessaires pour que le noeud cible puisse ensuite envoyer un message d'activation à $rac(SA_1(\alpha))$. Le message d'activation n'est donc plus émis par un noeud fonctionnel mais par un noeud séquence. Cette solution présente l'inconvénient d'allonger la taille du message α émis au noeud cible. En contre partie, le message d'activation est généré directement, donc plus rapidement. C'est donc cette troisième solution que nous retiendrons dans la suite. Avec cette solution, un noeud séquence traitant un message α doit donc émettre un message d'activation à destination de $rac(SA_1(\alpha))$, ce message comportant la mention *"générer des messages à faire suivre"* permettant de signaler au destinataire que le message émis à destination de son noeud cible devra comporter la mention *"faire suivre"*.

Tous les noeuds fonctionnels appartenant à $SA_1(\alpha)$ doivent émettre à destination de leur noeud cible un message comportant la mention *"faire suivre"*. Cela signifie donc que si le traitement d'un message d'activation comportant la mention *"générer des messages à faire suivre"*

génère d'autres messages d'activation, ces messages devront également comporter cette mention.

Le fait d'introduire des messages à *faire suivre* a des répercussions sur le mécanisme d'ordonnancement de messages. En effet, prenons l'exemple d'une fonction $F = \alpha f$ où f est une fonction primitive.

Dans une arborescence fonctionnelle, le noeud f serait alors le dernier noeud d'une branche séquentielle et le dernier noeud d'un chemin séquentiel. D'après l'algorithme d'exploration et le mécanisme d'ordonnancement, lorsque le noeud f est activé, il doit créer un message de fin d'exploration bloqué, en attente d'un accusé de réception, et émettre un message à destination de son noeud cible. Le noeud cible recevant un message en provenance du dernier noeud d'une branche séquentielle devra retourner un accusé de réception au message de fin d'exploration bloqué. Or, d'après cette solution, si le noeud cible du noeud fonctionnel α a n fils, le message f sera transmis aux n fils et donc, n messages accusé de réception seront retournés. On décide donc que lorsqu'un noeud cible reçoit en provenance d'un noeud fonctionnel, un message nécessitant un accusé de réception, si ce message doit être transmis aux frères, seul le noeud séquence en "fin de chaîne" transmet un accusé de réception.

Il reste encore à préciser cette solution dans le cas particulier où la sous-arborescence paramètre d'un noeud fonctionnel α contient une sous-arborescence ayant pour racine un autre noeud fonctionnel α.

Prenons l'exemple de la fonction $F = \alpha_1 \alpha_2 f$ pour illustrer ce problème (les α sont ici indicés de façon à pouvoir les nommer sans ambigüité). Soit n le noeud cible du noeud α_1 et n_1, ..., n_m les fils du noeud n_2. La figure IX.1 illustre les échanges de messages décrits ci-dessous.

Lorsque le noeud α_1 est activé, il transmet à n un message α_1. Le traitement de ce message α_1 par n va générer l'émission d'un message d'activation à destination du noeud fonctionnel α_2 comportant la mention *"générer des messages à faire suivre"*. L'activation du noeud α_2 va donc générer l'émission d'un message α_2 à faire suivre à destination

de n_1. Ce message α_2 sera donc transmis à l'ensemble des frères de n_1, qui retourneront chacun un message d'activation à destination de $rac(SA_1(\alpha_2))$, c'est-à-dire le noeud fonctionnel f dans la figure, générant une exploration multiple de $SA_1(\alpha_2)$. Ceci ne pose aucun problème, à condition de faire en sorte que le message créé bloqué par le noeud fonctionnel α_2 suite à son activation, soit créé bloqué en attente de m messages de fin d'exploration (où m est le nombre de fils du noeud n). Cela necéssite donc que le message d'activation émis par n à destination de α_2 contienne le nombre de fils de n (c'est-à-dire le nombre de sous-arbres auquel sera appliqué le paramètre de α_1).

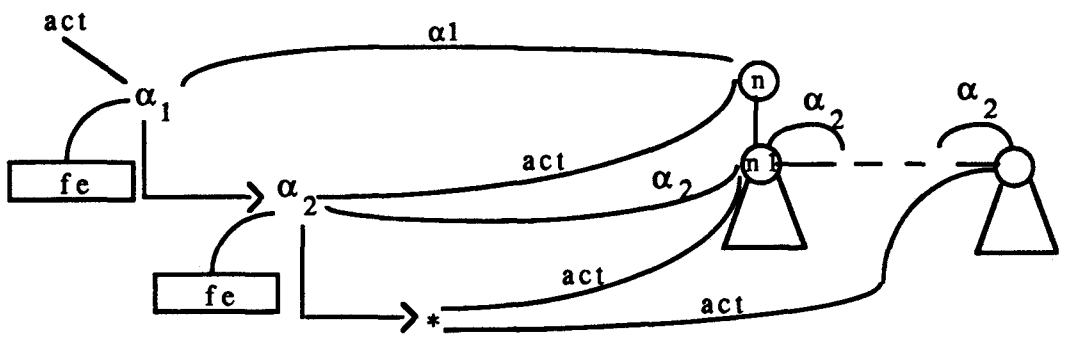


Figure IX.1 : illustration des échanges de messages

Tous les messages d'activation générés lors de l'exploration contiendront donc une information notée *NB-SS-ARB*, indiquant au destinataire le nombre de sous-arbres auquel il s'applique. Si un noeud fonctionnel α reçoit un message d'activation contenant l'information *NB-SS-ARB* = m, cela signifie donc que $SA_1(\alpha)$ sera explorée m fois et donc que le message créé bloqué par le noeud α devra attendre m messages de fin d'exploration.

Dans le cas où soit la première règle de réduction, soit la troisième règle de réduction doit être appliquée, le traitement du message α ne doit pas conduire à l'exploration de $SA_1(\alpha)$. Ainsi, si l'une de ces règles est appliquée, le noeud séquence retournera au message créé bloqué par le noeud fonctionnel α, un message particulier, ayant pour but de débloquer le message qui avait été créé bloqué en attente de messages de fin-d'exploration.

X : LES OBJETS ET LES MESSAGES

X : LES OBJETS ET LES MESSAGES

X.1 : représentation d'un noeud fonctionnel

Nous représenterons un noeud fonctionnel *nf* par l'ensemble des informations suivantes :

- *NOM* : le nom du noeud fonctionnel *nf*, cette information est constituée des informations suivantes :
  - *N-ARB* : le nom de l'arborescence à laquelle appartient *nf*,
  - *N-BS* : le numéro de la branche séquentielle à laquelle appartient *nf*,
  - *N-ORD* : le numéro d'ordre de *nf* dans la branche,
  - *IFB* : l'indicateur de fin de branche positionné à vrai ou faux selon que *nf* est ou n'est pas le dernier noeud de la branche,

- *TYPE* : cette information peut prendre les valeurs (fct-prim, f-fonct, obj, def). Elle indique donc si le noeud *nf* est un noeud fonction-primitive, forme-fonctionnelle, objet ou définition,

- *ETIQ* : l'étiquette de *nf*, c'est-à-dire un symbole de forme fonctionnelle, ou un symbole de fonction primitive, ou les références à un objet, ou le symbole Fcond, et si *nf* est un noeud définition, l'information ETIQ contient alors le nom du noeud racine de l'arborescence secondaire correspondante,

- *SUCC* : le nom du successeur de *nf* s'il existe, sinon, $\emptyset$,

- *PARAM* : le nom de $rac(SA_1(nf))$ si $TYPE(nf) = f\text{-}fonct$ sinon $\emptyset$,

- *SUIV* : si $nf = rac(SA_i(r))$ où r est un noeud forme fonctionnelle, alors *SUIV* contient le nom du noeud fonctionnel désigné par $rac(SA_{i+1}(r))$ s'il existe, sinon, *SUIV* contient $\emptyset$,

- *NUM-ACTIV* : si *nf* est le noeud racine d'une arborescence, alors *NUM-ACTIV* contient le numéro d'activation de l'arborescence qui sera incrémenté à chaque nouvelle activation, sinon, *NUM-ACTIV* contient $\emptyset$.

Exemple :

Soit $P = \alpha(+ \circ \alpha*)$. La figure IX.2.a montre la représentation de P sous la forme d'une arborescence fonctionnelle et la figure IX.2.b montre les noeuds fonctionnels composant cette arborescence. Les α sont indicés de façon à pouvoir les nommer sans ambiguïté et nous ne détaillons pas les informations constituant le nom d'un noeud fonctionnel. Nous donnons par exemple au noeud α_1 le nom α_1.

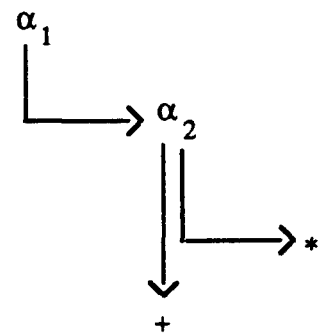


Figure IX.2.a : l'arborescence fonctionnelle

NOM	TYPE	ETIQ	SUCC	PARAM	SUIV	NUM-ACTIV
α_1	f-fonct	α	$\emptyset$	α_2	$\emptyset$	0
α2	f-fonct	α	+	*	$\emptyset$	$\emptyset$
*	fp	*	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
+	fp	+	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure IX.2.b : les noeuds fonctionnels composant l'arborescence fonctionnelle

Figure IX.2 : représentation du programme P

X.2 : représentation d'un noeud séquence

Nous représenterons un noeud séquence *ns* par l'ensemble des informations suivantes :

- *NOM* : le nom de *ns* (rappel : si *ns* est le destinataire d'un message émis par un noeud fonctionnel, l'application de la règle de réduction correspondante ne doit pas modifier l'information *NOM*, cf : section VII.2.2),

- *ETAT* : l'état du noeud *ns* qui vaut initialement *réduit* et qui respecte la contrainte donnée en section VIII.2.1,

- *VAL* : la valeur du noeud fonctionnel. *VAL* vaut *Nil* si le noeud *ns* n'est pas un noeud feuille et sinon, *VAL* contient une valeur atomique,

- *FILS* : le nom du fils du noeud *ns* (dans l'arbre binaire) s'il en a un, sinon, $\emptyset$,

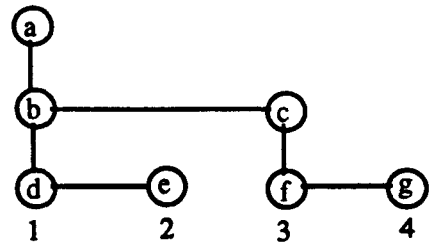
- *FRERE* : le nom du frère du noeud *ns* (dans l'arbre binaire), s'il en a un, sinon, $\emptyset$,

- *NB-FILS* : le nombre de fils du noeud *ns* (dans l'arbre n-aire),

- *DER-MSG-TRAITE* : les références du dernier message émis par un noeud fonctionnel traité par le noeud *ns*. Ceci sert à mettre en oeuvre le mécanisme d'ordonnancement décrit en section VII.3. A la création du noeud, cette information vaut $\emptyset$.

Exemple :

La séquence *<<1, 2>, <3, 4>>* sera initialement représentée par la séquence argument



et donc par l'ensemble des noeuds suivants :

NOM	ETAT	VAL	FILS	FRERE	NB-FILS	DER-MSG-TRAITE
a	réduit	nil	b	Ø	2	Ø
b	réduit	nil	d	c	2	Ø
c	réduit	nil	f	Ø	2	Ø
d	réduit	1	Ø	e	0	Ø
e	réduit	2	Ø	Ø	0	Ø
f	réduit	3	Ø	g	0	Ø
g	réduit	4	Ø	Ø	0	Ø

X.3 : les messages

Nous récapitulons dans cette section, différents messages nécessaires lors de l'exécution et nous donnons les informations devant figurer dans ces messages afin de pouvoir mettre en oeuvre l'exploration des arborescences fonctionnelles, l'ordonnancement des messages à destination de la séquence argument et les stratégies de réduction choisies dans la section IX.2.

Nous pouvons classer ces messages en quatre catégories :

- les messages émis par un noeud fonctionnel à destination d'un autre noeud fonctionnel, que nous appellerons les messages *NF-NF*,

- les messages émis par un noeud fonctionnel à destination d'un noeud séquence, que nous appellerons les messages *NF-NS*,

- les messages émis par un noeud séquence à destination d'un noeud fonctionnel que nous appellerons les messages *NS-NF*,

- les messages émis par un noeud séquence à destination d'un autre noeud séquence, que nous appellerons les messages *NS-NS*.

X.3.1 : les messages *NF-NF*

Les messages *NF-NF* sont ceux nécessaires à l'exploration des arborescences fonctionnelles, c'est-à-dire les messages d'activation et les messages de fin d'exploration.

X.3.1.1 : les messages d'activation

Un message d'activation contiendra les informations suivantes :

- *TYPE* : le type du message. Cette information vaut *"ACT"* pour tout message d'activation,

- *REF* : les références du message. Cette information sera utile si le message est créé bloqué, en attente de messages de fin d'exploration par exemple. *REF* sera alors le destinataire du message de fin d'exploration attendu. Si le message n'est pas créé bloqué, cette information, inutile, vaudra $\emptyset$,

- *DEST-ACT* : le nom du noeud fonctionnel destinataire du message d'activation,

- *NUM-ACTIV* : le numéro d'activation de l'arborescence,

- *CIBLE* : le nom du noeud séquence qui est le noeud cible de DEST-ACT (i.e. : le noeud cible du noeud fonctionnel destinataire du message d'activation),

- *NB-ACK-ATT* : le nombre d'accusés de réception attendus par le message d'activation. Cette information, définie dans la section relative à l'ordonnancement de messages (cf section VII.3) peut donc valoir 0 ou 1. Si elle vaut 1, le message est bloqué en attente d'un accusé de réception émis par le noeud cible de l'émetteur du message d'activation,

- *NB-FE-ATT* : le nombre de messages de fin d'exploration attendus par le message d'activation. Cette information a été définie dans la section relative à l'exploration des arborescences par messages (cf section VII.1) et permet de respecter l'ordre d'activation des noeuds fonctionnels (remarque : un message d'activation dont l'information *NB-ACK-ATT* ou *NB-FE-ATT* est différente de 0 est un message d'activation bloqué).

- *GEN-FAIRE-SUIVRE* : cette information, définie en section IX.2.3 vaut vrai ou faux selon que *DEST-ACT* doit ou non générer à destination de son noeud cible des messages à faire-suivre,

- *NB-SS-ARB* : le nombre de sous-arbres auquel le noeud fonctionnel *DEST-ACT* s'applique (cf section IX.2.3),

- *DEST-FE* : le destinataire du messages de fin d'exploration devant être émis par *DEST-ACT* s'il est en fin de chemin séquentiel (cf section VII.1).

X.3.1.2 : les messages de fin d'exploration

Un message de fin d'exploration contiendra les informations suivantes :

- *TYPE* : le type du message qui vaut *FE* pour tout message de fin d'exploration,

- *REF* : les références du message. De même que pour les messages d'activation, cette information vaudra $\emptyset$ si le message n'est pas créé bloqué,

- *DEST-FE* : les références du message bloqué, destinataire de ce message de fin d'exploration,

- *NB-ACK-ATT* : le nombre d'accusés de réception attendus par le message avant qu'il ne puisse effectivement être émis,

- *NB-FE-ATT* : le nombre de messages de fin d'exploration attendus par ce message.

X.3.2 : les messages *NF-NS*

Les messages *NF-NS* sont les messages émis par les noeuds fonctionnels, suite à leur activation, à destination de leur noeud cible. Dans cette partie, nous ne nous intéresserons qu'aux messages émis par les noeuds fonctionnels de type

- fonction-primitive (messages *fp*),
- forme-fonctionnelle (messages *f-fonct*),
- définition (messages *def*).

Un message *NF-NS* contiendra les informations suivantes :

- *TYPE* : le type du message qui vaut soit *fp*, soit *f-fonct*, soit *def*, selon que le noeud fonctionnel émetteur est un noeud fonction-primitive, forme-fonctionnelle, ou définition,

- *DEST* : le nom du noeud séquence destinataire du message,

- *ETIQ* : l'étiquette du noeud fonctionnel émetteur du message,

- *REF-EMET* : le nom du noeud fonctionnel émetteur du message et son numéro d'activation (c'est-à-dire soit le numéro d'activation présent dans le message d'activation reçu par le noeud fonctionnel émetteur si celui-ci n'est pas la racine d'une arborescence, soit le numéro d'activation contenu dans le noeud fonctionnel si celui-ci est la racine d'une arborescence),

- *DEST-ACK* : les références du message auquel doit être envoyé un accusé de réception si le noeud fonctionnel émetteur du message est en fin de branche séquentielle (si aucun accusé de réception ne doit être émis, *DEST-ACK* vaut $\emptyset$),

- *FAIRE-SUIVRE* vaut vrai ou faux selon que le noeud séquence doit ou non faire suivre le message à son frère (rappel : si *FAIRE-SUIVRE* vaut vrai et qu'un accusé de réception doit être retourné à un message bloqué, seul le noeud dont l'information *FRERE* est égale à $\emptyset$ renvoit un accusé de réception),

- *DEST-ACT* : les références d'un noeud fonctionnel auquel le noeud séquence doit renvoyer un message d'activation (dans le cas d'un message α par exemple). Si aucun message d'activation ne doit être émis par le noeud séquence destinataire, *DEST-ACT* vaut $\emptyset$.

X.3.3 : les messages *NS-NF*

Un message *NS-NF* est soit

- un message accusé de réception dans le cas où le noeud séquence a reçu un message émis par un noeud fonctionnel en fin de branche séquentielle,

- un message d'activation dans le cas, par exemple, où le noeud séquence a reçu un message émis par un noeud forme fonctionnelle étiqueté par le symbole α.

Les messages d'activation ont déjà été décrits dans la section X.3.1.1. Quant aux messages accusé de réception, ils contiendront les informations suivantes :

- *TYPE* qui vaut *ACK* pour tout message accusé de réception,

- *DEST-ACK* : les références du message bloqué, destinataire de l'accusé de réception.

X.3.4 : les messages *NS-NS*

Ce sont les messages permettant de mettre en oeuvre les règles de réduction, tels que des messages de copie, de création, de destruction de noeuds, etc... Le contenu de ces messages dépend des stratégies de réduction mises en oeuvre, nous ne les détaillerons donc pas.

XI : SCHEMA FONCTIONNEL DU MODELE

XI : SCHEMA FONCTIONNEL DU MODELE

XI.1 : schéma général

Le modèle peut, à un premier niveau, être vu comme le montre la figure XI.1 :

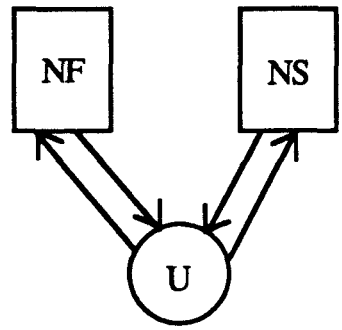


Figure XI.1 : le modèle au premier niveau

Une unité *U* permet d'appliquer, selon le modèle, un programme FP représenté par un ensemble de noeuds fonctionnels *NF* à une séquence argument representée par un ensemble de noeuds *NS*.

Cette unité doit donc permettre de mettre en oeuvre d'une part l'exploration des arborescences fonctionnelles et d'autre part, les mécanismes de réduction.

A un deuxième niveau, on peut alors diviser *U* en deux sous-unités *UF* et *US*, *UF* étant chargée de l'exploration et *US* étant chargée des mécanismes de réduction. Les deux unités *UF* et *US* ont alors besoin de communiquer, et selon le modèle, elles communiquent par échanges de messages. On associe donc à *UF* (respectivement *US*) un buffer d'entrée noté *BEF* (respectivement *BES*) où sont rangés les messages à destination de *UF* (respectivement *US*). Les messages émis par *UF* et *US* seront respectivement rangés dans les deux buffers de sortie notés *BSF* et *BSS*. Des outils de communication sont alors nécessaires. Un outil *CF* permet d'extraire un messages de *BSF* et de le ranger soit dans *BEF* (s'il s'agit d'un message *NF-NF*), soit dans *BES* (s'il s'agit d'un message *NF-NS*). De

même, un outil CS permet d'extraire les messages de BSS et de les ranger, soit dans BES s'il s'agit d'un message NS-NS, soit dans BEF, s'il s'agit d'un message NS-NF. La figure XI.2 montre le modèle ainsi représenté.

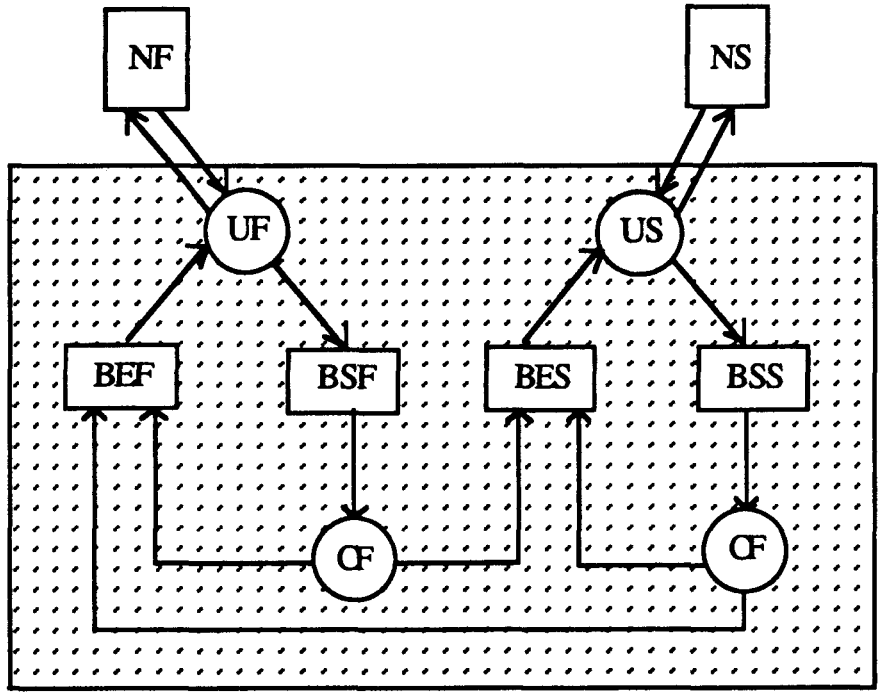


Figure XI.2 : le modèle au deuxième niveau

XI.2 : schéma du modèle dans un contexte multi-processeurs

Si on se place dans un contexte multi-processeurs sans mémoire commune, on dispose alors de n mémoires de noeuds fonctionnels NF_1, ..., NF_n et n mémoires de noeuds séquence NS_1, ..., NS_n et donc, de n unités UF_1, ..., UF_n et n unités US_1, ..., US_n. Le réseau de communication doit alors être tel que CF_i extrait de BSF_i un message qu'il range soit dans $BES_j, j \in [1, n]$, soit dans $BEF_k, k \in [1, n]$, et CS_i extrait de BSS_i un message qu'il range soit dans $BEF_j, j \in [1, n]$, soit dans $BES_k, k \in [1, n]$. On en déduit alors le schéma de la figure XI.3.

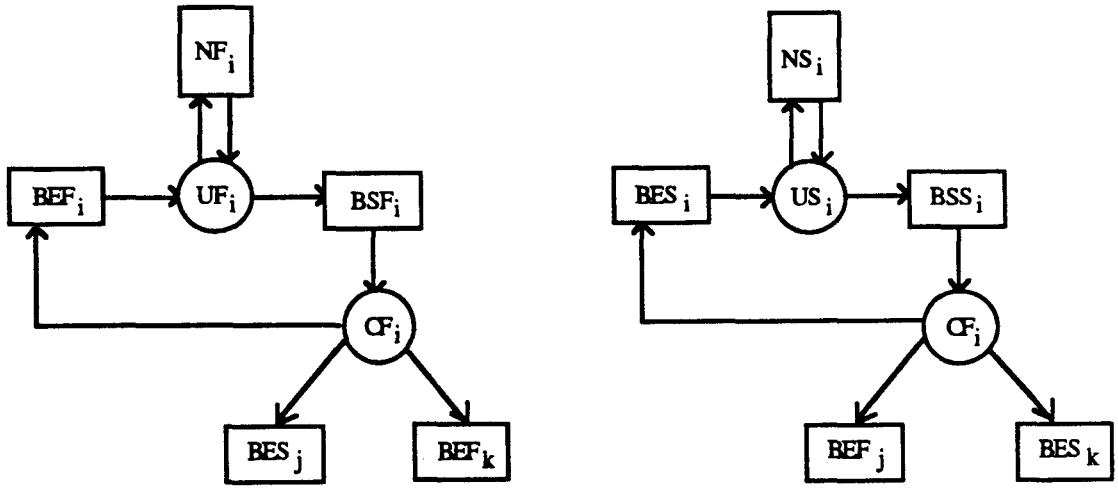


Figure XI.3 : les unités UF_i et US_i dans un contexte multi-processeurs

Si on se place dans un contexte architecture multi-processeurs avec une mémoire globale, le schéma est le même excepté que les unités UF_i et US_i ont accès aux mêmes ensembles NF et NS.

Dans la suite, nous nous placerons dans un contexte n processeurs, n mémoires.

XI.3 : découpage des unités en processus

Les unités UF_i et US_i peuvent être découpées en un ensemble de processus.

XI.3.1 : découpage des unités UF_i

La figure XI.4 montre le découpage des UF_i en processus.

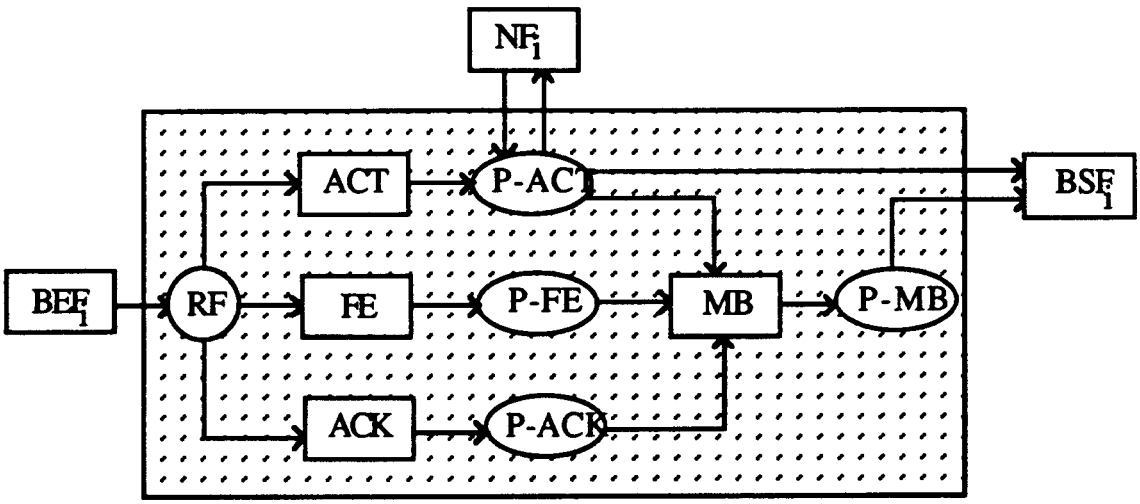


Figure XI.4 : découpage des UF_i en processus

BEF_i reçoit les messages d'exploration, c'est-à-dire les messages d'activation (message *NF-NF*), les messages de fin d'exploration (message *NF-NF*) et les accusés de réception (message *NS-NF*).

Le processus *RF*

Un premier processus noté *RF* sera uniquement chargé d'extraire les messages de BEF_i et, selon leur type, de les ranger dans la file d'attente correspondante : les messages de type ACT, FE et ACK seront ainsi respectivement rangés dans les files ACT, FE, ACK. Un processus est alors associé à chacune de ces files.

Le processus *P-ACT*

Le processus *P-ACT* extrait les messages de la file ACT et traite ce message selon le noeud fonctionnel destinataire du message (et selon l'algorithme d'exploration décrit en section VII.1.2.5). Le processus *P-ACT* peut ainsi être amené à émettre des messages à ranger dans BSF_i (c'est-à-dire un message *NF-NS* et éventuellement un message d'activation et un message de fin d'exploration). Il peut également être amené à créer un message d'activation bloqué ou un message de fin d'exploration bloqué qui sera alors rangé dans la file des messages bloqués (*MB*).

Ce processus sera plus amplement décrit en section XI.4.

Le processus *P-FE*

Le processus *P-FE* extrait de la file *FE* un message de fin d'exploration. Ce message est alors à destination d'un message bloqué rangé dans la file *MB*. Le processus *P-FE* est alors chargé de décrémenter l'information *NB-FE-ATT* du message destinataire rangé dans *MB*.

Le processus *P-ACK*

Le processus *P-ACK* extrait de la file *ACK* un accusé de réception. De même, ce message est alors à destination d'un message bloqué dans la file *MB*. Le processus *P-ACK* est alors chargé de décrémenter l'information *NB-ACK-ATT* du message destinataire rangé dans la file *MB*.

Le processus *P-MB*

Enfin, un processus *P-MB* est chargé d'extraire de la file *MB* les messages dont les informations *NB-ACK-ATT* et *NB-FE-ATT* sont égales à 0 (c'est-à-dire les messages "débloqués") et de les ranger dans BSF_i afin qu'ils soient effectivement émis à leur destinataire.

XI.3.2 : découpage des unités US_i

La figure XI.5 montre le découpage des US_i en processus.

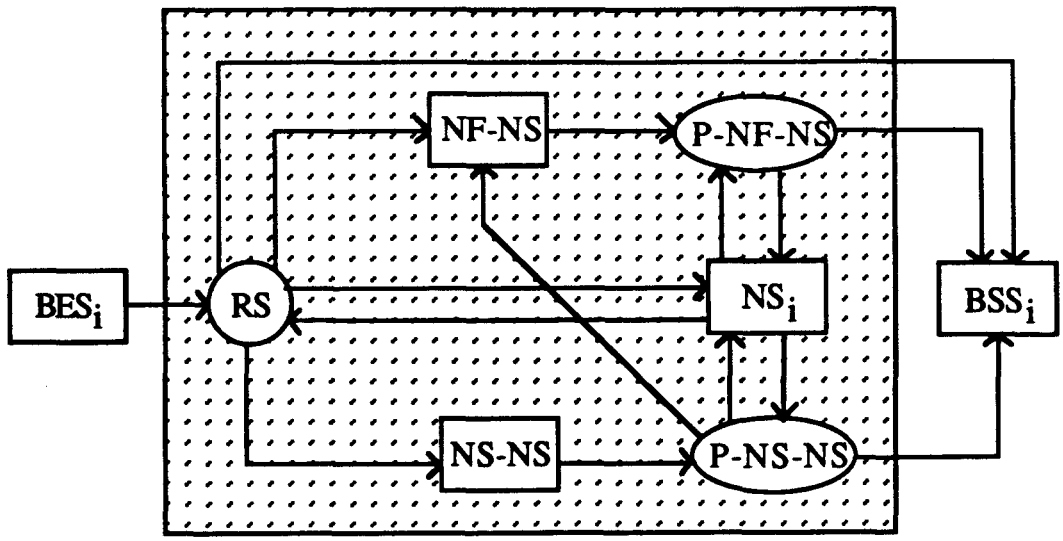


Figure XI.5 : découpage des US_i en processus

BES_i reçoit les messages "de réduction" c'est-à-dire les messages *NF-NS* et les messages *NS-NS*.

Le processus *R S*

Un premier processus *RS* est chargé d'extraire les messages de BES_i et de les ranger dans l'une des deux files *NF-NS* ou *NS-NS*. Si le message extrait est un message *NS-NS*, ce message sera rangé dans la file *NS-NS*. Si le message extrait est un message *NF-NS*, il sera rangé dans la file *NF-NS*. Si l'information FAIRE-SUIVRE du message extrait vaut vrai, le processus RS est alors chargé de transmettre le même message au frère du noeud séquence s'il existe. Si un message *NF-NS* a été émis par un noeud fonctionnel en fin de branche séquentielle, la réception de ce message nécessite alors l'envoi d'un accusé de réception à destination du message bloqué dont les références sont données par l'information *DEST-ACK* du message reçu. Le processus *RS* crée alors cet accusé de réception qu'il range dans BSS_i.

Le rangement d'un message dans la file *NF-NS* sera effectué selon l'algorithme d'ordonnancement décrit en section VII.3. Cet algorithme est donné pour le rangement des messages à destination d'un noeud séquence. Or, ici, NS_i contient éventuellement plusieurs noeuds séquence et donc, deux messages devant être rangés dans la file *NF-NS* peuvent être à destination de deux noeuds séquence différents. Nous

considèrerons donc que la file *NF-NS* est divisée en autant de sous-files qu'il y a de noeuds séquence dans NS_i et le processus *RS* range donc un message *NF-NS* dans la sous-file associée à son destinataire selon l'algorithme d'ordonnancement.

le processus *P-NF-NS*

Avec ces hypothèses, le processus *P-NF-NS* est alors chargé d'extraire un message de la file *NF-NS* pouvant être traité et de traiter ce message.

Qu'est-ce qu'un message *NF-NS* pouvant être traité ?

Un message *NF-NS* peut être traité si les deux conditions suivantes sont vérifiées :

1 - il n'y a pas un autre message *NF-NS* à destination du même noeud séquence devant être traité avant ce message (les règles permettant de déterminer si cette condition est vérifiée ont été énoncées en section VII.3.2).

2 - le noeud séquence destinataire du message est dans l'état réduit (condition énoncée en section IX.2.1).

Le traitement du message consiste alors à appliquer la règle de réduction correspondante, ce qui peut conduire à

- l'émission de messages *NS-NS*,
- l'émission d'un message d'activation à destination d'un noeud fonctionnel (par exemple dans le cas d'un message α).

Si le message extrait de la file *NF-NS* est un message α, le traitement de ce message par *P-NF-NS* sera le suivant :

Soit m = (f-fonct, DEST, α , REF-EMET, DEST-ACK, FAIRE-SUIVRE, DEST-ACT), le message extrait, où REF-EMET = (NOM, NA).

Soit *n* le noeud séquence destinataire du message (i.e. : *m.DEST* = *n*) tel que

n = (NOM, ETAT, VAL, FILS, FRERE, NB-FILS, DER-MSG-TRAITE).

Le processus *P-NF-NS* devra, selon la stratégie définie en section VIII.2.3, ranger dans BSS_i le message d'activation suivant :

TYPE = act
REF = Ø
DEST-ACT = m.DEST-ACT
NUM-ACTIV = m.NA
CIBLE = n.FILS
NB-ACK-ATT = 0
GEN-FAIRE-SUIVRE = vrai
NB-SS-ARB = n.NBFILS
DEST-FE = m.DEST-ACK

Le processus *P-NS-NS*

Enfin, le processus *P-NS-NS* est chargé d'extraire un message de la file *NS-NS* pouvant être traité et de le traiter. Deux conditions doivent être respectées pour qu'un message *NS-NS* puisse être traité :

1- le noeud séquence destinataire doit être dans l'état réduit,

2 - il n'existe pas dans la file *NF-NS* un message à destination du même noeud séquence.

Cette deuxième condition est une conséquence de la propriété démontrée en section VII.3.3. D'après cette propriété, si le noeud séquence *n'* est un fils d'un noeud séquence *n*, un message *NF-NS* à destination de *n'* doit être traité avant un message *NF-NS* à destination de *n*. Or, si un message *NS-NS* est à destination de *n'*, ce message aura été obligatoirement émis par un noeud *n* se trouvant à un niveau supérieur dans l'arbre représentant la séquence argument, suite au traitement d'un message *NF-NS* émis à destination d'un noeud séquence *n"* : *n"* peut éventuellement être différent de *n'*, il est dans ce cas à un niveau supérieur ou égal à celui du noeud *n*.

Le traitement de ce message *NF-NS* à destination de *n"* doit d'après la propriété, "passer après" le message *NF-NS* à destination de *n'*. Donc, le message *NS-NS* à destination de *n'* ayant été généré par le traitement du message *NF-NS* à destination de *n"*, il doit passer après le traitement du message *NF-NS* à destination de *n'*.

Dans le but de pouvoir décrire l'exécution d'un programme FP, nous détaillons dans la section suivante, les traitements effectués par le processus *P-ACT*.

XI.4 : le processus *P-ACT*

XI.4.1 : description détaillée

Le processus *P-ACT* extrait de la file un message d'activation de la forme :

(ACT, REF, DEST-ACT, NUM-ACTIV, CIBLE, NB-ACK-ATT, NB-FE-ATT, GEN-FAIRE-SUIVRE, NB-SS-ARB, DEST-FE).

Soit *m* ce message.

Le traitement de ce message va générer éventuellement la création d'un message bloqué. Dans ce cas, il faut générer une information *REF* (unique), qui servira de référence au message. Nous noterons *générer(REF)* cette opération.

Le processus *P-ACT* effectue alors les traitements suivants :

1 - rechercher dans NF_i le noeud fonctionnel ayant pour nom *DEST-ACT*. Soit n = (NOM, TYPE, ETIQ, SUCC, PARAM, SUIV, NUM-ACTIV) ce noeud fonctionnel où l'information *NOM* est constituée des informations (N-ARB, N-BS, N-ORD, IFB).

2 - SI n.NUM-ACTIV ≠ Ø alors

{*n* est le noeud racine d'une arborescence, le numéro d'activation doit être incrémenté. Ce numéro constituera alors le numéro d'activation de l'arborescence i.e. : le numéro véhiculé dans les messages d'activation lors de l'exploration des arborescences.}

n.NUM-ACTIV ← n.NUM-ACTIV + 1
NA ← n.NUM-ACTIV

SINON

{le numéro d'activation de l'arborescence est celui reçu dans le message d'activation}

NA ← m.NUM-ACTIV

FIN-SI

3 - traiter le message d'activation selon le type du noeud fonctionnel.

(Ces traitements, qui reprennent en grande partie ceux décrits dans l'algorithme d'exploration défini dans la section VII.1.2.5, sont résumés dans un tableau en section XI.4.2)

CAS 1 : n.TYPE = fct-prim

CAS 1.1 : n.SUCC = Ø

n est en fin de branche séquentielle et en fin de chemin séquentiel. Il doit donc créer à destination de *m.DEST-FE* un message de fin-d'exploration bloqué en attente d'un accusé de réception et émettre à destination de son noeud cible un message *NF-NS* (à la réception, le processus *RS* émettra l'accusé de réception). Les traitements sont alors les suivants :

1 - ranger dans MB le message de fin d'exploration bloqué suivant :

TYPE = fe
générer(REF)
DEST-FE = m.DEST-FE
NB-ACK-ATT = 1
NB-FE-ATT = 0

2 - ranger dans BSF_i le message $NF\text{-}NS$ suivant :

TYPE = fp
DEST = m.CIBLE
ETIQ = n.ETIQ
REF-EMET = (n.NOM, NA)
DEST-ACK = REF
FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
DEST-ACT = Ø

FIN-CAS-1.1

CAS 1.2 : n.SUCC ≠ Ø

Le processus $P\text{-}ACT$ doit uniquement émettre à destination du noeud cible un message $NF\text{-}NS$ et à destination de $n.SUCC$ un message d'activation. Les traitements sont alors les suivants :

1 - ranger dans BSF_i le message $NF\text{-}NS$ suivant :

TYPE = fp
DEST = m.CIBLE
ETIQ = n.ETIQ
REF-EMET = (n.NOM, NA)
DEST-ACK = Ø
FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
DEST-ACT = Ø

2 - ranger dans BSF_i le message d'activation suivant :
TYPE = act
REF = Ø
DEST-ACT = n.SUCC
NUM-ACTIV = NA
CIBLE = m.CIBLE
NB-ACK-ATT = 0
NB-FE-ATT = 0
GEN-FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
NB-SS-ARB = m.NB-SS-ARB
DEST-FE = m.DEST-FE

FIN-CAS 1.2

FIN-CAS 1

CAS 2 n.TYPE = def

le noeud fonctionnel *n* est alors obligatoirement en fin de branche séquentielle (cf section VII.3.2). S'il a un successeur, il doit créer à destination de son successeur un message d'activation bloqué en attente d'un accusé de réception et d'un message de fin d'exploration, sinon, il doit créer un message de fin d'exploration à destination de *m.DEST-FE*, ce message étant, de même, en attente d'un accusé de réception et d'un message de fin d'exploration. Dans les deux cas, un message *NF-NS* sera émis à destination du noeud cible et un message d'activation sera émis à destination du noeud racine de l'arborescence secondaire correspondante. Le processus *P-ACT* effectue donc les traitements suivants :

1 - SI n.SUCC = Ø ALORS
ranger dans *MB* le message de fin d'exploration bloqué suivant :
TYPE = fe
générer(REF)
DEST-FE = m.DEST-FE
NB-ACK-ATT = 1
NB-FE-ATT = 1

SINON

ranger dans MB le message d'activation bloqué suivant :

TYPE = act
REF = Ø
DEST-ACT = n.SUCC
NUM-ACTIV = NA
CIBLE = m.CIBLE
NB-ACK-ATT = 1
NB-FE-ATT = 1
GEN-FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
NB-SS-ARB = m.NS-SS-ARB
DEST-FE = m.DEST-FE

FSI

2 - REF-MSG-BLOQUE ← REF

3 - ranger dans BSF_i le message $NF\text{-}NS$ suivant :

TYPE = def
DEST = m.CIBLE
ETIQ = def
REF-EMET = (n.NOM, NA)
DEST-ACK = REF-MSG-BLOQUE
FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
DEST-ACT = Ø

4 - ranger dans BSF_i le message d'activation suivant :

TYPE = act
REF = Ø
DEST-ACT = n.ETIQ
NUM-ACTIV = NA
CIBLE = m.CIBLE
NB-ACK-ATT = 0
NB-FE-ATT = 0
GEN-FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
NB-SS-ARB = m.NB-SS-ARB
DEST-FE = REF-MSG-BLOQUE

FIN CAS 2

CAS 3 : n.TYPE = f-fonct

Le processus *P-ACT* doit effectuer dans ce cas différents traitements selon la forme fonctionnelle. Nous ne détaillerons que le cas où la forme fonctionnelle est l'*Apply-to-all*.

CAS 3.1 : n.ETIQ = α

Dans le cas où *n* n'a pas de successeur, un message de fin d'exploration est créé bloqué en attente d'un accusé de réception et *m.NB-SS-ARB* messages de fin d'exploration (cf section IX.2.3). Si *n* a un successeur, c'est un message d'activation qui est créé bloqué à destination du successeur, en attente de *m.NB-SS-ARB* messages de fin d'exploration. Dans les deux cas, un message *NF-NS* est envoyé au noeud-cible. Le processus *P-ACT* effectue donc les traitements suivants :

1 - SI n.SUCC = Ø ALORS

ranger dans *MB* le message de fin d'exploration bloqué suivant :

TYPE = fe
générer(REF)
DEST-FE = m.DEST-FE
NB-ACK-ATT = 1
NB-FE-ATT = m.NB-SS-ARB

SINON

ranger dans *MB* le message d'activation bloqué suivant :

TYPE = act
générer(REF)
DEST-ACT = n.SUCC
NUM-ACTIV = NA
CIBLE = m.CIBLE
NB-ACK-ATT = 0
NB-FE-ATT = m.NB-SS-ARB
GEN-FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
NB-SS-ARB = m.NB-SS-ARB
DEST-FE = m.DEST-FE

FIN-SI

2 - REF-MSG-BLOQUE ← REF

3 - ranger dans BSF_i le message *NF-NS* suivant :
TYPE = f-fonct
DEST = m.CIBLE
ETIQ = α
REF-EMET = (n.NOM,NA)
DEST-ACK = REF-MSG-BLOQUE
FAIRE-SUIVRE = m.GEN-FAIRE-SUIVRE
DEST-ACT = n.PARAM

FIN-CAS 3.1

etc ...

FIN CAS 3

XI.4.2 : résumé

Le tableau de la figure XI.6 résume les traitement effectués par le processus *P-ACT* lorsqu'un message d'activation a été extrait d'une file *ACT*, que le noeud fonctionnel destinataire a été recherché et que le numéro d'activation de ce noeud a éventuellement été mis à jour.

Soit m = (act, REF, DEST-ACT, NUM-ACTIV, CIBLE, NB-ACK-ATT, NB-FE-ATT, GEN-FAIRE-SUIVRE, NB-SS-ARB, DEST-FE),
le message d'activation extrait où *DEST-ACT* est un noeud fonctionnel *n* tel que
n = (NOM, TYPE, ETIQ, SUCC, PARAM, SUIV, NUM-ACTIV).

Le tableau est composé de 3 lignes principales et de 3 colonnes principales (séparées par un double trait). Chaque colonne principale correspond à un type de message à émettre (fin d'exploration, message

NF-NS, activation). Les 3 lignes principales du tableau correspondent aux trois cas de figure :

- n.TYPE = fct-prim,
- n.TYPE = def,
- n.TYPE = f-fonct.

Les "sous-lignes" (séparées par un trait simple) donnent alors, selon l'information *n.SUCC*, les messages devant être générés par *P-ACT*. Le symbole "_" en début de "sous-ligne" indique que les messages de la sous-ligne correspondante doivent être émis quelle que soit l'information *n.SUCC*. Les messages à émettre se trouvent aux intersections des sous-lignes et des colonnes.

Remarques :

- le symbole "*" dans une colonne *REF* indique que l'information *REF* doit être générée,
- l'information *GEN-FAIRE-SUIVRE* a été abrégée en *GFS* et l'information *NB-SS-ARB* a été abrégée en *NBSA*.

Exemple :

L'intersection de la première sous-ligne et de la première colonne montre que, si *n.TYPE* = *fct-prim* et *n.SUCC* = $\emptyset$, le message de fin d'exploration suivant doit être émis :

TYPE = fe
générer(REF)
DEST-FE = M.DEST-FE
NB-ACK-ATT = 1
NB-FE-ATT = 0

P-ACT		fin-d'exploration			NF-NS				activation			
		TYPE = fe, DEST-FE = m.DEST-FE			DEST = m.CIBLE, REF-EMET = (n.NOM, NA), FAIRE-SUIVRE = m.GFS				TYPE = act, NUMA = NA, CIBLE = m.CIBLE, GFS = m.GFS, NBSA = m.NBSA, DEST-FE = m.DEST-FE			
		REF	NB-ACK -ATT	NB-FE -ATT	TYPE	ETIQ	DEST-ACK	DEST-ACT	REF	DEST-ACT	NB-ACK	NB-FE
fct-prim	n.succ = Ø	*	1	0	fp	n.ETIQ	REF	Ø				
	n.succ ≠ Ø				fp	n.ETIQ	Ø	Ø	*	n.SUCC	0	0
def	n.succ = Ø	*	1	1								
	n.succ ≠ Ø								Ø	n.SUCC	1	1
	–				def	def	REF	Ø	Ø	n.ETIQ	0	0
f-fonct et etiq= α	n.succ = Ø	*	1	m.NB-SS-ARB								
	n.succ ≠ Ø								*	n.SUCC	0	m.NB-SS-ARB
	–				f-fonct	α	REF	n. PARAM				

Figure XI.6 : tableau résumant le processus P-ACT

Grâce aux descriptions de processus données en section X.3 et X.4, nous pouvons maintenant illustrer l'exécution d'un programme FP selon le modèle.

XII : EXEMPLE D'EXECUTION D'UN PROGRAMME FP

XII : EXEMPLE D'EXECUTION D'UN PROGRAMME FP

Dans ce chapitre, nous illustrons le "fonctionnement" du modèle sur un petit exemple de programme FP. Dérouler l'exécution d'un programme consiste à considérer qu'un message d'activation est émis au noeud racine de l'arborescence principale et à traiter ce message : le traitement de ce message génèrera l'émission d'autres messages qui seront à leur tour traités, etc... , et ce, jusqu'à ce qu'il n'y ait plus de message à traiter. Nous décrirons schématiquement l'exécution d'un programme. Pour cela,

- nous désignerons un noeud fonctionnel par son étiquette,
- nous donnerons des noms aux noeuds séquence, par exemple,

représentera la séquence $<1>$. Le noeud racine a pour nom a et le noeud feuille a pour nom b,

- nous donnerons un nom aux différents messages générés. Par exemple,

ack1 =

type	dest ack
ack1	m1

désigne un message "accusé de réception", nommé $ack1$, à destination du message nommé m_1,

- Nous découperons les pages suivantes en fenêtres comme le montre la figure XII.1.

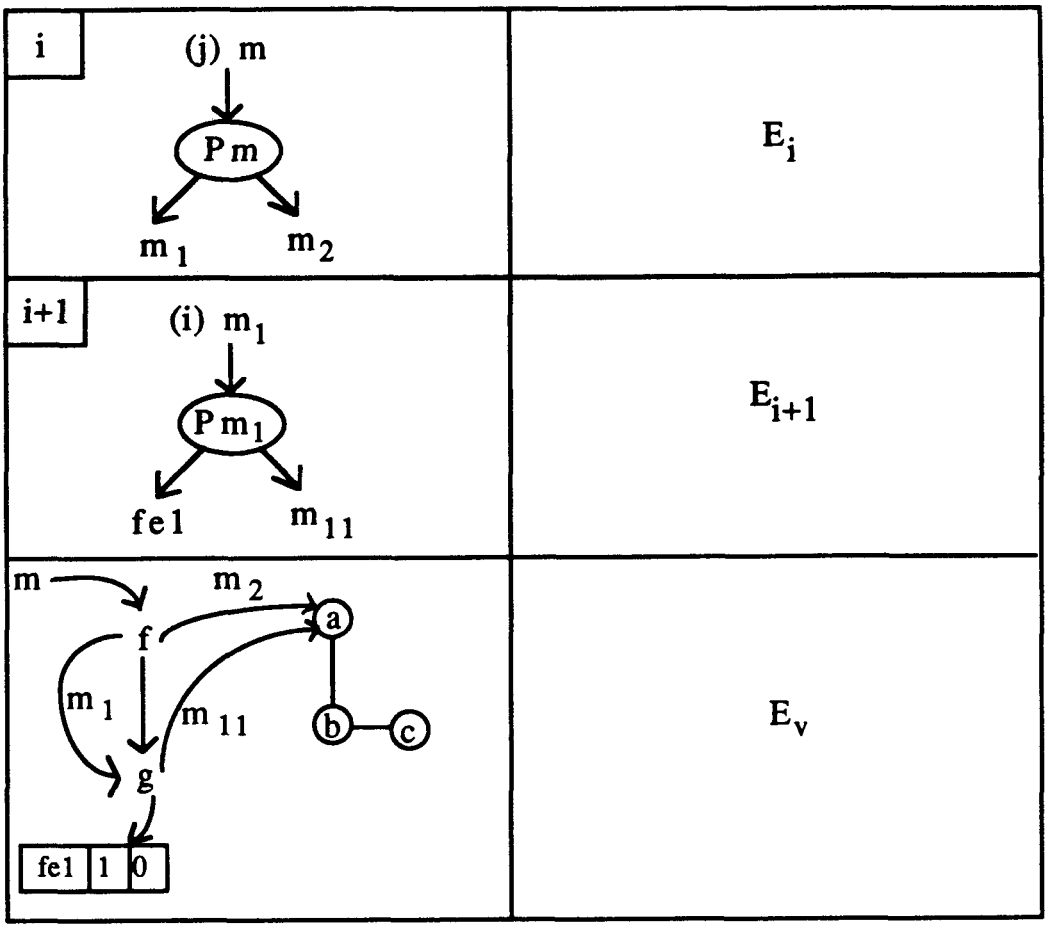


Figure XII.1 : découpage des pages en fenêtres

Sur cette figure, la page de gauche est divisée en 3 fenêtres.

- La fenêtre *i* montre que le message *m* créé dans la fenêtre *j* ($j \leq i$) est traité par le processus P_m qui émet les messages m_1 et m_2.

- De même, la fenêtre *(i+1)* montre que le message m_1 créé dans la fenêtre *(i)* est traité par le processus P_{m1} qui émet les messages fe_1 et m_{11}. Dans ces fenêtres, nous donnerons les informations contenues dans ces messages.

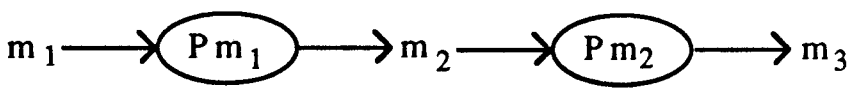
- La fenêtre au bas de la page gauche permet de visualiser les échanges de messages effectués. Les arborescences fonctionnelles du programme à exécuter et la séquence argument y sont représentés. L'exemple donné en figure XII.1 montre que le programme à exécuter

est la fonction $f \circ g$. Le message m est à destination du noeud fonctionnel f, le message m_1, émis suite au traitement de m est à destination de g tandis que le message m_2 est à destination du noeud séquence a.

Le message fe_1 émis suite au traitement de m_1 est un message bloqué en attente d'un accusé de réception et de 0 messages de fin d'exploration, tandis que le message m_{11} est émis à destination du noeud séquence a.

Sur la page de droite, nous donnerons quelques explications supplémentaires. Ainsi, la fenêtre E_i contiendra quelques explications sur les schémas représentés dans la fenêtre i de la page de gauche. Les noms des processus qui apparaîtront correspondent aux noms des processus décrits en section XI.3. Le lecteur pourra ainsi se référer à cette section pour trouver la justification des messages émis.

Nous ne ferons aucune hypothèse sur la machine cible et donc, nous ne ferons apparaître ni la façon dont sont répartis les noeuds fonctionnels et les noeuds séquence sur les différents processeurs, ni les phases de "routage" des messages. Ainsi, nous écrirons directement



au lieu d'un schéma montrant que le message m_1 est rangé dans un buffer d'entrée, qu'il est ensuite extrait par un processus RS ou RF qui le range dans une file, puis que ce message est extrait de la file par le processus P_{m1} correspondant qui range alors un message m_2 dans un buffer de sortie, que ce message m_2 est dirigé vers un buffer d'entrée etc...

L'exemple que nous traiterons est l'application du programme P à la séquence S définis ci-dessous :

$P = \alpha(+ \circ \alpha*)$

$S = \langle\langle\langle 0, 2\rangle, \langle 1, 3\rangle\rangle, \langle\langle 2, 1\rangle, \langle 3, 0\rangle\rangle\rangle$

Dans les schémas, les α seront indicés de façon à pouvoir les nommer sans ambiguïté. De plus, nous utiliserons les abréviations suivantes pour les informations contenues dans les messages :

GFS	désignera	GEN-FAIRE-SUIVRE
NBSA	"	NB-SS-ARB
ACK-ATT	"	NB-ACK-ATT
FE-ATT	"	NB-FE-ATT
FF	"	F-FONCT

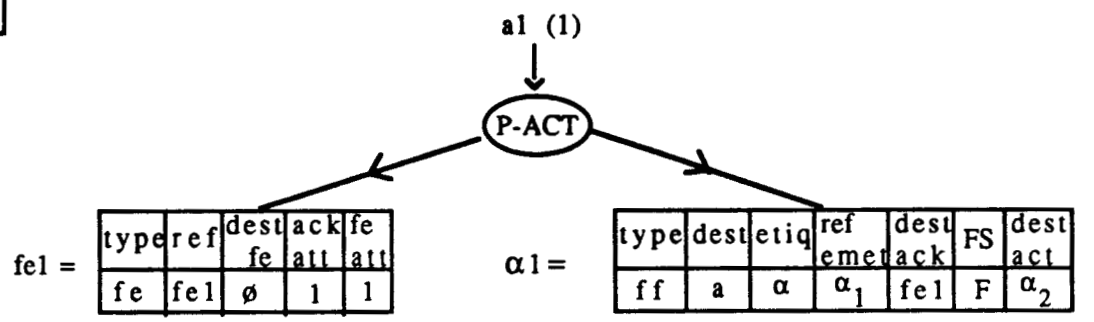
L'exemple est alors décrit ci-après.

1

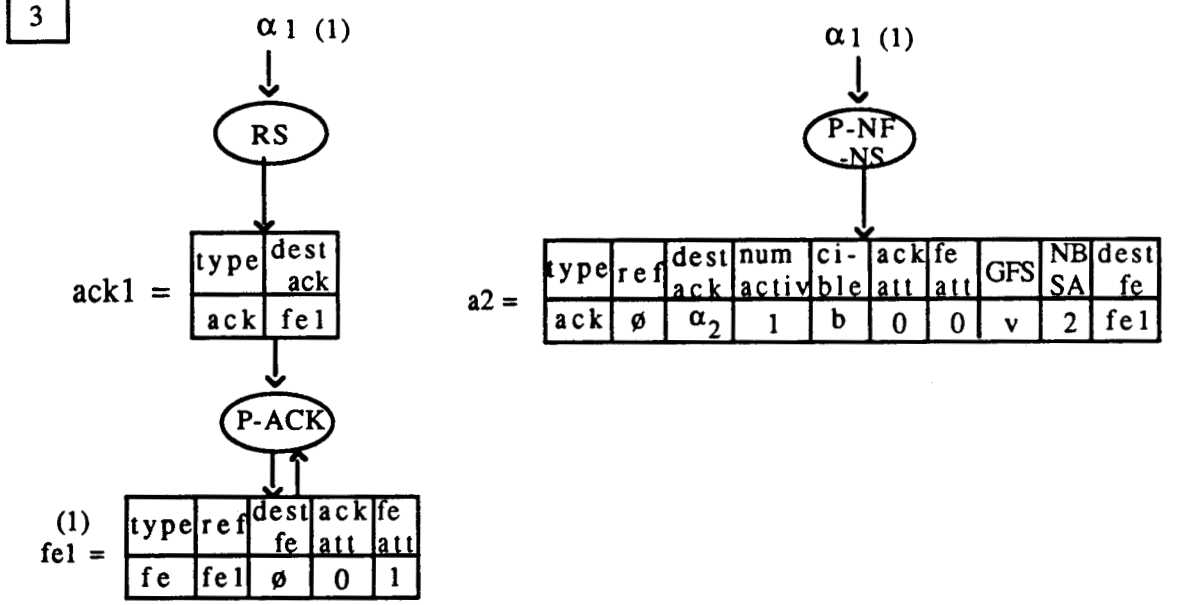
a1 =

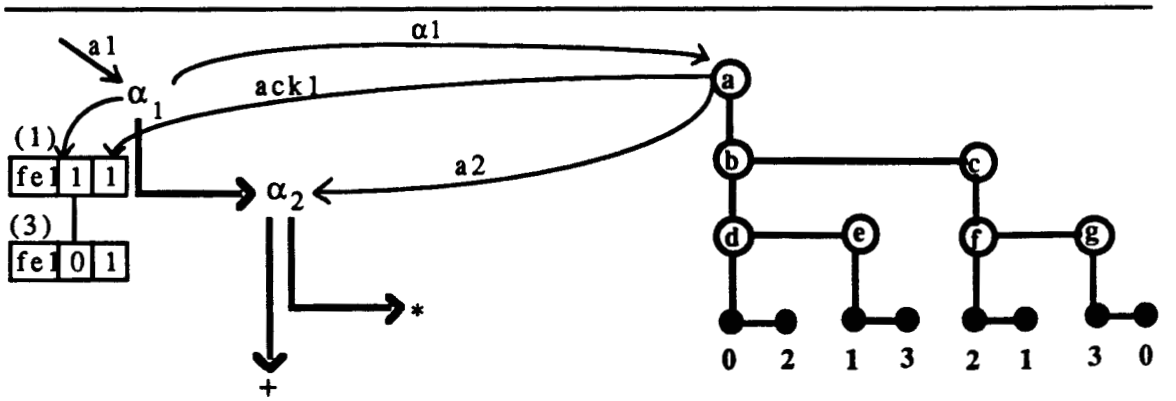
type	ref	dest ack	num activ	ci-ble	ack att	fe att	GFS	NB SA	dest fe
ack	ø	α_1	0	a	0	0	F	1	ø

2



3





L'exécution du programme débute par l'émission du message d'activation *a1* au noeud racine de l'arborescence.

Le message d'activation *a1* est traité par le processus *P-ACT* qui génère les messages *fe1* et $\alpha 1$. Le message *fe1* est en attente d'un accusé de réception et d'un message de fin d'exploration puisque le noeud α_1 est en fin de chemin séquentiel et en fin de branche séquentielle. Le message $\alpha 1$ est un message *NF-NS*, à destination du noeud séquence *a*.

Le traitement du message $\alpha 1$ par *RS* genère l'envoi d'un accusé de réception au message *fe1* puisque α_1 est en fin de branche séquentielle. Cet accusé de réception permet de décrémenter l'information *NB-ACK* du message *fe1* qui devient ainsi égale à 0. Le traitement du même message $\alpha 1$ par le processus *P-NF-NS* permet de générer l'émission du message d'activation *a2* à destination du noeud α_2. Le noeud α_2 devra générer à destination de son noeud cible, un message à faire suivre (*GFS* = *V*), $\alpha 2$ s'appliquera donc à deux sous-arbres (*NBSA* = *2*).

4

(2)a2 =

type	ref	dest ack	num activ	ci-ble	ack att	fe att	GFS	NB SA	dest fe
ack	ø	α_2	1	b	0	0	v	2	fe1

P-ACT

a3 =

type	ref	dest ack	num activ	ci-ble	ack att	fe att	GFS	NB SA	dest fe
act	a3	+	1	b	0	2	v	2	fe1

α_2^1 =

type	dest	etiq	ref emet	dest ack	FS	dest act
ff	b	α	α_2	a3	V	*

5

α_2^1 → RS → α_2^2 =

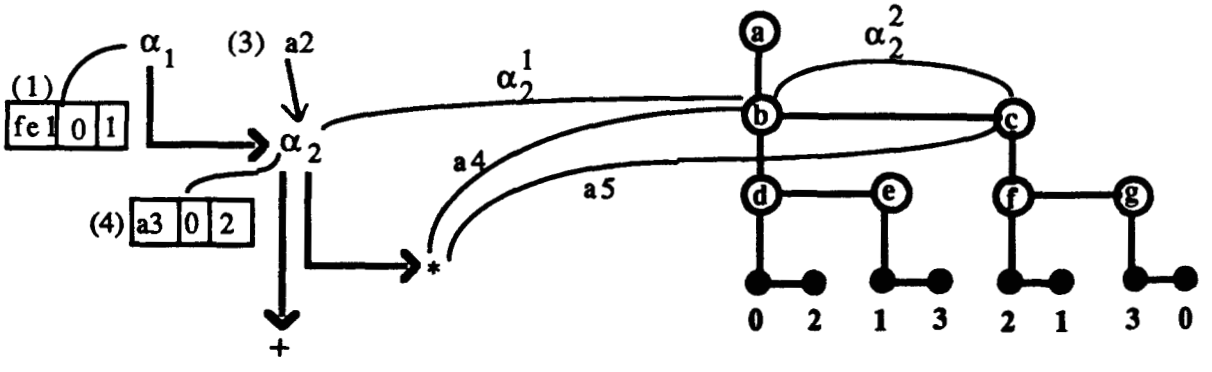
type	dest	etiq	ref emet	dest ack	FS	dest act
ff	c	α	α	a3	V	*

α_2^1 → P-NF -NS → a4 =

type	ref	dest ack	num activ	ci-ble	ack att	fe att	GFS	NB SA	dest fe
ack	ø	*	1	d	0	0	V	2	a3

α_2^2 → P-NF -NS → a5 =

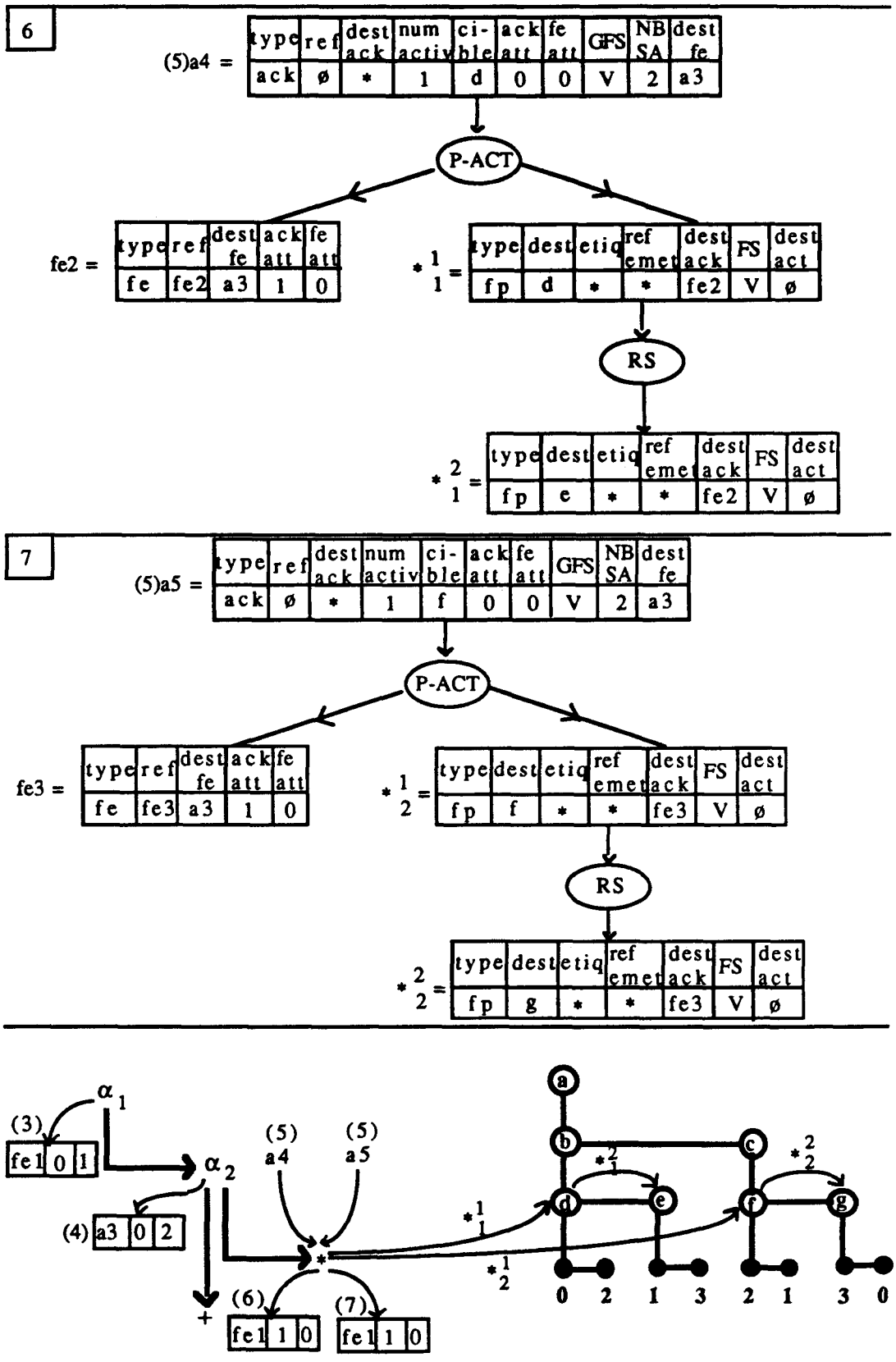
type	ref	dest ack	num activ	ci-ble	ack att	fe att	GFS	NB SA	dest fe
ack	ø	*	1	f	0	0	V	2	a3



Le traitement de $a2$ par $P\text{-}ACT$ génère :

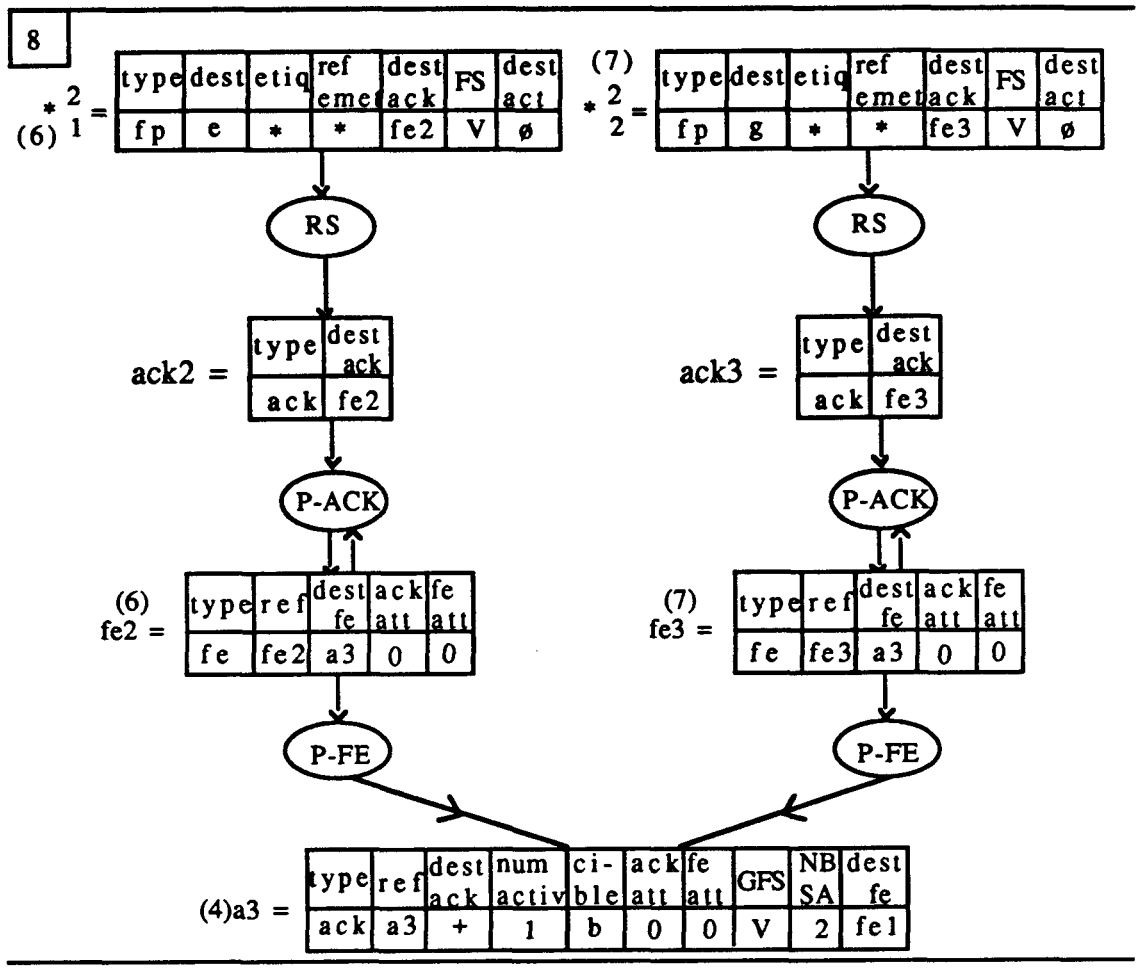
- la création du message d'activation bloqué à destination du noeud fonctionnel + : ce message est en attente de 2 messages de fin d'exploration puisque $\alpha 2$ s'applique à deux sous-arbres
- l'émission d'un message $NF\text{-}NS$ à destination du noeud séquence b.

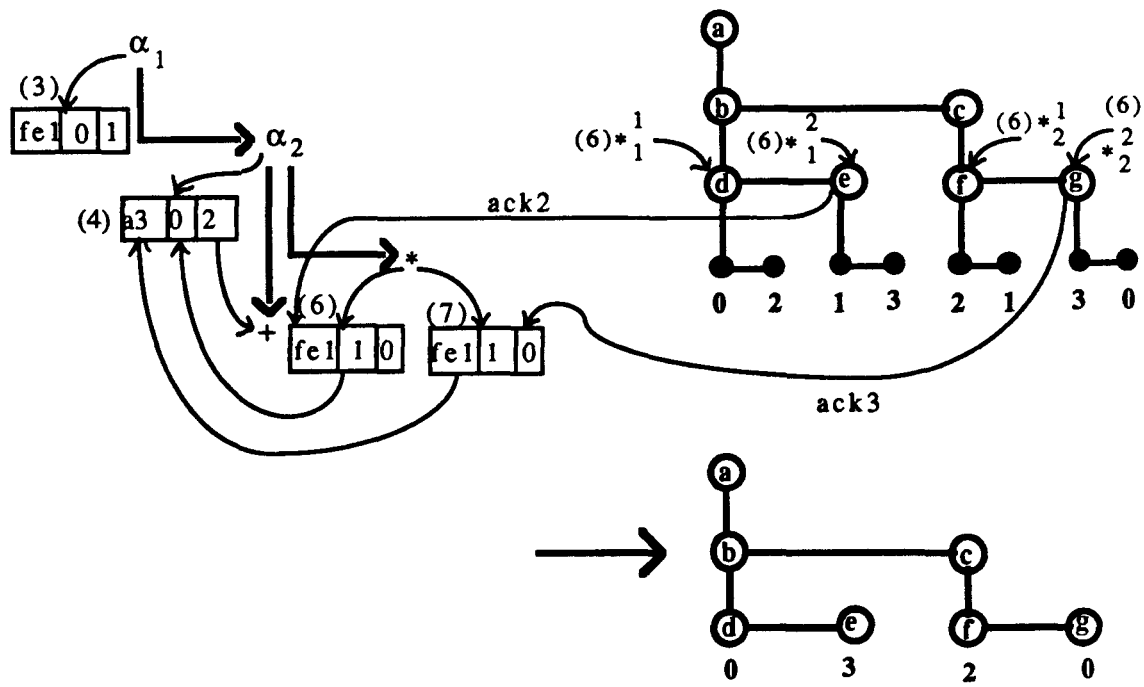
Le traitement du message $\alpha_2{}^1$ par RS permet de faire suivre ce message au noeud séquence c, c'est-à-dire d'émettre le message $\alpha_2{}^2$ à destination de c. Les traitements des messages $\alpha_2{}^1$ et $\alpha_2{}^2$ par $P\text{-}NF\text{-}NS$ génèrent deux messages d'activation à destination de $*$.


6
(5)a4 =
type | ref | dest ack | num activ | ci-ble | ack att | fe att | GFS | NB SA | dest fe
ack | ø | * | 1 | d | 0 | 0 | V | 2 | a3
P-ACT
fe2 =
type | ref | dest fe | ack att | fe att
fe | fe2 | a3 | 1 | 0
*1 1 =
type | dest | etiq | ref emet | dest ack | FS | dest act
fp | d | * | * | fe2 | V | ø
RS
*2 1 =
type | dest | etiq | ref emet | dest ack | FS | dest act
fp | e | * | * | fe2 | V | ø
7
(5)a5 =
type | ref | dest ack | num activ | ci-ble | ack att | fe att | GFS | NB SA | dest fe
ack | ø | * | 1 | f | 0 | 0 | V | 2 | a3
P-ACT
fe3 =
type | ref | dest fe | ack att | fe att
fe | fe3 | a3 | 1 | 0
*1 2 =
type | dest | etiq | ref emet | dest ack | FS | dest act
fp | f | * | * | fe3 | V | ø
RS
*2 2 =
type | dest | etiq | ref emet | dest ack | FS | dest act
fp | g | * | * | fe3 | V | ø
α1
(3)
fe1 | 0 | 1
α2
(4)
a3 | 0 | 2
(5) a4
(5) a5
(6)
fe1 | 1 | 0
(7)
fe1 | 1 | 0
+
*
a
b
c
d
e
f
g
*2 1
*2 2
*1 1
*1 2
0 2 1 3 2 1 3 0

Le message *a4* est l'un des deux messages d'activation qui ont été envoyés au noeud $*$. Le noeud fonctionnel $*$ est en fin de branche séquentielle et en fin de chemin séquentiel. Le processus *P-ACT* crée donc un message de fin d'exploration bloqué en attente d'un accusé de réception et émet à destination de *d* un message *NF-NS* que le processus *RS* fait suivre au noeud *e*.

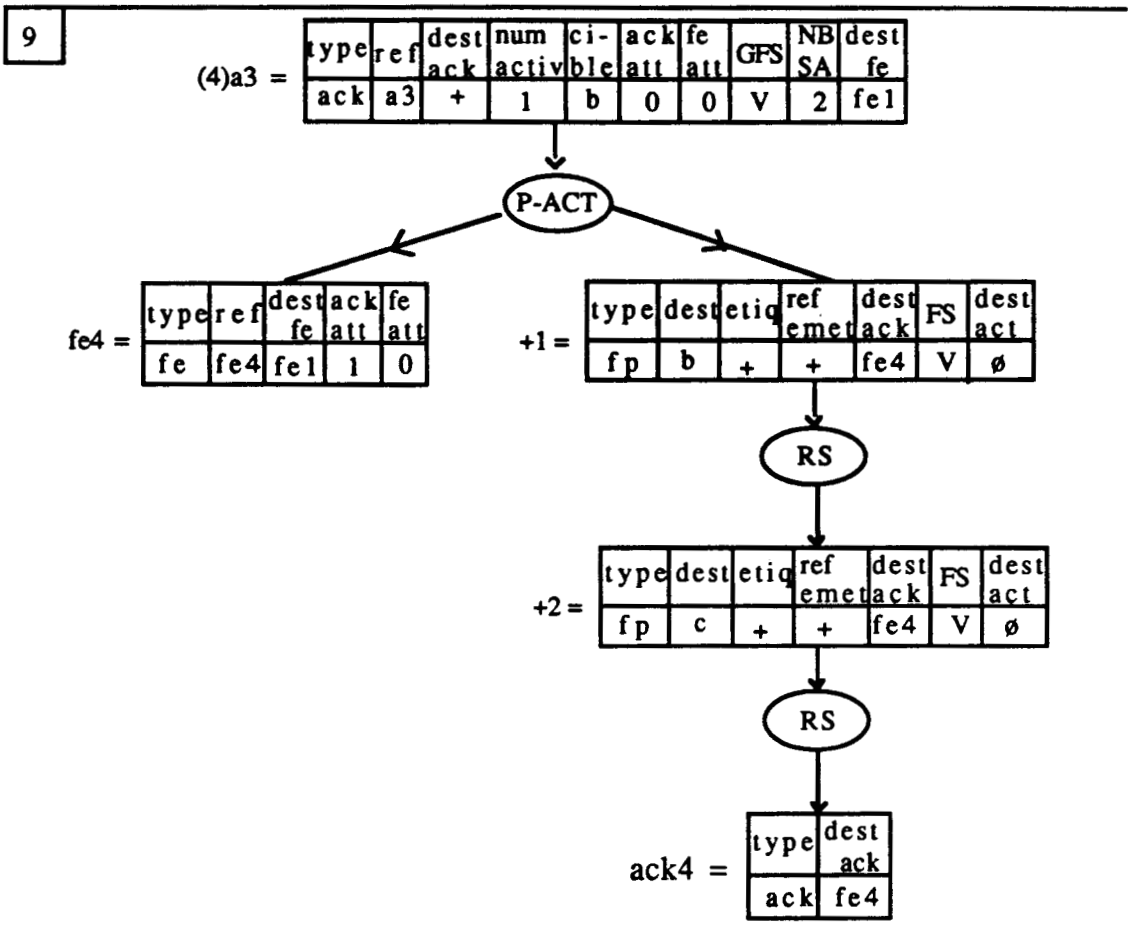
Le message *a5* est l'autre message d'activation envoyé au noeud fonctionnel $*$. Les traitements de ces deux message par *P-ACT* sont identiques.

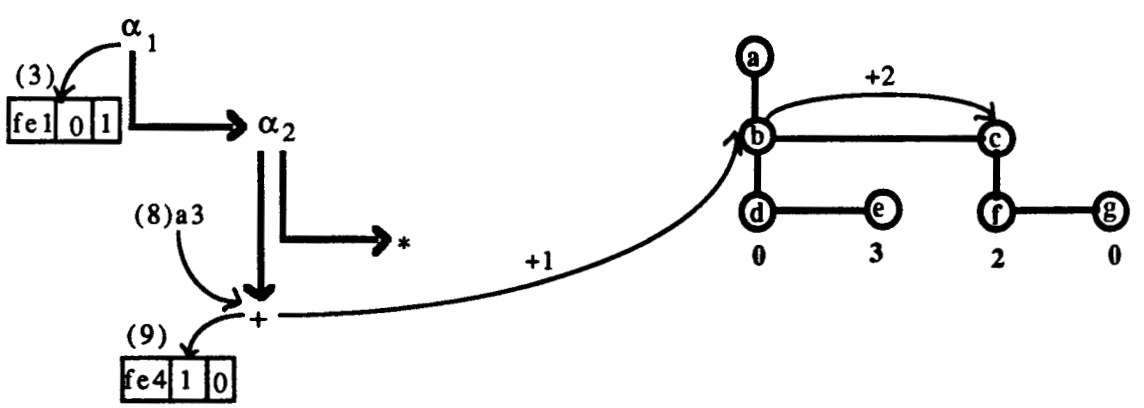

8
(6) *2 1 =
type dest etiq ref emet dest ack FS dest act
fp e * * fe2 V ø
(7) *2 2 =
type dest etiq ref emet dest ack FS dest act
fp g * * fe3 V ø
RS
RS
ack2 =
type dest ack
ack fe2
ack3 =
type dest ack
ack fe3
P-ACK
P-ACK
(6) fe2 =
type ref dest fe ack att fe att
fe fe2 a3 0 0
(7) fe3 =
type ref dest fe ack att fe att
fe fe3 a3 0 0
P-FE
P-FE
(4)a3 =
type ref dest ack num activ ci-ble ack att fe att GFS NB SA dest fe
ack a3 + 1 b 0 0 V 2 fe1


α1
(3)
fe1 0 1
α2
(4) a3 0 2
ack2
*
+
(6)
fe1 1 0
(7)
fe1 1 0
ack3
(6)*1 1
(6)*2 1
(6)*1 2
(6)*2 2
a b c d e f g
0 2 1 3 2 1 3 0
a b c d e f g
0 3 2 0

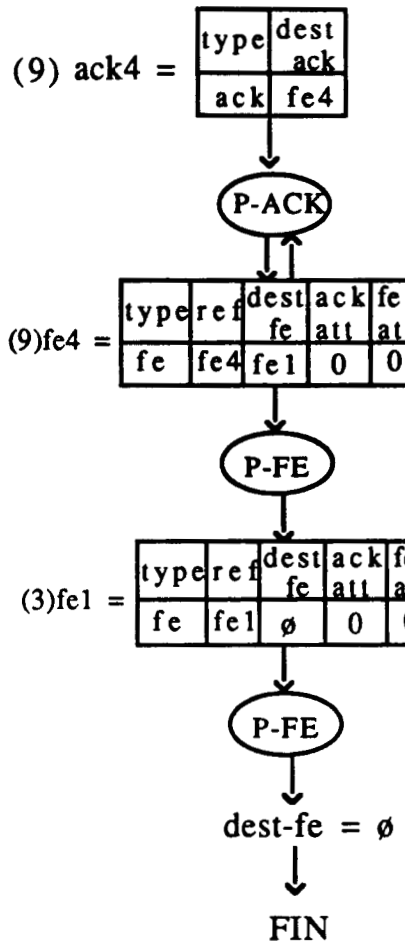
Les traitements des deux messages $*_1{}^2$ et $*_2{}^2$ par le processus *RS* permettent l'émission des accusés de réception *ack2* et *ack3*. Ces deux accusés de réception sont traités par le processus *P-ACK* qui met à jour l'information *ACK-ATT* des messages destinataires (i.e. : *fe2* et *fe3*). Les informations *ACK-ATT* et *FE-ATT* de ces deux messages étant égales à 0, les messages sont débloqués et traités par le processus *P-FE* qui met à jour le message d'activation bloqué *a3*. Ce message *a3* est alors débloqué et effectivement émis à son destinataire.

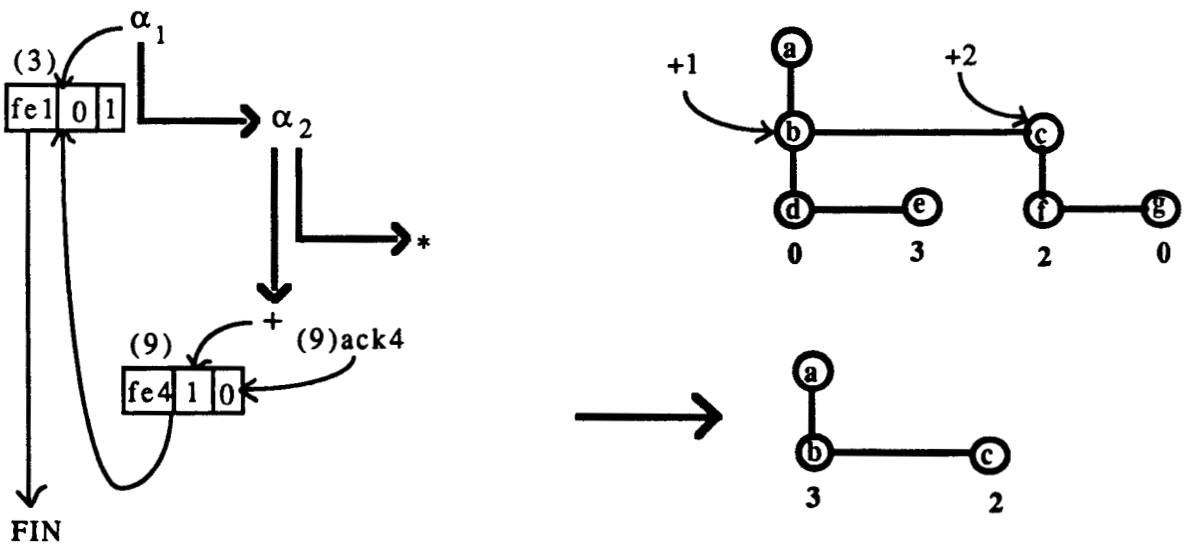
Le traitement des messages $*_1{}^1$, $*_1{}^2$, $*_2{}^1$ et $*_2{}^2$ par le processus *P-NF-NS* permet d'appliquer la règle de réécriture correspondant à un message émis par un noeud fonctionnel $*$ et donc, d'obtenir la séquence argument définie au bas de la page.


9
(4)a3 =
type | ref | dest ack | num activ | ci-ble | ack att | fe att | GFS | NB SA | dest fe
ack | a3 | + | 1 | b | 0 | 0 | V | 2 | fe1
P-ACT
fe4 =
type | ref | dest fe | ack att | fe att
fe | fe4 | fe1 | 1 | 0
+1 =
type | dest | etiq | ref emet | dest ack | FS | dest act
fp | b | + | + | fe4 | V | ø
RS
+2 =
type | dest | etiq | ref emet | dest ack | FS | dest act
fp | c | + | + | fe4 | V | ø
RS
ack4 =
type | dest ack
ack | fe4


α1
(3)
fe1 | 0 | 1
α2
(8)a3
*
+
(9)
fe4 | 1 | 0
+1
a
b
c
d
e
f
g
+2
0
3
2
0

Le message d'activation *a3* ayant été débloqué, il est traité par le processus *P-ACT*.


(9) ack4 =
type
dest ack
ack
fe4
P-ACK
(9)fe4 =
type
ref
dest fe
ack att
fe att
fe
fe4
fe1
0
0
P-FE
(3)fe1 =
type
ref
dest fe
ack att
fe att
fe
fe1
ø
0
0
P-FE
dest-fe = ø
FIN


(3)
α1
fe1
0
1
α2
*
+
(9)
(9)ack4
fe4
1
0
FIN
+1
+2
a
b
c
d
e
f
g
0
3
2
0
a
b
c
3
2

Les messages *+1* et *+2* sont traités par le processus *P-NF-NS* qui applique la règle de réécriture correspondante et permet d'obtenir le résultat de l'exécution du programme.

CONCLUSION

CONCLUSION

Pour conclure, nous allons essayer dans un premier temps de dégager les caractéristiques du modèle afin de montrer dans quelle mesure nous avons répondu à nos objectifs initiaux et dans un deuxième temps, nous dégagerons les perspectives de ce travail.

- Nous avons défini un modèle d'exécution parallèle du langage FP qui respecte la sémantique de ce langage.

- Dans ce modèle, le parallélisme est mis en oeuvre à deux niveaux : à un premier niveau, l'exploration parallèle des arborescences fonctionnelles génère à destination de la séquence argument des envois de messages en parallèle, qui donnent lieu, à un deuxième niveau, à l'application de règles de réécriture en parallèle.

Ces deux "niveaux de parallélisme" sont mis en oeuvre simultanément : les réductions et l'exploration sont effectuées en parallèle ; le surcoût introduit par l'exploration est donc en grande partie recouvert par les réductions.

- Le modèle est défini de telle façon que l'exploration des arborescences fonctionnelles ne modifie pas les arborescences : ceci permet une exploration multiple des arborescences dans le cas de fonctions récursives par exemple ; aucune copie de code n'est effectuée.

- Le modèle défini ne peut être considéré ni comme un modèle data-flow ni comme un modèle de réduction au sens classique du terme. Il peut plutôt être vu comme un distributeur de tâches générant des réductions en parallèle.

Notre but était de définir un modèle d'exécution parallèle de FP permettant de réduire les inconvénients que présente le modèle de réduction de graphe, à savoir

- la création et la gestion de nombreux noeuds application intermédiaires,

- le coût d'instanciation d'une application, puisque l'instanciation conduit à la transformation d'un sous-graphe,
- le coût de la recherche d'une expression réductible dans un graphe.

Ces inconvénients sont effectivement réduits dans le modèle :
- du fait de la séparation données programmes, on ne gère pas de noeuds application intermédiaires,
- il n'y a pas dans le modèle d'opération d'instanciation à proprement parler. Au lieu d'une transformation de graphe, l'exploration des arborescences génère l'émission de messages à destination de la séquence argument, ce qui est beaucoup moins coûteux,
- les messages émis à la séquence argument sont directement émis à destination d'un noeud séquence ce qui évite la recherche d'expressions réductibles : le traitement d'un message à destination d'un noeud séquence consiste à appliquer une règle de réécriture à l'arbre ayant pour racine, le noeud séquence destinataire du message.

Une simulation de ce modèle a débuté, qui nous permettra à court terme d'en évaluer les performances et de l'optimiser. Nous envisageons, également à court terme, d'approfondir la façon dont le modèle peut être implanté sur la machine N-ARCH et pour cela, de déterminer une stratégie de répartition des noeuds fonctionnels et des noeuds séquence sur le réseau, et de préciser l'architecture et les fonctionnalités de la machine.

A moyen terme, nous envisageons d'étendre ce modèle aux systèmes FFP de Backus afin de permettre la manipulation de fonctions d'ordre supérieur. Une étude partielle a déjà montré que cette extension était tout-à-fait envisageable.

A long terme, nous pouvons envisager d'étendre le modèle à d'autres langages sans variable et peut être même aux langages fonctionnels en général...

REFERENCES BIBLIOGRAPHIQUES

PUBLICATIONS ET COMMUNICATIONS

"A new parallel evaluation scheme of FP on the N-ARCH reduction machine"
N.DEVESA, G.GONCALVES, M.P.LECOUFFE, B.TOURSEL
publication LIFL n° 54 - Février 1988

"A controlled reduction model of functional programs on a distributed associative network"
N.DEVESA, G.GONCALVES, M.P.LECOUFFE, B.TOURSEL
Euromicro Congress 88 - Zurich (Suisse) - 29 Aug, 1 Sept 1988.

"The controlled reduction model : a new parallel evaluation scheme for FP"
N.DEVESA, M.P.LECOUFFE, B.TOURSEL
Publication LIFL n° 70 - Mars 1989

"Un algorithme parallèle d'exécution de programmes FP"
N.DEVESA, M.P.LECOUFFE, B.TOURSEL
Journées "algorithmes parallèles et architectures nouvelles" - Toulouse - Du 13 au 17 Mars 1989

"Un modèle formel d'exécution parallèle de programmes FP"
N.DEVESA, M.P.LECOUFFE, B.TOURSEL
Journées AFCET-GROPLAN : "développement de programmes pour machines parallèles" - Chamonix, 17-19 Mai 1989 - à paraître dans BIGRE+GLOBULE.

REFERENCES BIBLIOGRAPHIQUES

[ABRAMS85] S. ABRAMSKY, R. SYKES
SECD-m : a virtual machine for applicative programming. Lecture Notes in Computer Science n° 201, pp. 81-98, Nancy, 1985.

[AMAMIY84] M. AMAMIYA, R. HASEGAWA
Dataflow computing and eager and lazy evaluations. New Generation Computing, Vol. 2, pp. 105-129, 1984.

[ANDERS87] M. ANDERSON, F. BERMAN
Removing useless tokens from a data-flow computation. Proceedings of the International Conference on Parallel Processing, pp. 614-617, August 1987.

[ARNAUD89] P. ARNAUD
Proposition d'une méthode de répartition de la charge sur un réseau de processeurs : conséquences sur la topologie et simulation.
Thèse de Doctorat Informatique, LIFL, Université de Lille I, Nov. 1989.

[AUGUST84] L. AUGUSTSSON
A compiler for lazy ML.
Proceedings of the ACM Symposium on LISP and Functional Programming, Austin, pp. 218-227, August 1984.

[BACKUS78] J.W. BACKUS
Can programming be liberated from the von Neumann style ? A functional style and its algebra of programs.
Communications of the ACM, Vol. 21, n° 4, pp. 613-641, August 1978.

[BACKUS86] J.W. BACKUS, J.H. WILLIAMS, E.L. WIMMERS
FL Language Manual (preliminary version).
Research Report RJ 5339 (IBM Almaden Research), 1986

[BAREND84] H.P. BARENDREGT
The Lambda Calculus : its syntax and semantics.
North Holland 2nd edition, p. 24, 1984

[BELLEG85] F. BELLEGARDE
Utilisation des systèmes de réécriture d'expressions fonctionnelles comme outils de transformation de programmes itératifs.
Thèse d'Etat, Université de Nancy I, 1985.

[BELLIA86] M. BELLIA, G. LEVI
The relation between logic and functional languages : a survey.
The Journal of Logic Programming, Vol. 3, n° 3, Oct. 1986.

[BELLOT85] P. BELLOT
JYM : un langage de programmation sans variables et ses réalisations.
Actes des journées AFCET-GROPLAN 1985, Bulletin BIGRE+GLOBULE, J. ANDRE ed., Toulouse, Juillet 1985.

[BELLOT86] P. BELLOT
Sur les sentiers du GRAAL, Etude, Conception et Réalisation d'un langage de Programmation sans Variables.
Thèse d'Etat (Paris VI), Rapport LITP 86-62, Paris 1986.

[BELLOT88a] P. BELLOT
La programmation fonctionnelle sans variables et les captures d'application.
Actes des journées AFCET-GROPLAN "Langages et Algorithmes" BIGRE+GLOBULE n° 59, Avril 1988.

[BELLOT88b] P. BELLOT, B. ROBINET
La programmation sans variables.
AFCET / Interfaces n° 68, pp. 3-11, Juin 1988.

[BERKLI75] K. BERKLING
Reduction languages for reduction machines.
Proceedings IEEE International Symposium on Computer Architecture, pp. 133-140, 1975.

[BOHM64] C. BOHM
The CUCH as a formal and description language, Formal language, Description Language
T. STEEL Ed.,1964.

[BURSTA80] R. BURSTALL, Mc QUEEN, D. SANELLA
HOPE : an experimental applicative language
LISP conference, Stanford, California, pp. 136-143, 1980.

[BURTON81] F.W. BURTON, M.R. SLEEP
Executing functional programs on a virtual tree of processors.
Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, pp. 187-194, 1981.

[CASTAN86a] M. CASTAN
Mécanismes de base pour la réduction parallèle.
BIGRE + GLOBULE n° 50, GROPLAN, 1986.

[CASTAN86b] M. CASTAN, G. DURRIEU, B. LECUSSAN, M. LEMAITRE
Toward the design of a parallel graph reduction machine : the MaRS Project.
Graph Reduction Workshop, Santa Fe, USA, September 1986.

[CASTAN87] M. CASTAN, M.H. DURAND, M. LEMAITRE
A set of combinators for abstraction in linear space.
Information processing letters, Vol. 24, n° 3, pp. 183-188, February 1987.

[CHURCH41] A. CHURCH
The calculi of lambda conversion.
Annals of Mathematics studies n° 6, Princeton University Press, 1941.

[CLARKE80] T.J.W. CLARKE, P.J.S. GLADSTONE, C.D. Mac LEAN, A.C. NORMAN
SKIM : The S, K, I reduction machine.
Proceedings of LISP Conference, Stanford University, Stanford, CA, pp. 128-135, August 1980.

[COMYN79] G. COMYN
Types et lambda-calcul - Syntaxe et sémantique du calcul.
GROPER n° 4, Département Informatique IUT "A" de LILLE I, Nov. 1979.

[COUSIN85] G. COUSINEAU, P.L. CURIEN, M. MAURRY
The Categorical Abstract Machine.
Functional Programming Languages and Computer Architecture, J.P. JOUANNAUD (Ed.), Lecture Notes in Computer Science 201, Berlin-Springer, pp. 50-64, 1985.

[COUSIN86] G. COUSINEAU, P.L. CURIEN
La Machine Abstraite Catégorique (CAM).
BIGRE + GLOBULE n° 50, GROPLAN, 1986.

[CURRY58] H.B. CURRY, R. PEYS
Combinatory logic, Vol. 1.
North HOLLAND, 1958

[DARLIN81] J. DARLINGTON, M. REEVE
ALICE : a multiprocessor reduction machine for the parallel evaluation of applicative languages.
Proceedings of the ACM Conference on Functional Languages and Computer Architecture, New Hampshire, pp. 65-75, October 1981.

[DARLIN82] J. DARLINGTON, P. HENDERSON, D.A. TURNER Eds.,
Functional programming and its applications.
Cambridge, England, Cambridge University Press, 1982.

[DAVIS78] A.L. DAVIS
The architecture and system method of DDM1 : a recursively structured data driven machine.
Proceedings of the fifth International Symposium on Computer Architecture, pp. 210-215, April 1978.

[DENNIS79] J.B. DENNIS
The varieties of data-flow computers
Proceedings IEEE International Conference on Distributed System, pp. 430-439, 1979.

[DENNIS84] J.B. DENNIS, G.R. GAO, K.W. TODD
Modelling the weather with a data-flow supercomputer.
IEEE Transactions on Computers, Vol. C-33, pp. 592-603, July 1984.

[DILLER88] A. DILLER
Compiling functional Languages.
John WILEY & SONS LTD, 1988.

[DURAND86] M.H. DURAND
Etude et évaluation du parallélisme dans les langages fonctionnels. Une approche de la réduction de graphe par les combinateurs.
Thèse n° 123, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, Juin 1986.

[FAIRBA86] J. FAIRBAIRN, S.C. WRAY
Code generation techniques for functional languages.
Proceedings of the ACM Conference on Lisp and Functional Programming, Boston, pp. 94-104, August 1986.

[FRIEDM78] D.P. FRIEDMAN, D.S. WISE
Aspects of applicative programming for parallel processing.
IEEE Transactions on Computers, Vol. C-27, pp. 289-296, April 1978.

[FURBAC86] U. FURBACH, S. HOLLDOBLER
Modelling the combination of functional and logic programming languages.
Journal of Symbolic Computation, Vol. 2, pp. 123-138, 1986.

[GLASER87] H. GLASER, C. HANKIN, D. TIU
Principes de programmation fonctionnelle.
MASSON 1987.

[GONCAL88a] G. GONCALVES, M.P. LECOUFFE, B. TOURSEL
A distributed associative network for a parallel reduction architecture.
IFIP WG103, Working Conference on Parallel Processing Pisa (Italy), 25-27 April 1988.

[GONCAL88b] G. GONCALVES, M.P. LECOUFFE, B. TOURSEL
A parallel reduction machine with a distributed content adressable memory.
MIMI 88, San Feliu (Espagne), 27-29 June 1988.

[GONCAL89] G. GONCALVES, I. HANNEQUIN, M.P. LECOUFFE, B. TOURSEL
Prolog : a new parallel evaluation scheme
Euromicro Congress, Cologne, Sept. 89.

[GORDON79] M. GORDON, R. MILNER, C. WADSWORTH
Edinburgh LCF.
Lecture Notes in Computer Science 78, Berlin-Springer, 1979.

[GURD85a] J.R. GURD, C.C. KIRKHAM, I. WATSON
The Manchester prototype dataflow computer
Communications of the ACM, Vol. 28, n°1, pp. 34-52, January 1985.

[GURD85b] J.R. GURD
The Manchester dataflow machine.
Computer Physics Communications, Vol. 37, n° 1, pp. 49-62, July 1985.

[HARPER86] R. HARPER, D. MACQUEEN, R. MILNER
Standard ML.
Edinburgh University, Internal Report ECS-LFCS-86-2, 1986.

[HARPER87] R. HARPER, R. MILNER, M. TOFTE
The semantics of standard ML.
Draft report, Edinburgh University, 1987.

[HENDER80] P. HENSERSON
Functional programming : application and implementation.
Englewood Cliffs, Prentice-Hall,1980.

[HINDLE86] J.R. HINDLEY, J.P. SELDIN
Introduction to combinators and λ-calculus.
Cambridge University Press,1986.

[HUDAK84] P. HUDAK, D. KRANZ
A combinator based compiler for a functional language.
Proceedings of the 11th ACM Symposium on Principles of Programming Languages, pp. 122-132, January 1984.

[HUDAK85] P. HUDAK, B. GOLDBERG
Distributed execution of functional programs using serial combinators.
IEEE Transactions on Computer, Vol. C-34, n° 10, October 1985.

[HUGHES82] J. HUGHES
Supercombinators : a new implementation method for applicative languages.
Proceedings of the 1982 ACM Conference on LISP and Functional Programming, pp. 1-10, August 1982.

[JACQUE88] F. JACQUENET
Contribution à l'étude de concepts et de mécanismes de langages avancés pour la programmation logico-fonctionnelle : le langage VEGA.
Thèse n° 85, Faculté des Sciences et Techniques de l'Université de Bourgogne, Décembre 1988.

[JOHNSO80] D. JOHNSON & Al
Automatic partitioning of programs in multiprocessors systems.
Proceedings IEEE COMPCON, pp. 175-178, 1980.

[JOHNSS84] T. JOHNSSON
Efficient compilation of lazy evaluation.
Proceedings of the ACM Symposium of Compiler Construction, Montreal, pp. 58-69, June 1984.

[KELLER79] R.M. KELLER, G. LINDSTROM, S. PATIL
A loosely-coupled applicative multiprocessing system.
AFIPS Conference Proceedings, pp. 613-622, June 1979.

[KELLER84] R.M. KELLER, F.C.H. LIN, J. TANAKA
Rediflow multiprocessing.
Proceedings IEEE COMPCON, pp. 410-417, February 1984.

[KELLER85] R.M. KELLER
Rediflow Architecture Prospectus.
UUCS-85-105, Department of Computer Science, University of Utah, August 1985.

[KIEBUR81] R.B. KIEBURTZ, J. SHULTIS
Transformations of FP programs schemes.
Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, pp. 41-48, 1981.

[LANDIN64] P.J. LANDIN
The mechanical evaluation of expressions.
Computer Journal, Vol. 6, pp. 308-320, 1964.

[LEGRAN88] R. LEGRAND
Extension du langage GRAAL vers le calcul relationnel avec indéterminées.
Actes des journées AFCET-GROPLAN "langages et Algorithmes", BIGRE + GLOBULE n°59, Avril 1988.

[LEMETA84] D. LEMETAYER
Etude et réalisation de schémas d'évaluation parallèle pour langages fonctionnels.
Thèse, Série A, n° d'ordre 175, n° de série 36, Université de Rennes 1, Mars 1984.

[McCART60] J. Mc.CARTHY
Recursive functions of symbolic expressions and their computation by machine, Part I.
Communications of the ACM, Vol. 3, n°4, 1960

[McCART62] J. Mc CARTHY, P.W. ABRAHAMS, D.J. EDWARDS, T.P. HART, M.I. LEVIN
LISP 1.5 Programmer's Manual
MIT Press, 1962.

[MAGO79a] G.A. MAGO
A network of microprocessors to execute reduction languages - Part I.
International Journal of Computer and Information Sciences, Vol. 8, n° 5, pp. 349-385, 1979.

[MAGO79b] G.A. MAGO
A network of microprocessors to execute reduction languages, Part II.
International Journal of Computer and Information Sciences, Vol. 8, n° 6, pp. 435-471, 1979.

[MAGO84] G.A. MAGO, D. MIDDLETON
The FFP machine, a progress report.
Proceedings of the International Workshop on High Level Computer Architecture, Los Angeles, CA, May 21-25, 1984.

[MAGO85] G.A. MAGO
Making parallel computation simple : the FFP machine
Spring COMPCON, 1985.

[MARTI89] J.C. MARTI
Simulation du réseau N-ARCH.
Publication Interne n° 65, LIFL, Université de LILLE I, 1989.

[MAUNY86] M. MAUNY, A. SUAREZ
De la machine abstraite catégorique à l'implémentation de langages fonctionnels.
BIGRE + GLOBULE n° 50, GROPLAN, pp. 164-175, 1986.

[MYCROF81] A. MYCROFT
Abstract interpretation and optimising transformations for applicative programs.
Thesis, University of Edinburgh, December 1981.

[NIAR89] S. NIAR
Contribution à l'étude des architectures d'ordinateurs parallèles. Structure de la machine N-ARCH et émulation sur un réseau de transputers.
Thèse de Doctorat Informatique, Université de Lille I, Octobre 1989.

[NICOLA89a] J.C. NICOLAS
Une technique de hachage adaptée à la répartition d'une relation sur un réseau de processeurs et son utilisation dans un algorithme de jointure parallèle.
Publication Interne n° 68, LIFL, Université de Lille I, 1989.

[NICOLA89b] J.C. NICOLAS
Répartition d'un schéma logique d'objets complexes sur un réseau de processeurs.
Publication Interne n° 73, LIFL, Université de Lille I, 1989

[ONO86] S. ONO, N. TAKAHASHI, M. AMAMIYA
Optimized demand-driven evaluation of functional programs.
Proceedings of the 1986 International Conference on Parallel Processing, pp 421-428, 1986.

[PEYTON87a] S.L. PEYTON JONES
The implementation of functional programming languages.
Prentice Hall International Series in Computer Science, 1987.

[PEYTON87b] S.L. PEYTON JONES, C. CLARK, J. SALKILD, M. HARDIE
GRIP : A high performance architecture for parallel graph reduction.
Functional Programming languages and Computer Architecture, Gilles KAHN (Ed.), Lecture Notes in Computer Science n° 274, Springer Verlag, Portland Gregon, USA, Sept. 1987.

[REES82] J.A. REES, N.I. ADAMS
T : a dialect of Lisp.
Proceedings of the ACM Symposium on LISP and Functional programming, pp. 114-122, August 1982.

[RUGGIE87] C.A. RUGGIERO, J. SARGEANT
Control of parallelism in the Manchester data-flow machine.
Functional Programming Languages and Computer Architecture, Gilles KAHN (Ed.), Lecture Notes in Computer Science n° 274, Springer-Verlag, Portland Gregon, USA, Sept. 1987.

[SCHONF24] M. SCHONFINKEL
Uber die Bausteine der Mathematischen logik.
Mathematische Annalen, Vol. 92, pp. 305-316, 1924.
et aussi dans
From Frege to Gödel : a source book in mathematical logic 1879 - 1931.
J. van Heijenoorf, Harvard University Press, 1967.

[SLEEP80] M.R. SLEEP, F.W. BURTON
Towards a zero assignment parallel processor.
Proceedings of the Second International Conference on Distributed Computing Systems, 1980.

[SRINI86] V.P. SRINI
An architectural comparison of data-flow systems.
IEEE Computer, Vol. 19, pp. 68-88, 1986.

[STEELE78] G.L. STEELE, G.J. SUSSMAN
The revised report on SCHEME.
AI MEMO 452, MIT, January 1978.

[TARBOU70] J.C. TARBOURIECH
Un programme de simulation de circuits logiques.
Colloque International sur la Microélectronique avancée, Paris 1970.

[TRELEA82] P.C. TRELEAVEN, BROWN BRIDGE, R.P. HOPKINS
Data-driven and demand-driven computer architecture.
Computing Surveys, Vol. 14, n° 1, March 1982.

[TURNER76) D.A. TURNER
SASL Reference Manual.
University of St Andrews, Technical Report, 1976.

[TURNER79a] D.A. TURNER
A new implementation technique for applicative language.
Software Practise and Experience, Vol. 9, January 1979.

[TURNER79b] D.A. TURNER
Another algorithm for bracket abstraction.
Journal of Symbolic Logic, Vol. 44, n° 2, 1979.

[TURNER82] D.A. TURNER
Recursion equations as a programming language.
Functional programming and its applications, Cambridge. University Press, J. DARLINGTON, P. HENDERSON, D.A. TURNER (Eds), 1982.

[TURNER85] D.A. TURNER
MIRANDA : a non strict functional language with polymorphic types.
Functional Programming Languages and Computer Architecture, LNCS 201, Springer-Verlag, 1985.

[VEEN86] A.H. VEEN
Dataflow machine architecture.
ACM Computing Surveys, Vol. 18, pp. 365-396, December 1986.

[VEGDAH84] S.R. VEGDAHL
A survey of proposed architectures for the execution of functional languages.
IEEE Transactions on Computers, Vol. C-33, n° 12, December 1984.

[WADLER81] P.L. WADLER
Applicative style programming, program transformation and list operators.
Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, pp. 25-32, 1981.

[WATSON79] I. WATSON, J. GURD
A prototype data-flow computer with token labelling.
Proceedings AFIPS Nat. Comput. Conf., Vol. 48, pp. 623-628, 1979

[WATSON82] J. WATSON, J. GURD
A practical data-flow computer.
IEEE Computer, Vol.15, pp. 51-57, February 1982.

[WATSON87] P. WATSON, I. WATSON
Evaluating functional programs on the FLAGSHIP machine.
Functional Programming and Computer Architecture, Gilles KAHN (Ed.), Lecture Notes in Computer Science n° 274, Springer-Verlag, Portland Gregon, USA, September 1987.

[WEI88] Y.H. WEI, J.L. GAUDIOT
Demand-driven interpretation of FP programs on a data-flow multiprocessor.
IEEE Transactions on Computers, Vol. 37, n° 8, August 1988.

[WIKSTR87] A. WIKSTROM
Functional programming using standard ML.
Prentice Hall International Series in Computer Science, C.A.R. HOARE series Editor, 1987.

[WILLIA82] J.H. WILLIAMS
Notes on the FP style of functional programming.
Functional Programming and its Applications, Cambridge University Press, Ed. by J. DARLINGTON, P. HENDERSON, D.A. TURNER, 1982.

[WIRSIN87] M. WIRSING, D. SANELLA
Une introduction à la programmation fonctionnelle : HOPE et ML.
Technique et Science Informatiques, Vol. 6, n° 16, 1987.

Résumé :

Dans le cadre du projet N-ARCH ayant pour objectif, la conception d'une machine parallèle, adaptée à l'exécution de programmes déclaratifs, nous nous sommes intéressés à l'évaluation parallèle du langage FP, le langage fonctionnel sans variable défini par Backus. Un programme FP peut être assimilé à une expression combinatoire qui, dans la plupart des implantations, est exécutée selon le modèle de réduction de graphe présentant, à notre avis, un certain nombre d'inconvénients. C'est pourquoi nous avons défini un nouveau modèle d'évaluation parallèle du langage FP.

Dans ce modèle, on a d'une part la représentation du programme source en un ensemble d'arborescences et d'autre part, la représentation de l'argument (la séquence argument). Une exploration parallèle des arborescences est définie formellement par un graphe de transition d'état des noeuds de l'arborescence, qui peut être décrit par un système de réécriture. On peut alors en déduire un algorithme d'exploration parallèle par envois et réceptions de messages. Cette exploration permet d'envoyer en parallèle des messages à la séquence argument, générant des réductions en parallèle sur la séquence argument, dont la sémantique est également donnée par des règles de réécriture. L'ensemble du modèle peut alors être décrit par un ensemble de processus communiquant par messages.

Le modèle défini ne peut être considéré ni comme un modèle data-flow, ni comme un modèle de réduction au sens classique du terme : l'exploration parallèle des arborescences joue le rôle de "moteur d'exécution" ou de "distributeur de tâches" générant des réductions en parallèle sur la séquence argument.

Mots-clés :

langages fonctionnels, langages sans variable, parallélisme, réduction, schéma d'évaluation