Numéro d'ordre :

Année : **1994**

LIFL *Laboratoire d'Informatique Fondamentale de Lille*

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Nadia BENNANI

Proposition d'un modèle d'évaluation parallèle des langages fonctionnels sans variables

Thèse soutenue le 17 Novembre 1994, devant la commission d'examen :

J.-M. Geib	Professeur	Université de Lille I, LIFL (*président*)
P. Bellot	Professeur	ENST de Paris (*rapporteur*)
P. Jouvelot	Chargé de recherche	Ecole des Mines de Paris (*rapporteur*)
R. Pino-perez	Maître de Conférences	EUDIL (*examinateur*)
N. Devesa	Maître de Conférences	EUDIL (*codirecteur de thèse*)
M.P. Lecouffe	Maître de Conférences	Université de Lille1, LIFL (*codirecteur de thèse*)
B. Toursel	Professeur	EUDIL, LIFL(*directeur de thèse*)

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

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. PARREAU, J. LOMBARD, M. MIGEON. J. CORTOIS. A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
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	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
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 électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
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. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Je remercie

- Monsieur **Jean Marc Geib**, professeur à l'université de Lille1 et directeur du LIFL, d'avoir accepté la pr'esidence du jury de cette thèse.
- Monsieur **Patrick Bellot**, Professeur à l'ENST de Paris, qui m'a fait l'honneur d'être l'un des rapporteurs de cette thèse. Ses remarques encourageantes ont renforcé ma motivation.
- Monsieur **Pierre Jouvelot**, Professeur à l'école des mines de Paris pour sa participation au jury en tant que rapporteur. Ses critiques et observations ont permis de corriger ce document.

Je remercie tout particulièrement

- Monsieur **Bernard Toursel**, mon directeur de thèse, de m'avoir fait confiance en m'accueillant dans son équipe. Ses conseils et ses remarques, tout au long de la réalisation de ce travail m'ont été très précieux.

Mes remerciements vont également à

- monsieur **Ramon Pino Perez**, Maitre de conférences à l'EUDIL, pour son réel intérêt pour mes travaux. Ses suggestions et ses conseils ont largement contribué à l'avancée de mes travaux.
- mademoiselle **Nathalie Devesa** qui a suivi de très près mon travail. Je la remercie pour ses conseils enrichissants. Ses encouragements et son amitié m'ont beaucoup apporté.
- madame **Marie Paule Lecouffe**, Maitre de conférences à l'Université de Lille1, qui a coencadré ce travail. Je la remercie pour sa disponibilité et pour la lecture particulièrement attentive de ce document.

Mes remerciements s'adressent aussi

- aux membres de l'équipe Paloma.
- aux membres du personnel technique et du personnel administratif qui sont avant tout mes amis, pour leur bonne humeur et leur sympathie.
- à tous mes amis, je pense à tous ceux qui m'ont témoigné leur soutien et leur sympathie tout au long de la réalisation de cette thèse, et ils sauront se reconnaître.
- à mes collègues de l'IUT de Lens. Je vous remercie d'avoir réaménagé mon emploi du temps durant les dernières semaines, ce qui m'a permis de me consacrer entièrement à la préparation de la soutenance.
- à Monsieur Henri Glanc qui a imprimé cette thèse pour la qualité et le sérieux de son travail.

Enfin, mes remerciements vont à ma famille pour son soutien et son encouragement, et plus particulièrement à mes parents pour tous leurs efforts. Ce travail leur est dédié.

Table des matières

Liste des figures

Introduction

Plusieurs décennies de recherches et de développement visant à perfectionner les ordinateurs autour du concept originel de von Neuman ont permis d'offrir à l'utilisateur des machines dont les performances brutes ne cessent de s'améliorer pour des prix de plus en plus faibles.

Cependant l'obtention de performances réelles d'exécution nécessite une programmation visant à optimiser l'utilisation des ressources de la machine. Ceci se fait au prix d'un effort de la part du programmeur qui doit accorder autant d'importance à l'élaboration optimisée de son algorithme qu'aux détails d'implantation de l'algorithme sur machine. La cause de cet état de fait est l'utilisation des langages impératifs où la programmation s'inspire du style "Von Neumann" i.e où le programme est une suite de commandes exécutées par la machine. L'efficacité dépend donc beaucoup de l'approche utilisée pour le développement du programme.

Les langages fonctionnels, contrairement aux langages impératifs, permettent une programmation plus "propre" en ce sens que les programmes sont développés à partir de concepts formels et non à partir de considérations liées à l'implantation. Les langages fonctionnels expriment ainsi l'intention du programme i.e ce qu'il est sensé calculer.

L'exécution des programmes fonctionnels dans le "contexte von Neumann" qui ne leur est pas adapté, entraîne une chute des performances d'exécution, en comparaison avec les langages impératifs. C'est une des raisons qui entraînent la diminution de la crédibilité des langages fonctionnels. Les travaux de recherche des vingt dernières années ont prouvé le contraire, par l'élaboration de machines-langages performantes mais aussi par la définition de compilateurs de code puissants. Ceci permet l'amélioration des performances obtenues tout en adoptant une programmation déclarative, qui se rapproche plus d'une formulation mathématique.

Les langages fonctionnels se caractérisent par leur puissance d'expression et par l'expression d'un parallélisme intrinsèque. Cette dernière caractéristique traduit la liberté quant à l'ordre d'évaluation d'un programme fonctionnel. Il existe donc

plusieurs façons d'évaluer un même programme, en séquentiel mais aussi en parallèle. Ces stratégies d'évaluation sont à l'origine des schémas d'évaluation définis pour l'exécution des langages fonctionnels, et dont le plus connu et le plus adapté à l'évaluation des langages fonctionnels est le modèle de réduction de graphe.

La réduction de graphe est basée sur la représentation de l'expression fonctionnelle sous forme d'un graphe syntaxique contenant des nœuds intermédiaires appelés *nœuds application*. Ces nœuds symbolisent les sous-expressions réductibles. L'évaluation de l'expression s'effectue en transformant le graphe, en remplaçant chaque sous-graphe réductible par le graphe résultat, jusqu'à l'obtention d'un graphe irréductible.
La réduction de graphe présente cependant quelques inconvénients se résumant en un parcours pénalisant des graphes syntaxiques qui retarde l'évaluation des expressions fonctionnelles. En outre, la représentation statique des redex, par la présence des nœuds application, dès la phase initiale de l'évaluation, alourdit la gestion des nœuds application intermédiaires.

C'est dans le but de pallier ces inconvénients que le modèle P^3 a été défini par N.Devesa, au sein de l'équipe PALOMA au LIFL [Dev90]. Ce modèle s'apparente aux modèles de réduction, tout en proposant une représentation différente de l'expression fonctionnelle par l'utilisation de deux structures : les arborescences fonctionnelles et les arbres de données. L'évaluation des expressions est basée sur un mécanisme d'exploration des arborescences fonctionnelles qui génère des ordres de réduction pris en compte par les arguments. La production et la prise en compte des ordres de réductions sont deux activités asynchrones ce qui permet l'accélération de l'évaluation.
Le modèle P^3 a été défini pour l'évaluation d'expressions fonctionnelles sans variables dans le langage FP de J.W. Backus [Bac78].

- L'objectif principal de ce travail est d'étendre le modèle P^3 à l'évaluation d'expressions fonctionnelles sans variables quelconques en intégrant les fonctions d'ordre supérieur. Le modèle étendu doit être aussi performant sinon plus performant que le modèle de réduction de graphe et doit offrir les mêmes possibilités d'exécution.

- Le deuxième objectif de cette thèse est de valider le fonctionnement du modèle P^3 et mettre en évidence ses possibilités ainsi que son parallélisme potentiel.

- Enfin, le troisième objectif est d'introduire un aspect plus pratique du modèle P^3 qui concerne l'étude des problèmes de répartition qui est un deux-

ième axe de recherche actuellement à l'étude au sein de notre équipe et dont quelques aspects sont étudiés dans le cadre de cette thèse.

Structure de la thèse

Cette thèse est donc composée de 3 parties.

- La première partie décrit le contexte du travail proposé. Elle se divise en trois chapitres.

  Dans le chapitre 1, nous évoquons les propriétés mathématiques des langages fonctionnels, qui sont à l'origine d'une programmation fonctionnelle déclarative, puis nous présentons les 3 classes de langages fonctionnels : les λ-langages, basés sur la théorie du λ-calcul, les langages de combinateurs et les langages fonctionnels sans variables, basés sur la logique combinatoire. Dans la catégorie des langages sans variables, nous distinguons les concepts communs à tous les langages fonctionnels sans variables et les concepts évolués. L'utilisation des fonctions d'ordre supérieur figure parmi cette dernière catégorie de concepts.

  Le chapitre 2 présente le langage GRAAL, un exemple de langage fonctionnel sans variables possédant les concepts évolués des langages fonctionnels sans variables, que nous choisissons comme langage support pour la conception du modèle P^3 étendu.

  Avant de présenter le modèle P^3 étendu, le chapitre 3 fait le point sur le modèle P^3 originel et sa situation par rapport aux autres modèles d'évaluation existants. Les objectifs de l'extension sont détaillés à la fin de ce chapitre.

- La deuxième partie décrit formellement le modèle P^3 étendu. Elle se compose des chapitres 4 et 5.

  Dans le chapitre 4, nous présentons la fonction de représentation des expressions fonctionnelles à partir de la représentation des objets les composant. Le système de réécriture décrivant les mécanismes d'exploration et de réduction permettant l'évaluation des expression fonctionnelles est ensuite proposé. Le modèle ainsi décrit prend en compte les fonctions d'ordre

supérieur et permet l'exploitation, sous toutes ses formes, du parallélisme des langages fonctionnels. L'exploration des arborescences fonctionnelles proposée dans ce chapitre, contrairement au modèle P^3 originel, permet l'anticipation de l'application d'une fonction à ses arguments.

Le chapitre 5 présente une optimisation du système de réécriture qui conserve toutes les caractéristiques du modèle P^3 étendu. Dans ce chapitre est proposé également un algorithme d'ordonnancement des requêtes de réduction qui permet, contrairement à celui proposé dans la première version du modèle, de classer les requêtes de réduction sans besoin de synchronisation lors de l'exploration.

- La troisième partie permet la validation ainsi que l'étude préliminaire des problèmes de répartition qui se posent pour le modèle P^3. Elle se compose des chapitres 6 et 7.

  Dans le chapitre 6, le modèle P^3 étendu est comparé au modèle de réduction de graphe, en se basant, d'une part, sur les coûts de gestion des deux modèles; cette analyse des coûts montre le gain dans le modèle P^3 sur le coût de parcours et le coût d'occupation mémoire. D'autre part, cette comparaison porte sur les possibilités offertes par les deux modèles; la comparaison s'établit sur les points suivants : le type d'expression évaluée, le parallélisme exploité, les modes d'évaluation possibles et la possibilité du partage.

  Une simulation du modèle P^3 a été réalisée. Le but en est de valider le fonctionnement du modèle, de mesurer le parallélisme potentiel du modèle P^3 et d'aborder les problèmes de placement de données dans le modèle P^3. Cette simulation met donc en évidence la nature très dynamique de ces structures. Les résultats théoriques de cette simulation, l'introduction aux problèmes de placement et une méthode de placement dynamique des copies d'arguments contribuant à l'amélioration du placement dynamique des données sont présentés au chapitre 7.

  Le chapitre 8 est la conclusion de cette thèse. Dans ce chapitre, nous montrons quelles sont les modifications apportées au modèle originel pour permettre l'extension. Les perspectives futures de ce travail sont également abordées.

Partie I

Classification des langages fonctionnels

Chapitre 1

Les langages fonctionnels: Généralités

Les langages fonctionnels sont des langages informatiques basés sur la notion de fonction au sens mathématique du terme. En effet, concevoir une solution informatique pour un problème donné, dans un langage fonctionnel, consiste à définir un ensemble de fonctions. La figure 1.1 permet de situer les langages fonctionnels par rapport aux autres langages informatiques. Nous pouvons distinguer 3 grandes classes de langages : les langages *impératifs*, les langages *déclaratifs* et les langages *à objets*. Les langages fonctionnels au même titre que les langages logiques et plus récemment les langages logico-fonctionnels sont des langages déclaratifs i.e ce sont des langages où la façon de concevoir le programme ne traduit pas l'ordre dans lequel il s'exécutera.
Les langages fonctionnels possèdent des propriétés mathématiques intéressantes. Ils sont naturellement parallèles du fait de l'absence d'effets de bord : la chronologie adoptée pour évaluer les sous expressions fonctionnelles n'a aucune importance. Cette indépendance offre la possibilité d'un traitement parallèle. Nous distinguons 3 sous-classes de langages fonctionnels : Les λ-langages, les langages de combinateurs et les langages fonctionnels sans variable. Ce chapitre est donc consacré à la description de ces différentes classes de langages. Il est organisé comme suit : les sections 1 et 2 présentent respectivement les propriétés mathématiques et le parallélisme des langages fonctionnels. La section 3 traite des caractéristiques générales de la programmation fonctionnelle. Les sections 4, 5

et 6 décrivent respectivement les λ-langages, les langages de combinateurs et les langages fonctionnels sans variable.

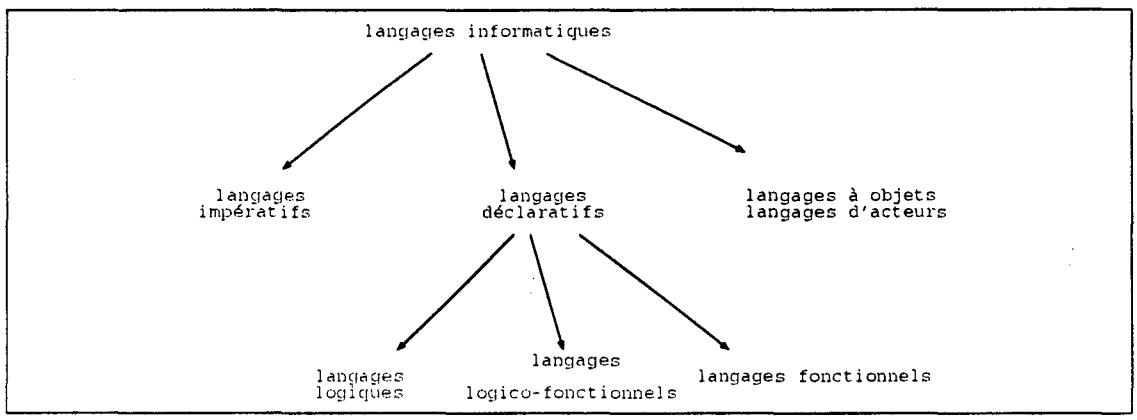


Figure 1.1 : Classification des langages informatiques

1 Caractéristiques générales des langages fonctionnels

1.1 Propriétés mathématiques

La fonction est une notion mathématique fondamentale. Elle permet d'associer à chaque élément de son domaine d'application une image unique qui est le résultat de l'application de la fonction à un élément donné. Soit f une fonction, l'élément auquel f associe une image est appelé *argument* de f.
La composition fonctionnelle (notée o) permet de combiner des fonctions afin d'obtenir d'autres fonctions.
Grâce à ces deux notions, la fonction et la composition fonctionnelle, on peut écrire des fonctions de complexité diverse. Le résultat obtenu est une fonction qui, appliquée à ses arguments $x_1\ x_2\ \ldots\ x_n$, constitue une expression fonctionnelle de la forme $f : x_1\ x_2\ \ldots\ x_n$. Dans un langage fonctionnel, tout programme est une expression fonctionnelle au sens ci-dessus.

Outre la notion de fonction, les langages fonctionnels se caractérisent par des propriétés mathématiques intéressantes dont la notion de *transparence référentielle* [Bac78] et la vérification du théorème de Church-Rosser [Chu41].

- Dans un langage fonctionnel, le résultat de l'application d'une fonction aux

mêmes arguments donne toujours le même résultat. Le résultat d'une application ne dépend donc pas d'un contexte d'évaluation particulier. C'est la propriété de *transparence référentielle*. Cette propriété traduit l'absence d'effet de bord dans l'évaluation des expressions fonctionnelles. On ne peut pas en dire autant des langages impératifs. En voici un exemple :

```
Fonction f(x:entier):entier;
debut
f:= x+y;
y:=y+1;
fin;
```

où le programme appelant se présente comme suit :

```
...
y:=0;
z:= f(1);
z:= f(1);
...
```

Cet exemple illustre l'utilisation successive d'une même fonction *f*, avec les mêmes paramètres, dans un programme quelconque.

Le premier appel à la fonction *f*, dans le programme principal retourne la valeur 1. Le deuxième appel retourne la valeur 2. Le résultat obtenu pour la deuxième utilisation dépend de celui obtenu au premier appel. Il en découle que l'ordre d'exécution est fixe et ne peut donc s'effectuer en parallèle.

- Les langages fonctionnels vérifient le théorème de Church-Rosser i.e quelle que soit la stratégie d'évaluation d'une expression fonctionnelle, le résultat obtenu est unique.

Les deux propriétés citées ci-dessus ont les conséquences suivantes :

- Les preuves de programmes fonctionnels i.e les preuves qui démontrent l'adéquation du programme au cahier des charges, grâce au concept de fonction et aux propriétés citées ci-dessus, se font de façon plus naturelle que dans d'autres langages notamment les langages impératifs. Ceci est vrai parce que le programme fonctionnel fait intervenir les notions de fonction et de composition de fonctions qui sont déjà des outils mathématiques.
- Ces deux propriétés garantissent également la cohérence du résultat dans un contexte d'évaluation parallèle par rapport à une exécution séquentielle.

1.2 Parallélisme

Les langages fonctionnels sont naturellement parallèles. Ce parallélisme est exploitable pour les raisons évoquées dans le paragraphe précédent. Nous distinguons deux formes de parallélisme dans les langages fonctionnels : le parallélisme *horizontal* et le parallélisme *vertical.*

- Le parallélisme horizontal provient de la possibilité d'évaluer simultanément les arguments d'une fonction.
- Le parallélisme vertical provient de la possibilité d'appliquer une fonction à des arguments non encore évalués ou dont l'évaluation est en cours.

Pour illustrer ces deux types de parallélisme, nous proposons l'exemple suivant où l'expression à évaluer est la suivante :

$$f \quad exp_x \quad exp_y$$

exp_x, exp_y sont des expressions fonctionnelles quelconques et où la fonction f est définie comme suit :

$$\text{f x y} \equiv \text{if (g y) then x+1 else x}$$

Le parallélisme horizontal dans cet exemple provient de l'évaluation parallèle des expressions exp_x et exp_y

L'évaluation en parallèle de exp_x et $(g\ y)$ est un exemple de parallélisme vertical. Le parallélisme vertical est plus couramment désigné par l'*anticipation sur l'évaluation de la fonction.*

Remarques
Certains langages fonctionnels permettent au programmeur de contrôler le parallélisme par l'ajout d'annotations dans le programme.
Le style de programmation et le choix de l'algorithme contribuent également à faire apparaître le parallélisme dans les programmes fonctionnels.

1.3 Conception des programmes

Les langages fonctionnels permettent la conception de programmes complètement indépendants d'un modèle de machine physique. Cette indépendance se traduit par les deux aspects suivants :

- Ecrire un programme dans un langage fonctionnel consiste à écrire une série de fonctions chacune correspondant à une fonctionalité de l'algorithme. Les fonctions écrites ne font référence à aucun détail technique i.e à ce niveau de la conception, on ne s'occupe pas du choix de la machine cible, de l'occupation mémoire...etc.

- Les variables utilisées dans un programme servent uniquement à **nommer** les arguments, contrairement aux langages impératifs où la variable symbolise un emplacement mémoire destiné à recevoir une valeur. Les variables utilisées dans les langages fonctionnels sont appelées *variables de discours.*

Dans la section précédente, nous avons évoqué les caractéristiques générales communes à tous les langages fonctionnels. Il existe également des caractéristiques propres à chaque langage. Pour en discuter, nous pouvons d'ores et déjà regrouper les langages fonctionnels en trois classes de langages : les lambda-langages, les langages de combinateurs et les langages fonctionnels sans variable. La différence fondamentale entre ces types de langages est leur base théorique. La base théorique des lambda-langages est le *lambda-calcul* tandis que celle des langages sans variable et des langages de combinateurs est la logique combinatoire.
Dans la section suivante, nous aborderons les caractéristiques, avantages et inconvénients de chaque classe de langages.

2 Les lambda-langages

Ces langages sont tous issus de la théorie du lambda-calcul. Avant de présenter quelques exemples de lambda-langages, rappelons brièvement les principes du lambda-calcul.

2.1 Le lambda-calcul

Fondée par Church en 1930 [Chu41], cette théorie fut reconnue comme un des plus puissants modèles de calculabilité qui existent. Dans ce modèle, une fonction est construite par abstraction d'une variable dans une expression. L'expression résultante porte le nom de **λ-expression**. En concaténant une λ-expression et une suite d'arguments, on obtient une expression fonctionnelle évaluée par application d'un algorithme de réduction jusqu'à l'obtention d'une expression irréductible.

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

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

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

- $S \longrightarrow \alpha$	$\alpha \in W$	
- $S \longrightarrow (SS)$		règle dite d'application
- $S \longrightarrow \lambda v.S$	$v \in V$	règle dite d'abstraction

2.1.1 La syntaxe du λ-calcul

Le λ-calcul est un modèle de calculabilité. Sa sémantique peut être décrite principalement à l'aide des deux règles de conversion suivantes : la β-conversion et l'α-conversion que nous détaillons dans ce paragraphe. Pour cela, nous introduisons les définitions suivantes :

Variables libres et variables liées
Soit E et F deux λ-expressions, on peut définir inductivement les variables libres (Varlib) et liées (Varlie) de ces expressions de la façon suivante :

Varlib(x)	=	{x}
Varlib(E F)	=	Varlib(E) ∪ Varlib(F)
Varlib(λx.E)	=	Varlib(E) - {x}

Varlie(x)	=	∅
Varlie(E F)	=	Varlie(E) ∪ Varlie(F)
Varlie(λx.E)	=	Varlie(E) ∪ {x}

Pour une λ-expression, une variable libre peut être assimilée à une variable globale. Une variable liée peut être assimilée à une variable locale.

La substitution
Soit M une λ-expression quelconque. [N/x]M dénote la substitution de toutes les occurences libres de la variable x dans M par N. La substitution est définie par :

[N / x] x	=	N

$$
\begin{array}{lcl}
[\ N\ /\ x\ ]\ a & = & a\ (\text{si } a \neq x \text{ et } a \in W) \\
[\ N\ /\ x\ ](M1\ M2) & = & ([N\ /\ x]M1)\ ([\ N\ /\ x\ ]M2) \\
[\ N\ /\ x\ ]\ (\lambda v.X) & = & \lambda z([N\ /\ x]([\ z/\ v]\ X))
\end{array}
$$

où z est une nouvelle variable : en particulier $z \neq x$ et $z \notin$ Varlib(N).

2.1.2 La β-conversion

définition
On appelle β-radical (ou redex) toute expression de la forme (λx. M)N. La règle de β-réduction consiste à remplacer tout rédex de cette forme par la substitution [N/x] M. On note :

$$(\lambda x.M)N \xrightarrow{conv_\beta} [N/x]M$$

L'opération inverse de la β-réduction consiste à retrouver le terme de gauche à partir de celui de droite. Cette transformation est appelée règle de la β-abstraction. Une β-conversion est, d'après [Pey87], soit une β-réduction soit une β-abstraction.

2.1.3 La α-conversion

définition
On appelle α-radical toute expression de la forme (λy.X) . La règle de l'α-conversion consiste à remplacer cet α-radical par λv.([v/y]X) avec v $\notin$Varlib(X). On note :

$$(\lambda y.X) \xrightarrow{conv_\alpha} \lambda\ v.([v/y]X)$$

2.2 Exemples de λ-langages

Etant données la puissance et la simplicité du λ-calcul, de nombreux langages basés sur cette théorie ont été définis dont LISP, MIRANDA, SASL, KRC, HOPE, ML, CAML, HASKELL etc .... Nous donnons dans la suite de cette section, les principales caractéristiques des langages LISP, KRC, HOPE, ML et HASKELL.

2.2.1 LISP

Créé par John Mc CARTHY [MAE+62] au début des années 60, le langage LISP est un des λ-langages les plus populaires. Le but initial était de concevoir un langage dont les entités de base sont des entités mathématiques de manière à pouvoir y inclure la théorie des fonctions récursives développée pendant les 20 ou 30 années précédentes. LISP est un langage naturellement polymorphique : tous les objets, y compris les fonctions, sont représentés de manière identique ce qui permet la manipulation des fonctions comme des objets quelconques, d'où la possibilité d'utiliser des fonctions d'ordre supérieur.
Les premières versions de LISP offraient un jeu très réduit de fonctions primitives : CONS, CAR, CDR, EQ et ATOM. Cependant, petit à petit, le langage LISP a été enrichi par des fonctions à effet de bord telles que l'affectation, les fonctions de gestion mémoire et les fonctions de contrôle de l'évaluation et a donc perdu ses propriétés fonctionnelles.

Le langage LISP a une sémantique stricte : une fonction ne peut être évaluée que si ses arguments sont évalués. L'évaluation peut être volontairement inhibée par la fonction *quote* notée '.

2.2.2 KRC

Le langage KRC, développé par D. TURNER [Tur82], est basé sur une représentation des fonctions comme un ensemble d'équations. Un exemple d'expression à évaluer est donné ci-dessous :

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

Dans cet exemple l'équation représente les couples (x,y) où x appartient à l'intervalle [0..2] et y à l'intervalle [-2,2] et où x est inférieur à y. L'évaluation de cette expression a pour résultat, l'ensemble des couples (0 1) (0 2) (1 2).
Les particularités du langage KRC sont d'abord la possibilité de manipuler des données infinies grâce à la notation sous forme d'intervalle des ensembles de données. L'évaluation de ces expressions peut alors se faire de façon paresseuse. La deuxième particularité est la possibilité d'utiliser les fonctions d'ordre supérieur.

2.2.3 Les langages HOPE et ML

Ces deux langages ont été développés respectivement par R. BURSTALL & Al dans [BMS80] et GORDON dans [GMW79]. Conçu au départ pour servir de

méta-langage pour le projet LCF (Logic for Computable Functions) destiné à la construction de preuves formelles de fonctions récursives, ML fut redéfini en reprenant les principaux concepts du langage HOPE. Cette nouvelle version de ML [HMM86] porte le nom de "Standard ML".

ML et HOPE possèdent donc des propriétés communes qui ont été étudiées dans [WS87] :

- Ils se caractérisent par un aspect déclaratif des programmes écrits.
- Il s'agit de langages typés.
- Les fonctions sont définies inductivement par des équations récursives sur les constructeurs des types de données.
- Ils permettent tous deux la manipulation des fonctions d'ordre supérieur.

2.2.4 Le langage HASKELL

Haskell [HF92]- baptisé du prénom de H.B.CURRY- est un langage fonctionnel défini en 1987 à l'issue d'un colloque [1] organisé par un comité de chercheurs dont Paul Hudak, S. Peyton-Jones, J. Kairbain et beaucoup d'autres. Leur but a été de définir un langage supportant les principaux concepts des lambda-langages. De ce fait, le langage Haskell se veut être un standard de la programmation fonctionnelle. Haskell est un langage fonctionnel pur où toute entité évaluable est une expression. Ses principales caractéristiques sont les suivantes :

- C'est un langage fortement typé : toute valeur possède un type. Il en est de même pour les fonctions et les expressions. Dans ce cas, on parle de *signature*.
- La définition des fonctions est intrinsèquement déclarative : toute fonction définie par l'utilisateur est un ensemble de clauses. Ainsi, la définition de la fonction f exprimée dans d'autres langages fonctionnels de la manière suivante :

```
f x y = si (x == 0) alors y
sinon si (x == 1) alors y+1
            sinon y+2
```

est exprimée en Haskell de la manière suivante :

[1] Functional Programming and Computer Architecture

```
f 0 y = y
f 1 y = y+1
f x y = y+2
```

Le choix de la clause adéquate se fait par le mécanisme du *filtrage*.

- La définition en compréhension des listes permet de définir des gardes conditionnant l'utilisation des différentes clauses définissant une même fonction.
- Haskell est un langage polymorphique; les programmes sont définis pour des familles de types et sont par conséquent plus généraux.
- Haskell autorise l'utilisation des fonctions d'ordre supérieur.
- Haskell est un langage non strict permettant ainsi la manipulation des structures potentiellement infinies.
- Haskell est un langage fonctionnel orienté objet. Il gère une hiérarchie de types où les opérateurs sont surchargés [2]. En revanche, la surcharge d'un type nouvellement défini est possible. Cet aspect d'Haskell est sans doute le plus innovateur pour la programmation fonctionnelle.

3 Les langages de combinateurs

Les langages de combinateurs et les langages fonctionnels sans variables se basent sur une théorie appelée *logique combinatoire* ou *théorie des combinateurs*. Cette théorie fut fondée par M. Schönfinkel [Sch24] au début du siècle. Elle fut étendue par A.B. CURRY dans [CF58]. Elle introduit la notion de combinateur grâce à laquelle une expression fonctionnelle ne contient pas de variable. La puissance d'expression de la logique combinatoire est comparable à celle du λ-calcul. En effet, le théorème de complétude de la théorie des combinateurs, énoncé dans [Bel86], assure qu'à partir d'une λ-expression quelconque, il est possible de construire algorithmiquement une expression combinatoire équivalente sans variables.

La signification de ce théorème est qu'à partir d'une λ-expression $\lambda x_1.\lambda x_2...\lambda x_n.\mathrm{E}$, on peut construire une expression sans variables qui soit équivalente à l'expression d'origine. Cette expression est appelée *expression combinatoire*. Une expression combinatoire se caractérise par le fait de ne contenir aucune variable : elle se compose uniquement de combinateurs, de constantes et de fonctions prédéfinies. Le rôle des combinateurs dans une expression combinatoire consiste à replacer

[2] au sens utilisé dans les langages à objets

les données à leur place initiale dans l'expression et ce, dynamiquement en cours d'exécution. Ce travail est effectué par des transformations successives de l'expression selon un ensemble de règles de réécriture précisant l'action de chaque combinateur.
Les langages de combinateurs manquent de lisibilité et servent essentiellement de langages intermédiaires ou de langages machine pour une machine abstraite plutôt que de langages de programmation. Ils permettent la normalisation du traitement des programmes fonctionnels quel que soit le langage source dans lequel ces derniers sont écrits.
Il existe plusieurs familles de combinateurs, à chacune d'elle sont associés un algorithme d'abstraction permettant la conversion des λ-expressions en une expression combinatoire et un système de réécriture décrivant le rôle de chaque combinateur, à l'exécution. Dans la suite de ce paragraphe nous présentons quelques exemples de familles de combinateurs.

3.1 Les combinateurs SKI

Cette famille de combinateurs créée par Curry [CF58] et optimisée par la suite par TURNER [Tur79], se caractérise par un jeu de combinateurs simples: **S, K, I** et par un système de réduction réduit:

a) **S** f g x	$\Longrightarrow$	f x (g x)
b) **K** x y	$\Longrightarrow$	x
c) **I** x	$\Longrightarrow$	x

L'algorithme d'abstraction d'une λ-expression en une expression combinatoire à base de combinateurs SKI, peut être décrit de façon informelle de la façon suivante

a)λ x. e1 e2	$\longrightarrow$	**S** (λ x.e1)(λ x.e2)
b)λ x. c	$\longrightarrow$	**K** c Si $x \neq c$
c)λ x.x	$\longrightarrow$	**I**

Le code combinatoire obtenu est plus long que le code fonctionnel d'origine : pour la version originale de l'algorithme d'abstraction [CF58, Tur79], cette expansion est de l'ordre de n^2, n étant la longueur du code source. Ce qui explique, en partie, leur inefficacité. Cette expansion excessive s'explique d'après [CDL87] en partie par le caractère récursif du processus d'abstraction impliquant ainsi une abstraction séquentielle des variables et l'obtention d'un code récursif s'adaptant mal à une exécution parallèle. Elle fut ramenée à o($nlogn$) grâce à une optimisation de

l'algorithme d'abstraction proposée dans [NH85], puis à une expansion linéaire dans [Bur82] et [NH85].

3.2 Les combinateurs de MARS

Ces combinateurs ont été inspirés des combinateurs SKI indicés d'ABDALI [Abd76] et des combinateurs SKI(BC). Contrairement à la famille des combinateurs décrite précédemment, l'algorithme d'abstraction associé aux combinateurs MARS, permet de traiter les variables en une seule passe en fonction de leur apparition dans l'expression permettant ainsi d'obtenir, plus rapidement, un code combinatoire plus court (au plus deux fois plus long que le code originel), qui en plus, se prête aisément à une exécution parallèle. Une description du jeu de combinateurs indicés et de l'algorithme standard d'abstraction se trouvent dans [CDL87].
Les travaux d'E.Cousin [Cou91] ont permis une optimisation de la longueur du code. L'idée de l'optimisation consiste à remarquer que l'algorithme d'abstraction standard commence par traiter les expressions les plus internes en extrayant les variables locales uniquement dans les sous expressions : les variables globales seront extraites lors de l'abstraction des expressions englobantes. Ceci a pour conséquence de rallonger considérablement le code produit puisque l'extraction de ces variables plus tard dans le processus d'abstraction, nécessite l'adjonction de combinateurs supplémentaires. L'algorithme proposé par E.Cousin consiste à extraire simultanément **toutes** les variables apparaissant dans une expression. Dans la plupart des cas, l'expansion du code est linéaire. Cependant, il peut arriver que le code obtenu soit plus long mais l'augmentation est bornée par le nombre de variables globales différentes apparaissant dans l'expression interne.

Le code combinatoire généré sert de langage machine pour la machine MARS [LC90] qui est une machine dédiée au traitement symbolique.

3.3 Les supercombinateurs

La description des supercombinateurs et de l'algorithme d'abstraction correspondant sont décrits dans [Hug82] ainsi que dans [Pey87].
Par définition, un supercombinateur est une λ-expression de la forme
$\lambda\ x_1\ \lambda\ x_2\ \ldots\ \lambda\ x_n$.E, vérifiant les deux propriétés suivantes :

- $\lambda x_1 \lambda x_2 \ldots \lambda x_n$.E ne contient pas de variables libres.
- Toute λ-expression appartenant à E est un supercombinateur.

L'algorithme d'abstraction permet d'obtenir à partir d'une λ-expression quelconque, un ensemble de définitions de supercombinateurs et une expression à évaluer. Les définitions des supercombinateurs obtenues servent également de système de réécriture décrivant les réductions à effectuer pour une occurrence d'un combinateur donné dans l'expression à évaluer.
L'algorithme d'abstraction associé aux supercombinateurs se trouve dans [Hug82].

L'originalité des supercombinateurs est que le système de réécriture les définissant n'est pas préalablement défini comme c'est le cas pour les autres familles de combinateurs : il dépend de la λ-expression à transformer.
L'avantage du code combinatoire obtenu est la taille très réduite du code ainsi que la granularité moyenne des supercombinateurs. Ces deux facteurs ont fait la réputation des supercombinateurs : le code combinatoire permet l'obtention de performances remarquables. I.Watson dans [WW87], affirme que l'exécution d'un code combinatoire constitué de supercombinateurs est 10 à 100 fois (selon les programmes) plus efficace qu'un code à base de combinateurs SKI.
Les supercombinateurs sont utilisés comme code combinatoire pour la G-machine [Kie87] ainsi que pour la machine FLAGSHIP [WW87].

3.4 Les combinateurs décurryfiés

Ces combinateurs sont basés sur une théorie des combinateurs nommée TGE présentée dans [BJ88]. Ils se caractérisent par le fait qu'ils sont décurryfiés i.e la curryfication y a été abolie ce qui simplifie le nombre d'étapes de calcul et améliore donc l'efficacité de l'évaluation des expressions fonctionnelles. Ces combinateurs permettent une représentation plus naturelle des fonctions. Un algorithme d'abstraction en expression combinatoire à base de tels combinateurs est proposé dans [BJ87].

4 Les langages fonctionnels sans variables

Les Langages Fonctionnels Sans Variables ou LFSV sont eux aussi basés sur la théorie des combinateurs au même titre que les langages de combinateurs mais sont plutôt considérés comme des langages de programmation à part entière. Quantitativement, ils ne sont pas nombreux. Le premier langage du genre fut le langage CUCH défini par C.BOHM [Boh64] mais il passa inaperçu. Les vrais débuts de ces langages correspondent à la définition par J.W.BACKUS [Bac78] des systèmes FP. Les concepts des systèmes FP sont reconnus être à la base de tout langage fonctionnel sans variables. D'autres langages sans variables furent

créés par la suite dont : JYM de P.BELLOT [Bel85] qui est une extension des systèmes FP, FL [Bac86] qui intègre les fonctionnelles de JYM puis le langage fonctionnel GRAAL [Bel86]. D'autres langages plus spécialisés ont vu le jour également. C'est le cas du langage GRIFFON [LS89] qui est un langage fonctionnel de manipulation de données. Nous distinguons deux types de concepts des LFSV que nous présentons dans ce paragraphe : les concepts de base que l'on retrouve au niveau de tous les LFSV et les concepts évolués. Nous terminerons par la présentation de quelques LFSV dont les systèmes FP, les systèmes FFP, le langage GRIFFON et le langage GRAAL.

4.1 Les concepts de base des LFSV

- Les LFSV se définissent par la donnée d'un ensemble O d'objets, d'un ensemble F de fonctions et d'une opération unique : l'*application*. Nous verrons par la suite que pour certains LFSV, l'ensemble F est inclus dans l'ensemble O.

- L'ensemble O se compose d'objets de types divers qui diffèrent selon les langages. Comme exemples d'objets, nous pouvons citer les atomes, les listes ou les agrégats.

- L'ensemble des fonctions F se compose de 3 types de fonctions : les **fonctions primitives** (P), les **formes fonctionnelles** (FF) et les **définitions** (D).

- Les fonctions primitives sont les fonctions prédéfinies du langage. Leur nombre est fixé dès le départ. Comme exemple de fonctions primitives, nous pouvons citer les fonctions arithmétiques.

- Les formes fonctionnelles sont des fonctions particulières qui permettent de combiner des fonctions existantes du langage afin d'en obtenir d'autres. Les fonctions combinées sont appelées les **paramètres** de la forme fonctionnelle. Chaque LFSV présente un ensemble de formes fonctionnelles prédéfinies. En voici quelques exemples :

  1. La distribution: α

$$\alpha \text{ f:}\langle x_1\ x_2\ \ldots\ x_n\rangle \quad \Longrightarrow \quad \langle f:x_1\ \ldots\ f:x_n\rangle$$
$$\alpha \text{ f:}\langle\rangle \quad \Longrightarrow \quad \langle\rangle$$

Cette définition doit être interpretée comme suit :
L'application de la fonction αf, quelle que soit la fonction f, à la liste composée de n éléments $x_1\ x_2\ \ldots\ x_n$ a pour résultat une liste composée de n éléments où l'élément n_i est l'application de la fonction f à l'élément x_i.

2. La construction: []
   $[f_1\ f_2\ \ldots\ f_n]$:x $\Longrightarrow < f_1 : x, f_2 : x, \ldots\ldots, f_n : x >$
3. La conditionnelle: **if**

   (if p f g):x $\Longrightarrow$ f:x si p:x=true
   $\Longrightarrow$ g:x si p:x=false
   $\Longrightarrow \perp$ [a] sinon.

[a] $\perp$ désigne l'objet indéfini

Il existe des LFSV qui permettent la définition d'autres formes fonctionnelles.

- Les définitions sont des fonctions définies par l'utilisateur. Chaque fonction définie pourra être utilisée dans un programme au même titre qu'une fonction prédéfinie. Chaque définition est une composition de fonctions de la forme $f_1 \circ f_2 \circ \ldots \circ f_n$ avec $n \geq 1$, où chaque f_i est une fonction de l'ensemble F. L'utilisation du nom d'une fonction définie dans une composition de fonctions est une *occurrence d'utilisation* de la définition. Dans un LFSV, définir une fonction s'effectue de la façon suivante:

  Def nom_de_la_definition = corps_de_la_fonction

  Exemple

  Def f = car o car

  L'application de la fonction f à la séquence << 1 2 > 3 > revient à appliquer le corps de la fonction f à la séquence << 1 2 > 3 >.

  cdr o car : << 1 2 > 3 >
  car : < 1 2 >
  < 2 >

- L'unique opération des LFSV est l'opération binaire *application*. Son premier opérande est une fonction de l'ensemble F et son deuxième opérande est un objet de l'ensemble O. L'application d'une fonction f à l'objet x,

appelé aussi l'**argument** de la fonction f, est notée **f:x**. Cette notation désigne également l'objet résultat de cette application. La fonction f est dite **en position active** puisqu'elle **s'applique** à un argument.

- Dans un LFSV, un programme fonctionnel est constitué d'un ensemble de définitions. L'une d'elles est la **définition principale** du programme. Cette définition d_p est une composition de fonction $f_1 \circ f_2 \circ \ldots \circ f_n$. Les définitions utilisées dans d_p sont les **définitions secondaires** du programme.

  Remarque
  Par abus de langage, un programme écrit dans un LFSV quelconque s'appellera *programme LFSV*. Similairement, l'application d'une fonction à ses arguments sera appelée *expression LFSV*.

- Les formes fonctionnelles d'un LFSV ont un rôle important dans la conception du programme mais aussi à l'exécution de celui-ci.
  Rôles à la conception

  - Les formes fonctionnelles permettent la combinaison de fonctions dans le but d'obtenir d'autres fonctions. Le pouvoir d'expression des LFSV provient donc essentiellement de la présence des formes fonctionnelles.
  - Les formes fonctionnelles permettent l'introduction de parallélisme dans les LFSV. Si nous prenons comme exemple la forme fonctionnelle *distribution* plus connue sous le nom de l' *apply-to-all*,

    $$\alpha \text{ f:} < x_1, x_2, \ldots, x_n > \Longrightarrow < f : x_1, \ldots, f : x_n >$$

    l'évaluation des applications f:x_i pour les i entre 1 et n sont indépendantes les unes des autres et peuvent donc s'effectuer en parallèle.

  Rôles à l'exécution

  - Les formes fonctionnelles assurent la distribution aux paramètres de leurs arguments respectifs. Prenons l'exemple de la forme fonctionnelle *construction* :

    $$[f_1\ f_2\ \ldots\ f_n]\text{:x} \Longrightarrow < f_1 : x, f_2 : x, \ldots\ldots, f_n : x >$$

    L'application de la forme fonctionnelle $[f_1\ f_2\ \ldots\ f_n]$ à l'argument x consiste à distribuer l'argument x à chacun des paramètres f_i. Les f_i deviennent alors chacune en position active.

– Elles assurent également le contrôle de l'exécution comme en témoigne l'exemple suivant où le résultat retourné est le premier élément de la liste si celle-ci est vide ou son premier élément sinon :

$$null \circ 1 \longrightarrow car;\ car \circ car: << 1 > 2 >$$

cette fonction retourne la valeur 1 puisque l'évaluation de l'application

$null \circ 1 :<< 1 > \ 2 >$

a pour résultat l'atome False. L'expression à évaluer est alors

$car \circ car :<< 1 > \ 2$

qui a pour résultat l'atome 1. Dans ce cas, seule la deuxième alternative de la conditionnelle est appliquée.

Il existe une autre catégorie de concepts que nous qualifions d'évolués que nous retrouvons dans certains LFSV. Ces concepts sont énumérés dans le paragraphe suivant.

4.2 Les concepts évolués des LFSV

- **Unification des objets**
  Dans les LFSV supportant ce concept, **il n'y a pas de différence fondamentale entre les données et les fonctions.** Autrement dit, l'ensemble O des objets inclut l'ensemble des fonctions F. L'unification des objets a les conséquences suivantes :

  – Une fonction peut désormais jouer deux rôles. Elle peut être en position active et dans ce cas, elle est applicable à un ensemble d'arguments, ou en position passive c.à.d qu'elle est l'un des arguments d'une autre fonction en position active.

  – Une expression peut admettre pour résultat une fonction. C'est la notion de *calculabilité* des fonctions. La fonction devient alors *calculable*.
  Dans un LFSV, nous pouvons désormais étendre la notion d'expression en définissant 3 formes d'expressions fonctionnelles à partir desquelles nous pouvons généraliser la notion d'expression.

  1. Le premier type d'application est :

$$\text{P: } x_1\ x_2\ \ldots\ x_n$$

où P est une composition de fonctions et ∀ i, x_i est un objet quelconque de l'ensemble O.

2. Le deuxième type d'application est

   P: $x_1\ x_2\ \ldots\ x_n$

   où P est une fonction calculable de la forme $P_1 : a_1\ a_2\ \ldots\ a_k$ et ∀ i, x_i est un objet quelconque de l'ensemble O.
3. Dans le troisième type d'application, un des objets x_i est calculable donc x_i est une application de la forme $P_2 : b_1\ b_2\ \ldots\ b_k$ où P_2 est une composition de fonctions et ∀ i, b_i est un objet quelconque de l'ensemble O.

- **Polyadicité des fonctions**

La polyadicité est le fait qu'une fonction peut avoir plus d'un argument. Les fonctions sont polyadiques de nature. Dans les langages ne prenant pas en compte ce concept, on a recours à l'opération de curryfication qui permet d'obtenir une fonction monadique à partir d'une fonction polyadique de la manière suivante :

$$(f : a_1 ... a_n) \equiv (((f\ :\ a_1)\ :\ a_2 ...))$$

L'opération de curryfication entrainera des transformations supplémentaires à l'évaluation qui nuisent à l'efficacité des langages fonctionnels. La solution adoptée dans les systèmes FP est de représenter une suite de n arguments par une séquence de n objets.

- **Généricité des fonctions**

Les LFSV dotés du concept de généricité des fonctions, permettent de définir des fonctions génériques appelées *schéma d'opération* ou *fonctionnelle* illustrant un comportement générique d'une fonction. Ce type de fonctions est paramétrable. Les paramètres effectifs sont des fonctions. En remplaçant les paramètres formels par les paramètres effectifs, on obtient une *instance* du schéma d'opération. Ce procédé est illustré par le schéma de la figure 1.2, emprunté à la thèse de P. Bellot :

Ce schéma d'opération est une fonction admettant 2 listes $< x_1, x_2, \ldots, x_n >$ et $< y_1, y_2, \ldots, y_n >$ pour arguments. Le résultat produit est la liste $< c_1, c_2, \ldots, c_n >$ où chaque $c_i = (x_i\ \ op\ \ y_i)$ où *op* est le paramètre formel de ce schéma d'opération. En remplaçant le paramètre formel *op* par une fonction (dans l'exemple la fonction +), on obtient une instance du schéma d'opération qui réalise le calcul de la liste $< x_1 + y_1, \ \ldots\ , x_n + y_n >$.

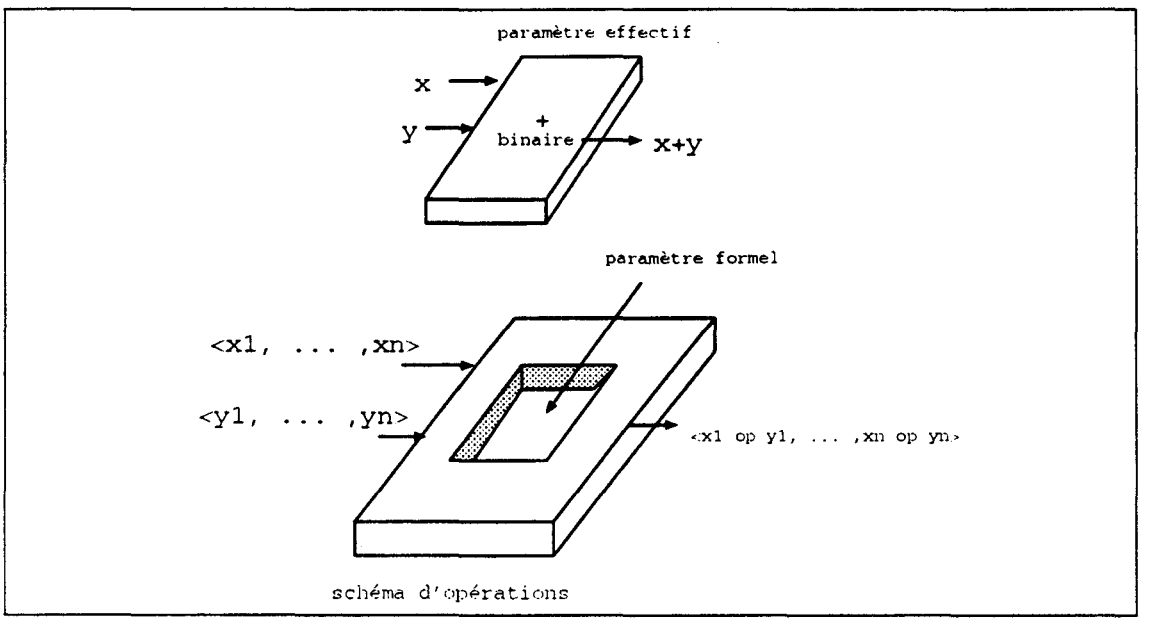


Figure 1.2 : Exemple de fonctionnelle

- **Pluralité du résultat**
  Il existe des LFSV qui autorisent l'obtention de résultats multiples. Ces langages comptent parmi leurs fonctions, des fonctions primitives appelées **séquences** qui se définissent comme suit :

$$[\mathrm{i}, \mathrm{j}] : X_1\ X_2\ \ldots\ X_n \Longrightarrow X_i \ldots X_j$$

  i et j sont deux entiers tels que i> 0 et j<n et $X_1\ X_2\ \ldots\ X_n$ sont des objets de l'ensemble O. Le langage VAAL [BB92] fait partie de ces langages.

4.3 Puissance d'expression des LFSV

La puissance d'expression des LFSV peut être comparée à celle du lambda-calcul. Ceci est prouvé par l'existence d'algorithmes qui convertissent toute λ-expression en une expression LFSV. Un de ces algorithmes a été décrit par P. Bellot. Il permet la conversion de toute λ-expression en une expression FP. Cet algorithme est décrit dans [Bel86] et [OB92]. Il est énoncé comme suit : soit a_1,a_2,...,a_n un ensemble de variables.

Remarque $\lambda < a_1\ a_2\ \ldots\ a_n >$.E est une écriture simplifiée de $\lambda a_1.\lambda a_2.\ldots.\lambda a_n$.E.

Algorithme

1. $\lambda < a_1 \ldots a_n > .a_i \xrightarrow{abs} i$

2. $\lambda < a_1 \ldots a_n > .c \xrightarrow{abs} /c$

   où c est une donnée constante par rapport aux variables a_1,a_2,...,a_n.

3. $\lambda < a_1 \ldots a_n > .(Si\ \ u\ alors\ v\ sinon\ w) \xrightarrow{abs} (if\ p\ f\ g)$

   où u, v, w sont des lambda-expressions et où p, f, g sont obtenues par abstraction des variables a_1,a_2,...,a_n dans les lambda expressions u, v et w.

   $$\begin{array}{lcl} \text{p} & \equiv & \lambda < a_1\ a_2\ \ldots\ a_n > .\ \text{u} \\ \text{f} & \equiv & \lambda < a_1\ a_2\ \ldots\ a_n > .\ \text{v} \\ \text{g} & \equiv & \lambda < a_1\ a_2\ \ldots\ a_n > .\ \text{w} \end{array}$$

4. $\lambda < a_1 \ldots a_n > \ f(e_1 \ldots e_k) \xrightarrow{abs} f \circ [g_1 \ldots g_k]$

   où [] est la forme fonctionnelle *construction*

   $$g_i \equiv \lambda \quad < a_1\ a_2\ \ldots\ a_n > .e_i \qquad 1 \leq \text{i} \leq \text{k}$$

Remarque

Les langages sans variables offrent certes une puissance d'expression équivalente à celle des λ-langages. Cependant, ils ne peuvent prétendre à une meilleure lisibilité, ce qui explique le scepticisme des programmeurs à leur égard.

4.4 Exemples de LFSV

4.4.1 Les systèmes FP

Créés par J.W.Backus durant les années 70 [Bac78], les systèmes FP peuvent être considérés comme les premiers LFSV. L'entité de base des systèmes FP est l'objet. Les systèmes FP sont des langages monadiques où les fonctions ne sont pas considérées comme des objets. L'ensemble O se compose de deux types d'objets : les atomes et les séquences. L'ensemble des atomes regroupe les nombres, les littéraux, les atomes booléens **T** et **F** et l'atome indéfini $\perp$. Une séquence permet aussi bien de représenter une liste d'éléments qu'une suite d'arguments. Son contenu sera interprété selon la fonction qui lui est appliquée. Par exemple, la fonction *tail* définie comme suit :

$$\begin{array}{lll} tail\ :\ x \equiv & x =< x_1 > \cdot & \Longrightarrow\ <>; \\ & x =< x_1\ \ldots\ x_n > et\ n \geq 2 & \Longrightarrow\ < x_2\ \ldots\ x_n > . \end{array}$$

où tous les x_i sont des objets de l'ensemble O, illustre un cas où la séquence x est interprétée comme une liste.
Une fonction comme la fonction *selecteur* définie par :

$$\forall s \geq 1, s : x \quad \equiv \quad x =< x_1 \ \ldots \ x_n > et \ \ n \geq s \Longrightarrow x_s; \ \perp .$$

illustre un cas où la séquence x composée de n objets quelconques est considérée comme une suite d'arguments. Les systèmes FP ne contiennent pas les fonctions d'ordre supérieur. Ce concept a été introduit par l'extension des systèmes FP aux systèmes FFP.

4.4.2 Les systèmes FFP

Les systèmes FFP sont conceptuellement des sur-ensembles des systèmes FP et possèdent donc tous les concepts de base des LFSV. En revanche, ils se caractérisent par la définition d'une entité plus globale que l'objet : l'expression.
Les systèmes FFP proposent une fonction de représentation φ qui associe à tout atome, une fonction donnée. Une deuxième fonction, la fonction μ définit la sémantique de l'évaluation des expressions FFP. Elle est définie par les 4 règles suivantes :
Soit A l'ensemble des atomes :

1. Un atome est invariable par la fonction μ.

   $\forall x \in A, \mu(x) = x$

2. L'évaluation d'une séquence x de n éléments a pour résultat, par μ, la séquence des résultats de l'évaluation des éléments.

   $\forall x =< x_1, \ x_2, \ \ldots \ , x_n >, \mu(x) =< \mu(x_1), \ \mu(x_2), \ \ldots \ , \mu(x_n) >$

3. l'évaluation de l'expression $x : y$ où x est un atome entraine l'application de la fonction associée à l'atome x par la fonction φ à l'expression y.

   $si \ x \ \in A \ \ \mu(x : y) = \mu(\varphi(x) : y)$

4. l'évaluation de l'expression $x : y$ où x est une séquence $< x_1, \ x_2, \ \ldots \ , x_n >$ entraîne l'application de la fonction $\varphi(x)$ à la séquence :

   $<< x_1, \ x_2, \ \ldots \ , x_n > \ \ y >$

   $si \ x =< x_1, \ x_2, \ \ldots \ , x_n > \mu(< x_1, \ x_2, \ \ldots \ , x_n >: y) = \mu(\varphi(x_1) :<< x_1, \ x_2, \ \ldots \ ,$

   C'est la règle de la *métacomposition*

Les systèmes FFP sont monadiques. Ils supportent les fonctions d'ordre supérieur. Elles sont appelées *métafonctions*. Ces fonctions permettent entre autre la redéfinition des formes fonctionnelles usuelles.

4.4.3 Le langage GRIFFON

GRIFFON défini dans [Le 88] est un langage fonctionnel de manipulation de base de données. La conception du langage GRIFFON a été amplement influencée par celle des deux LFSV : JYM [Bel86] et IFP [Rob87]. L'originalité du langage GRIFFON est son application à la gestion des bases de données. C'est pourquoi, en plus des éléments qui composent habituellement un LFSV s'ajoute une base de données. GRIFFON propose d'autres types d'objets tels que les *ensembles* et les *agrégats*. GRIFFON prend en compte les fonctions d'ordre supérieur : les fonctions sont également des objets. GRIFFON a une sémantique paresseuse : les *références* sont des objets dont le calcul est retardé jusqu'à ce qu'il devienne nécessaire.

4.4.4 Le langage GRAAL

Le langage GRAAL (General Recursive Applicative and Algorithmic Language) défini par P. BELLOT est un langage polyadique basé sur le système des combinateurs décurryfiés. De ce fait, contrairement aux systèmes FP, la représentation des listes et celle d'un ensemble d'arguments sont distinctes. GRAAL intègre les fonctions d'ordre supérieur. Il existe deux types de fonctions primitives d'ordre supérieur : les combinateurs et les fonctions de métaévaluation. De plus l'utilisateur peut se définir des fonctions d'ordre supérieur mais aussi des schémas d'opération appelés dans GRAAL **fonctions user**.
GRAAL possède aussi bien les concepts de base que les concepts évolués des LFSV. On peut donc le considérer comme un bon représentant de cette classe de langages fonctionnels. Le chapitre suivant décrit les composants de ce langage.

5 Conclusion

Les langages fonctionnels constituent une famille de langages informatiques possédant de très bonnes propriétés mathématiques et un parallélisme potentiel important. Ils se caractérisent par une totale indépendance par rapport aux contraintes techniques liées au matériel.
Nous avons cité trois classes de langages fonctionnels :

Les lambda-langages basés sur la théorie du λ-calcul offrent une grande puissance d'expression grâce à l'intégration des fonctions d'ordre supérieur et des fonctions non strictes permettant la gestion des structures potentiellement infinies.
Les langages de combinateurs quant à eux sont des langages de bas niveau servant la plupart du temps de langages intermédiaires pour une exécution efficace des langages fonctionnels. Cette exécution est identique, quel que soit le langage fonctionnel source.
Enfin les LFSV sont basés sur la théorie des combinateurs. Cette théorie a la réputation d'être moins complexe mais aussi puissante que le λ-calcul. Cette puissance d'expression est également prouvée par l'éxistence d'algorithmes de transcription de programmes écrits en un λ-langage en un programme LFSV équivalent. Ces langages présentent deux types de concepts : des concepts de base présents dans tous les LFSV et des concepts évolués.
Dans ce document, nous nous intéressons plus précisément à cette catégorie de langages et plus particulièrement aux langages possédant les concepts évolués des LFSV. GRAAL semble être un bon représentant de cette classe de langages. L'étude de l'extension du modèle sera donc basée sur l'étude de ce langage qui sera donc plus amplement détaillé au chapitre 2.

Chapitre 2

Le langage GRAAL

GRAAL est un LFSV polyadique intègrant les fonctions d'ordre supérieur. Dans GRAAL, aussi bien les données que les fonctions sont des objets. Dans ce chapitre, nous ne reviendrons pas sur les concepts de GRAAL (cf. chapitre1), mais nous le consacrerons à la description [1] des objets de GRAAL que nous avons divisé en 3 classes : les données, les fonctions "ordinaires" [2] et les fonctions d'ordre supérieur. Nous décrirons également le procédé de définition et d'utilisation des schémas d'opération. Nous terminerons enfin, par un deuxième type de formes fonctionnelles appelées *métaformes* dont l'utilisation est plus efficace que les schémas d'opération quand la fonction à définir est récursive.

1 Les données

Une donnée en GRAAL est :

- soit un *atome*, nous distinguons comme types d'atomes les symboles par exemple : toto, nil, et les nombres, par exemple : 1, 4.5.
- soit une *liste* hétérogène d'objets où chaque objet est un élément de l'ensemble O des objets.
  exemples :
  $< toto >$
  $< car \ o \ car \quad 1 \quad toto >$
  $<< 1 >< toto \ \ 10 >>$
- soit un *stream* : un *stream* est une liste infinie d'éléments appartenant à l'ensemble O. Il est représentée en compréhension par le quadruplet : $< x_0, m, d, e >$ où :

[1] Certains aspects du langage ont été volontairement omis. Le lecteur averti peut consulter [Bel86] pour davantage d'informations

[2] qui ne sont pas d'ordre supérieur

- x_0 est le premier élément du stream
- m est un prédicat que tout élément du *stream* doit vérifier.
- d est une fonction que l'on applique à tous les éléments de la suite.
- e est la fonction à appliquer à l'élément x_i pour obtenir l'élément x_{i+1}. (i $\geq$ 0)

exemple Le stream des nombres naturels est représenté par le quadruplet $< 0, True, id, add1 >$.

2 Les fonctions "ordinaires"

2.1 Les fonctions primitives

L'ensemble des fonctions primitives comprend les fonctions arithmétiques, les fonctions de manipulation de listes, les prédicats [3], les sélecteurs ... etc. Dans la suite, X=< x_1 x_2 ... x_n > et Y=< y_1 y_2 ... y_m > représentent deux listes de n et m éléments respectivement. Les objets $a, b, z_1, z_2, \ldots, z_n$, $x_1, x_2, \ldots, x_n$, $y_1, y_2, \ldots, y_n$ représentent des objets quelconques. Le symbole L représente une liste quelconque.

- Les fonctions arithmétiques
  Généralement, les fonctions arithmétiques admettent 2 arguments tous deux des nombres du même ensemble i.e des entiers, des réels...etc.

  $$+: \text{x} \quad \text{y} \Longrightarrow \text{x+y}$$

  Remarque : les fonctions commutatives + et * ont une arité variable c.à.d qu'elles peuvent admettre un nombre quelconque d'arguments supérieur ou égal à 2.

  $$+: a_1\ a_2\ \ldots\ a_n \Longrightarrow a_1 + a_2 + \ldots + a_n$$

  $$*: a_1\ a_2\ \ldots\ a_n \Longrightarrow a_1 * a_2 * \ldots * a_n$$

- Les fontions de manipulation de listes

  La fonction **car**

[3] les fonctions primitives dont le résultat est booleen

$car : < x_1\ x_2 \ldots x_n > \quad \Longrightarrow \quad x_1 \qquad si\ n \geq 1$
$car : <> \quad \Longrightarrow \quad <>$

La fonction **cdr**

$cdr : < x_1\ x_2 \ldots x_n > \quad \Longrightarrow \quad < x_2 \ldots x_n > \qquad si\ n \geq 1$
$cdr : <> \quad \Longrightarrow \quad <>$

La fonction **cons**

cons: a z $\quad \Longrightarrow \quad < a.z >$

La fonction **list**

$list : a_1\ a_2 \ldots a_n \quad \Longrightarrow \quad <a_1\ a_2 \ldots a_n>$

La fonction **append**

append : X Y $\quad \Longrightarrow \quad < x_1\ x_2 \ldots x_m\ y_1\ y_2 \ldots y_n>$

Le résultat est la liste composée successivement des éléments de la liste X suivis de ceux de la liste Y. Cette liste résultat a donc $m+n$ éléments.

La fonction **length**

$length : < a_1\ a_2 \ldots a_n > \quad \Longrightarrow \quad n$
$length : < > \quad \Longrightarrow \quad 0$

- Les sélecteurs
  Les sélecteurs sont des fonctions primitives qui permettent de sélectionner un argument parmi plusieurs. Elles se définissent comme suit :

$k : a_1\ a_2 \ldots a_i \ldots a_n \quad \Longrightarrow \quad a_k \qquad 1 \leq k \leq n$

- La fonction identité :*id*

$id : z_1\ z_2 \ldots z_n \quad \Longrightarrow \quad z_1$

- Les prédicats
  Il existe plus de 100 prédicats dans GRAAL. Ces prédicats servent entre autres à tester l'égalité de deux symboles (exemple le prédicat *eq*), à tester la valeur d'un symbole (ex :*eq1, null*...etc) ou son type. En voici quelques exemples.

  Le prédicat **null**

  null : L $\Longrightarrow$ True si *L* est une liste vide
  $\Longrightarrow$ False sinon

  Le prédicat **eq**

  eq : < x y > $\Longrightarrow$ True si x=y
  $\Longrightarrow$ False sinon.

  Le prédicat **atomp**

  atomp : s $\Longrightarrow$ True si *s* est un atome
  $\Longrightarrow$ False sinon.

  Le prédicat **listp**

  listp : s $\Longrightarrow$ True si *s* est une liste
  $\Longrightarrow$ False sinon.

  Le prédicat **formp**

  formp : s $\Longrightarrow$ True si *s* est un symbole de forme fonctionnelle
  $\Longrightarrow$ False sinon.

2.2 Les formes fonctionnelles

Les formes fonctionnelles prédéfinies de GRAAL sont : la forme *constante*, la forme *conditionnelle* (if), la forme *tant que*, les formes fonctionnelles *distribution*, les formes fonctionnelles *binaires-unaires*, la forme fonctionnelle *cond*, la forme fonctionnelle *composition* et la forme fonctionnelle "::".

Dans ce qui suit, les symboles f, g, p, h, f_i, $(i \geq 0)$ désignent des symboles de fonction. Les symboles x, y, a_i, x_i, $(i \geq 0)$ désignent des objets quelconques.

La constante notée /x

$/x : y \quad \Longrightarrow \quad x$

La conditionnelle **if**

$(if\ p\ \ f\ \ g) : x \quad \Longrightarrow$ f : x si p : x $\neq$ <>
$\Longrightarrow$ g : x si p : x = <>

La conditionnelle **while**

$(while\ \ p\ \ f) : x \quad \Longrightarrow$ (if p (while p f) o f) id) : x

Les formes fonctionnelles de distribution

La forme distribution α

c'est une généralisation de la forme fonctionnelle *distribution* [4] des systèmes FP présentée au chapitre précédent. Elle se note également *dist.* Elle se définit par :

$$\alpha\, f : < a_{11} \ \ldots\ a_{1m_1} > \ \ldots\ < a_{n1} \ \ldots\ a_{nm_n} >$$

$$\Longrightarrow\ <$$
$$f : a_{11}\ \ a_{21} \ldots\ a_{n1}$$
$$f : a_{12}\ \ldots\ \ a_{n2}$$
$$\ldots$$
$$f : a_{1m_i}\ \ \ldots\ a_{nm_i}$$
$$>$$

$m_i = \min_{j=1}^{n}(m_j)$

Pour résumer sa sémantique, l'application de la fonction αf à n arguments, arg_1, arg_2, ... , arg_n, chacun étant une liste de m_i éléments, est équivalente à l'application simultanée de la fonction f aux suites d'arguments s_1, s_2, ... , s_{m_i} où chaque suite s_j se compose des jièmes composantes des listes arg_i.

La forme distribute-right **distr**

$$(distr\ f) : a \quad < b_1\ b_2 \ldots\ b_n > \ \Longrightarrow\ <$$
$$f : a\ \ b_1$$
$$f : a\ \ b_2$$
$$\ldots$$
$$f : a\ \ b_n$$
$$>$$

[4] "apply-to-all"

La forme distribute-left **distl**

$$(distl\ \mathrm{f}) : < b_1\ b_2\ \ldots\ b_n > \quad \mathrm{a} \quad \Longrightarrow \quad < \begin{array}{l} f : b_1\ \ a \\ f : b_2\ \ a \\ \ldots \\ f : b_n\ \ a \end{array} >$$

La forme **dista**

$$(\mathrm{dista\ f\ g}) : a_1\ a_2\ \ldots\ a_n \quad \Longrightarrow \quad f : \begin{array}{l} (g : a_1) \\ (g : a_2) \\ \ldots \\ (g : a_n) \end{array}$$

Remarque 1
Les parenthèses sont utilisées dans le but unique de rendre plus claire la description de la forme fonctionnelle.
Remarque 2
La forme *dista* peut également être décrite à l'aide de la forme *distribution* des systèmes FP:

$$(distafg)_{GRAAL} \equiv (f \circ \alpha g)_{FP}$$

Les formes binaire-unaires
Il existe deux formes fonctionnelles *binaire-unaire* de noms *binu* et *binul*, toutes deux ont deux paramètres; le premier est nécessairement une fonction, le second est un objet quelconque. Ces formes fonctionnelles s'appliquent à un seul argument.

La forme **binu**

$$(binu\ \mathrm{f\ x}) : \mathrm{y} \quad \Longrightarrow \quad \mathrm{f} : \mathrm{x\ y}$$

La forme **binul**

$$(binul\ \mathrm{f\ x}) : \mathrm{y} \quad \longrightarrow \quad \mathrm{f} : \mathrm{y\ x}$$

La forme **cond**

Cette forme est inspirée de la structure de contrôle *cond* du langage LISP. C'est une forme généralisée de la conditionnelle au sens des systèmes FP. Sa sémantique est décrite comme suit :

$$
\begin{array}{lll}
(cond \quad f_1\ g_1\ \ f_2\ g_2\ \ldots\ f_n\ g_n) : a_1\ a_2\ \ldots\ a_n & & \\
\Longrightarrow \quad g_1 \text{: } a_1\ a_2\ \ldots\ a_n & \text{si } f_1 \text{: } a_1\ a_2\ \ldots\ a_n \neq \text{nil sinon} & \\
\Longrightarrow \quad g_2 \text{:} a_1\ a_2\ \ldots\ a_n & \text{si } f_2 \text{:} a_1\ a_2\ \ldots\ a_n \neq \text{nil sinon} & \\
\quad \ldots & & \\
\Longrightarrow \quad g_n \text{: } a_1\ a_2\ \ldots\ a_n & \text{si } f_n \text{:} a_1\ a_2\ \ldots\ a_n \neq \text{nil} & \\
\Longrightarrow \quad \text{nil} & \text{sinon} &
\end{array}
$$

La forme **composition**

$$
\begin{array}{l}
(comp\ \mathrm{f}\ g_1\ g_2\ \ldots\ g_n) : a_1\ a_2\ \ldots\ a_m \\
\qquad \Longrightarrow \quad f : \quad (g_1 \text{: } a_1\ a_2\ \ldots\ a_m) \\
\qquad\qquad\qquad\quad\ (g_2 \text{:} a_1\ a_2\ \ldots\ a_m) \\
\qquad\qquad\qquad\quad\ \ldots \\
\qquad\qquad\qquad\quad\ (g_n \text{: } a_1\ a_2\ \ldots\ a_m)
\end{array}
$$

La forme *composition* admet $n+1$ paramètres. Le résultat retourné est celui de l'application du premier paramètre f à l'ensemble des objets $g_1 \text{:} a_1\ a_2\ \ldots\ a_m$, $g_2 \text{: } a_1\ a_2\ \ldots\ a_m$ et $g_n : a_1\ a_2\ \ldots\ a_m$. La forme fonctionnelle *composition* est notée plus fréquemment $\{\ \mathrm{f}\ g_1\ g_2\ \ldots\ g_n\ \}$. On peut également l'exprimer par les formes *composition* et *construction* des systèmes FP:

$$\{\ \mathrm{f}\ g_1\ g_2\ \ldots\ g_n\ \}_{GRAAL} \equiv (f \circ\ \ [\ g_1\ g_2\ \ldots\ g_n\ ])_{FP}$$

La forme '::'

$$(\ \mathrm{f}\ \ :: a_1\ a_2\ \ldots\ a_n) : b_1\ b_2\ \ldots\ b_k \Longrightarrow \mathrm{f}\text{:}a_1\ a_2\ \ldots\ a_n$$

On peut l'exprimer à l'aide la forme *composition* GRAAL par:

$$(\ \mathrm{f}\ ::\ a_1\ a_2\ \ldots\ a_n\ ) \equiv \{\ \mathrm{f}\ \ 'a_1\ \ 'a_2\ \ \ldots\ \ 'a_n\ \}$$

où $'a_i$ est une abréviation de l'expression (cste a_i). La forme fonctionnelle '::' admet $n+1$ paramètres. Le premier, f, est nécessairement une fonction; $a_1, a_2, \ldots, a_n$ sont des objets quelconques. Appliquée à une suite d'arguments $b_1, b_2, \ldots, b_k$, elle entraine l'application de la fonction f aux objets $a_1\ a_2\ \ldots\ a_n$.

2.3 Les définitions

Dans GRAAL, l'utilisateur peut définir de nouvelles fonctions, chacune d'elle est une composition de fonctions dans laquelle chaque fonction peut être "ordinaire" ou d'ordre supérieur. La définition de fonction se fait selon le modèle suivant :

$$\text{Def} \quad nom - de - la - fonction \quad \equiv \quad corps - de - la - fonction$$

3 Les fonctions d'ordre supérieur

Comme pour les fonctions ordinaires, il existe des fonctions primitives, des formes fonctionnelles et des définitions, toutes d'ordre supérieur. Les fonctions primitives d'ordre supérieur sont de deux types : les *combinateurs* et les *fonctions de méta-évaluation.*

3.1 Les combinateurs

Les combinateurs sont des fonctions primitives d'ordre supérieur admettant comme arguments des fonctions quelconques de l'ensemble F. Le résultat de l'application d'un combinateur à ses arguments est une forme fonctionnelle, dont *les paramètres sont les arguments du combinateur.* A chaque fois qu'un combinateur donné c est appliqué à ses arguments, il permet d'obtenir toujours la même forme fonctionnelle. Il n'y a que les paramètres de la forme fonctionnelle qui changent. Il existe donc une association entre la forme fonctionnelle et le combinateur qui permet sa construction.
Dans GRAAL, il existe autant de combinateurs que de formes fonctionnelles prédéfinies.

exemples

- le combinateur *if*
  Le combinateur *if* permet l'obtention de la forme fonctionnelle "conditionnelle". Il admet 3 arguments, tous des fonctions.

$$if : \text{p f g} \Longrightarrow (\text{if p f g})$$

- Le combinateur **binu** associé à la forme fonctionnelle *binu.* Il admet deux arguments le premier est une fonction, le deuxième est un objet quelconque.

binu: f c $\Longrightarrow$ (binu f c)

- Le combinateur *comp*
  Le combinateur *comp* appliqué à plusieurs fonctions permet d'obtenir la composition de ces fonctions. Le nombre d'arguments du combinateur *comp* est variable. Il est supérieur ou égal à 2.

  comp: f g $\Longrightarrow$ { f g } ou f ∘ g.

  comp: $f_1\ f_2\ \ldots\ f_n \Longrightarrow \{\ f_1\ f_2\ \ldots\ f_n\ \}$

3.2 Les fonctions de méta-évaluation

Les combinateurs permettent la construction de formes fonctionnelles et contribuent donc, au cours de l'évaluation d'une expression, à la construction d'une fonction. Les fonctions de méta-évaluation permettent de forcer l'évaluation d'une expression en cours d'exécution. Dans GRAAL, il existe 3 fonctions de *métaévaluation* : *eval, apply, et funcall.*

- La fonction *eval*
  La fonction *eval* admet un seul argument (e) qui est une expression fonctionnelle GRAAL. Appliquée à son argument, elle retourne le résultat de l'évaluation de celui-ci. Deux cas sont possibles:

  - e est un symbole. Le résultat retourné est le symbole e. Il en est de même dans le cas d'un nombre.
  - e est une application du style: (F : $v_1\ v_2\ \ldots\ v_n$) où F, $v_1, v_2, \ldots, v_n$ sont des expressions. Dans ce cas,

  $eval : e$

  $\equiv eval : (F\ :\ v_1\ \ \ldots\ \ v_n)$

  $\equiv (eval\ \ F)\ :\ (eval\ \ v_1)\ \ \ldots\ \ (eval\ \ v_n)$

- La fonction *apply*

  apply : f $<a_1\ a_2\ \ldots\ a_n> \Longrightarrow$ f: $a_1\ a_2\ \ldots\ a_n$

  La fonction *apply* admet 2 arguments, le premier est une fonction, le deuxième est une liste d'objets quelconques. L'arité de la fonction f doit correspondre à la longueur de la liste $<a_1\ a_2\ \ldots\ a_n>$.

- La fonction *funcall*
  f est une fonction quelconque, a un objet quelconque.

$$\textit{funcall: f a} \Longrightarrow \text{f: a}$$

Remarque
La fonction *funcall* est un cas particulier de la fonction *apply*.

$$\text{funcall: f a} \Longleftrightarrow \text{apply: f} < a >$$

4 Les fonctionnelles

4.1 Définition

Dans un langage fonctionnel, il est intéressant de pouvoir se définir ses propres formes fonctionnelles quand les formes fonctionnelles prédéfinies ne répondent pas aux besoins du programmeur. La définition de nouvelles formes fonctionnelles est rendue possible dans GRAAL grâce à la propriété de calculabilité des programmes. Définir une forme fonctionnelle revient à définir une façon différente de combiner les fonctions. Ce procédé devient encore plus attrayant quand ce type de combinaison est fréquemment utilisé aux paramètres près. En GRAAL, pour définir une nouvelle forme fonctionnelle, le programmeur écrit un programme appelé *fonction user*. Cette définition symbolise un schéma d'opération. Appliquée à ses arguments (qui sont dans la plupart des cas des fonctions), elle permet l'obtention d'une instance du schéma d'opération. Ce nouveau programme illustre une nouvelle combinaison des arguments de la définition. C'est pour cette raison que nous parlons de formes fonctionnelles définies.

4.2 Exemple d'utilisation

Imaginons que nous ayons besoin d'un programme qui effectue une opération quelconque sur deux arguments le premier étant fixé à la valeur constante 2. En d'autres termes, quelle que soit la fonction f, nous désirons arriver au programme *(binul f 2)* pour pouvoir l'appliquer à son argument. Il nous faut donc écrire une fonction user qui permette l'obtention du programme (binul f 2) $\forall$ f $\in$ F. Une fonction *user* est définie à l'aide de la fonction **dc** qui la distingue des autres types de définitions et qui spécifie qu'il s'agit d'une fonction décrivant un schéma d'opération. La fonction recherchée est donc définie comme suit:

(dc op2 {binul 1 '2 })

L'application de la fonction user *op2* à une fonction *f* permet l'obtention de la fonction (binul f 2):

op2 : f

1.remplacement par le corps de la fonction op2
$\Longrightarrow$ {binul 1 '2} : f

2.application de la composition GRAAL
$\Longrightarrow$ binul : (1 : f) ('2 : f)

3.application du sélecteur 1 et de la constante 2
$\Longrightarrow$ binul : f 2

4.application du combinateur associé a la forme fonctionnelle binul
$\Longrightarrow$ (binul f 2)

Remplaçons *f* par une fonction concrète : la fonction +; et appliquons cette nouvelle fonction à la constante *5* par exemple :

(binul + 2) : 5
$\Longrightarrow$ + : 5 2
$\Longrightarrow$ 7

En définissant la fonction *op2*, un nouveau type de combinaisons ou de forme fonctionnelle existe désormais. Elle est baptisée du même nom que la fonction qui l'a définie. De ce fait, on l'appellera **forme user**.

Dans un programme GRAAL, on peut dorénavant utiliser cette nouvelle forme de la même manière que l'on utilise les formes fonctionnelles prédéfinies. Ainsi, on peut écrire comme suit le programme qui calcule la surface d'un triangle utilisant la *forme user op2* :

de surface-triangle *(op2 /)* o *

L'évaluation de l'expression suivante, permet d'obtenir la surface d'un triangle ayant pour base 10 et pour hauteur 15 .

(op2 /)o *: 10 15
1.composition
$\Longrightarrow$ (op2 /) : (* : 10 15)
2.multiplication
$\Longrightarrow$ (op2 /): 150
3.remplacement du litteral op2 par le corps de la fonction correspondante
$\Longrightarrow$ ({binul 1 '2 }: /): 150
4.après construction de la nouvelle fonction
$\Longrightarrow$ (binul / 2):150
5.application de la forme binul
$\Longrightarrow$ /: 150 2
6.application de la fonction /
$\Longrightarrow$ 75

Pour récapituler, la définition d'une fonction user permet donc d'ajouter une nouvelle forme fonctionnelle de type user à l'ensemble initial des formes fonctionnelles. L'utilisation des nouvelles formes permet la conception de programmes clairs et concis. La nouvelle combinaison de fonctions est calculable en cours d'exécution en fonction des paramètres fournis. Cependant, ce calcul est très pénalisant dans des programmes récursifs puisqu'à chaque appel, la combinaison est recalculée comme nous pouvons le voir à travers l'exemple suivant donné par P.Bellot [Bel86], et définissant le comportement de la forme fonctionnelle *while*

dc while {if 1 {comp {user [5] 'while 1 2} 2 } 'id }}

L'application de la forme *while* ayant 2 paramètres : les fonctions p et f donne la série de transformations suivantes :

(while p f): $a_1\ a_2\ \ldots\ a_n$
1.application du schéma d'opération pour construire le programme décrivant le comportement de la forme while
$\Longrightarrow$ (while: p f): $a_1\ a_2\ \ldots\ a_n$

[5] *user* est le combinateur qui appliqué à un symbole définissant une forme user *su* et une suite de fonctions de longueur quelconque, permet l'obtention d'une forme user.
user : $su\ f_1\ f_2\ \ldots\ f_n \Longrightarrow (su\ f_1\ f_2\ \ldots\ f_n)$

$\Longrightarrow$ ({if 1 {comp { user 'while 1 2 } 2 } 'id } : p f) : a_1 a_2 ... a_n
2.application de la forme fonctionnelle {} *où*
$\Longrightarrow$ (if: A B C):a_1 a_2 ... a_n
application de la forme fonctionnelle {} *où*
A ≡ 1: p f
B ≡ {comp { user 'while 1 2 } 2 }: p f
C ≡ 'id: p f
$\Longrightarrow$ id

Evaluation de l'expression B

B ≡ comp: ({ user 'while 1 2 }: p f) (2: p f)
1.application de {}
$\Longrightarrow$ comp: (user: ('while: p f) (1: p f) (2: p f)) f
2.user est le combinateur qui permet de construire une forme user de nom son premier argument et de paramètre ses autres arguments
$\Longrightarrow$ comp: (user: while p f) f
$\Longrightarrow$ comp:(while p f) f
$\Longrightarrow$ $(while\ p\ f) \circ f$

Dans le cas où la condition $p : a_1\ a_2 \ldots\ a_n$ retourne True, on évalue l'expression $(while\ \ p\ \ f) \circ f$: $a_1\ a_2 \ldots\ a_n$ ce qui veut dire que l'on recalcule le programme *while: p f* autant de fois que la condition p est vérifiée. Dans ce cas précis, le langage GRAAL propose d'écrire autrement la forme *while* en utilisant un autre type de formes: *les métaformes.*

5 Les métaformes

5.1 Définition

Les métaformes trouvent leurs origines dans les systèmes FFP [Bac78]. Elles tiennent leur nom de la règle de métacomposition de ces systèmes. Pour définir une métaforme, nous avons besoin d'écrire une définition. Une telle fonction est définie à l'aide de la fonction **dm** et porte le nom de *fonction méta.* Une forme méta admet comme arguments une liste de fonctions (qui sont les futurs paramètres de la métaforme) suivie d'un ensemble d'arguments quelconques. Les fonctions méta se caractérisent par une utilisation fréquente des fonctions de méta-évaluation. Par l'écriture d'une fonction méta, on définit donc une nouvelle

forme fonctionnelle de type *méta* portant le même nom. Une forme méta de nom m est utilisée dans un programme de façon identique à une forme fonctionnelle prédéfinie:

$$(\text{m } p_1\ p_2\ \ldots\ p_n)$$

où p_1,p_2,...,p_n sont ses paramètres. Appliquée à une suite d'arguments, elle fait intervenir la fonction méta correspondante de la façon suivante:

$$(m_{fm}\ p_1\ p_2\ \ldots\ p_n)\text{: } a_1\ a_2\ \ldots\ a_k \Longrightarrow m_{fct} :<p_1\ p_2\ \ldots\ p_n>\ a_1\ a_2\ \ldots\ a_k$$

m_{fm} est la métaforme de nom m. m_{fct} est la fonction méta de nom m qui décrit le comportement de la forme m_{fm}.

5.2 Exemple d'utilisation

Réécrivons une fonction méta de nom *op2* définissant une forme méta qui effectue une opération f sur un argument quelconque et un argument fixe qui est la constante 2. Cette fonction est définie comme suit:

$$(\text{dm } op2\ \{apply\quad car\ \circ\ 1\quad (binul\quad list\quad '2)\ \circ\ 2\})$$

La métaforme ainsi définie est utilisée de la même façon dans le programme de calcul de la surface d'un triangle:

$$(\text{de surf-triangle} \equiv (op2\quad /)\circ *)$$

L 'évaluation du programme est par contre différente:
surf-triangle: 10 15
$\Longrightarrow$ ($op2$ /) $\circ$ *: 10 15
$\Longrightarrow$ (op2 /) : 150
$\Longrightarrow$ op2: < / > 150
$\Longrightarrow$ {apply car o1($binul$ $list$ '2) o 2} :< / > 150
$\Longrightarrow$ apply: (car o 1 : < / > 150) (($binul$ $list$ '2) o 2: < / > 150)
$\Longrightarrow$ apply: / ((binul list '2): 150)
$\Longrightarrow$ apply: / (list:150 2)
$\Longrightarrow$ apply: / < 150 2 >
$\Longrightarrow$ /: 150 2

$\Longrightarrow$ 75

Remarque:
La définition d'une nouvelle forme fonctionnelle par l'écriture d'une fonction user est plus performante: en effet, la fonction user est une combinaison plus simple de fonction comparée à la fonction méta. Par conséquent le nombre d'étapes effectuées lors de l'évaluation d'une forme user est moins important. Il n'y a qu'un cas où l'utilisation des métaformes est préférable. C'est le cas des programmes récursifs: les métaformes évitent le calcul répété du corps de la forme.

Réécrivons une fonction méta qui décrit le comportement de la forme *while* [Bel86].

(dm while (if { funcall *car* o 1 2 } {while 1 {funcall *cadr* o 1 2 }} 2))

(while p f): a
$\Longrightarrow$ while : $< pf >$ a
application de la fonction méta
$\Longrightarrow$ (if { funcall *car* o 1 2 } {while 1 {funcall *cadr* o 1 2 }} 2)) : $< pf >$ a
$\Longrightarrow$ { funcall *car* o 1 2 }: $< pf >$ a
$\Longrightarrow$ funcall: (*car* o 1:$< pf >$ a) (2: $< pf >$ a)
$\Longrightarrow$ funcall: (car: $< p \quad f >$) a
$\Longrightarrow$ funcall: p a
$\Longrightarrow$ p: a

Si l'objet résultat de l'application *p: a* est l'atome *True*, on applique la fonction {while 1 funcall *cadr* o 1 2 }} aux arguments:

{ while 1 {funcall *cadr* o 1 2}}: $< p \quad f >$ a
$\Longrightarrow$ while: (1: $< p \quad f >$ a) ({funcall *cadr* o 1 2 }: $< pf >$ a)
$\Longrightarrow$ while: $< pf >$ (funcall: (*cadr* o 1: $< pf >$ a) (2: $< p \quad f >$ a))
$\Longrightarrow$ while: $< p \quad f >$ (funcall: f a)
$\Longrightarrow$ while: $< p \quad f >$ (f: a)
$\Longrightarrow$ while: $< p \quad f > a_1$
$a_1 \equiv$ f: a
... etc.

On réapplique donc la fonction méta récursivement et non pas la forme fonctionnelle récursivement d'où l'avantage par rapport aux formes user dans ce cas précis.

Remarque:
Les *fonctions user* peuvent être utilisées directement dans un programme comme

des définitions ordinaires. Elles permettent l'obtention comme résultat d'une combinaison de fonctions éventuellement applicable par méta-évaluation. Ce qui n'est pas le cas des fonctions méta auxquelles on fait appel uniquement quand la *forme méta* correspondante intervient dans un programme.

6 Exemple de programme : La multiplication matricielle

Elle est définie par les deux fonctions *ps* et *pm*:

$(de\ \ pm\ \ \{(distl\ \ (distr\ \ ps))1(binu\ \ apply\ \ (dist\ \ list))\circ\ 2\}\ )$

$(de\ \ ps\ \ \ (binu\ \ apply\ \ +)\circ(dist\ \ *))$

Appliquons ce programme aux deux matrices A et B suivantes de dimension 2 :

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} et\ B = \begin{pmatrix} -1 & 1 \\ 0 & 3 \end{pmatrix}$$

chacune représentée par une liste de listes. Chaque sous liste représente une ligne de la matrice.

$A =<< 1\ \ 3 >< 2\ \ 4 >>\qquad B =<< -1\ \ 1 >< 0\ \ 3 >>$

pm : $<< 1\ \ 3 >< 2\ \ 4 >>\ \ << -1\ \ 1 >< 0\ \ 3 >>$

1.le symbole pm est remplacé par la définition de nom pm
$\Longrightarrow$ { (distl (distr ps) 1 (binu apply (dist list)) ∘ 2 }: A B

2.application de la composition GRAAL
$\Longrightarrow$ (distl (distr ps)):
(1: A B)
((binu apply (dist list)) ∘ 2: A B)

3.application des sélecteurs 1 et 2
$\Longrightarrow$ (distl (distr ps)): A

```
                    ((binu apply (dist   list)) :    B )
```

4.application de la forme fonctionnelle binu

```
    ⟹ (distl (distr ps)):    A
                    (apply :
                            (dist   list)
                            B
                    )
```

5.application de la fonction apply

```
    ⟹ (distl (distr ps)):
                    << 2   3 >< 2   4 >>
                    ((distl   list):
                            < −1   1 >
                            < 0   3 >
                    )
```

6.application de la forme distribution généralisée

```
    ⟹ (distl(distr ps)):
                    << 1   3 >< 2   4 >>
                    < list: -1
                            0
                      list: 1
                            3
                    >
```

7.on obtient donc la transposée de la matrice B

```
    ⟹ (distl (distr ps)):
                    << 1   3 >< 2   4 >>
                    << −1   0 >< 1   3 >>
```

8.application de la forme distl

```
    ⟹ <
          (distr   ps): < 1   3 >
                        << −1   0 >< 1   3 >>
          (distr ps):   < 2   4 >
                        << −1   0 >< 1   3 >>
       >
```

9.application de la forme distr

$\Longrightarrow <$
$<$
ps: $< 1 \quad 3 >< -1 \quad 0 >$
ps: $< 1 \quad 3 >< 1 \quad 3 >$
$>$
$<$
ps: $< 2 \quad 4 > \; < -1 \quad 0 >$
ps:$< 2 \quad 4 > \; < 1 \quad 3 >$
$>$
$>$

10.remplacement de l'appel de la fonction par le corps de la fonction ps
ps: $< 1 \quad 3 > \; < -1 \quad 0 >$
$\Longrightarrow (binu \;\; apply \;\; +) \circ (dist \;\; *)$: $< 1 \quad 3 > \; < -1 \quad 0 >$

11.application de la distribution généralisée
$\Longrightarrow$ (binu apply +): $< * : 1 \quad -1 * : 3 \quad 0 >$

*12.application de la fonction **
$\Longrightarrow$ (binu apply +): $< -1 \quad 0 >$

13.application de binu
$\Longrightarrow$ apply: $+ < -1 \quad 0 >$
14. $\Longrightarrow$ + : -1 0
15 $\Longrightarrow$ -1

Les autres composantes de la matrice sont calculables simultanément. On obtient finalement la liste finale suivante comme résultat de l'expression

$pm : AB$

$<< -1 \quad 10 > \quad < -2 \quad 14 >$

ce qui équivaut à la matrice :

$$A = \begin{pmatrix} -1 & 10 \\ 2 & 14 \end{pmatrix}$$

7 Conclusion

Dans ce chapitre, nous avons décrit succintement le langage GRAAL, un LFSV possédant les propriétés évolués de cette classe de langages, spécifiées au chapitre 1. Nous nous sommes particulièrement attardés sur la polyadicité des fonctions et l'utilisation des fonctions d'ordre supérieur. Grâce à ces deux concepts, GRAAL permet l'évaluation d'expressions de la forme :

$$Exp_f \ : \ Exp_{a_1} \ \ Exp_{a_2} \ \ \ldots \ \ Exp_{a_n}$$

où :

- Exp_{a_i} est une expression quelconque pour tout i.
- Exp_f est de la forme $Exp_{f_1} \circ Exp_{f_2} \circ \ldots \circ Exp_{f_k}$
où chaque $Exp_{f_j} \forall j \in [1, k]$ est une expression ayant pour résultat une fonction.

Le modèle P^3 défini par N. Devesa [Dev90] permet l'évaluation de programmes fonctionnels sans variables. L'extension de ce modèle aux expressions de la forme spécifiée est un de nos principaux objectifs dans le cadre de cette thèse. GRAAL est justement un LFSV où ce type d'expressions est exprimé et évalué. Pour cette raison, le choix du langage GRAAL en tant que représentant des LFSV pour l'extension du modèle P^3 a été retenu.

Il existe d'autres schémas d'exécution des langages fonctionnels. Le chapitre suivant rappelle les principaux concepts de ces modèles et tente de situer le modèle P^3 et de préciser, en les justifiant, les objectifs que nous nous sommes fixés en proposant une extension du modèle P^3.

Chapitre 3

Les modèles d'évaluation

Introduction

Les langages fonctionnels sont reconnus être intrinsèquement parallèles (cf. chapitre 1). Il existe deux approches pour exécuter ces langages en parallèle. La première approche consiste à introduire un parallélisme explicite en ajoutant des annotations aux programmes [Rab93]. Cette approche facilite la sélection des traitements à effectuer en parallèle mais le programme fonctionnel perd son aspect déclaratif et son indépendance vis à vis de tous les aspects liés à la machine cible. Une deuxième approche consiste à exploiter le parallélisme implicite des langages fonctionnels en adoptant une stratégie d'évaluation décrite par un schéma d'exécution ou modèle d'évaluation adéquat. Le travail présenté dans ce mémoire s'inscrit plutôt dans ce deuxième cadre, aussi nous ne consacrerons pas ce chapitre à la comparaison de ces deux approches mais nous y décrirons les principales familles de modèles d'évaluation. Il existe deux familles de modèles d'évaluation qui exploitent le parallélisme intrinsèque des langages fonctionnels : les modèles dataflow et les modèles de réduction.

Le principe d'évaluation dans le modèle dataflow consiste à représenter un programme à l'aide d'un graphe dirigé où les nœuds sont des opérateurs et où les arcs illustrent les flots de données entre les opérateurs. Chaque nœud prend en entrée, les valeurs qui se trouvent sur ses arcs en entrée. Il effectue leur traitement en fonction de l'opérateur qu'il contient puis délivre le résultat sur l'arc en sortie. Dans le cas où un même résultat est utilisé comme entrée de plusieurs nœuds, un opérateur de duplication reproduit le nombre de copies nécessaires d'un même argument. Durant la phase de traitement, un nœud est dit *actif*.

Dans le modèle dataflow, plusieurs opérateurs peuvent être activés simultanément. Le parallélisme dans le modèle dataflow consiste à activer en parallèle

plusieurs nœuds du graphe dataflow. L'activation de plusieurs nœuds, produisant chacun une valeur en entrée d'un même opérateur, est possible. De ce fait, les arguments d'une même fonction sont évalués en parallèle.

Dans le modèle dataflow pur [Den90], une fonction ne peut être appliquée avant réception de ses arguments. Par conséquent, le modèle dataflow ne permet pas l'exploitation du parallélisme vertical. De ce fait, le seul mode d'évaluation pouvant être utilisé est l'appel par valeur. L'appel par nécessité peut être utilisé pour le modèle dataflow à sémantique paresseuse [WG88] où la présence de tous les opérandes sur les arcs en entrée n'est pas indispensable.

Le principe de la réduction, quant à lui, consiste à réécrire une expression en une expression équivalente en utilisant un système de réécriture. Le processus de réduction se termine par l'obtention d'une expression à laquelle on ne peut appliquer aucune règle de réécriture. Cette expression est appelée **forme finale de l'expression** et est mathématiquement équivalente à l'expression initiale. Tout modèle qui adopte le principe de la réduction est appelé **modèle de réduction**. Un modèle de réduction se doit de vérifier la propriété de Church-Rosser [Chu41] i.e quel que soit l'ordre d'application des règles de réécriture, l'expression finale obtenue est unique.

Chaque sous expression à laquelle on peut appliquer une règle de réécriture est un *redex* [1]. Le parallélisme dans les modèles de réduction provient du traitement parallèle de plusieurs redex. De ce fait, a priori, tout modèle de réduction exploite le parallélisme horizontal des langages fonctionnels. Pour cela, il suffit que les redex choisis représentent les arguments d'une même fonction. De même, pour le parallélisme vertical à condition que les fonctions non strictes soient prises en compte. Selon les modèles de réduction, ce type de parallélisme est exploité de façon plus ou moins efficace. Nous reviendrons plus en détail sur ce point dans la suite de ce chapitre. Les modes d'évaluation utilisés dans un modèle de réduction varient d'un modèle à un autre. Nous reviendrons également sur cet aspect dans ce chapitre.

Les modèles de réduction les plus connus sont le modèle de réduction de chaîne et le modèle de réduction de graphe. Il existe un autre modèle nommé P^3 défini par N. Devesa dans sa thèse [Dev90]. Ce modèle est basé sur les principes de réduction. En effet, l'évaluation d'une expression est décrite par un système de réécriture. Nous y reviendrons au chapitre 4. Ce modèle permet l'évaluation

[1] reducible expression

d'expressions simples de la forme

$$f : x$$

où f est une composition de fonctions et x est un argument non fonctionnel et réduit. L'extension du modèle P^3 pour permettre l'évaluation d'expressions plus générales fait l'objet d'une grande partie du travail présenté dans ce mémoire. De ce fait, l'objectif de ce chapitre est donc de situer le modèle P^3, tel qu'il a été défini par N. Devesa, par rapport aux autres modèles de réduction. Dans ce but, nous décrivons successivement le modèle de réduction de chaîne, le modèle de réduction de graphe et le modèle P^3 avant de faire le point sur ces modèles. Pour situer ces différents modèles les uns par rapport aux autres, nous nous baserons sur la représentation de l'expression, sur la localisation des redex et sur le contrôle de l'évaluation des redex.

1 Le modèle de réduction de chaîne

1.1 Principes du modèle

La principale caractéristique du modèle de réduction de chaîne [Mag79] est une représentation de l'expression sans référencement et ce tout au long de l'évaluation. Quelle que soit la représentation de l'expression adoptée, l'expression à évaluer est en forme préfixée i.e le symbole de fonction précède les arguments dans la chaîne. Tout symbole de fonction d'arité n suivi de n expressions représente un redex. Les redex potentiels sont repérés par un parcours de la chaîne du début jusqu'à la fin. L'application d'une règle de réécriture à une sous expression doit permettre l'obtention d'une sous expression équivalente représentée de la même façon. De ce fait, toute forme de partage est interdite. Ainsi, l'identification d'un symbole de fonction dans la chaîne entraîne une duplication systématique du code correspondant pour préserver cette propriété.

1.2 Le parallélisme exploité

Le parallélisme dans ce modèle s'exprime par l'application simultanée de règles de réécriture aux redex. De ce fait les deux formes de parallélisme des langages fonctionnels sont exploitées.

- **Le parallélisme horizontal**
  L'application simultanée de règles de réécriture à des sous expressions ad-

jacentes dans la chaîne et précédées d'un symbole de fonction est possible d'après le principe même de la réduction. Le parallélisme horizontal est donc exploité.

- **Le parallélisme vertical**

  L'exploitation de ce parallélisme est inefficace. En effet, si dans une expression on exploite le parallélisme horizontal et si de plus on réécrit l'expression toute entière par une règle de réécriture, toute duplication d'argument est en fait une duplication de calcul.

1.3 Modes d'évaluation possibles

L'appel par valeur est le seul mode d'évaluation utilisé efficacement pour la réduction de chaîne. L'appel par nécessité ne peut être utilisé puisqu'il contredit les principes d'évaluation de la réduction de chaîne. En effet, si l'application d'une règle de réécriture à une expression où la réduction des arguments est retardée conduit à une duplication d'un argument, cet argument sera évalué plus d'une fois.

2 Le modèle de réduction de graphe

2.1 Représentation d'une expression

Dans le modèle de réduction de graphe [Sch86, Kie85, Pey87], l'expression à réduire est représentée par un graphe syntaxique. Ce graphe se compose de deux types de nœuds : des nœuds intermédiaires appelés *nœuds application* représentés par des "@", et des nœuds feuilles contenant des symboles de fonctions ou des atomes. Un nœud application référence deux sous-graphes : le sous graphe de gauche représente une fonction, le sous graphe de droite représente un argument quelconque. Le graphe constitué d'un nœud application et de ses sous graphes représente l'application de la fonction à cet argument. L'application d'une fonction f à un argument a est représentée par le graphe de la figure 3.1.a. L'application d'une fonction f aux arguments $x_1, x_2, \ldots, x_n$ est représentée par le graphe de la figure 3.1.b.

Un graphe de racine un nœud application symbolise donc un redex.
L'évaluation d'une expression consiste à traiter tous les redex par l'application de règles de réécriture. Chaque sous graphe réécrit par une règle de réécriture

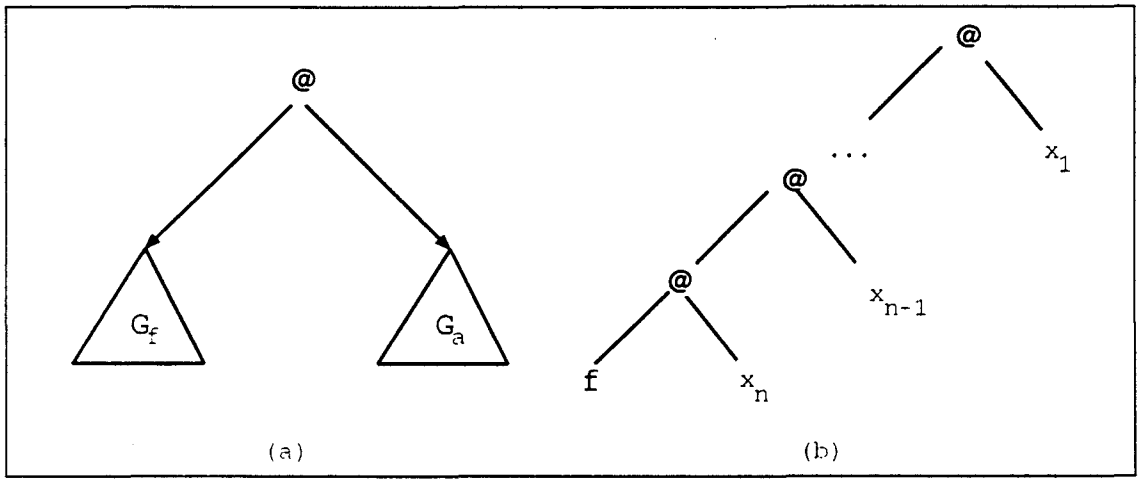


Figure 3.1 : (a).Représentation de l'application *f : a.* (b). Représentation d'une fonction polyadique

remplace l'ancien dans le graphe global.

2.2 Evaluation d'une expression

L'évaluation d'une expression représentée par un graphe commence à la racine de ce graphe. Celui ci est parcouru de haut en bas en commençant par la branche gauche à chaque fois. Le but d'un tel parcours est de rechercher les redex près à être réduits i.e dont la réécriture ne dépend pas de celle d'autres redex. Ces redex se trouvent en profondeur dans le graphe. Le parcours en profondeur continue jusqu'à rencontrer un symbole de fonction. Ce symbole rencontré permet de déterminer le nombre d'arguments impliqués, et par conséquent le sous graphe concerné par la réécriture. La rencontre d'une fonction simple conduit au remplacement du sous graphe par le résultat de l'application de la fonction à ses arguments. La rencontre d'une fonction complexe conduit à la réécriture du graphe. Le nouveau sous graphe est réorganisé similairement au graphe initial i.e les redex qui peuvent immédiatement être réécrits sont situés en profondeur dans le sous graphe.

2.3 Le partage de graphes

Le partage de sous graphes est un des points forts du modèle de réduction de graphe. En effet, il permet d'éviter la duplication de sous graphes mais plus généralement la duplication de calcul car les sous graphes dupliqués ne sont pas toujours dans l'état irréductible. Ce partage est autorisé pour une raison bien simple : le graphe tel qu'il est défini pour le modèle de réduction de graphe est tel que chaque nœud intermédiaire du graphe référence deux sous graphes. Le partage de sous graphes ne modifie en rien cette définition. Ce partage de sous graphes permet donc de pallier les deux principaux inconvénients du modèle de réduction de chaîne, à savoir, la duplication de calcul et la duplication du code pour remplacer un symbole de fonction.

2.4 Contrôle de l'évaluation

Les nœuds application dans le modèle de réduction de graphe permettent de contrôler l'évaluation. En effet, par sa présence dans le graphe, un nœud application indique l'existence d'un redex qu'il faut traiter par une règle de réécriture. Les nœuds application permettent aussi de guider le parcours de recherche de redex. Enfin ces nœuds assurent également la synchronisation des réécritures effectuées sur le graphe. Cette synchronisation n'empêche pas le traitement de redex où un ou plusieurs arguments ne sont pas réduits. Le partage de graphe évite dans ce cas la duplication éventuelle de redex. Ainsi, un graphe dont le sous graphe gauche n'est pas réduit ne peut lui-même être réduit avant que ce dernier ne devienne irréductible. Inversement, dès qu'un sous graphe est réduit, les graphes qui le référencent peuvent être pris en compte par une règle de réécriture. Les nœuds application permettent donc la synchronisation des réécritures effectuées sur le graphe.

2.5 Réduction de graphe parallèle

Dans le modèle de réduction de graphe, plusieurs réductions peuvent être déclenchées simultanément [CDL87, LKID88, BVP+87, AJ89] en déclenchant en parallèle l'évaluation de plusieurs sous graphes. De ce fait, le modèle de réduction de graphe exploite les deux formes de parallélisme des langages fonctionnels : en effet, la représentation curryfiée des applications illustrée par le graphe de la figure 3.1.b, montre bien que les sous graphes représentant chacun des arguments de la fonction sont indépendants et se situent sur le chemin parcouru à la recherche

d'expressions réductibles. L'évaluation de ces sous expressions peut donc être déclenchée en parallèle lors du parcours. Si de plus la fonction appliquée est une fonction non stricte, la transformation du sous graphe peut se faire en même temps que celle des sous graphes représentant les arguments.

2.6 Modes d'évaluation utilisés

Le modèle de réduction de graphe permet l'utilisation des modes d'évaluation suivants : l'appel par valeur [2], l'appel par nom [3] et l'appel par nécessité [4].

- L'appel par valeur consiste à ne réécrire un sous graphe que lorsque tous ses sous graphes sont irréductibles. L'utilisation de ce mode d'évaluation offre la possibilité de déclencher dans la phase de descente dans le graphe, la réduction de tous les sous graphes rencontrés. On retrouve donc les avantages et les inconvénients de ce mode d'évaluation : l'avantage en est que tous les sous graphes peuvent être évalués en parallèle. L'inconvénient est que pour les fonctions non strictes, des sous graphes peuvent être évalués inutilement.

- Grâce au partage possible de graphes, chaque sous graphe est évalué au plus une fois. L'utilisation de l'appel par nécessité est donc possible. La réduction des arguments est cependant quelque peu retardée, le temps d'identifier la fonction appliquée. La réduction inutile de sous graphes est par contre évitée.

- Du fait du partage, l'utilisation de l'appel par nom est identique à l'appel par nécessité dans le modèle de réduction de graphe.

En résumé, le modèle de réduction de graphe est donc un modèle où l'expression est représentée par un graphe syntaxique constitué d'un ensemble de redex préétablis. Les nœuds application symbolisent ces redex. Ces mêmes nœuds établissent le contrôle dans le graphe.

[2] l'*appel par valeur* consiste à évaluer tous les paramètres effectifs d'une fonction avant l'application de celle-ci

[3] l'*appel par nom* permet de retarder au maximum l'évaluation des arguments en les transmettant sous leur forme non évaluée aux fonctions qui les utilisent.

[4] l'*appel par nécessité* est une forme optimisée de l'appel par nom. L'évaluation de l'expression est retardée au maximum comme dans le cas précédent sauf que dans ce cas, une expression est évaluée une seule fois.

Nous pouvons d'ores et déjà remarquer pour les modèles de réduction présentés, que la représentation de l'expression a beaucoup d'influence sur le déroulement des réductions et sur les possibilités du modèle. De plus, cette représentation "contient" les redex suceptibles d'être traités même si leur traitement ne peut se faire à cause de la dépendance vis à vis d'autres redex. Dans le modèle de réduction de chaîne, chaque symbole de fonction d'arité n suivi de n expressions est un redex potentiel. Dans le modèle de réduction de graphe, chaque nœud application symbolise un redex potentiel. D'autres redex seront introduits en cours d'évaluation lors du remplacement d'un symbole de fonction par le corps de la fonction dans le modèle de réduction de chaîne ou lors d'une transformation d'un sous graphe en un sous graphe non réduit - contenant des redex- dans le modèle de réduction de graphe. Ces redex sont insérés dans la chaîne dans le modèle de réduction de chaîne. Dans le modèle de réduction de graphe, ceci se traduit par la construction d'un nouveau graphe selon le modèle du graphe initial. Ces redex ne sont pas tous exploités immédiatement. A la différence de ces modèles, dans le modèle P^3 présenté dans la section suivante, seuls les redex prêts à être traités existent dans la représentation initiale. Les autres redex seront constitués au fur et à mesure, dans l'ordre dans lequel ils doivent être traités.

3 Le modèle P^3

Le modèle P^3 est un modèle d'évaluation parallèle des langages fonctionnels sans variables défini par N. DEVESA dans sa thèse [Dev90]. Le modèle P^3 peut être considéré comme un modèle de réduction. En effet, le passage d'une étape à une autre, pour évaluer une expression, peut être décrit à l'aide d'un système de réécriture. Pour le prouver, le chapitre 4 est dédié à la description du système de réécriture associé au fonctionnement de P^3.

Dans ce modèle, la représentation de l'expression à réduire se fait à l'aide d'un ensemble d'*arborescences fonctionnelles* et d'*arbres de données*. Les arborescences fonctionnelles servent à la représentation des fonctions et les arbres de donnée à la représentation des données [5] présentes dans l'expression à évaluer. Dans le modèle P^3, les fonctions sont placées dans les arborescences fonctionnelles dans l'ordre où elles doivent être appliquées aux arguments. La recherche d'expressions réductibles consiste donc en un parcours de haut en bas des arborescences fonctionnelles. Ce parcours est assuré par le mécanisme d'exploration des arborescences fonctionnelles. Chaque nouvelle fonction de l'arborescence fonctionnelle explorée symbolise la présence d'une sous expression non réduite. La réduction

[5] le langage FP n'étant pas d'ordre supérieur, les seuls arguments possibles sont des données

de cette sous expression consiste donc dans un premier temps à transmettre aux arguments concernés la fonction explorée. Cette transmission symbolise une demande de réduction qui sera prise en compte par les arguments.

Le modèle P^3 est un modèle de réduction mixte en ce sens qu'il possède quelques caractéristiques du modèle de réduction de chaîne et du modèle de réduction de graphe. Avant de situer plus précisément P^3 par rapport à ces deux modèles, nous donnons d'abord un bref aperçu de P^3. Cette description succinte de P^3 aborde successivement la représentation des expressions et les mécanismes d'exploration et de réduction du modèle.

3.1 Le point sur le modèle P^3

La première version du modèle P^3 a été faite en restreignant le domaine d'étude aux systèmes FP. De ce fait, le modèle intègre uniquement les concepts de base des LFSV définis au chapitre 1 et permet **uniquement** l'évaluation des expressions ayant la forme suivante :

P: a

où P est un programme LFSV de premier ordre i.e il s'agit d'un ensemble de définitions où chaque définition est une composition de fonctions ordinaires. Ces fonctions sont soit des fonctions primitives, des définitions ou des formes fonctionnelles. L'argument *a* est **nécessairement** une donnée et est l'**argument** de P. Les fonctions dans le modèle P^3 sont représentées à l'aide des arborescences fonctionnelles. Les données quant à elles sont représentées à l'aide des arbres de données. De ce fait, programme et argument sont nécessairement représentés différemment puisque, pour les LFSV considérés, un argument ne peut être une fonction. Par conséquent, le modèle propose une représentation séparée des programmes et des arguments dans deux espaces de représentation : l'**espace d'exploration** pour la représentation des programmes et l'**espace de réduction** pour la représentation de l'argument comme le montre la figure 3.2.

3.2 Représentation d'un programme

Un programme LFSV est constitué d'un ensemble de définitions. Parmi ces définitions, l'une peut être considérée comme le moteur du programme. Elle est appelée la **définition principale** du programme. Par opposition, les autres définitions sont considérées comme les **définitions secondaires** du programme.

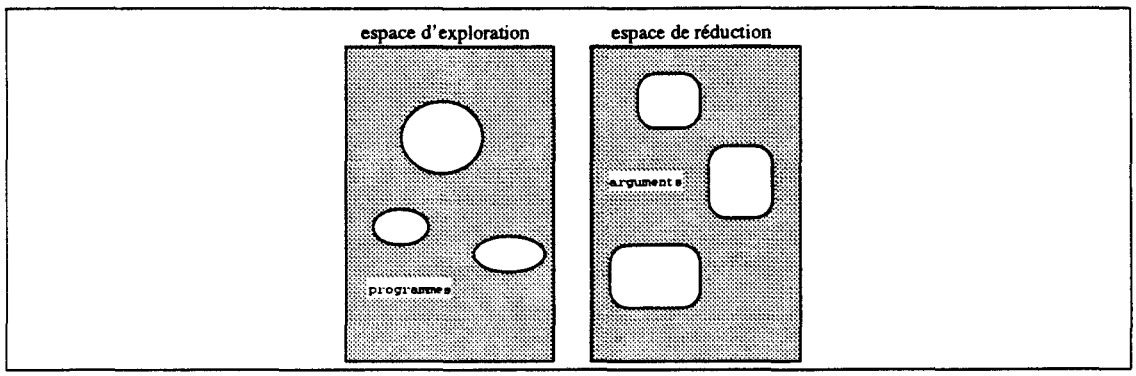


Figure 3.2 : Représentation générale du modèle

Le programme est représenté dans l'espace d'exploration par un ensemble d'arborescences fonctionnelles. L'une d'elle, appelée l'**arborescence fonctionnelle principale**, représente la définition principale. Les autres représentent chacune une des définitions secondaires du programme.

Chaque définition faisant partie du programme est une composition de fonctions où chaque fonction est soit une fonction primitive, une occurrence d'utilisation d'une définition ou une forme fonctionnelle.
Une arborescence fonctionnelle est composée d'un ensemble de nœuds reliés par trois types d'arcs: $\downarrow$,$\hookrightarrow$ et $\longrightarrow$.

L'arborescence fonctionnelle représentant une définition est constituée de la façon suivante:

- Chaque fonction primitive est représentée par un **nœud fonctionnel** appelé **nœud fonction primitive.**

- Chaque occurrence d'utilisation d'une définition est représentée par un nœud fonctionnel appelé **nœud définition.** Ce nœud référence l'arborescence fonctionnelle de la définition utilisée.

- Une forme fonctionnelle admet n paramètres. Chaque paramètre est une composition de fonctions représentée chacune par une sous arborescence fonctionnelle.
  La représentation complète d'une forme fonctionnelle est constituée d'un nœud contenant le symbole de la forme fonctionnelle appelé **nœud forme fonctionnelle.** Un arc de type "$\hookrightarrow$" relie ce nœud à la sous arborescence fonctionnelle représentant le premier paramètre. Les arborescences fonctionnelles représentant deux paramètres consécutifs, sont reliées par un

arc de type "⟶".

- L'arborescence fonctionnelle représentant la définition d=$f_1 \circ \ldots \circ f_n$ s'obtient en reliant par des arcs de type "↓" les représentations des fonctions f_n, f_{n-1}, ..., f_1.

définitions

1. Un nœud extrémité d'un arc de type ↓ ayant comme origine un autre nœud fonctionnel n, est le **successeur** du nœud n.
2. Les nœuds fonctionnels reliés par des arcs de type "↓" constituent un **chemin séquentiel** de l'arborescence fonctionnelle.

Exemple
La fonction qui calcule le produit scalaire se définit par :

$$def \;\; ps \equiv (binul \;\; apply \;\; +) \;\; \circ \;\; \alpha \;\; *$$

L'arborescence fonctionnelle représentant cette fonction est illustrée par la figure 3.3.

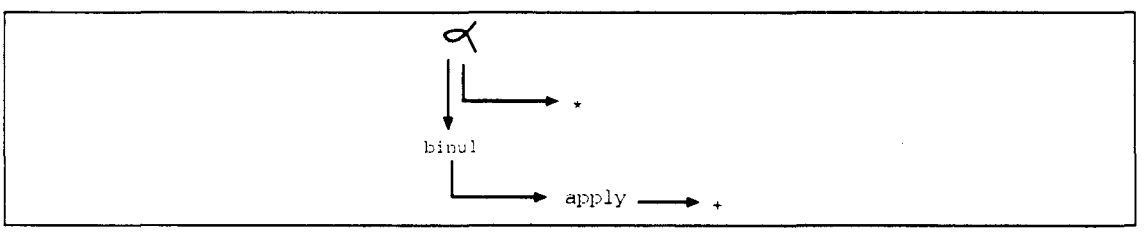


Figure 3.3 : Représentation du produit scalaire

1. Le noeud fonctionnel α est la **racine** de l'arborescence fonctionnelle.
2. Les noeuds α et *binul* constituent un chemin séquentiel.

4 Représentation d'un argument

Dans les systèmes FP, un argument est soit un atome soit une séquence. Dans les deux cas, il est représenté à l'aide d'une structure appelée **arbre de données**.

Cet arbre de données est construit de la façon suivante :

- Un atome est représenté par un **nœud de donnée** contenant la valeur de l'atome.
- Chaque élément d'une séquence est soit un atome soit une séquence. Une séquence est représentée par un arbre binaire dont les feuilles sont des atomes.

La figure 3.4 schématise l' arbre de données représentant la séquence

<< 1 2 > < 3 4 >>

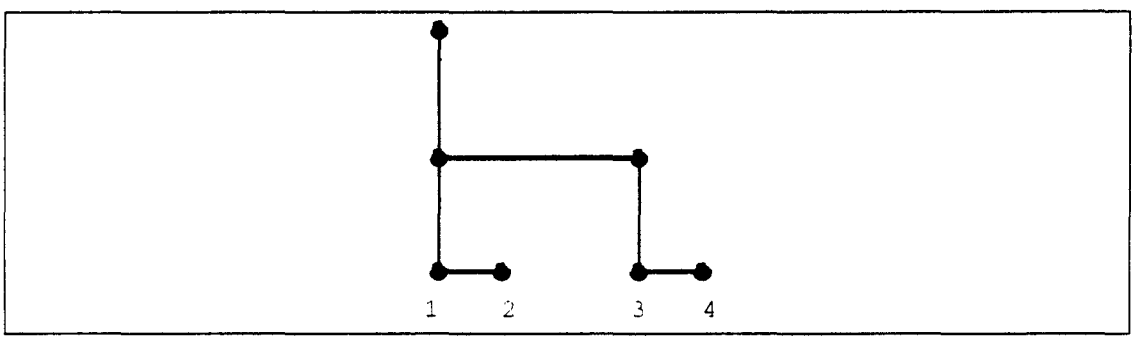


Figure 3.4 : Représentation d'une séquence

4.1 Evaluation d'une expression

L'évaluation de l'application d'un programme à son argument dans le modèle P^3 se base sur deux activités parallèles : l'exploration des arborescences fonctionnelles par **activation** des nœuds fonctionnels et la **réduction** de l'argument. L'exploration d'une arborescence fonctionnelle a pour but de permettre à chaque nœud fonctionnel d'une arborescence fonctionnelle de "prendre connaissance" de son argument. Cet argument est désigné par le nœud racine de l'arbre de donnée le représentant. Ce nœud est appelé **nœud cible** du nœud fonctionnel. Activer un nœud fonctionnel revient donc à lui associer son nœud cible. L'exploration d'une arborescence fonctionnelle s'effectue en respectant les règles suivantes:

- L'exploration commence à la racine de l'arborescence fonctionnelle principale.
- Chaque nœud fonctionnel activé déclenche l'exploration de son successeur, s'il existe.

- Chaque nœud définition activé déclenche l'exploration de l'arborescence fonctionnelle principale de la définition utilisée.
- Chaque nœud forme fonctionnelle activé déclenche l'exploration parallèle des sous arborescences fonctionnelles représentant ses paramètres.

Ainsi l'exploration d'une arborescence fonctionnelle engendre l'exploration d'autres arborescences fonctionnelles. Elle provoque également dans le cas où le programme comporte des formes fonctionnelles, l' exploration parallèle des sous arborescences fonctionnelles de l'arborescence.
Chaque nœud activé effectue une demande de réduction sur l'argument désigné par le nœud cible qu'il a reçu grâce au mécanisme de l'exploration. La réduction à effectuer dépend de la fonction représentée par le nœud fonctionnel. Ces réductions sont de deux types: des réductions par des fonctions primitives qui permettent l'obtention d'un résultat, par exemple l'application de la fonction "+" à l'argument < 1 2 > permet l'obtention de l'objet résultat "3". Le deuxième type de réduction est engendré par des formes fonctionnelles. Il s'agit plutôt, dans ce cas, de préparer les arguments associés ultérieurement aux paramètres de la forme fonctionnelle à partir de l'argument de celle-ci. Par exemple une requête de réduction issue d'un nœud fonctionnel représentant la forme fonctionnelle "[]" (cf. chapitre 1, section 4.1, ayant deux paramètres, à l'argument < 1 2> permet la construction de deux copies de cet argument pour les deux paramètres de la forme fonctionnelle afin de pouvoir appliquer à l'argument les deux paramètres de la forme fonctionnelle en parallèle.
L'évaluation d'une expression consiste donc à satisfaire **toutes** les demandes de réductions issues des nœuds fonctionnels activés. L'arbre de données obtenu suite à l'accomplissement de ces réductions est la forme *réduite* de l'expression évaluée.

4.2 Contrôle de l'évaluation

Le contrôle dans le modèle P^3 est assuré par deux mécanismes différents :

- L'exploration : elle permet de construire les redex dans l'ordre où ils doivent être exécutés.
- L'ordonnancement du traitement des redex : en effet, la construction des redex et le traitement de ces derniers étant deux activités asynchrones, il faut préserver un ordre de traitement identique à l'ordre de construction des redex, pour obtenir des résultats cohérents. L'algorithme d'ordonnancement est décrit dans [Dev90].

4.3 Le parallélisme dans le modèle P^3

Dans le modèle P^3, trois activités s'effectuent en parallèle : l'exploration des arborescences fonctionnelles, la construction des redex, et l'évaluation des redex ce qui lui a valu le nom de P^3.

Le modèle P^3 permet d'évaluer des expressions simples où chaque fonction admet un seul argument. Dans ces expressions exprimées dans le langage FP [Bac78], la polyadicité est introduite par la notion de séquence. En effet, une fonction d'arité n, sémantiquement polyadique, admet pour argument dans FP une séquence (cf. chapitre 1) dont les éléments sont les arguments de la fonction. Le parallélisme horizontal est exploité grâce à la présence de formes fonctionnelles dont les paramètres s'appliquent simultanément à ces éléments. Cette situation est illustrée par la figure 3.5 représentant une composition de fonctions contenant une forme fonctionnelle *ff* admettant n paramètres, $p_1, p_2, \ldots, p_n$. La fonction g dans le chemin admet pour argument, le résultat des redex formés à partir des fonctions $p_1, p_2, \ldots, p_n$. Il s'agit là du seul cas où le parallélisme horizontal est exploité.

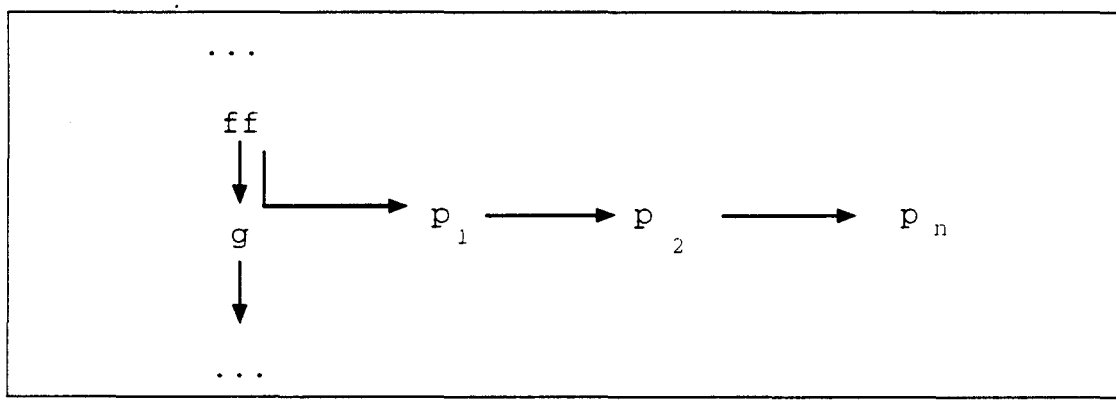


Figure 3.5 : Exemple de fonctionnelle

Par ailleurs, le parallélisme vertical n'est pas exploité puisque d'une part le traitement d'expressions de la forme $E_f : E_{x1} \ldots E_{xn}$ n'est pas pris en compte dans P^3. De plus, dans le cas illustré par la figure 3.5, le redex impliquant la fonction g ne peut être pris en compte pendant l'application des paramètres à leurs arguments respectifs et ce pour deux raisons :

- La synchronisation de l'exploration [Dev90] empêche l'exploration du nœud g avant la fin de l'exploration des sous arborescences représentant les fonctions $p_1, p_2, \ldots, p_n$.
- L'ordonnancement des requêtes ne permet à ce redex d'être pris en compte qu'une fois les derniers redex impliquant des fonctions sont pris en compte.

Pour résumer, le modèle P^3 est un modèle de réduction où l'expression à réduire est représentée par un ensemble d'arborescences fonctionnelles et d'arbres de données. Les redex se constituent au fur et à mesure par le mécanisme d'exploration, en établissant un lien dynamique entre la fonction appliquée et ses arguments. La réduction de l'expression consiste à prendre en compte ces redex en appliquant chaque fonction à ses arguments.

5 Le point sur les modèles existants

L'objectif de cette section est de situer le modèle P^3, tel qu'il a été défini par N. Devesa, par rapport aux autres modèles de réduction.

5.1 P^3 par rapport au modèle de réduction de chaîne

Un redex dans P^3 constitue une liaison dynamique entre une fonction et ses arguments. Ce lien peut être interprété comme une duplication de la fonction appliquée. Il s'agit donc là, d'une forme de duplication de code. Vu sous cet angle, nous pouvons considérer qu'il s'agit là d'un point commun du modèle P^3 et du modèle de réduction de chaîne. La duplication de code dans P^3 se fait de façon progressive comparée à la duplication de code dans le modèle de réduction de chaîne où le code se duplique entièrement. L'avantage de la première forme de duplication est que seules les portions du code utilisées sont dupliquées. En effet dans le cas d'un programme de la forme $(if\ \ P\ \ \ F\ \ \ G)$ où P, F et G sont des fonctions quelconques, seule la fonction F ou la fonction G est dupliquée.

5.2 P^3 par rapport au modèle de réduction de graphe

Analysons la représentation de l'expression dans le modèle P^3. Une expression est représentée dans le modèle P^3 par un ensemble d'arbres de deux types : les arborescences fonctionnelles et les arbres de données. Initialement ces éléments composant la représentation sont indépendants. Les liens entre les fonctions et les arguments auxquels elles s'appliquent s'établissent au fur et à mesure grâce au mécanisme de l'exploration. Chaque fonction liée à ses arguments constitue un redex. Autre élément important, les fonctions sont situées dans les arborescences fonctionnelles dans l'ordre normal dans lequel elles s'appliquent. Il en découle que les premiers redex constitués par l'exploration sont ceux qui peuvent logiquement être traités sans attente. En tenant compte de ces constatations, nous pouvons considérer que le modèle P^3 s'apparente au modèle de réduction de graphe, et ce

pour différentes raisons. D'abord par sa représentation; en effet, la représentation de l'expression dans P^3 est un graphe qui est construit dynamiquement en cours d'évaluation tandis que dans le modèle de réduction de graphe, ce graphe est construit à la compilation. La construction du graphe dans P^3 est contrôlée par le mécanisme d'exploration. Les premiers redex construits dans P^3 sont ceux qui peuvent s'exécuter en premier, autrement dit ceux qui se trouvent en profondeur dans le graphe syntaxique du modèle de réduction de graphe. Le graphe est donc construit en débutant par le bas ce qui évite la phase de parcours en profondeur du graphe à la recherche de redex dans P^3.

Une deuxième constatation concerne le traitement des redex impliquant des fonctions complexes. Dans le modèle de réduction de graphe, le traitement de ce type de redex conduit à une réorganisation du graphe. Dans le modèle P^3, cette réorganisation est en quelque sorte évitée, grâce à la structure même des arborescences fonctionnelles. Ce point sera repris dans le chapitre 6 après description plus détaillée de la représentation des fonctions dans le modèle P^3 (cf. chapitre 4).

6 Critiques du modèle P^3

Le modèle P^3 a été réalisé pour permettre l'évaluation d'expressions simples de la forme :

P: x

où :

- P est une composition de fonctions de la forme $f_1 \circ f_2 \circ \ldots \circ f_n$ telle que chaque fonction f_i n'est pas une composition de fonctions et telle que chaque fonction f_i est une fonction réduite.
- x est un argument réduit.

Il ne permet pas l'évaluation de l'expression fonctionnelle dans sa forme la plus générale que nous rappelons ci-dessous :

$$\text{P} : x_1\ x_2\ \ldots\ x_n$$

où P, x_1,x_2,...,x_n sont des expressions dans un état quelconque (réduit ou non réduit). Cette restriction s'explique en partie par le fait que les fonctions d'ordre

supérieur ne sont pas prises en compte. En effet, si la fonction P est une expression non réduite de la forme g: y1, y2, ... , yn, la fonction g se compose nécessairement de fonctions d'ordre supérieur puisque le résultat de l'expression P est une fonction.

Les fonctions d'ordre supérieur ne sont pas gérées dans le modèle P^3 pour deux raisons :

- L'utilisation des fonctions d'ordre supérieur suppose l'utilisation des fonctions comme arguments d'autres fonctions. Ceci pose un problème de représentation dans le modèle P^3 puisque l'espace de réduction ne contient que des arbres de données.

- l'utilisation des fonctions d'ordre supérieur suppose la possibilité d'obtenir une fonction en résultat. D'une part, il se pose alors le problème de savoir dans quel espace se retrouve ce résultat fonctionnel, d'autre part, cette fonction résultat est représentée à l'aide d'une arborescence fonctionnelle. La construction des arborescences fonctionnelles n'est pas prise en compte dans le modèle défini par N. Devesa.

Le deuxième aspect indispensable au traitement de telles expressions est la synchronisation de l'évaluation de plusieurs expressions dépendantes i.e dont le résultat de l'une constitue la fonction ou l'un des arguments de l'autre application. Cette synchronisation doit être suffisamment souple pour permettre l'application de fonctions non strictes aux arguments dont l'évaluation n'est pas achevée. De même, on doit pouvoir anticiper l'application d'une fonction, obtenue en résultat, sans attendre la fin de l'évaluation de celle-ci et ce, pour des raisons évidentes d'efficacité.

Le modèle P^3 ne permet cependant pas l'exploitation du parallélisme vertical i.e l'anticipation de l'application d'une fonction à ses arguments quel que soit l'état d'avancement de l'évaluation de ces derniers. En effet, :

- Initialement, ceci est vrai puisque par hypothèse, la fonction appliquée et son argument sont réduits. Il n'y a donc pas d'application anticipée de la fonction à ses arguments.

- En cours d'évaluation, il n'y a pas d'application anticipée d'une fonction à ses arguments puisque l'ordonnancement interdit la prise en compte d'un redex dont l'argument n'est pas réduit.

Conclusion

Dans ce chapitre, nous avons tenté de situer le modèle P^3 par rapport aux autres modèles de réduction. P^3 ressemble plus à un modèle de réduction de graphe où les redex sont constitués en cours d'évaluation dans l'ordre où ils devraient être évalués. Ceci évite tout parcours préalable du graphe à la recherche des premiers redex à évaluer. De plus, les transformations du graphe, en graphes non réduits, dans le modèle de réduction de graphe, sont très coûteuses dans le modèle de réduction de graphe, dans la mesure où ces graphes sont reconstruits en cours d'évaluation puis ensuite explorés sur le même modèle que le graphe d'origine c.à.d avec les redex succeptibles d'être réduits en premier en profondeur dans le graphe. Dans P^3, ces transformations du graphe sont prévues à la compilation dans la construction des arborescences fonctionnelles. Ainsi, ces transformations n'ont pas lieu. De plus, les redex sont encore une fois formés dans l'ordre logique d'évaluation. Ces deux aspects de P^3 portent à croire que ce dernier présente quelques avantages par rapport au modèle de réduction de graphe classique.
Le modèle P^3 présente cependant quelques limitations qui réduisent considérablement sa puissance. Notre objectif sera donc de l'étendre pour généraliser l'évaluation des expressions et permettre l'exploitation du parallélisme intrinsèque des langages fonctionnels. Cette extension se fera par la prise en compte des fonctions d'ordre supérieur et des fonctions polyadiques et par une synchronisation assez souple de l'évaluation des expressions interdépendantes. Le modèle P^3 étendu est décrit dans le chapitre 4 et optimisé dans le chapitre 5. Le chapitre 6 sera consacré à une comparaison du modèle P^3 étendu et du modèle de réduction de graphe.

Partie II

Le modèle P^3

Chapitre 4

Le Modèle P^3 : définition formelle

Dans ce chapitre, nous décrivons formellement et en détail, le modèle P^3. Cette description concerne la version étendue du modèle qui intègre les concepts évolués des LFSV, notamment la polyadicité des fonctions et l'utilisation des fonctions d'ordre supérieur. Ce chapitre couvre deux aspects importants qui caractérisent le modèle : la représentation des objets et les mécanismes d'évaluation. A cet effet, nous avons défini une fonction de représentation $\Re$ qui associe une représentation à chaque objet d'un LFSV. Cette représentation est basée sur une seule structure : les arbres de réduction [1].
Le deuxième aspect important de cette description concerne les mécanismes fondamentaux grâce auxquels une expression peut être évaluée dans le modèle. Dans P^3, il en existe deux : l'*exploration* des arborescences fonctionnelles et la *réduction* des arguments. Le modèle P^3 étant considéré comme un modèle de réduction, nous avons choisi de décrire ces deux mécanismes à l'aide d'un système de réécriture.
Les objets du langage sont classés en deux groupes : les objets qui se présentent sous une forme réduite, (non calculable) et les applications qui sont les objets calculables du langage. Ce chapitre est donc structuré de la manière suivante : la section 1 définit la fonction de représentation $\Re$ pour les objets non calculables d'un LFSV. La section 2 est consacrée à la représentation des applications qui sont les objets calculables du langage. Cette représentation nous sera utile

[1] les arborescences fonctionnelles et les arbres de données, définis au chapitre précedent, sont des cas particuliers d'arbres de réduction

en section 3 pour la description de la représentation des formes fonctionnelles définies qui sont des *cas particuliers d'objets calculables.* Nous présentons ensuite le système de réécriture décrivant les mécanismes du modèle en section 4. Enfin, dans la section 5, nous présentons une série d'exemples à travers lesquels nous mettons en évidence l'exploitation dans P^3 des deux formes de parallélisme des langages fonctionnels : le parallélisme horizontal et le parallélisme vertical.

1 La fonction de représentation $\Re$

Dans cette section, nous définissons la fonction de représentation $\Re$. Cette fonction associe à chaque objet o de l'ensemble O caractérisant un LFSV donné, une représentation dans le modèle P^3 notée $\Re$(o). Nous nous limitons dans un premier temps aux objets "réduits" [2] du langage. Trois entités sont essentielles dans la représentation des objets d'un LFSV : les arborescences fonctionnelles et les arbres de données et les arbres de réduction. Dans un premier temps, nous définissons donc ces deux notions et ensuite, nous nous intéresserons successivement à la représentation des trois principaux types d'objets d'un LFSV c.à.d les fonctions, les atomes et les listes.

1.1 Les arborescences fonctionnelles

La notion d'arborescence fonctionnelle a déjà été introduite au chapitre précédent. Nous en donnons une définition plus formelle.

Définition
Une arborescence fonctionnelle est un arbre dont les nœuds sont appelés **nœuds fonctionnels**, dont les arcs sont orientés et de trois types différents notés : "$\downarrow$", "$\longrightarrow$" et "$\hookrightarrow$" tels qu'il n'existe pas deux arcs de même origine qui soient de même type.

définitions

- On note **Rac(A)** le noeud racine de l'arborescence fonctionnelle A.
- **successeur d'un nœud:**
  Soient n_1 et n_2 deux nœuds fonctionnels distincts. Le nœud n_2 est dit le

[2] Rappel : par objet réduit, nous entendons les fonctions simples, les atomes, les listes et les compositions de fonctions simples.

successeur du nœud n_1 si et seulement si il existe un arc de type "↓" de n_1 vers n_2.

- **chemins séquentiels**
  Soient n_1,n_2,...,n_k un ensemble de nœuds fonctionnels appartenant à une même arborescence fonctionnelle A. Les nøeuds n_1,n_2,...,n_k (dans cet ordre) constituent un chemin séquentiel de l'arborescence fonctionnelle A si et seulement si pour tout i appartenant à l'intervalle [2,k], le nœud n_i est le successeur du nœud n_{i-1}. Le nœud n_k n'admet pas de successeur. Ce chemin est noté CS($n_1 \mapsto n_k$).

- n_1 est appelé le **sommet** de ce chemin séquentiel et est noté **S(CS**($n_1 \mapsto n_k$)).

- Le chemin CS($n_1 \mapsto n_k$) est appelé **chemin séquentiel maximal** si et seulement si le nœud fonctionnel n_1 n'est le successeur d'aucun nœud de l'arborescence fonctionnelle.

- Le chemin séquentiel CS($n_1 \mapsto n_k$) est appelé **chemin séquentiel principal** de l'arborescence fonctionnelle A ssi CS($n_1 \mapsto n_k$) est un chemin séquentiel maximal ayant pour sommet le nœud fonctionnel Rac(A). On le note alors **CSP(A)**.

Exemples la figure 4.1 représente une arborescence fonctionnelle quelconque A où les nœuds r_i, r_{ij} et r_{ijk} sont des nœuds fonctionnels.

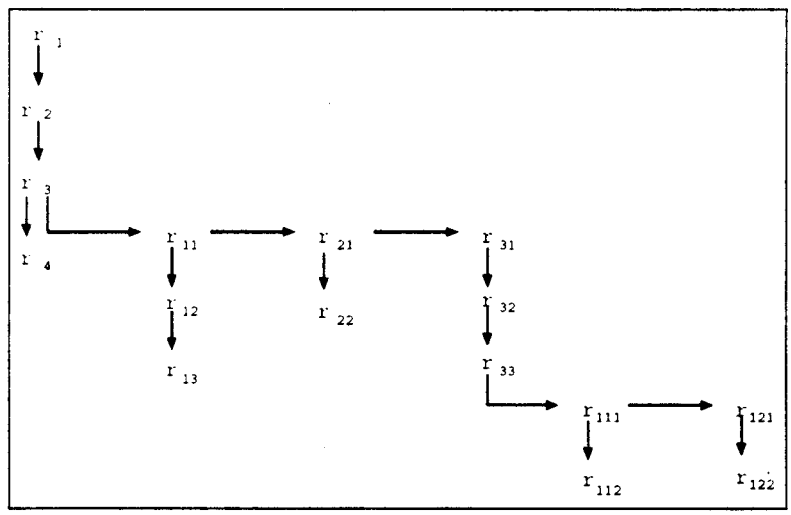


Figure 4.1 : Exemple d'arborescence fonctionnelle

1. Le nœud fonctionnel r_1 est la racine de l'arborescence fonctionnelle.
2. $CS(r_2 \mapsto r_4)$ est un chemin séquentiel de l'arborescence fonctionnelle.
3. L'arborescence fonctionnelle A admet comme chemins séquentiels maximaux $CS(r_1 \mapsto r_4)$, $CS(r_{11} \mapsto r_{13})$, $CS(r_{21} \mapsto r_{22})$,$CS(r_{31} \mapsto r_{33})$, $CS(r_{111} \mapsto r_{112})$ et $CS(r_{121} \mapsto r_{122})$.
4. $CS(r_1 \mapsto r_4)$ est le chemin séquentiel principal de l'arborescence fonctionnelle A.

L'arborescence fonctionnelle représentant une composition de fonctions est obtenue à partir des arborescences fonctionnelles représentant chaque fonction en utilisant les fonctions de constructions $\downarrow_n$, $\longrightarrow_n$, $\hookrightarrow_n$ définies comme suit :

Soit A_1 et A_2 deux arborescences fonctionnelles disjointes i.e telles qu'il n'existe aucun arc dont l'origine est un nœud de l'arborescence fonctionnelle A_1 et dont l'extrémité est un nœud de l'arborescence fonctionnelle A_2 ou vice-versa. Soit n un noeud quelconque de l'arborescence fonctionnelle A_1

On notera :

- $\boldsymbol{A_1 \downarrow_n A_2}$ la structure obtenue en reliant A_1 et A_2 par l'arc $n \downarrow Rac(A_2)$
- $\boldsymbol{A_1 \longrightarrow_n A_2}$ la structure obtenue en reliant A_1 et A_2 par l'arc $n \longrightarrow Rac(A_2)$, et
- $\boldsymbol{A_1 \hookrightarrow_n A_2}$ la structure obtenue en reliant A_1 et A_2 par l'arc $n \hookrightarrow Rac(A_2)$.

Propriétés :

- Si, dans A_1, le noeud fonctionnel n n'a pas de successeur, alors $\boldsymbol{A_1 \downarrow_n A_2}$ est une arborescence fonctionnelle.
- Si, dans A_1, le noeud fonctionnel n n'est pas l'origine d'un arc $\longrightarrow$, alors $\boldsymbol{A_1 \longrightarrow_n A_2}$ est une arborescence fonctionnelle.
- Si, dans A_1, le noeud fonctionnel n n'est pas l'origine d'un arc $\hookrightarrow$, $\boldsymbol{A_1 \hookrightarrow_n A_2}$ est une arborescence fonctionnelle.

Démonstration :
la démonstration de ces propriétés est triviale; prenons le cas de la liaison $A_1 \downarrow_n A_2$. La propriété se démontre similairement pour les 2 autres types de liaisons.

A_1 et A_2 sont par hypothèse toutes deux des arborescences fonctionnelles et par conséquent elles vérifient les 3 points de la définition des arborescences fonctionnelles (cf. section 1.1). Pour que $A_1 \downarrow_n A_2$ soit une arborescence fonctionnelle il suffit de vérifier :

1. L'unicité du nœud racine : $A_1 \downarrow_n A_2$ admet pour nœud racine $\mathrm{Rac}(A_1)$. En effet, c'est le seul nœud de $A_1 \downarrow_n A_2$ qui ne reçoit aucun arc d'un autre nœud ($\mathrm{Rac}(A_2)$ reçoit un arc de type $\downarrow$ en provenance du nœud n).

2. Du nœud n part exactement un arc de type $\downarrow$ dans $A_1 \downarrow_n A_2$ puisque par hypothèse, il n'avait pas de successeur dans l'arborescence A_1.

3. Le nœud $\mathrm{Rac}(A_2)$ reçoit **un** arc; c'est l'arc de liaison entre A_1 et A_2.

Donc $A_1 \downarrow_n A_2$ est une arborescence fonctionnelle.

Notations

- La liaison $A_1 \downarrow_{Rac(A_1)} A_2$ est notée plus simplement $A_1 \downarrow A_2$. Il en est de même pour les liaisons $A_1 \longrightarrow_{Rac(A_1)} A_2$ et $A_1 \hookrightarrow_{Rac(A_1)} A_2$ notées respectivement $A_1 \longrightarrow A_2$ et $A_1 \hookrightarrow A_2$.

- Dans le cas où l'arborescence fonctionnelle A_1 est composée d'un seul nœud fonctionnel n, l'arborescence fonctionnelle $A_1 \downarrow_n A_2$ (resp.$A_1 \longrightarrow_n A_2$ ou $A_1 \hookrightarrow_n A_2$) est notée $n \downarrow A_2$ (resp. $n \longrightarrow A_2$ ou $n \hookrightarrow A_2$).

1.2 Les arbres de données

Un arbre de données est un arbre de degré quelconque tel que :

- les noeuds sont de deux types : les noeuds fonctionnels et les noeuds de données.

- les noeuds fonctionnels appartiennent à une arborescence fonctionnelle : ils sont origine d'arcs typés vers d'autres nœuds fonctionnels (cf. définition 1.1).

1.3 Les arbres de réduction

Chaque argument d'une fonction est représenté plus généralement par un **arbre de réduction**. Cet arbre de réduction est :

- soit une arborescence fonctionnelle si l'argument à représenter est une fonction.
- soit un arbre de données constitué d'un noeud de donnée unique si l'argument est un atome.
- soit un arbre de données de racine un noeud de degré k si l'argument représenté est une liste de k éléments.

La fonction $\Re$, définie dans cette section, permet d'affecter à chaque objet un arbre de réduction. Dans la suite de cette section, nous présentons donc la représentation des objets par $\Re$. Mais, auparavant, nous définissons un ensemble d'informations qui caractérisent les nœuds. Ces informations concernent les nœuds tous types confondus et sont indispensables à la représentation des objets.

1.4 Caractéristiques des nœuds

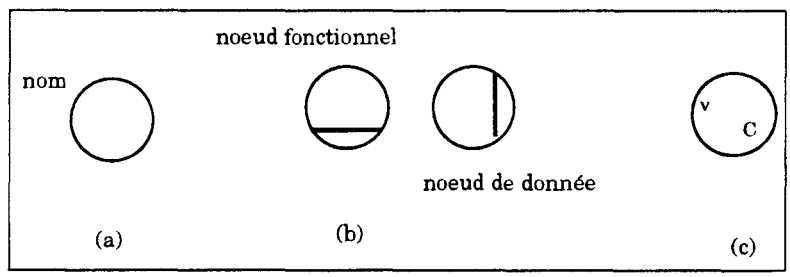


Figure 4.2 : Caractéristiques des nœuds

A chaque nœud quel que soit son type (fonctionnel ou donnée), nous associons les informations suivantes :

- *Un nom* permettant de le désigner de manière unique. Le nom d'un nœud est toujours représenté comme le montre la figure 4.2.a
- *Un type :* on distingue 2 types de nœuds : les nœuds fonctionnels, et les nœuds de donnée (figure 4.2.b)

- *Une classe c* : à chaque type de nœud, correspondent plusieurs classes possibles.
  Un nœud fonctionnel peut appartenir à l'une des classes suivantes : la classe des nœuds fonction primitive (**P**), la classe des nœuds forme fonctionnelle (**FF**) ou la classe des nœuds définition (**D**).
  Un nœud de donnée appartient soit à la classe des nœuds atomes (**A**) ou à celle des nœuds liste (**L**).

- *Une valeur* : chaque nœud admet une valeur *v*. C'est la classe d'un nœud qui détermine l'ensemble des valeurs possibles pour un nœud. Par exemple pour les nœuds fonction primitive appartenant à la classe des nœuds fonctions primitives, les valeurs possibles sont les symboles de fonctions primitives du LFSV utilisé.

  Les informations valeur *v* et classe *c* d'un nœud sont représentées comme le montre la figure 4.2.c

- L'*état de création* : Dans le but d'anticiper l'évaluation des expressions, nous sommes amenés à créer des nœuds n'ayant pas de valeur. Cette information permet donc de renseigner sur la présence ou l'absence de valeur. Des explications plus détaillées sont données en section 2.6.

Détaillons maintenant la représentation des objets d'un LFSV dans le modèle P^3 en utilisant les arborescences fonctionnelles et les arbres de réduction.
Nous commençons par la représentation des fonctions.

1.5 Représentation des fonctions

Un programme LFSV est un ensemble de définitions dont une est appelée **définition principale.**
Une définition dans un LFSV est une composition de fonctions $f_1 \circ f_2 \circ \ldots \circ f_n$ telle que pour tout i, f_i n'est pas une composition de fonctions i.e

$$f_i \neq f_{i_1} \circ f_{i_2} \circ \ldots \circ f_{i_m}$$

Chaque fonction f_i est soit :

— une fonction primitive simple,
— une fonction primitive d'ordre supérieur appelée plus couramment un combinateur,
— une occurrence d'utilisation d'une définition,

— une forme fonctionnelle prédéfinie ou
— une forme fonctionnelle définie.

La fonction $\Re$ associe à chaque programme une forêt d'arborescences fonctionnelles dont une est appelée **arborescence fonctionnelle principale**. Chaque occurrence d'utilisation d'une définition dans le corps de la définition principale fait appel à une définition appelée **définition secondaire**, elle-même représentée par une forêt d'arborescences fonctionnelles. La forêt d'arborescences fonctionnelles représentant un programme **P** est donc constituée de :

- l'arborescence fonctionnelle **principale** notée aussi $\Re_p$(P).
- L'union [3] des ensembles d'arborescences fonctionnelles représentant les définitions secondaires utilisées dans le programme. Ces arborescences fonctionnelles sont appelées **les arborescences fonctionnelles secondaires** de la représentation du programme.

Chaque fonction f_i d'une composition de fonctions est représentée par une arborescence fonctionnelle. L'arborescence fonctionnelle d'une composition est établie à partir des représentations des fonctions qui la constituent. Dans ce paragraphe, nous décrivons successivement les représentations des différents types de fonctions cités ci-dessus. Ensuite, nous décrivons la représentation d'une composition de fonction et celle d'un programme.

1. **Les fonctions primitives**

   L'arborescence fonctionnelle représentant une fonction primitive f_p par $\Re$ se réduit à un seul nœud. Ce nœud est de *type* **fonctionnel**. Il appartient à la *classe* **P** des nœuds fonctions primitive et a pour valeur le symbole de la fonction primitive. La figure 4.3 illustre la représentation d'une fonction primitive par la fonction $\Re$.

   Remarque
   Il est à noter que les combinateurs sont des fonctions primitives d'ordre supérieur. Ils admettent donc une représentation identique à celles des fonctions primitives i.e ils sont représentés par un nœud de type fonctionnel appartenant à la classe P et de valeur le symbole du combinateur.

[3] On parle d'*union* parce que les arborescences fonctionnelles faisant partie de la représentation de plusieurs définitions n'existent qu'en un seul exemplaire dans la représentation du programme

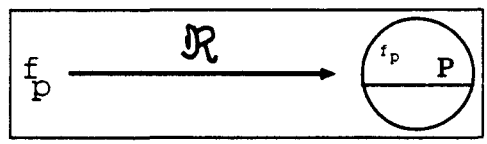


Figure 4.3 : Représentation d'une fonction primitive

2. Les définitions

Une occurrence d'utilisation d'une définition est représentée par une arborescence fonctionnelle composée d'un seul nœud fonctionnel. Ce nœud a pour valeur le nom du nœud racine de l'arborescence fonctionnelle principale représentant la définition utilisée. Un nœud définition appartient à la *classe* **D** des **nœuds définitions**. Si nous représentons schématiquement une arborescence fonctionnelle de racine le nœud fonctionnel n par le schéma de la figure 4.4.a, la représentation d'une occurrence d'utilisation de cette définition notée occ(d) est illustrée par la figure 4.4.b :

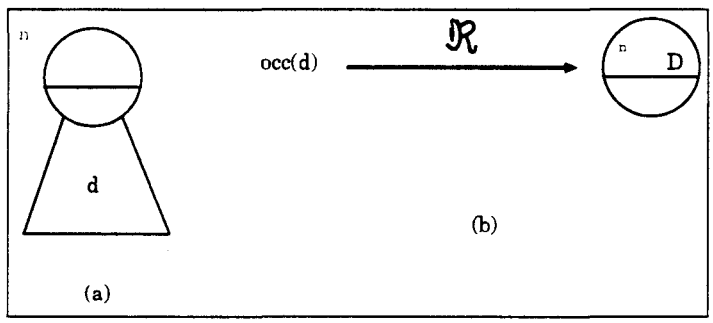


Figure 4.4 : Représentation d'une définition

3. Les formes fonctionnelles prédéfinies

On considère une forme fonctionnelle *ff* ayant k paramètres $p_1, p_2, \ldots, p_k$. Chaque p_i est une composition de fonctions $p_{i_1} \circ \ldots \circ p_{i_{m_i}}$. p_i admet donc comme représentation par $\Re$ une forêt d'arborescences fonctionnelles dont l'arborescence fonctionnelle principale est $\Re_p(p_i)$.
L'arborescence fonctionnelle représentant la forme fonctionnelle (ff p_1 p_2 ... p_k) est construite à partir des arborescences fonctionnelles principales représentant les paramètres p_i :

$$n \hookrightarrow \Re_p(p_1) \longrightarrow \ldots \longrightarrow \Re_p(p_k)$$

schématisée par la figure 4.5, où n est un nœud fonctionnel ayant pour valeur le symbole de la forme fonctionnelle *ff*. Le nœud fonctionnel n appartient à la classe des **nœuds forme fonctionnelle** notée **FF**.

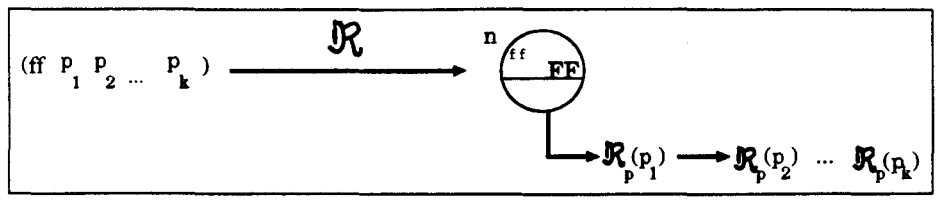


Figure 4.5 : Représentation d'une forme fonctionnelle

4. **Représentation d'un programme**

Un programme P se compose d'un ensemble de définitions dont une est la définition principale. Chacune de ces définitions est une composition de fonctions $f_1 \circ f_2 \circ \ldots \circ f_n$ où chaque fonction f_i est représentée par une arborescence fonctionnelle $\Re_p(f_i)$ telle que CSP($\Re_p(f_i)$) est réduit à un seul nœud qui est **Rac**(f_i) puisque par hypothèse chaque fonction f_i n'est pas une composition de fonctions. La représentation d'une composition de fonctions est obtenue à partir de celles des fonctions f_i en utilisant l'opérateur de liaison $\downarrow$. Elle est égale à :

$$\Re_p(f_n) \downarrow (\Re_p(f_{n-1}) \downarrow (\ldots \downarrow \Re_p(f_1)\ldots))$$

L'arborescence fonctionnelle ainsi construite pour la définition principale est appelée **arborescence fonctionnelle principale ou représentation principale** du programme P et est notée $\Re_p(P)$. Les arborescences fonctionnelles construites de la même manière pour chacune des définitions secondaires sont appelées **arborescences fonctionnelles secondaires** du programme P. L'ensemble de ces arborescences constituent la représentation du programme P. $\Re_p$(P) présente les caractéristiques suivantes :

(a) $\Re_p$(P) admet pour racine le nœud racine de l'arborescence fonctionnelle représentant f_n .

$$\text{Rac}(\Re_p(\text{P}))=\text{Rac}(\Re(f_n))$$

(b) $\Re_p$(P) admet donc pour chemin séquentiel principal le chemin constitué des racines des arborescences fonctionnelles $\Re_p(f_n), \Re_p(f_{n-1}), \ldots, \Re_p(f_1)$.

$$\text{CSP}(\Re_p(\text{P}))=\text{CS}(\text{Rac}(\Re_p(f_n)) \mapsto \text{Rac}(\Re_p(f_1))$$

Avantage : l'avantage de cette représentation est que les fonctions sont placées dans l'arborescence fonctionnelle dans l'ordre dans lequel elles s'appliquent aux arguments.

1.6 Représentation des atomes

Un atome est représenté par un arbre de données composé d'un nœud unique. Ce nœud est un **nœud de donnée** appartenant à la classe **A** des **nœuds atomes** et a pour valeur la valeur de l'atome. La représentation de l'atome *toto* est illustrée par la figure 4.6.

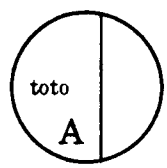


Figure 4.6 : Représentation de l'atome toto

1.7 Représentation d'une liste d'objets

Une liste l composée des objets o_1,o_2,...,o_n où chaque objet o_i est soit un atome, une liste ou une fonction, est représentée par un arbre de réduction dont la racine s appartient à la *classe* **L** des nœuds liste.
L'arbre de réduction représentant la liste l peut être décrit comme suit :

- Son nœud racine est un nœud de donnée s de classe L.
- Il admet n sous arbres de réduction, chacun représentant un élément de la liste. Un sous arbre de réduction est :

  - Un arbre de données constitué d'un seul nœud si l'élément o_i est un atome.
  - Un arbre de réduction représentant une liste si l'élément o_i est une liste.
  - Une arborescence fonctionnelle si o_i est une fonction. Cette arborescence fonctionnelle est la représentation principale de la fonction o_i : $\Re_p(o_i)$.

Remarque
Les représentations par $\Re$ et par $\Re_p$ d'un atome ou d'une liste sont identiques i.e : si o_i est un atome ou une liste, $\Re(o_i) = \Re_p(o_i)$.

La représentation générale d'une liste est illustrée par la figure 4.7.

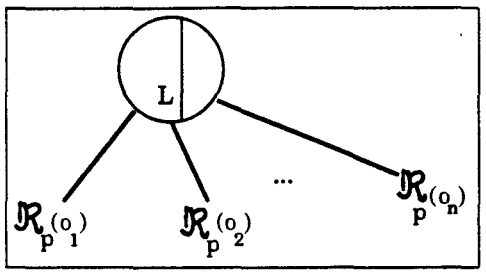


Figure 4.7 : Représentation d'une liste d'objets

Exemple
La liste < toto < 1 2> car o car > est représentée par l' arbre de réduction de la figure 4.8.

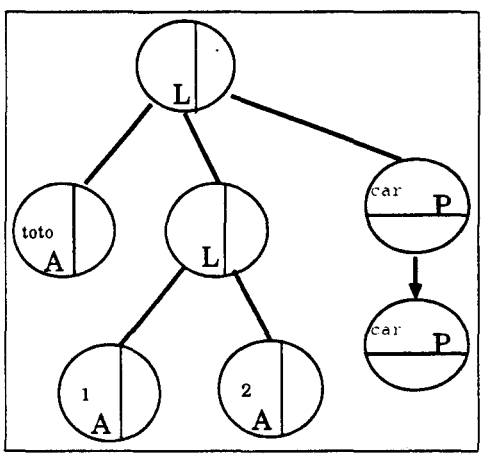


Figure 4.8 : Représentation de la liste < toto < 1 2> car o car >

La section suivante décrit la représentation des applications par la fonction de représentation $\Re$.

2 L'application

2.1 Définition d'une application

L'application est l'unique opération des LFSV. Cette opération admet deux opérandes. Le premier est nécessairement une fonction, c'est la **fonction à appliquer**. Le second est une suite d'objets du langage. Chaque objet de cette suite est un **argument** de la fonction à appliquer. L'ordre des arguments d'une fonction

est important; c'est la raison pour laquelle on parle de suite d'arguments et non pas d'ensemble d'arguments. La fonction à appliquer ainsi que ses arguments sont appelés les **composants de l'application.** La notation f : o_1 o_2 ... o_n symbolise aussi bien l'application d'une fonction f à une suite d'arguments o_1,o_2,...,o_n que l'objet résultat de cette même application. f : o_1 o_2 ... o_n est **la forme non réduite** de l'objet résultat de cette même application. L'évaluation d'une application permet d'aboutir à la forme réduite de l'objet résultat de l'application.

2.2 Types d'applications

Rappel : Nous distinguons trois types d'applications :

- les applications de type 1

  $$\text{f} : o_1\ o_2\ \ldots\ o_n$$

  où tous les composants de l'application sont des objets dans leur forme réduite.

- les applications de type 2

  $$(\text{f} : o_1\ o_2\ \ldots\ o_n) : a_1\ a_2\ \ldots\ a_k$$

  La fonction à appliquer est un objet non réduit. Le **résultat** de l'application de la fonction f aux arguments o_1,o_2,...,o_n est une fonction P qui s'applique à a_1 a_2 ... a_k, une suite d'arguments dans leur forme réduite. La fonction f est par hypothèse dans sa forme réduite.

- les applications de type 3

  $$\text{f} : a_1\ \ldots\ a_{i-1}(g\ :\ o_{i_1}\ o_{i_2}\ \ldots\ o_{i_k})a_{i+1}\ \ldots\ a_n$$

  où il existe au moins un indice i tel que le i^{eme} argument de la fonction f est un objet calculable c.à.d non réduit.

Toute application est obtenue par combinaison de ces 3 cas de figure.

2.3 Notion de nœud cible

Soit la fonction à appliquer f, représentée par un nœud fonctionnel m, et $o_1, o_2, \ldots, o_n$ ses arguments. Chaque argument o_i admet une représentation par $\Re$ et sera désigné par le nœud racine de sa **représentation principale**. Ce nœud est appelé **nœud cible** de la fonction f ou du nœud fonctionnel m. Une fonction admettant n arguments a n nœud cibles. L'ordre des nœud cibles est important tout comme celui des arguments.

Remarque : Deux nœuds fonctionnels distincts ont leurs nœuds cibles disjoints exception faite d'un nœud définition et du nœud racine de la définition qui ont les mêmes nœuds cibles (cf. section 8).

2.4 Notions de nœuds résultats

- Soit l'application d'une fonction f, représentée par un nœud fonctionnel m, aux arguments $o_1, o_2, \ldots, o_n$; la forme réduite du résultat de cette application est un objet du langage. Il admet par $\Re$ une représentation dans le modèle P^3. On appelle **nœud résultat** du nœud m, le nœud racine de cette représentation.

- Soit $CS(c_1 \mapsto c_k)$, un chemin séquentiel. On appelle *nœud résultat du chemin séquentiel* $CS(c_1 \mapsto c_k)$, le noeud résultat du nœud fonctionnel c_k.

2.5 Drapeaux d'activation

Dans le modèle P^3, l'objectif est d'associer à chaque fonction du programme l'ensemble de ses arguments. Cette association que l'on appelera *un redex* est effectuée grâce au mécanisme d'exploration des arborescences fonctionnelles qui sera décrit à la section 4.1. Elle est symbolisée par la création de **drapeaux d'activation**.
Nous distinguons deux activités asynchrones dans le modèle P^3 : l'exploration des arborescences fonctionnelles, créatrice de redex, et la réduction qui prend en compte ces redex en appliquant la fonction à ses arguments. Dans ce but, un même redex sera représenté différemment au moment de sa création et au moment de sa prise en compte par le mécanisme de réduction respectivement par un **Drapeau d'Activation de l'Exploration ou DAE** et par un **Drapeau d'Activation de la Réduction ou DAR**.

- **Drapeaux d'Activation de l'Exploration (DAE)** : Le mécanisme d'exploration permet d'attribuer un DAE à chaque noeud fonctionnel en commençant par la racine. L'affectation d'un tel DAE à un noeud fonctionnel n de valeur une fonction f signifie la création d'un redex impliquant la fonction f et ses arguments. Pour désigner les arguments de la fonction f, le DAE contient l'ensemble des nœuds cibles de la fonction. Un nœud fonctionnel auquel on associe un DAE est dit **en cours d'exploration.**
  L'application d'une fonction F quelconque à l'ensemble de ses arguments, se déclenche par la création d'un premier DAE affecté au noeud racine de la représentation principale de la fonction F. La présence d'un DAE sur un noeud fonctionnel signifie de plus que la sous arborescence dont il est la racine sera explorée. L'ordre dans lequel les DAE sont attribués aux nœuds fonctionnels est décrit par les règles d'exploration du système de réécriture (cf. section 4).

  Un DAE affecté à un noeud fonctionnel n contient les informations suivantes :

  - les noeuds cibles de la fonction contenue dans le noeud fonctionnel n.
  - La référence du résultat produit par l'application des fonctions contenues dans la sous arborescence fonctionnelle dont n est le nœud racine. Cette référence correspond au noeud résultat du chemin séquentiel principal de la sous arborescence.

- **Drapeaux d'Activation de la Réduction (DAR)** : un DAR est la deuxième forme de représentation d'un redex. Il est associé à une suite d'arbres de réduction $A_1,A_2,\ldots,A_n$. Un DAR contient un symbole de fonction f ; sa présence sur les racines des arbres de réduction $A_1,A_2,\ldots,A_n$ déclenche la réduction par la fonction f des objets $o_1,o_2,\ldots,o_n$ dont $A_1,A_2,\ldots,A_n$ sont les représentations respectives par $\Re$ et ce afin d'obtenir l'objet résultat f : o_1 o_2 ... o_n. Le DAR contient également le nom du nœud racine de la forme réduite de l'expression f : o_1 o_2 ... o_n, appelé **nœud résultat intermédiaire.**
  La figure 4.9.a illustre un DAE contenant k nœuds cibles n_1 n_2 ... n_k et un nœud résultat R. La figure 4.9.b illustre un DAR contenant un symbole de fonction f et le nom d'un nœud résultat intermédiaire R.

Notion de drapeau complet Un drapeau **complet** doit contenir toutes les informations spécifiées selon son type. Il est dit incomplet sinon. Concrètement, un

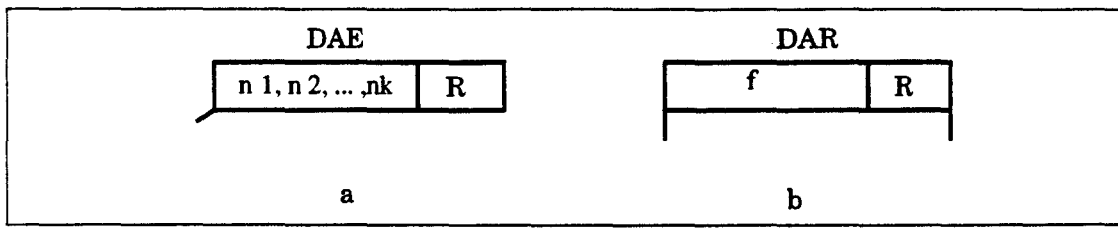


Figure 4.9 : Drapeaux d'activation

drapeau incomplet ne représente pas une application et ne peut être considéré en tant que tel que s'il devient complet.

2.6 Création partielle d'un nœud

Dans P^3, la forme réduite et la forme calculable d'une même expression peuvent parfois "coexister" dans le but d'anticiper l'évaluation. Comme l'objet résultat n'est pas encore obtenu, la forme réduite de l'objet se représente par un nœud unique dont la valeur n'est pas connue. On dit qu'il s'agit d'un nœud résultat en état de **création partielle**. Par opposition, un nœud, quel que soit son type et ayant une valeur, est dit en état de **création effective**. Ce concept de double représentation de l'objet est utilisé également dans l'interprétation des langages QLISP [GRC89] et Multilisp [Hal85] qui sont respectivement des extensions des langages LISP et SCHEME par des structures exprimant explicitement le parallélisme. Un nœud fonctionnel en état de création partielle est représenté par le symbole de la figure 4.10.a. Un nœud de donnée en état de création partielle est représenté par le symbole de la figure 4.10.b. Enfin, un nœud de type quelconque en état de création partielle est représenté par le symbole de la figure 4.10.c. Un noeud en état de création partielle devient en état de création effective dès qu'il acquiert une valeur.

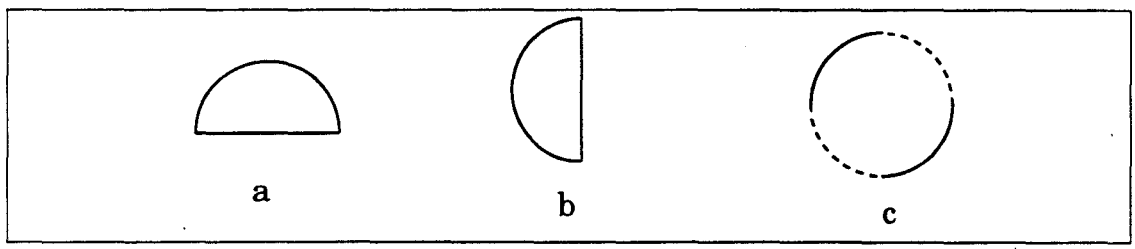


Figure 4.10 : Noeuds en état de création partielle

2.7 Représentation initiale d'une application

Dans ce paragraphe, nous nous intéressons à la représentation initiale d'une application. Celle-ci est obtenue à partir de celles de ses composants.

Détaillons la représentation des composants :

1. **La fonction à appliquer**

   La fonction à appliquer est soit un objet dans sa forme réduite, dans ce cas, il s'agit d'une composition de fonctions; soit un objet calculable : il s'agit alors d'une application.

   - **cas1** : f est une composition de fonctions
     f est donc représentée par une forêt d'arborescences fonctionnelles.
   - **cas2** : f est une application de la forme g : $o_1\ o_2\ \ldots\ o_n$
     f est donc le résultat d'une application. A ce stade de la représentation, l'application g : $o_1\ o_2\ \ldots\ o_n$ n'a pas encore été évaluée, et donc la forme réduite de f n'est pas connue. Dans ce cas, f est représentée par un nœud fonctionnel unique . Ce nœud est un nœud résultat dans l'état de *création partielle*. Il s'agit de la racine de la future arborescence fonctionnelle principale devant représenter f.

2. **les arguments**
   Un argument o_i est soit une composition de fonctions, un atome, une liste ou une application.

   - **cas1** : o_i est une composition de fonctions
     Il admet donc pour représentation une forêt d'arborescences fonctionnelles dont l'une est l'arborescence fonctionnelle principale.
   - **cas2** : o_i est un atome ou une liste.
     Il est donc représenté par un arbre de réduction.
   - **cas3** : o_i est une application de la forme g : $a_1\ a_2\ \ldots\ a_k$. Sa forme finale n'est pas connue au moment de la représentation. L'argument o_i est dans ce cas représenté par un nœud unique en état de *création partielle*. Ce nœud est également le nœud résultat de l'application g : $a_1\ a_2\ \ldots\ a_k$. Le type de l'objet résultat est également inconnu, par conséquent l'unique nœud représentant o_i "n'a pas de type". Il sera typé au moment de l'obtention de la forme réduite de l'argument.

En résumé, une application f : $a_1\ a_2\ \ldots\ a_n$ est représentée par :

1. La représentation de la fonction à appliquer *f*. Cette représentation est selon le cas une forêt d'arborescences fonctionnelles ou un nœud fonctionnel dans l'état de création partielle.

2. La représentation de chaque argument c.à.d un arbre de réduction ou un nœud dans l'état de *création partielle.*

3. Un DAE contenant l'ensemble des nœud cibles de la fonction à appliquer et qui est associé au nœud racine de l'arborescence fonctionnelle principale représentant *f*.

Remarque
Pour une application de type 2, i.e de la forme (f :o_1 o_2 ... o_n) : a_1 a_2 ... a_k, l'application qui constitue la fonction à appliquer est représentée similairement. Il en est de même pour toute application représentant un argument qui est une application de type 3.

Si nous utilisons le symbole de la figure 4.11 pour représenter un arbre de réduction. Nous pouvons représenter initialement, les trois types d'application par les figures 4.12.a, 4.12.b et 4.12.c :

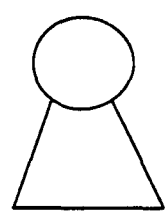


Figure 4.11 : Symbole représentant un arbre de réduction

2.8 Exemple d'application

Soit à représenter l'application A suivante :

$$
\begin{array}{lll}
\text{A} \equiv (d_0 :_2 & f_1 \circ f_2 & d_1) :_1 \\
 & & d_1 \\
 & & (d_1 :_3 \quad < 1 \;\; 2 > \quad \text{toto}) \\
 & & < d_1 \quad 5 >
\end{array}
$$

où :

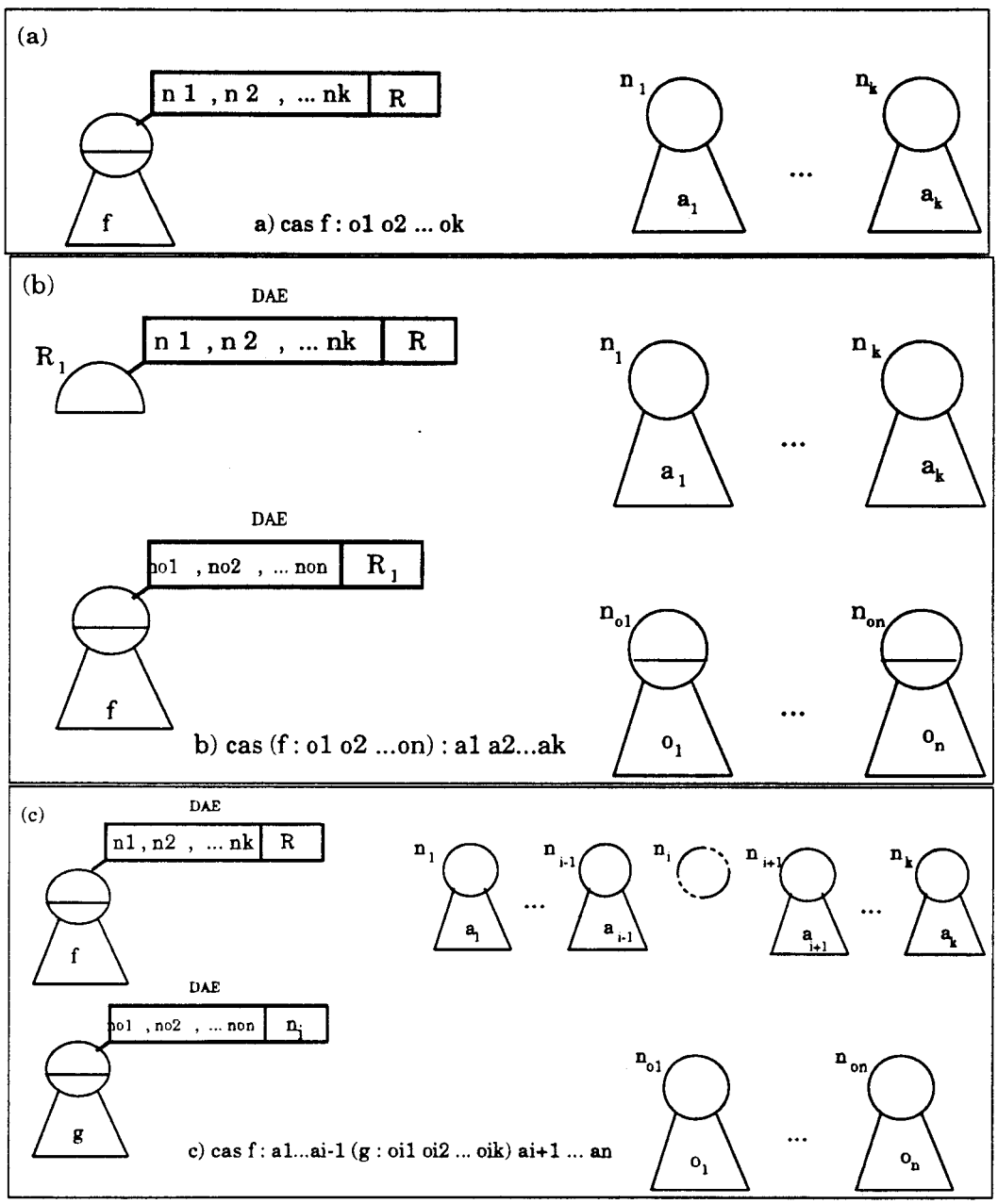


Figure 4.12 : Représentation de 3 types d'applications

1. tous les f_i sont des fonctions primitives,

2. tous les d_i sont des définitions définies par :
   $d_0 \equiv f_1 \circ f_5$
   $d_1 \equiv \alpha d_2$
   $d_2 \equiv f_6 \circ f_7$

3. Les symboles d'application sont numérotés dans l'unique but de pouvoir les différencier.

4. Représentation de l'application 1
   La fonction à appliquer F est calculable ; c'est le résultat de l'application 2. Nous reviendrons plus loin sur la représentation de l'application 2. A ce stade, la fonction F est représentée par le nœud np_1 en état de création partielle. La fonction F admet 3 arguments o_1, o_2 et o_3 :

   $o_1 = d_1$
   $o_2 = d_1 :_3 < 1\ \ 2 > \ toto$ qui est un objet calculable
   $o_3 = < d_1\ \ 5 >$

   L'objet o_1 est représenté par les deux arborescences fonctionnelles de racines respectives les nœuds nd_1 et nd_2 où la première est l'arborescence fonctionnelle principale.
   L'objet o_2 est une application. Il est donc représenté par le nœud no_2 en état de création partielle.
   L'objet o_3 est représenté par l' arbre de réduction de racine no_3. La représentation de l'application 1 est illustrée par la figure 4.13.

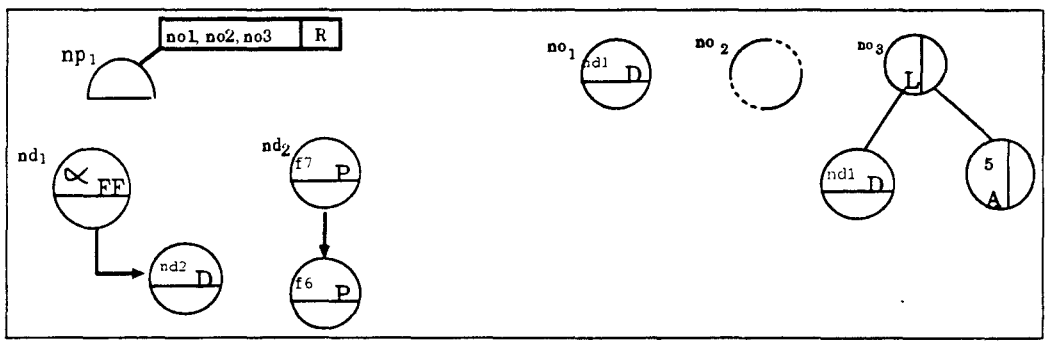


Figure 4.13 : Représentation initiale de l'application 1

5. Représentation de l'application 2

L'arborescence fonctionnelle de racine le nœud fonctionnel np_2 est l'arborescence fonctionnelle principale représentant la fonction à appliquer. Les arborescences fonctionnelles de racines respectives les nœuds nd_0, nd_1 et nd_2 sont les arborescences fonctionnelles secondaires représentant cette même fonction. Cette fonction admet deux arguments o_4 et o_5.

$$o_4 = f_1 \circ f_2$$
$$o_5 = d_1$$

L'objet o_4 est représenté par l'arborescence fonctionnelle de racine le nœud no_4.

L'objet o_5 est représenté par l'arborescence fonctionnelle de racine le nœud no_5, qui est l'arborescence fonctionnelle principale de cette représentation, et par les arborescences fonctionnelles secondaires de racines respectives nd_1 et nd_2. La représentation de l'application 2 est illustrée par la figure 4.14.

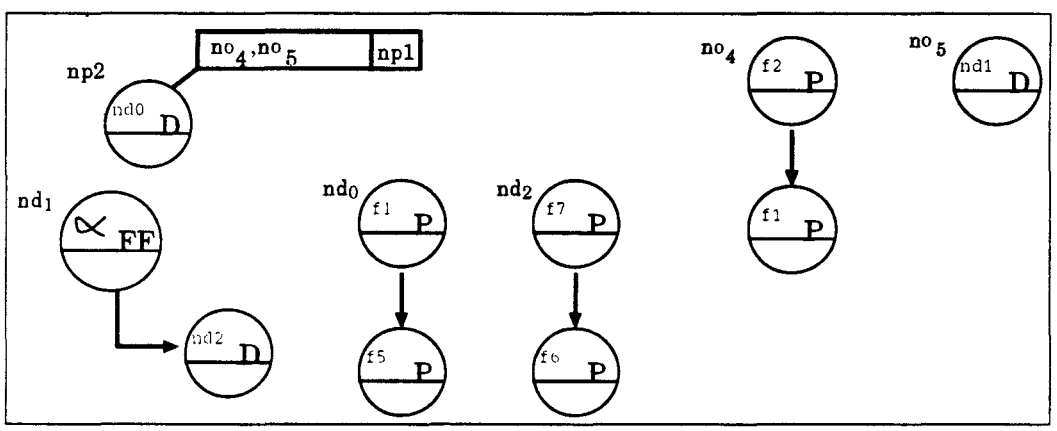


Figure 4.14 : Représentation initiale de l'application 2

6. Représentation de l'application 3

La figure 4.15 illustre la représentation de l'application 3. La fonction à appliquer dans cette application est représentée par les arborescences fonctionnelles de racines np_3, nd_1 et nd_2 où l'arborescence fonctionnelle de racine np_3 est l'arborescence fonctionnelle principale. Les arguments :

$$o_6 = < 1 \ \ 2 >$$
$$o_7 = toto$$

sont représentés respectivement par les arbres de données de racine no_6 et no_7.

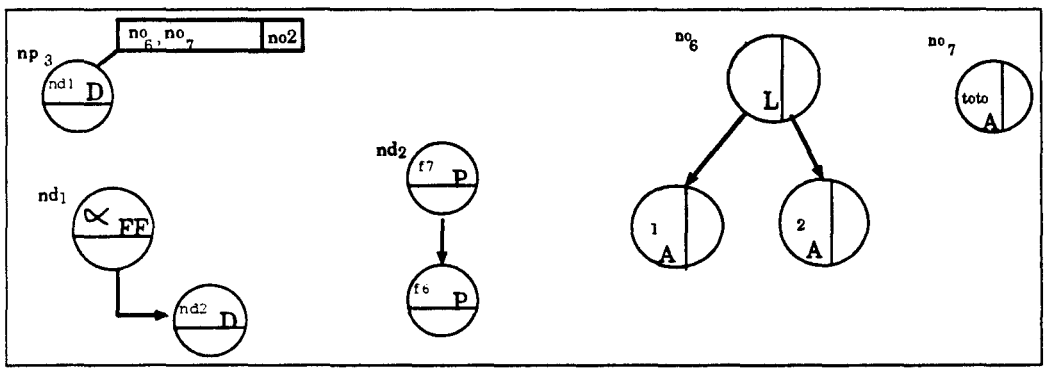


Figure 4.15 : Représentation initiale de l'application 3

La figure 4.16 résume la représentation simultanée des 3 applications dans le modèle P^3.

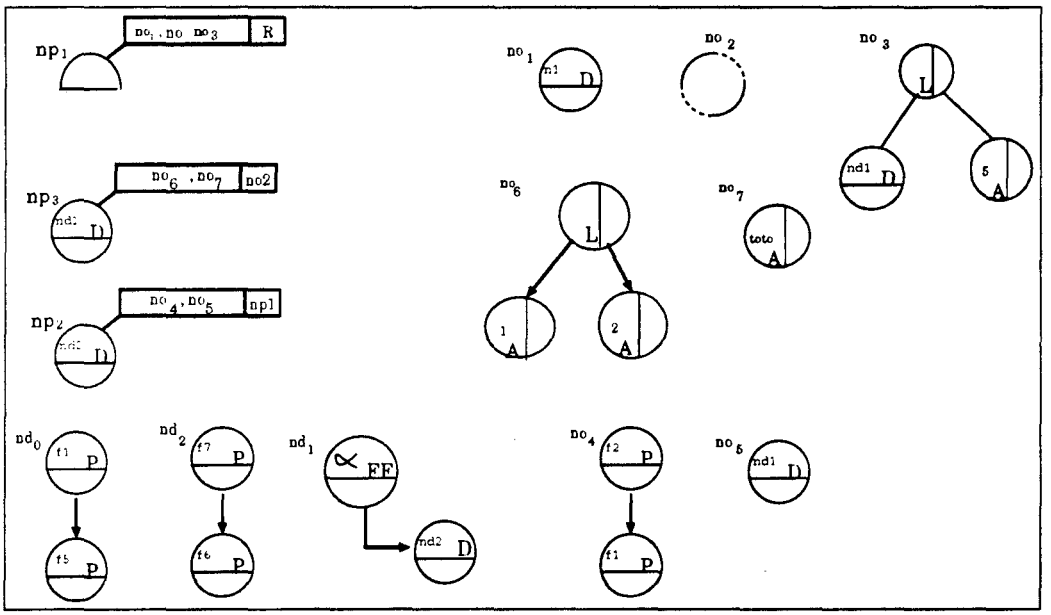


Figure 4.16 : Représentation simultanée des trois applications

Remarques

1. Les nœuds résultat no_2 et np_1 établissent le lien entre les représentations de ces trois applications.

2. Les fonctions admettent une représentation unique quel que soit le rôle qu'elles jouent dans une application.

3. Si une même fonction est utilisée plusieurs fois dans une ou plusieurs applications simultanément, sa représentation figure une seule fois dans l'espace de représentation. Le modèle P^3 permet donc un réél partage du code.

Les formes fonctionnelles définies du langage GRAAL [Bel86] permettent, quand elles sont appliquées à des arguments, l'obtention d'expressions dont la représentation et l'évaluation ne diffère pas de celles des expressions représentées dans cette section. La seule vraie différence est que l'obtention de ces expressions est basée sur une sémantique propre à chaque classe de formes définies. La section suivante décrit la représentation donc des formes fonctionnelles définies.

3 Les formes fonctionnelles définies

Dans les LFSV d'ordre supérieur, le programmeur a la possibilité de se définir ses propres formes fonctionnelles. L'écriture des programmes se trouve nettement simplifiée par l'utilisation des nouvelles formes fonctionnelles. La définition d'une nouvelle forme fonctionnelle consiste à écrire un programme qui décrit son comportement en utilisant les fonctions prédéfinies ainsi que les fonctions préalablement définies.
Dans GRAAL, il existe deux types de formes fonctionnelles définies : les formes *méta* et les formes *user*. Ces deux types de formes fonctionnelles diffèrent par leurs sémantiques.

3.1 Les formes méta

Soit m une forme méta dont le comportement est défini par la composition de fonctions :

$$\mathrm{m}{=}m_1 \circ \ldots \circ m_l.$$

La sémantique d'application d'une forme méta m ayant n paramètres $p_1\ p_2\ \ldots\ p_n$ à ses arguments $a_1,a_2,\ldots,a_k$ est la suivante :

$$(\mathrm{m}\ p_1\ p_2\ \ldots\ p_n){:}a_1\ a_2\ \ldots\ a_k \equiv \mathrm{m}{:}{<}p_1\ p_2\ \ldots\ p_n{>}\ a_1\ a_2\ \ldots\ a_k$$

Autrement dit, l'application d'une forme méta m à une suite d'arguments revient à appliquer le corps de la fonction décrivant le comportement de la forme méta m à la suite d'arguments composée de la liste $<p_1\ p_2\ \ldots\ p_n>$ suivie des autres arguments. C'est la règle classique de la *métaévaluation*. Une forme méta n'est donc rien d'autre qu'un appel de la fonction m dont le premier argument est constant et donc connu à l'avance. Par conséquent, sa représentation est un nœud définition dont la valeur est le nom du nœud racine de l'arborescence fonctionnelle principale représentant la fonction m. Ce nœud est muni d'un DAE incomplet contenant le premier nœud cible du nœud définition. La liste des paramètres $< p_1\ p_2\ \ldots\ p_n\ >$, est représentée par un arbre de réduction. Le DAE associé au nœud définition ne déclenche aucune exploration puisqu'il est incomplet. Il deviendra complet quand le nœud définition recevra un DAE de son prédécesseur. Une application F : $a_1\ a_2\ \ldots\ a_n$ où F=f_1o...of_n tel qu'il existe f_i=(m $p_1\ p_2\ \ldots\ p_l$) une forme méta ayant l paramètres est représentée comme le montre la figure 4.17.

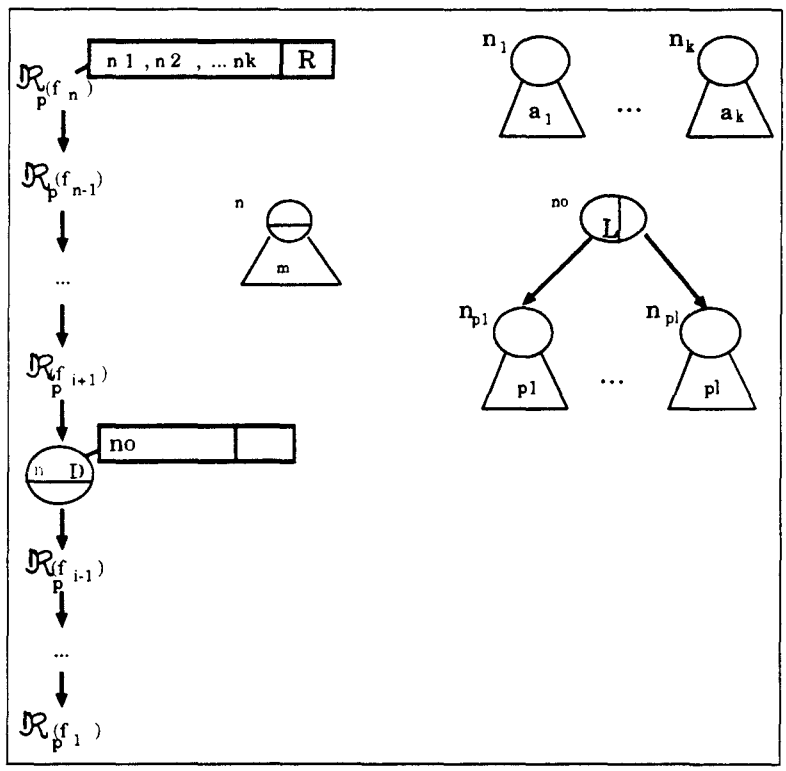


Figure 4.17 : Représentation d'une forme méta

3.2 Les formes user

Soit u une forme user dont le comportement a été défini par la fonction de même nom

$$\mathrm{u}{=}u_1 \mathrm{o} \ldots \mathrm{o} u_l$$

La sémantique d'application d'une forme user u ayant n paramètres $p_1, p_2, \ldots, p_n$ est la suivante :

$$(\mathrm{u}\ p_1\ p_2\ \ldots\ p_n) : a_1\ a_2\ \ldots\ a_k \equiv (\mathrm{u} : p_1\ p_2\ \ldots\ p_n) : a_1\ a_2\ \ldots\ a_k$$

Autrement dit, l'application d'une forme *user* (u p_1 p_2 ... p_n) à ses arguments, conduit d'abord au calcul d'une fonction U, par application de la fonction user associée, aux paramètres $p_1, p_2, \ldots, p_n$.

La fonction U est obtenue en évaluant l'expression Exp_U suivante :

$$Exp_U = u_1 \circ \ldots \circ u_l \ : \ p_1 \ \ p_2 \ldots p_n$$

Plus généralement la fonction U peut faire partie d'une composition de fonctions F telle que :

$$F \ = \ f_1 \circ f_2 \circ \ldots \circ f_{i-1} \circ Exp_U \circ f_{i+1} \circ \ldots \circ f_c$$

Il s'agit donc, plus généralement, de proposer une représentation pour une composition de fonctions dans laquelle au moins une fonction est une application. L'expression Exp_U admet une représentation par la fonction $\Re$. La fonction U n'est pas connue au moment de la représentation de la fonction F. Par conséquent, on ne peut déterminer $\Re_p$(U). Afin d'anticiper la représentation de la fonction F sans se préoccuper du résultat de l'évaluation de l'expression Exp_U, nous représenterons la fonction U dans $\Re_p$(F) par un nœud définition de valeur le noeud résultat de l'expression Exp_U i.e le noeud racine de l'arborescence fonctionnelle $\Re_p$(U). Ce noeud est en état de création partielle tant que l'expression Exp_U n'est pas évaluée.
Par ailleurs, cette représentation a l'avantage non seulement de permettre la représentation de la fonction F quelles que soient les fonctions qui la composent mais aussi l'application de la fonction F à ses arguments. La représentation de l'application :

$$F : o_1 \ \ o_2 \ldots o_s$$

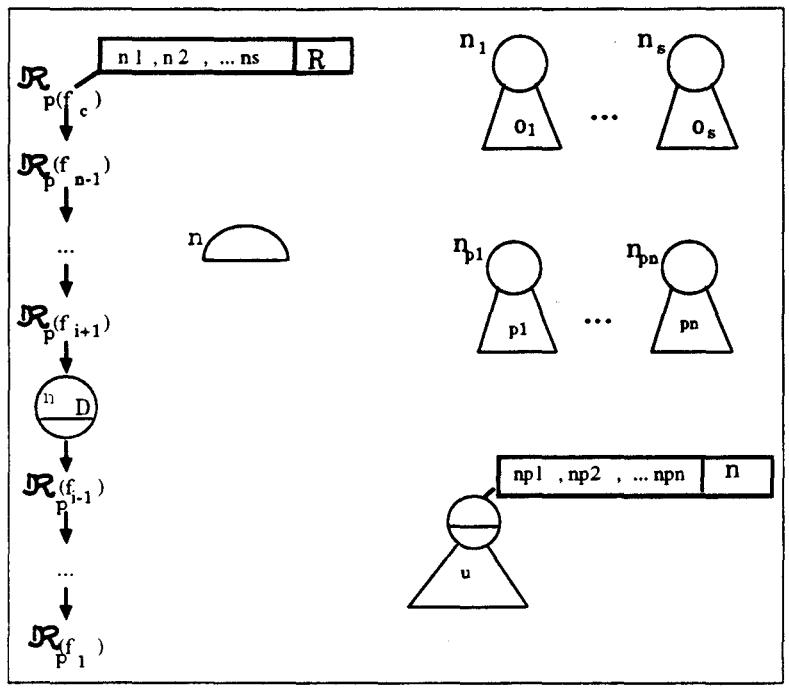


Figure 4.18 : Représentation de la forme user

est illustrée par la figure 4.18.

Cette représentation permet une anticipation au niveau de l'évaluation de l'application u : $p_1 \ldots p_n$. Si le résultat de l'application est obtenu avant l'activation du nœud définition de valeur le nœud n, il n'y aura pas d'attente au niveau de l'exploration de l'arborescence de racine n.

Sur la figure 4.18, seules les représentations principales des objets apparaissent étant donné que les fonctions intervenant dans l'application sont définies dans le cas général. Les arborescences fonctionnelles secondaires, s'il y en a, font bien sûr partie intégrante de la représentation.

Dans cette section, nous avons décrit la représentation d'une application f : $a_1\ a_2 \ldots\ a_n$ quels que soient ses composants. Les composants de l'application qui sont les résultats d'autres applications sont représentés par des nœuds résultats qui établissent le lien entre les applications. Chaque DAE de cette représentation matérialise une demande d'application d'une fonction à ses arguments. Le nombre initial de DAE correspond au nombre d'applications qui nécessitent d'être évaluées. Ce nombre augmente au fur et à mesure, par l'exploration parallèle des sous arborescences et des arborescences fonctionnelles secondaires.

4 Evaluation d'une application

Soit une application f : a_1 a_2 ... a_n représentée dans le modèle par un ensemble d'arborescences fonctionnelles, un ensemble d'arbres de réduction et un ensemble de DAE associés à certains nœuds fonctionnels.

L'évaluation d'une application consiste en une exploration des arborescences fonctionnelles et la réduction des arguments par les fonctions contenues dans les nœuds fonctionnels explorés.
Nous avons décrit ces deux mécanismes à l'aide d'un système de réécriture comportant deux types de règles : des *règles d'exploration* qui permettent la description de la propagation de l'exploration dans les arborescences fonctionnelles et des *règles de réduction* décrivant l'application des fonctions à leurs arguments.

Dans cette section, nous définissons les conditions d'application des règles d'exploration et des règles de réduction, puis, nous montrons quelques exemples d'application de ces règles. Enfin, nous démontrons la correction du système de réécriture i.e l'obtention d'un résultat unique d'évaluation quelque soit l'ordre d'application adopté.

L'annexe A, qui se trouve à la fin de ce chapitre, commente plus en détail la composition des règles de réécriture proposées, leurs conditions respectives d'application et des exemples de règles de chaque type.

4.1 Les règles d'exploration

- Ces règles permettent d'associer à chaque nœud d'une arborescence fonctionnelle ses nœuds cibles par l'intermédiaire de l'affectation d'un DAE.

- Toute arborescence ou sous arborescence fonctionnelle contenant au moins un nœud fonctionnel **ayant une valeur** et possédant **un DAE complet** est candidate à l'application d'une règle d'exploration.

- La règle à appliquer est choisie en fonction de :

  — la classe du nœud fonctionnel possédant le DAE,
  — la valeur du nœud, si pour une même classe de nœuds la propagation se fait différemment pour des valeurs de nœuds différentes,
  — l'existence ou l'absence d'un successeur.

- L'application d'une règle d'exploration permet le traitement **d'un DAE** et

peut engendrer la création de zéro, un ou plusieurs DAE contenant chacun, les nœuds cibles et le nœud résultat du nœud fonctionnel possédant le DAE à traiter.

- Le traitement d'un DAE par une règle d'exploration permet de créer le DAR représentant le même redex. Le DAE ainsi traité est supprimé pour que d'une part il ne soit pas redondant (le DAR généré représente le même redex) et d'autre part parce qu'un DAE complet affecté à un noeud fonctionnel peut toujours être traité par une règle d'exploration, ce qui entraînerait une incohérence au niveau de l'évaluation.

- Un nœud fonctionnel n appartenant aux classes P ou FF, racine d'une arborescence fonctionnelle A et possédant un DAE (n_1,n_2,...,n_k;R), où n_1,n_2,...,n_k sont les nœud cibles du nœud fonctionnel n, crée à la suite de l'application d'une règle de propagation, un DAR qu'il affecte à ses arguments a_1,a_2,...,a_k désignés par les racines respectives de leurs représentations principales n_1,n_2,...,n_k. Ce DAR contient comme symbole de fonction la valeur v du nœud fonctionnel n et comme nœud résultat, un nœud NI qui est la racine de la représentation de la forme réduite de l'objet résultat de l'application $v : a_1\ a_2\ \ldots\ a_k$.

- Chaque nœud fonctionnel n possédant un DAE (n_1,n_2,...,n_k; R) transmet à son successeur, s'il en a un, un DAE (m_1,m_2,...,m_l;R) où m_1,m_2,...,m_l sont les nœud cibles du successeur. Le nœud résultat est identique pour le nœud n et son successeur puisqu'ils appartiennent au même chemin séquentiel (cf. section 2.4).

La figure 4.19 montre les règles d'exploration R_1 et R_2 correspondant aux nœuds fonction primitive avec (figure 4.19.a) et sans (figure 4.19.b) successeur. L'ensemble des règles d'exploration est présenté en annexe à la fin de ce chapitre.

Cas particulier : Gestion des DAE incomplets

Soient n et Succ(n) deux nœuds fonctionnels consécutifs d'un chemin séquentiel, tel qu'à Succ(n), est associé, à la construction de l'arborescence fonctionnelle, un DAE incomplet ($m_1,\ldots,m_i\ldots$; $-$), tel que $m_1,\ldots,m_i$ sont les noeuds cibles constants du noeud fonctionnel succ(n). Le traitement par une règle d'exploration d'un DAE ($n_1, n_2, \ldots, n_k$; R) associé à un noeud fonctionnel n conduit à la création d'un DAE (($m_1,\ldots,m_i, R_1, R_2, \ldots, R_s$; R) associé au noeud fonctionnel succ(n) où $R_1, R_2, \ldots, R_s$ sont les noeuds cibles déterminés par l'exploration du noeud n et ce quel que soit le type et la valeur du noeud n.

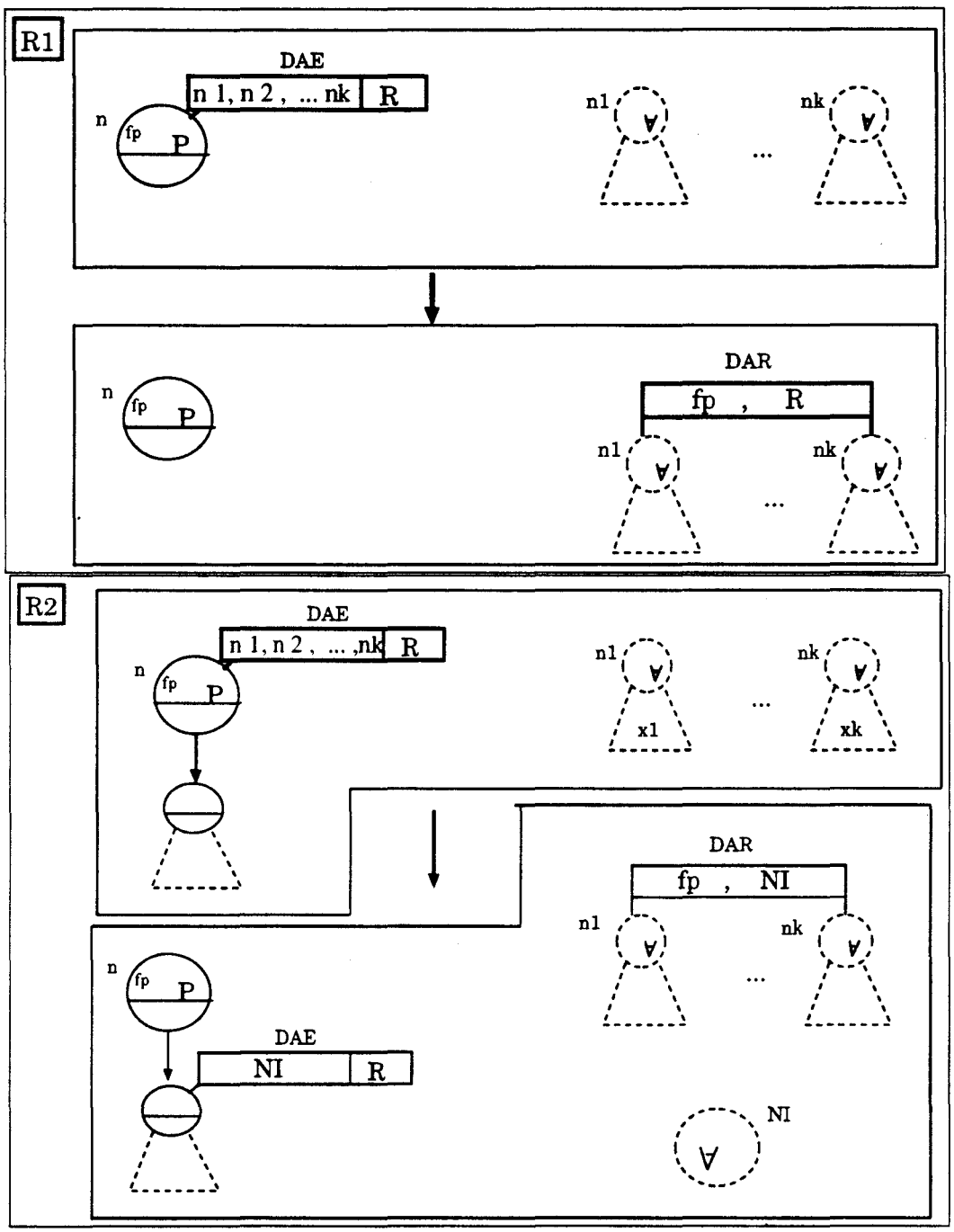


Figure 4.19 : Règles d'exploration pour un nœud fonction primitive

Ce comportement est illustré par la règle suivante (cf. figure 4.20) :

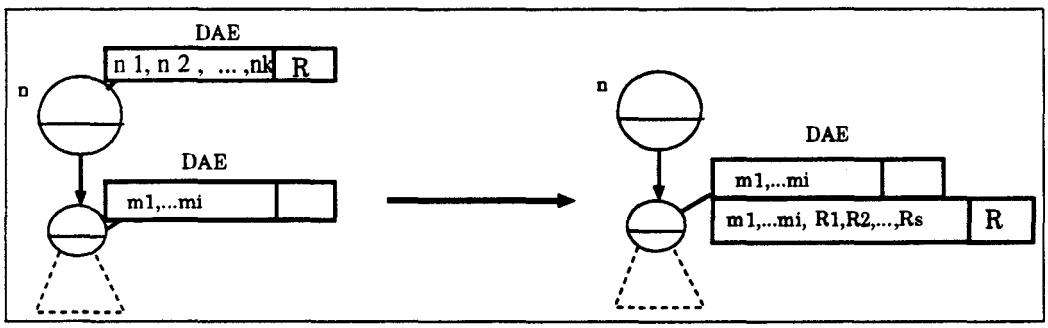


Figure 4.20 : Exploration d'un noeud en possession d'un DAE incomplet

Le DAE incomplet a le rôle de compléter les DAE associés au noeud fonctionnel Succ(n) par le mécanisme d'exploration. Il reste inchangé pour permettre la transmission des mêmes informations en cas d'explorations successives ou simultanées.

4.2 Les règles de réduction

- En s'appliquant à une suite d'arguments munis d'un DAR, une règle de réduction permet l'obtention de la forme réduite de l'objet résultat de l'application de la fonction spécifiée dans le DAR.

- Chaque suite d'arguments possédant un DAR est donc candidate à l'application d'une règle de réduction.

- La règle à choisir dépend :

  — du symbole de fonction contenu dans le DAR.
  — des objets arguments, (si pour deux types d'objets différents le traitement à effectuer n'est pas le même.)
  — de l'état de création des nœuds racines des arbres de réduction représentant les arguments.
  — de l'existence ou non du nœud résultat spécifié dans le DAR. Ce nœud s'il n'existe pas déjà est créé par l'application de la règle de réduction. Dans le cas contraire, il change d'état à la suite de l'application de la règle de réduction puisqu'il a désormais une valeur donc il passe dans l'état de création effective.

- Un DAR représente une application d'une fonction à ses arguments. Le traitement par une règle de réduction fournit le résultat de cette application. Le redex - c.à.d le DAR- est supprimé après application d'une règle de réduction à ce DAR.

La figure 4.21 illustre un exemple de règles de réduction pour le traitement d'un DAR contenant le symbole de fonction primitive *add.* De même que pour les règles d'exploration, des exemples de règles de réduction se trouvent en annexe, à la fin de ce chapitre.

Remarque Le système de réécriture ainsi défini est déterministe i.e pour un drapeau particulier associé à un ensemble de noeuds, une seule règle est applicable. Pour un DAE, cette règle dépend du type, de la valeur, de la présence d'un successeur et de la présence d'un DAE incomplet sur le successeur. Pour un DAR, la règle adaptée dépend du type du noeud et de son état de création et, le cas échéant, de sa valeur.

5 Exemples

Dans le but d'éclaircir encore plus le fonctionnement du modèle P^3, les exemples suivants illustrent l'évaluation de plusieurs expressions dans le modèle P^3. Pour en faciliter la compréhension, nous noterons que :

- Chaque drapeau d'activation sur les schémas, illustre un traitement potentiel et donc le nombre de drapeaux dans une étape de l'évaluation montre les activités que l'on peut effectuer en parallèle à un moment donné.

- Dans un souci de clarté, les arguments quel que soit leur type seront représentés à droite sur les schémas illustrant les étapes du calcul.

5.1 Exemple1

Cet exemple illustre l'évaluation de l'expression suivante :

$$\alpha\ (\text{binul}\quad \text{div}\quad (\text{cste}\quad 5))\ :\ <35\quad 40\quad 10>$$

qui permet de diviser par 5 tous les éléments de la liste $<\ 35\quad 40\quad 10\ >$.

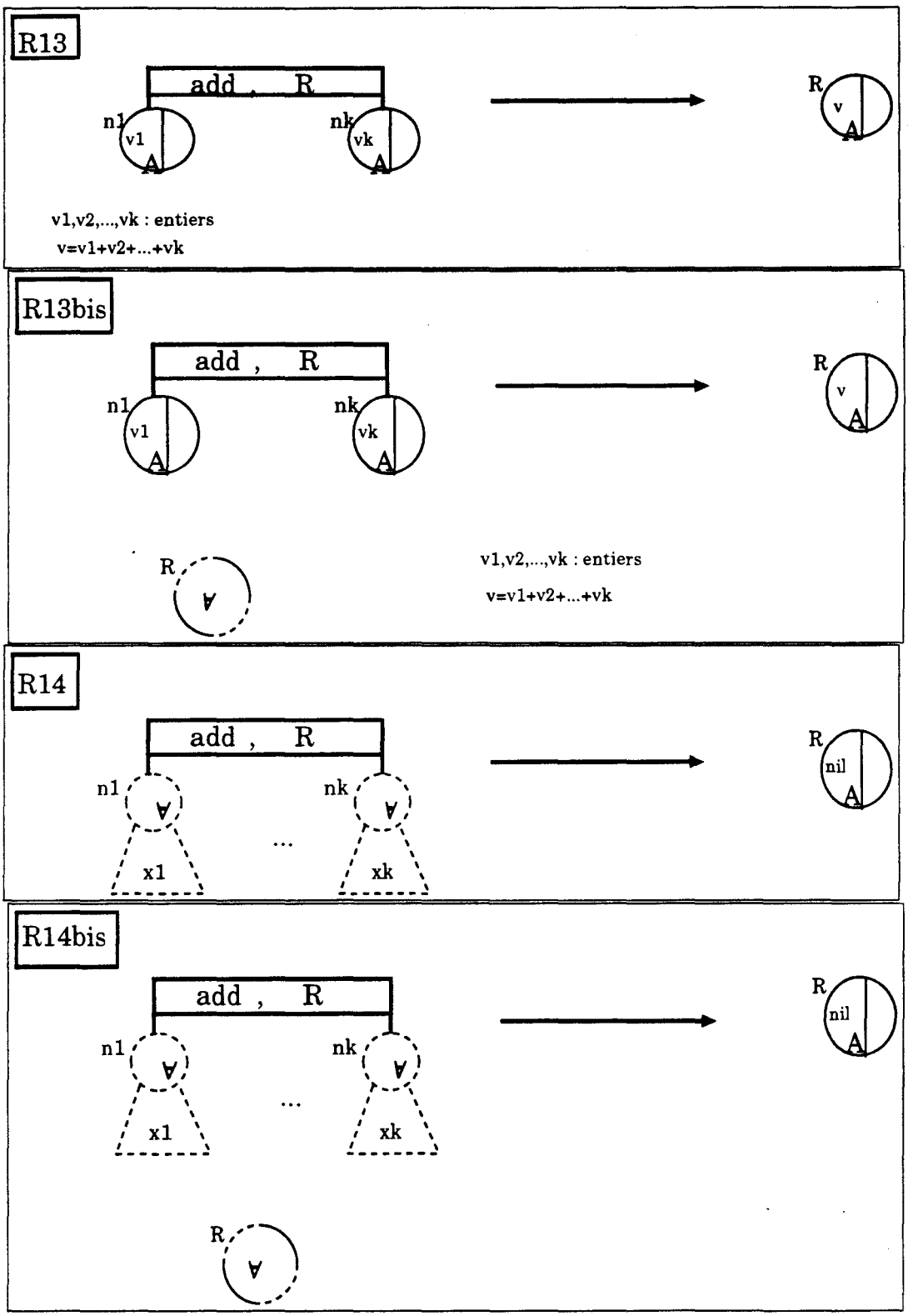


Figure 4.21 : Règles de réduction pour la fonction primitive *add*

Cet exemple met l'accent sur la multi-exploration d'une même arborescence fonctionnelle par propagation simultanée d'un *groupe* de DAE [4] (les DAE 2 et 3 sur les figures 4.22.b et 4.22.c). Cet exemple illustre également le parallélisme de réduction engendré par l'exploration d'une même arborescence fonctionnelle (traitement simultané des DAR 1, 2 et 3 sur la figure 4.23).

Toutes les transformations intermédiaires sont illustrées par les figures 4.22 et 4.23.

Etapes d'évaluation

- L'application de la règle d'exploration R6 (cf. section 8), permet de passer de l'état décrit par la figure 4.22.a à celui de la figure 4.22.b. Le nœud R racine de l' arbre de réduction résultat est créé.
- Le DAE2 est ensuite traité par application de la règle d'exploration R10. Le nœud de donnée de valeur 5 est dupliqué de façon parallèle 3 fois. On obtient les nœuds de donnée n4, n5 et n6. Le DAE3 est alors créé (figure 4.22.c).
- Les DAE3 sont traités simultanément par la règle d'exploration R1. Les DAR 1, 2 et 3 sont alors créés (figure 4.23.a).
- Ces DAR sont ensuite traités simultanément par application de la même règle. Cette étape illustre bien le parallélisme de réduction du modèle P^3.

5.2 Exemple 2

Cet exemple illustre l'évaluation de l'expression suivante :

$$\{ + \quad car \quad cdr \circ car \} : A$$

où A est un objet quelconque.
L'exemple met en évidence la possibilité d'explorer **simultanément** plusieurs sous arborescences distinctes (DAE 2, 3 et 4 sur la figure 4.24), mais aussi la possibilité d'anticiper sur l'exploration, illustrée par le traitement possible du DAE4 avant les DAE 2 et 3.

Etapes d'évaluation

[4] c'est un ensemble de DAE affecté simultanément au même nœud fonctionnel

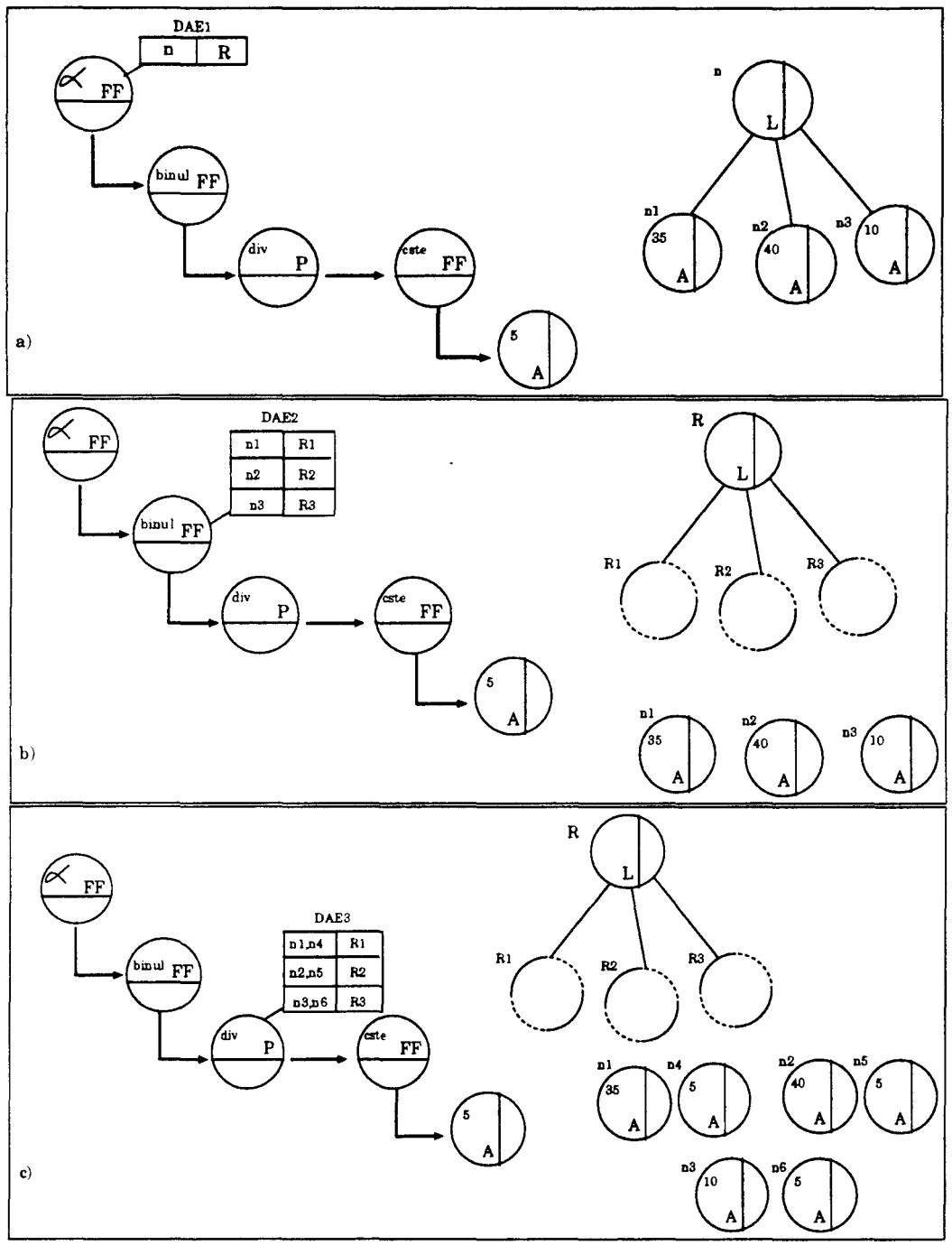


Figure 4.22 : Exemple 1

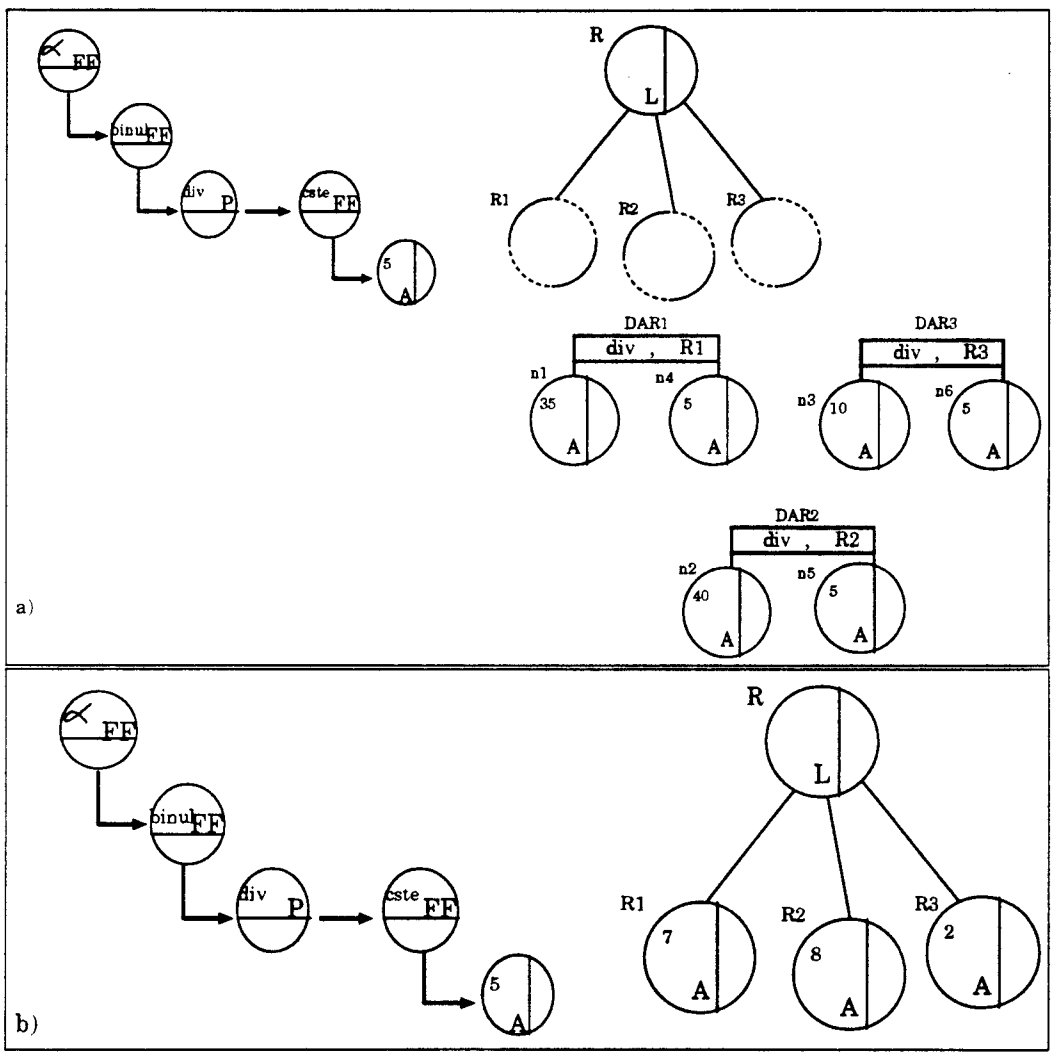


Figure 4.23 : Exemple 1 (suite)

Pour cet exemple, nous ne décrivons pas toutes les étapes de réduction. De plus, nous suivrons les transformations effectuées sur la représentation du programme **uniquement**. La représentation initiale de l'expression de l'exemple 2 est donnée par la figure 4.24.a. L'application de la règle d'exploration R5 pour traiter le DAE1 entraine la création des DAE 2, 3 et 4 (figure 4.24.b). Ces trois DAE peuvent être traités indépendamment les uns des autres par la règle d'exploration R1, puisqu'ils sont placés sur 3 nœuds fonction primitive.

5.3 Exemple 3

Cet exemple illustre l'évaluation de l'expression :

$$\{\text{list} \quad (\text{cste} \quad 5) \quad \text{id}\} : \text{A}$$

où A est un objet quelconque.

Cet exemple met en avant la possibilité d'anticipation sur l'application des règles d'exploration (traitement du DAE4 sur la figure 4.25.b) et des règles de réduction (traitement du DAR1 sur la figure 4.25.c). Ceci est faisable grâce à la création anticipée des nœuds résultats qui permet la transmission de DAE complets, prêts à être traités par des règles d'exploration. Sans cette possibilité, le DAE4 ne peut être traité **avant** les DAE 2 et 3.

La représentation initiale de cette expression est illustrée par la figure 4.25.a.

Etapes d'évaluation

Le traitement du DAE1 par application de la règle d'exploration R5 entraine comme dans l'exemple précédent, la création de 3 DAE (figure 4.25.b). Supposons que, pour une raison quelconque, nous décidions de traiter le DAE4 d'abord, par application de la règle R1 (figure 4.25.c). Le DAR1 est alors créé. Pour ce dernier la règle qui s'applique est la règle de réduction R15. Cette règle est applicable quels que soient le type et l'état des nœuds cibles. L'application de cette règle est illustrée par la figure 4.26.

5.4 Exemple 4

Dans cet exemple, nous évaluerons deux expressions, d'abord l'expression 1 :

$$\alpha \quad \circ \quad (binul_{ff} \quad binul_{c} \quad div) : \text{A}$$

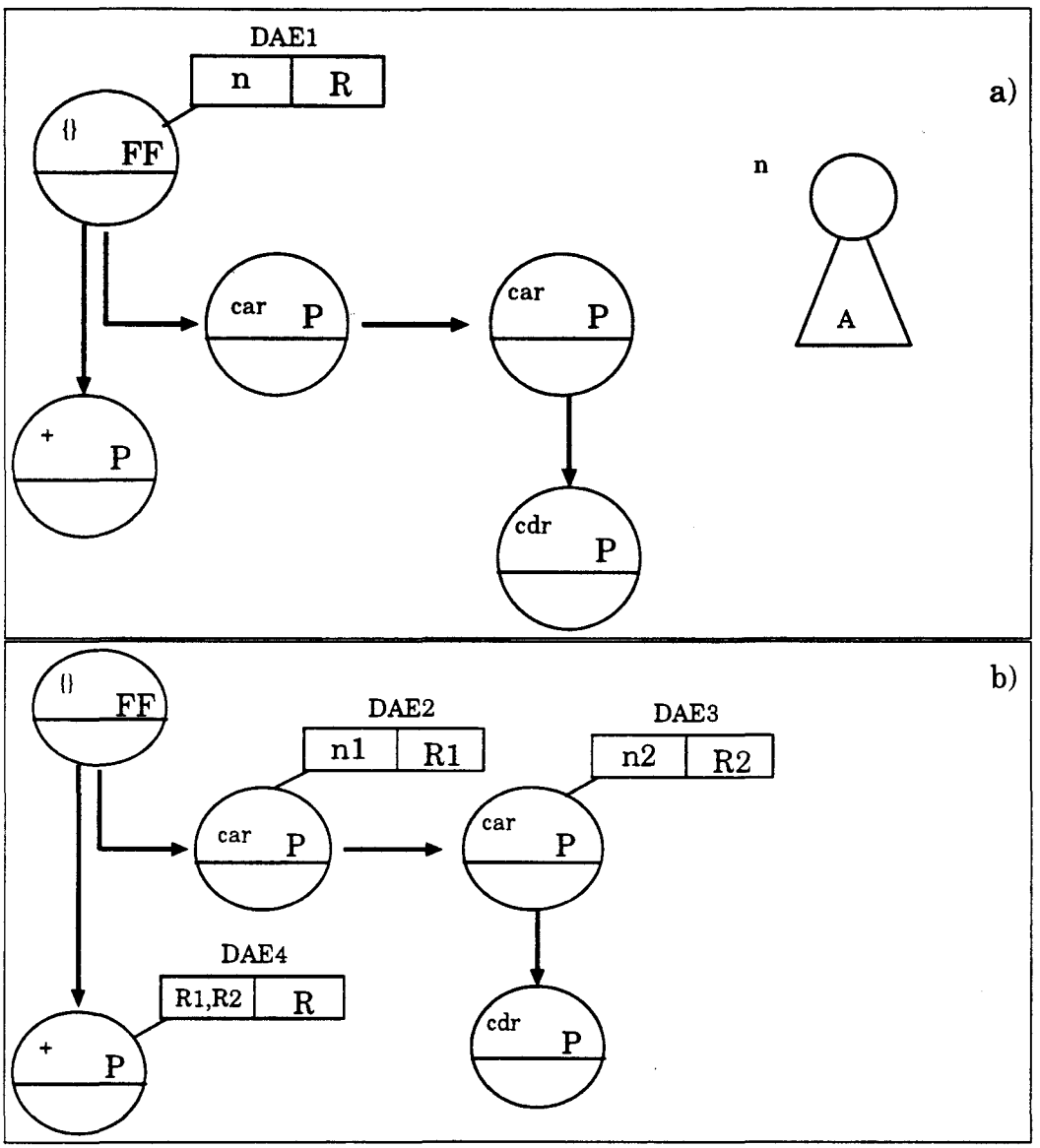


Figure 4.24 : Exemple 2

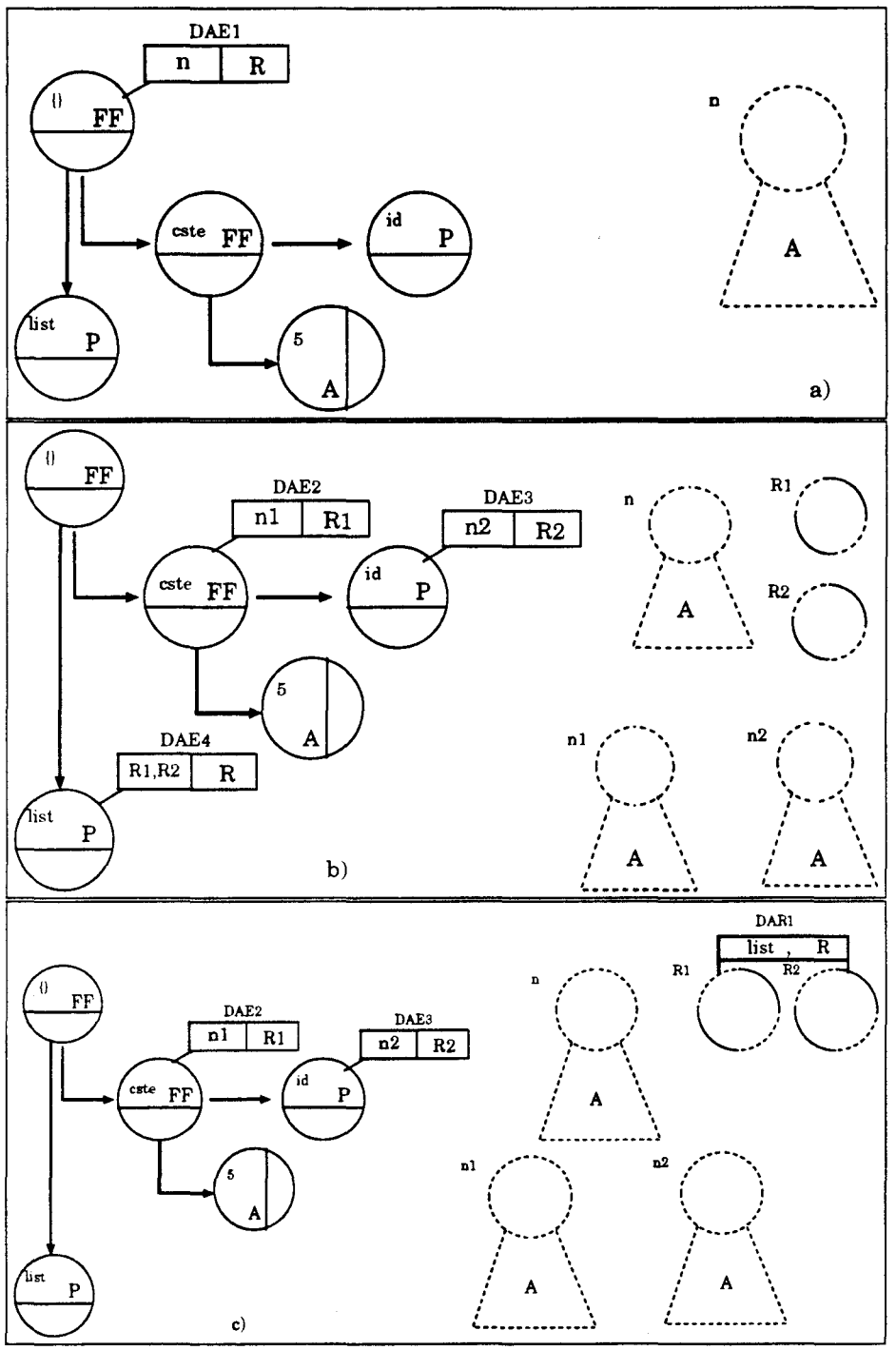


Figure 4.25 : Exemple3

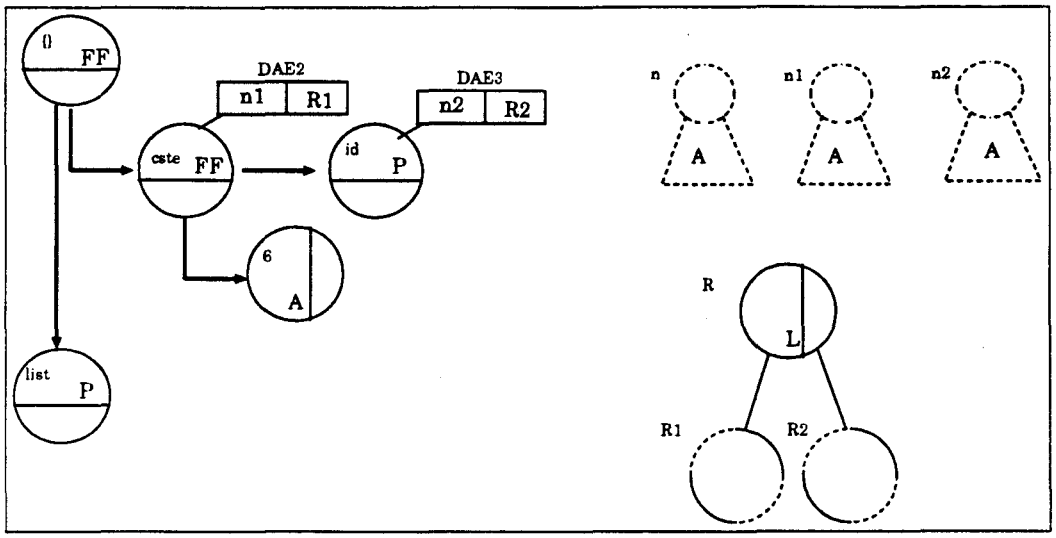


Figure 4.26 : Exemple 3 (suite)

où $binul_{ff}$ est la forme fonctionnelle *binul*, $binul_c$ est le combinateur associé à la forme fonctionnelle *binul* utilisé ici comme paramètre de la forme fonctionnelle $binul_{ff}$. A est un argument quelconque représentée par la structure de racine le nœud n; puis l'expression 2 :

$$(\alpha \circ (binul_{ff} \ binul_c \ div) : A):B$$

où B est un objet quelconque.

Cet exemple montre l'utilisation des fonctions en arguments (le nœud n_1 sur la figure 4.27.b). Nous montrons également la prise en compte des fonctions d'ordre supérieur (traitement des DAE 1 et 2 respectivement sur les figures 4.27.b et 4.27.c) et l'obtention des fonctions en résultat (l'arborescence fonctionnelle de racine le nœud R_1 sur la figure 4.28). Il illustre aussi un deuxième cas d'anticipation (traitement du DAE3 sur la figure 4.27.b). Dans le cas du traitement de l'expression 2 (voir plus haut), en anticipant sur le traitement du DAE3, on peut anticiper sur l'exploration de l'arborescence fonctionnelle de racine R_1 (figure 4.29.b) même si celle-ci n'est pas complètement construite. A remarquer également que la création du nœud R_1 se fait seulement pour le traitement de l'expression 2 dans le but d'anticiper sur le traitement.

Etapes de l'évaluation

- **expression 1**
  La représentation initiale de cette expression est illustrée par la figure 4.27.a.

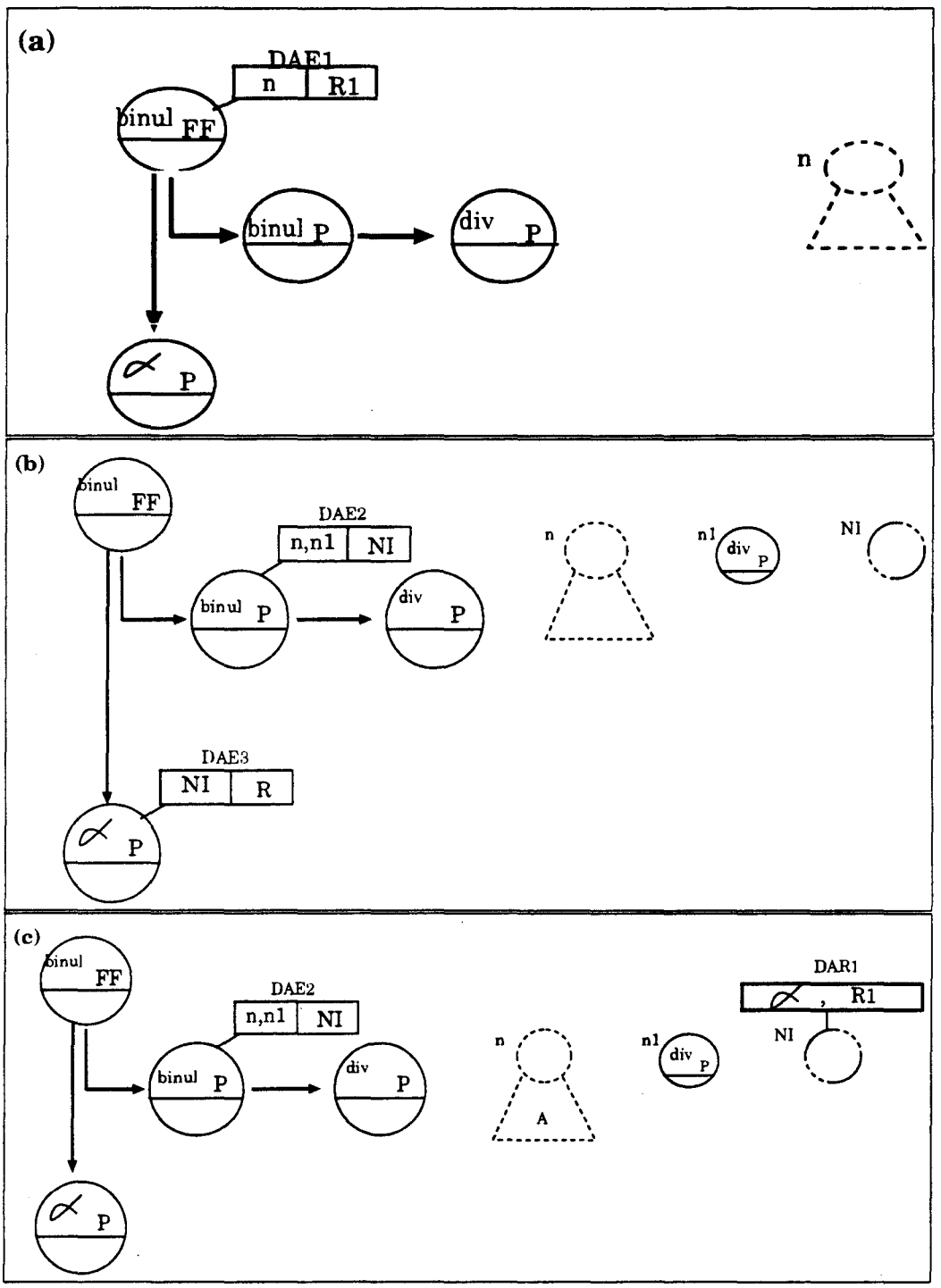


Figure 4.27 : Exemple 4

En appliquant la règle R11, on obtient la représentation de la figure 4.27.b. Si pour une raison quelconque, nous choisissons d'appliquer la règle R1 au DAE3 avant de traiter le DAE2, le DAR1 est alors créé (voir figure 4.27.c). Ce DAR contient un symbole de combinateur. Les règles applicables dans ce cas de figure s'appliquent quel que soit l'état des nœuds cibles. On obtient l'arborescence fonctionnelle de racine le nœud R_1 en état de *construction partielle* (voir figure 4.28).

- **expression 2**

L'expression 2 est représentée initialement par la figure 4.29.a. L'anticipation au niveau de l'exploration apparaît sur la figure 4.29.b, où la règle d'exploration R6 a pu être appliquée, avant que la totalité de l'arborescence fonctionnelle de racine R_1 soit entièrement construite.

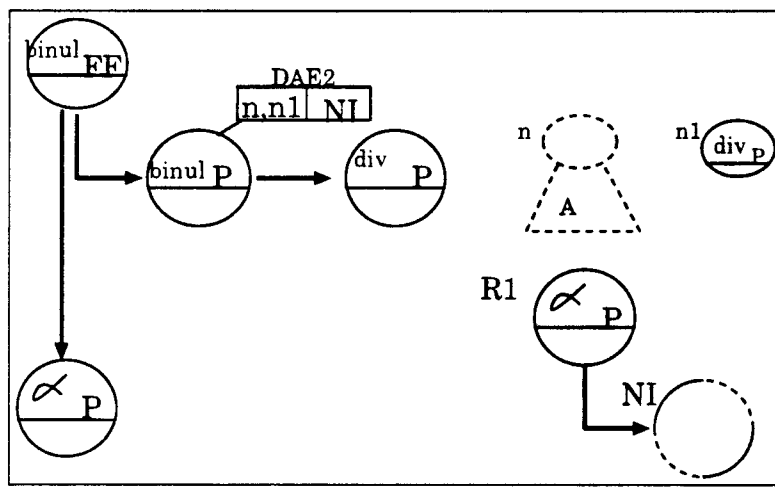


Figure 4.28 : Exemple 4 (suite)

6 Preuve de vérification de la propriété de Church Rosser du système de réécriture

De par sa définition, le système de réécriture traduit les deux contraintes d'application suivantes :

1. Une règle d'exploration ne peut être appliquée à un DAE que sous deux conditions :

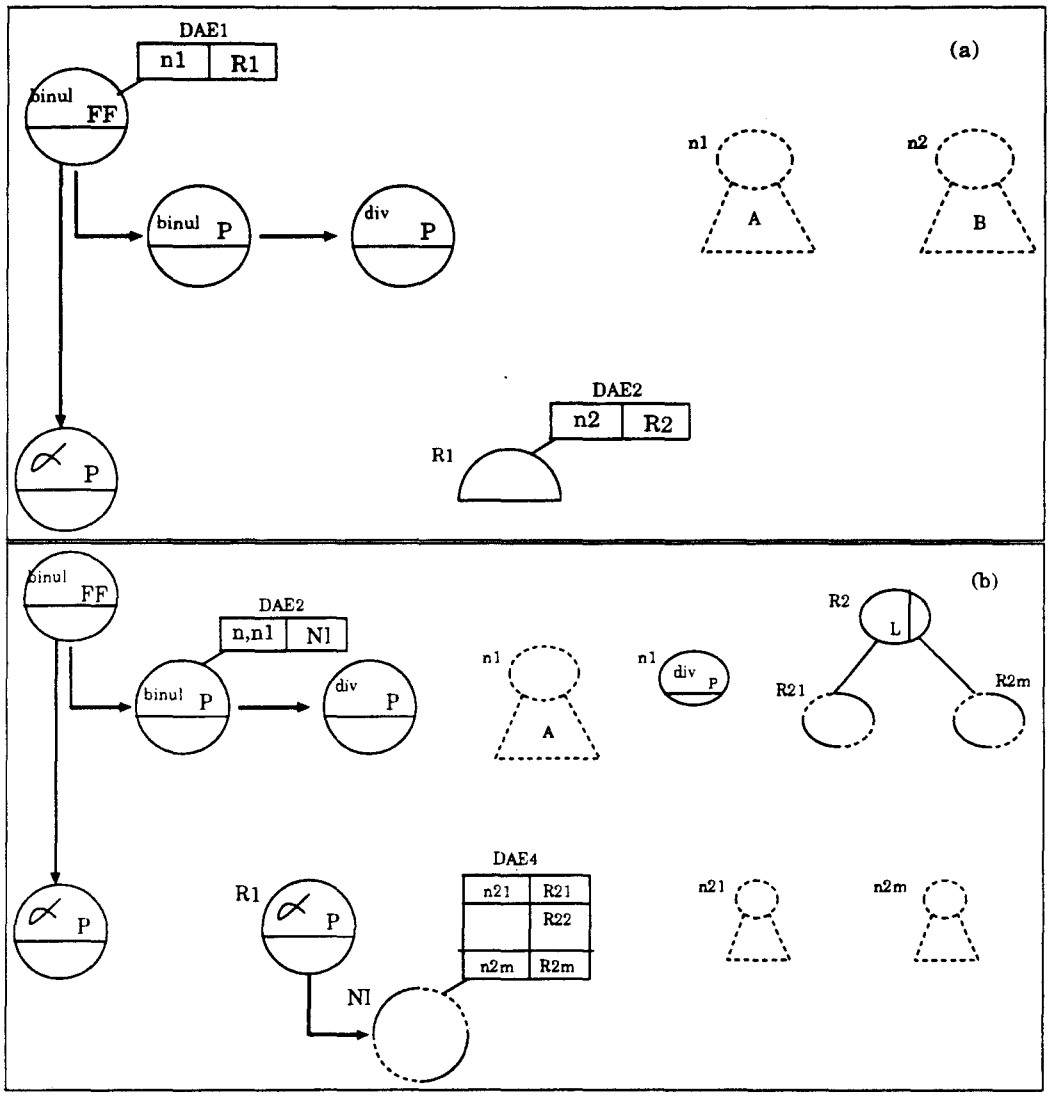


Figure 4.29 : Exemple 4 (suite)

- le nœud fonctionnel auquel le DAE est affecté, est en état de création effective.
- le DAE est complet.

2. Une règle de réduction ne peut être appliquée à un DAR (f,R) que si la fonction est non stricte sur ses arguments ou si les nœuds cibles désigant les fonctions sur lesquels la fonction est stricte sont dans l'état de création effective.

Pour prouver la correction du système de réécriture, il faut prouver que quel que soit l'ordre d'application des règles de réécriture énoncées plus haut, le résultat est unique. Formulé autrement, il faudrait prouver que pour chaque paire de drapeaux d'activation D_1 et D_2 que l'on peut respectivement traiter par deux règles du système de réécriture R_1 et R_2 :

- soit l'ordre d'application des règles est unique,
- soit l'ordre d'application des règles n'a aucune importance : le résultat obtenu après application des deux règles de réécriture quel que soit l'ordre d'application retenu est identique.

Nous démontrons cette propriété en distinguant 3 cas :

- D_1 et D_2 sont des DAE.
- D_1 est un DAE et D_2 est un DAR.
- D_1 et D_2 sont des DAR.

Les cas auxquels nous nous intéressons sont ceux où les deux drapeaux ont un lien commun i.e ils contiennent des informations communes ou alors l'ensemble des nœuds auxquels ils sont attribués ne sont pas disjoints. Le cas où les deux drapeaux sont indépendants est un cas évident où l'ordre d'application des règles n'a pas d'importance.

- **Cas 1** :
  $D_1 = (n_1, n_2, \ldots, n_k;\ NR_1)$ affecté à un noeud fonctionnel N_1.
  $D_2 = (m_1, m_2, \ldots, m_l;\ NR_2)$ affecté à un noeud fonctionnel N_2.
  Nous pouvons assurer les propriétés suivantes :

  - $\forall i, j\ \ n_i \neq m_j,\ \ si\ N_1 \neq N_2$ puisque les noeuds cibles de deux noeuds fonctionnels distincts sont toujours disjoints.

- $NR_1 \neq NR_2$ En effet, l'information *nœud résultat* est commune à tous les DAE associés aux nœuds fonctionnels composant un chemin. Or l'exploration d'un chemin séquentiel s'effectuant séquentiellement, il ne peut y avoir deux DAE *non traités* appartenant au même chemin.

Cas 1.1 : $N_2 = NR_1$
Autrement dit le drapeau D_2 est affecté au noeud résultat du chemin séquentiel auquel appartient le noeud N_1. Soit C ce chemin. Tant que le DAR, créé par application d'une règle d'exploration au dernier DAE du chemin séquentiel C, n'est pas traité, le noeud NR_1 est en état de création partielle. Dans ce cas, d'après la contrainte 1, l'ordre d'application des règles est unique : La règle R_1 est prioritaire.

Le cas où $N_1 = NR_2$ est similaire.

Cas 1.2 : $NR_1 = m_i \;\; i \in [1, l]$.
i.e Le noeud résultat du chemin séquentiel C est le ième noeud cible du noeud fonctionnel N_2. Pour la même raison évoquée pour le cas 1.1, le noeud NR_1 est en état de création partielle. L'application de la règle R_1, produit le DAR (f_1, NI_1), associé aux noeuds $n_1, n_2, \ldots, n_k$; l'application de la règle R_2 produit le DAR (f_2, NI_2), associé aux noeuds $m_1, m_2, \ldots, m_l$ et ce quel que soit l'ordre d'application. Plusieurs cas sont alors possibles pour les DAR créés (cf. Cas 3).

Le cas où $N_2 = n_i \;\; i \;\; \in [1, k]$ est similaire.

- **Cas 2** :
  $D_1 = (n_1, n_2, \ldots, n_k;\ NR_1)$ affecté à un noeud fonctionnel N_1.
  $D_2 = (f, NR_2)$ associé aux noeuds $m_1, m_2, \ldots, m_l$.

Nous pouvons assurer les propriétés suivantes :

- $NR_1 \neq NR_2$ Pour les mêmes raisons évoquées dans le cas 1.
- $\forall\, i, j \;\; n_i \neq m_j, \;\; si \;\; N_1 \neq N_2$ puisque les noeuds cibles de deux noeuds fonctionnels distincts sont toujours disjoints.

Cas 2.1 : $NR_2 = n_i \;\; i \in [1, k]$.
Le noeud NR_2 est dans l'état de création partielle puisque le drapeau D_2 n'est pas encore traité. Les deux règles de réécriture peuvent s'appliquer dans un ordre quelconque; l'application de la règle d'exploration R_1 au DAE

entraîne la création d'un DAR= (f_1, NI_1) associé aux noeuds $n_1, n_2, \ldots, n_k$.
L'application de la règle de réduction R_2 au DAR détermine la valeur du noeud NR_2. Ces deux actions ne sont pas conflictuelles et peuvent, par conséquent, s'effectuer dans un ordre quelconque.
Cas 2.2 : $NR_2 = N_1$
Dans ce cas, pour respecter la contrainte 1, l'ordre d'application des règles est unique : d'abord R_2 (qui permettra au nœud NR_2 de devenir en état de création effective), puis R_1.
Cas 2.3 : $NR_1 = m_j,\ j \in [1, l]$.
Dans ce cas, le jième argument de la fonction f est en état de construction. Si la fonction f est stricte sur cet argument, l'ordre d'application des règles est unique. Dans le cas contraire, l'ordre d'application des règles est quelconque; le résultat obtenu est le même.

- **Cas 3 :**

D_1 $= (f_1, NR_1)$ associé aux noeuds $n_1, n_2, \ldots, n_k$.
D_2 $= (f_2, NR_2)$ associé aux noeuds $m_1, m_2, \ldots, m_l$.

Le seul cas à étudier est le cas où :
$NR_1 = m_i,\ \ i \in [1, l]$.
Dans ce cas, si la fonction f_2 est stricte sur son ième argument, alors l'ordre d'application des règles est unique (R_1 puis R_2). Dans le cas contraire, quel que soit l'ordre dans lequel ces drapeaux sont traités, le résultat obtenu est identique.

Le cas où $NR_2 = n_i,\ \ i \in [1, k]$ peut être traité de façon similaire.

Nous avons démontré que pour chaque paire de drapeaux complets : soit l'ordre d'application est unique ou sans importance, le résultat obtenu est identique après application des deux règles. En réitérant ce raisonnement tout au long de l'évaluation pour chaque paire de drapeaux, nous en concluons que le résultat final obtenu est unique quel que soit l'ordre d'application des règles de réécriture. Cette propriété met en évidence le parallélisme potentiel du modèle. En effet, les drapeaux appartenant à un ensemble de drapeaux pour lesquels deux à deux, l'ordre d'application n'a pas d'importance peuvent être pris en compte simultanément.

7 Conclusion

Nous avons présenté dans ce chapitre, une description formelle du modèle P^3 en définissant une fonction $\Re$ de représentation des objets (simples ou des expressions) et un système de réécriture, dont nous avons démontré la correction, qui décrit les mécanismes du modèle : l'exploration et la réduction.

Les caractéristiques principales du modèle P^3 sont d'abord la représentation des expressions proposée. En effet, les fonctions et leurs arguments sont représentés de manière séparée. Les liens entre fonctions et arguments se font progressivement grâce au mécanisme d'exploration et sont symbolisés par les drapeaux d'activation.
Le modèle P^3 se caractérise également par ses deux activités asynchrones qui s'effectuent en parallèle.
La description que nous avons faite du modèle est celle de la version étendue. Les fonctions d'ordre supérieur y sont utilisées permettant un elargissement de l'ensemble des expressions fonctionnelles pouvant être prises en compte dans le modèle P^3.

Nous avons également montré dans ce chapitre, à travers quelques exemples, comment sont exploitées les deux formes de parallélisme des langages fonctionnels. Le modèle présente plusieurs formes d'anticipation : l'anticipation à l'exploration et l'anticipation à la réduction. La première forme d'anticipation est possible grâce à la création anticipée des nœuds résultats. L'anticipation à la réduction est possible grâce à la nature non stricte de certaines fonctions.

Le chapitre suivant présente une optimisation du système de réécriture visant à minimiser les créations de nœuds résultats intermédiaires tout en gardant l'aspect attrayant de l'anticipation.

8 Annexe A : Le système de réécriture

Avant de décrire les règles d'exploration et les règles de réduction, nous définissons d'abord un ensemble de symboles que l'on utilisera dans cette description.

8.1 Symboles utilisés

Un nœud se caractérise par les informations suivantes :

1. Son nom qui permet de le désigner sans équivoque,
2. son type : fonctionnel ou donnée,
3. Sa classe :
   pour un nœud fonctionnel, nous distinguons les classes suivantes :

P	pour la classe des nœuds fonction primitive.
FF	pour les nœuds forme fonctionnelle.
D	pour la classe des nœuds définitions

   Pour un nœud de donnée nous avons les classes suivantes :

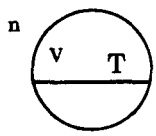
L	pour la classe des nœuds liste.
A	pour la classe des nœuds atome.

4. sa valeur,
5. son état de création : partielle si le nœud n'a pas de valeur et état de création effective sinon.

Une arborescence fonctionnelle, un arbre de données ou plus généralement, un arbre de réduction se caractérisent par :

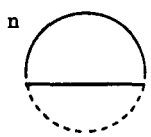
1. leur nœud racine contenant toutes les informations décrites ci-dessus.
2. **leur état de construction** : un arbre de réduction est *partiellement construit* s'il existe au moins un nœud leur appartenant qui est dans l'état *création partielle*. Dans le cas contraire, ils sont dits *complètement construits*.

1. Les nœuds fonctionnels



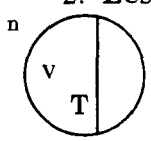
nœud fonctionnel de nom n dans l'état de création effective. Il a une valeur v et appartient à la classe T

nœud fonctionnel de nom n dans l'état création partielle. La valeur et la classe du nœud sont inconnues au moment où ce nœud est représenté

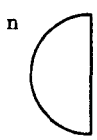


nœud fonctionnel de nom n ; son état de création est quelconque i.e il est en état de création effective ou partielle.

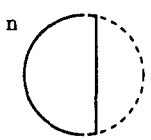
2. Les nœuds de données



nœud de donnée de nom n dans l'état de création effective ayant une valeur v et appartenant à la classe T

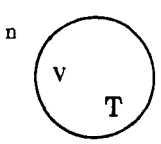


nœud de donnée de nom n dans l'état création partielle.

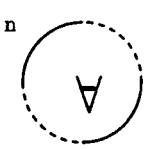


nœud de donnée de nom n dont l'état de création est quelconque.

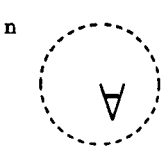
3. Noeuds quelconques



nœud de type quelconque i.e c'est un nœud fonctionnel ou nœud de donnée, de nom n de valeur v et de classe T (quelconque) dans l'état de création effective.

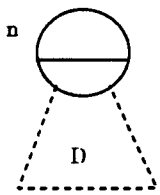


nœud de type quelconque de nom n dans l'état création partielle.

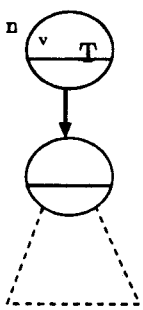


nœud de type quelconque de nom n dont l'état de création est quelconque.

4. Les arborescences fonctionnelles

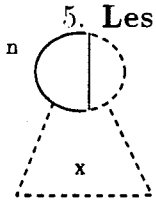


arborescence fonctionnelle principale représentant la définition D ayant pour sommet le nœud fonctionnel de nom n. L'état de construction de l'arborescence fonctionnelle est quelconque.



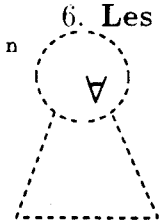
Arborescence fonctionnelle ayant pour nœud sommet un nœud fonctionnel dans l'état de création effective ayant pour valeur v et appartenant à la classe T. Ce nœud admet au moins un successeur. L'état de construction du reste de l'arborescence fonctionnelle est quelconque.

5. Les arbres de données



Arbre de donnée ayant pour sommet un nœud de donnée de nom n. Cet arbre représente un objet x dont l'état de création est quelconque.

6. Les arbres de réduction



arbre de réduction de sommet un nœud n dans l'état de construction quelconque.

Le diagramme de la figure 4.30 illustre tous les symboles décrits précédemment, du plus général au plus particulier en précisant pour chacun quels sont les symboles dont il est la représentation générale.

Le diagramme de la figure 4.31 montre les dépendances entre ces différents symboles.

8.2 Le système de réécriture

Le système de réécriture décrit ci-dessous comprend deux types de règles :

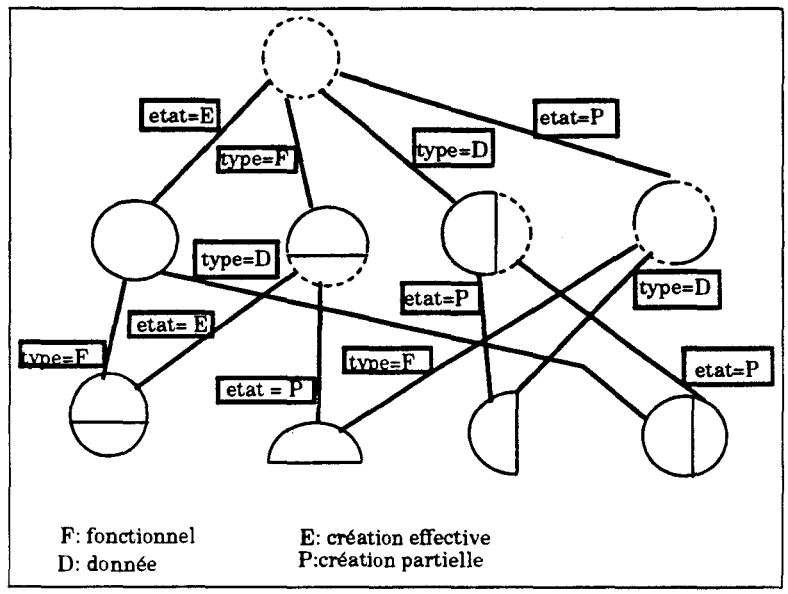


Figure 4.30 : Hiérarchie des symboles utilisés pour la représentation des nœuds

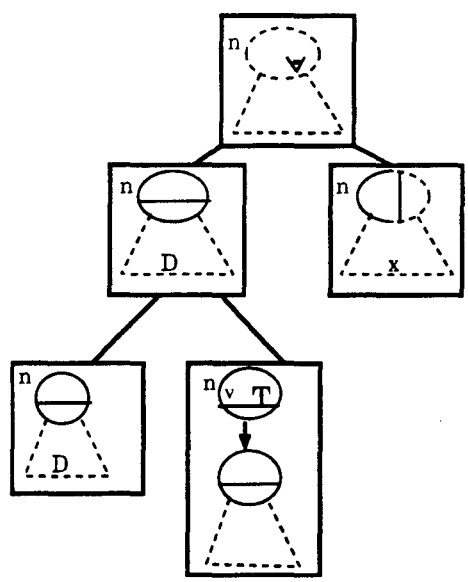


Figure 4.31 : Hiérarchie des symboles utilisés pour la représentation des arbres de réduction

- **des règles d'exploration**
- **des règles de réduction**

8.2.1 Les règles d'exploration

- Ces règles permettent d'associer à chaque nœud d'une arborescence fonctionnelle ses nœuds cibles par l'intermédiaire de l'affectation d'un DAE.
- Toute arborescence ou sous-arborescence fonctionnelle contenant au moins un nœud fonctionnel **ayant une valeur** et possédant **un DAE complet** est candidate à l'application d'une règle d'exploration.
- La règle à appliquer est choisie en fonction de :

  — la classe du nœud fonctionnel possédant le DAE,
  — la valeur du nœud, si pour une même classe de nœuds, la propagation se fait différemment pour des valeurs de nœuds différentes,
  — l'existence ou l'absence d'un successeur.

- L'application d'une règle d'exploration permet le traitement **d'un DAE** et peut engendrer la création de zéro, un ou plusieurs DAE contenant chacun, les nœuds cibles et le nœud résultat du nœud fonctionnel possédant le DAE à traiter.
- Le traitement d'un DAE par une règle d'exploration permet de créer le DAR représentant le même redex. Le DAE ainsi traité est supprimé pour que d'une part il ne soit pas redondant (le DAR généré représente le même redex) et d'autre part parce qu'un DAE complet affecté à un noeud fonctionnel peut toujours être traité par une règle d'exploration, ce qui entraînerait une incohérence au niveau de l'évaluation.
- Un nœud fonctionnel n appartenant aux classes P ou FF, racine d'une arborescence fonctionnelle A et possédant un DAE (n_1,n_2,...,n_k;R) où n_1,n_2,...,n_k sont les nœud cibles du nœud fonctionnel n, crée à la suite de l'application d'une règle de propagation un DAR qu'il affecte à ses arguments a_1,a_2,...,a_k désignés par les racines respectives de leurs représentations principales n_1,n_2,...,n_k. Ce DAR contient comme symbole de fonction la valeur v du nœud fonctionnel n et comme nœud résultat, un nœud NI qui est la racine de la représentation de la forme réduite de l'objet résultat de l'application v : $a_1\ a_2\ \ldots\ a_k$.

- Chaque nœud fonctionnel n possédant un DAE (n_1,n_2,...,n_k; R) transmet à son successeur, s'il en a un, un DAE (m_1,m_2,...,m_l;R) où m_1,m_2,...,m_l sont les nœud cibles du successeur. Le nœud résultat est identique pour le nœud n et son successeur puisqu'ils appartiennent au même chemin séquentiel (cf. section 2.4).

Cas particulier : Gestion des DAE incomplets

Soient n et Succ(n) deux nœuds fonctionnels consécutifs d'un chemin séquentiel, tel qu'à Succ(n), est associé, à la construction de l'arborescence fonctionnelle, un DAE incomplet ($m_1,\ldots,m_i\ldots$; $-$), tel que $m_1,\ldots,m_i$ sont les noeuds cibles constants du noeud fonctionnel Succ(n). Le traitement par une règle d'exploration d'un DAE ($n_1,n_2,\ldots,n_k$; R) associé à un noeud fonctionnel n conduit à la création d'un DAE ($(m_1,\ldots,m_i,R_1,R_2,\ldots,R_s)$; R) associé au noeud fonctionnel Succ(n) où $R_1,R_2,\ldots,R_s$ sont les noeuds cibles déterminés par l'exploration du noeud n et ce quel que soit le type et la valeur du noeud n.

Ce comportement est illustré par la règle suivante (cf. figure 4.32) :

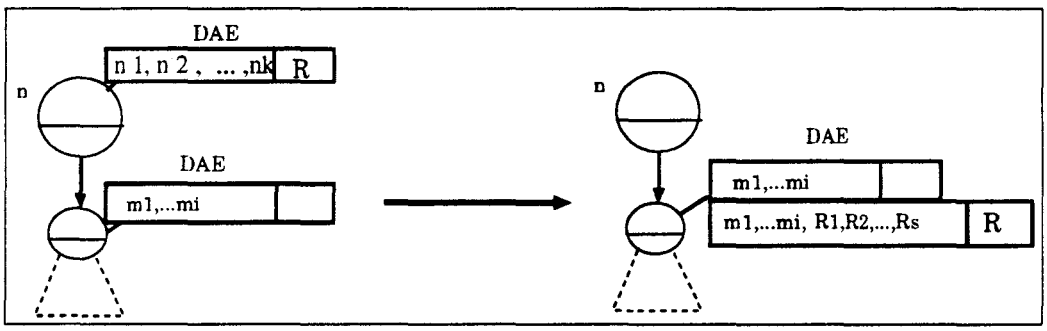


Figure 4.32 : Exploration d'un noeud en possession d'un DAE incomplet

Le DAE incomplet a le rôle de compléter les DAE associés au noeud fonctionnel Succ(n) par le mécanisme d'exploration. Il reste inchangé pour permettre la transmission des mêmes informations en cas de multi-exploration ou d'explorations successives.

Nous distinguons trois groupes de règles d'exploration suivant la classe de nœuds fonctionnels.

1. **Les nœuds fonction primitive**
   Les règles applicables dans ce cas sont les règles R_1 et R_2 de la figure 4.33.

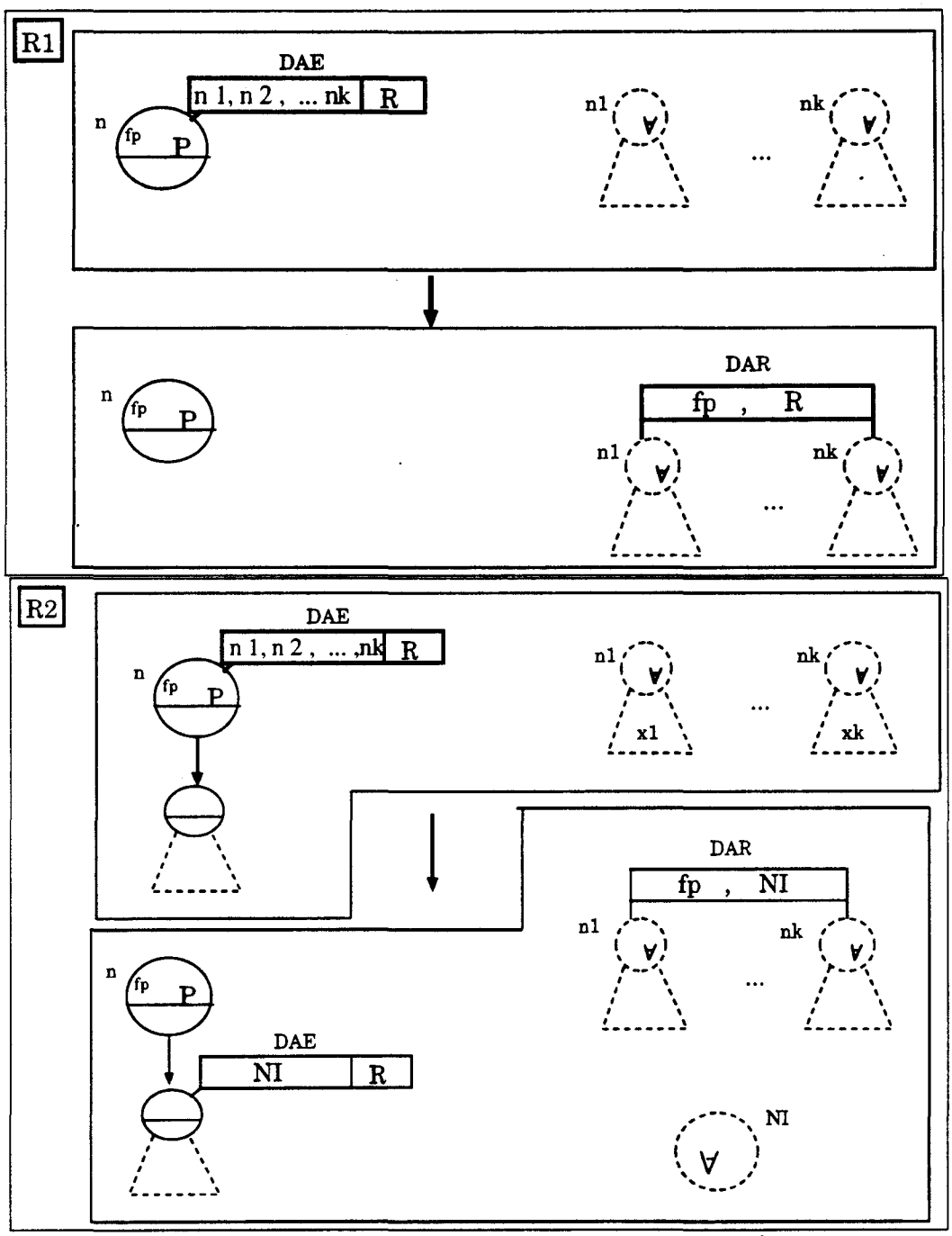


Figure 4.33 : Règles d'exploration pour un nœud fonction primitive

Quelle que soit leur valeur, les nœuds fonction primitive possédant un DAE donnent lieu à l'application de la règle R_2 si le nœud fonction primitive a un successeur et à l'application de la règle R_1 dans le cas contraire. Dans les deux cas, le nœud fonction primitive crée un DAR qu'il affecte à ses nœud cibles. Ce DAR contient la valeur f_p du nœud fonction primitive et le nom du *nœud résultat* de l'application de la fonction f_p aux arguments désignés par les nœud cibles dans le DAE. Si le nœud fonction primitive n'a pas de successeur, le nœud résultat intermédiaire est le même que celui contenu dans le DAE traité par la règle R_1 [5]. Dans le cas contraire, le nœud résultat intermédiaire est un nœud NI créé par l'application de la règle R_2. Le nœud résultat NI est dans l'état de *création partielle* puisqu'il n'a pas encore de valeur. Le successeur du nœud fonction primitive admet pour nœud cible le nœud NI qui lui sera associé par l'intermédiaire d'un DAE. Ce dernier peut donc déjà être traité par une autre règle d'exploration.

2. **Les nœuds définition**
Les règles applicables dans le cas d'un nœud définition sont : les règles R_4 si le nœud définition a un successeur et R_3 s'il n'en a pas (figure 4.34).

   (a) Si la règle R3 est appliquée :
   Le nœud fonctionnel n à qui est affecté le DAE traité n'a pas de successeur. Un seul DAE est alors créé. Il est affecté au nœud n' racine de l'arborescence fonctionnelle principale de la définition utilisée. Le nom du nœud n' est aussi la *valeur* du nœud fonctionnel n. Les contenus du DAE traité et celui associé au noeud n' sont identiques.

   (b) Si la règle R4 est appliquée :
   Deux DAE sont créés : le premier à destination du nœud fonctionnel n' comme dans le premier cas, le deuxième à destination du successeur du nœud n. A la différence du cas précédent, le nœud résultat contenu dans le DAE associé au nœud fonctionnel n' est un nœud résultat intermédiaire créé par application de la règle R4. Ce nœud est le nœud cible du nœud fonctionnel Succ(n).

3. **Les nœuds forme fonctionnelle**
La règle à appliquer pour un nœud forme fonctionnelle en possession d'un DAE dépend, en plus de la présence ou de l'absence du successeur, de la valeur du nœud forme fonctionnelle. Pour cette raison, nous retrouvons dans le système de réécriture pour chaque valeur possible d'un nœud forme

[5] Il s'agit dans ce cas, du nœud résultat du chemin séquentiel, auquel appartient le nœud fonction primitive.

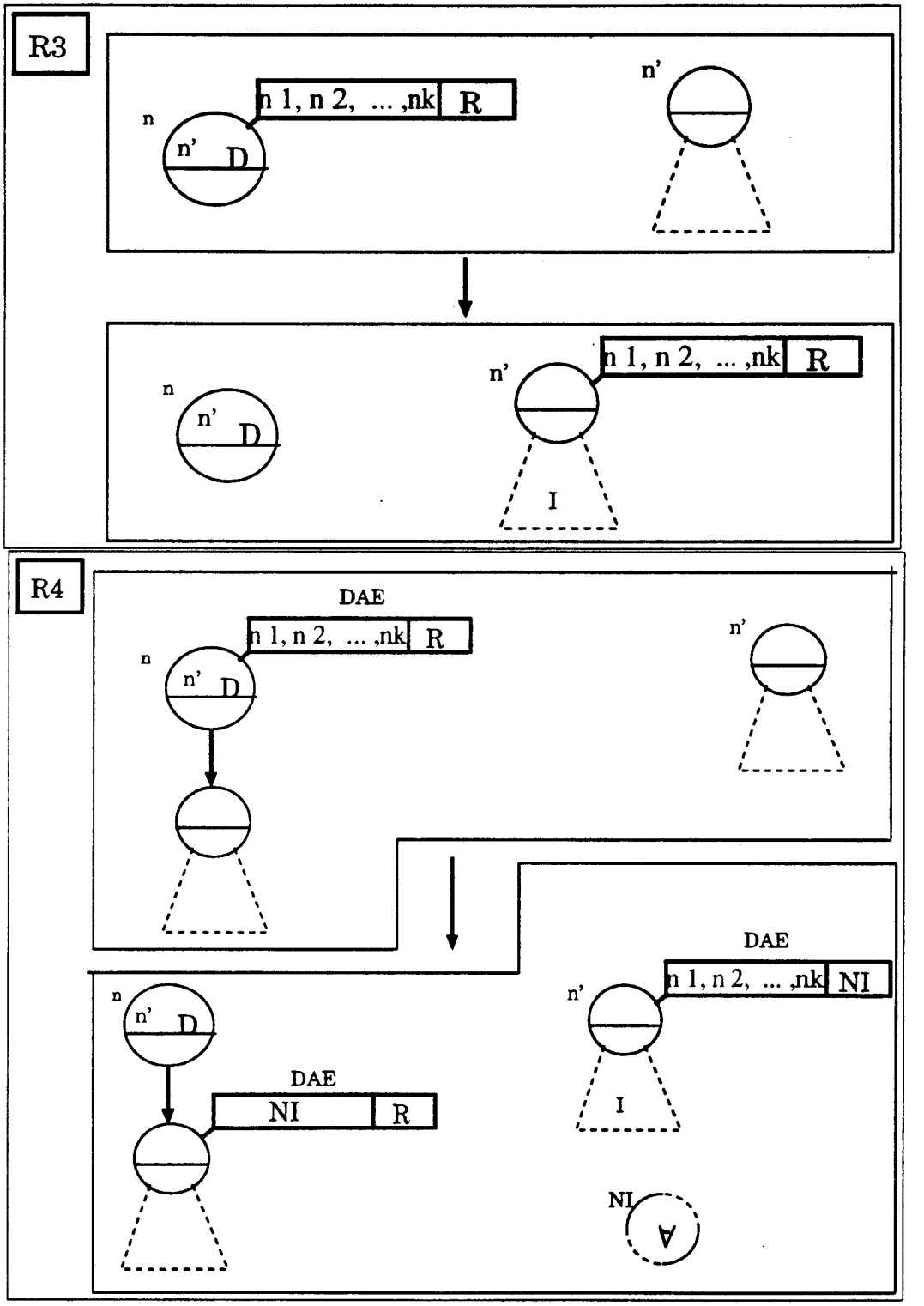


Figure 4.34 : Règles d'exploration pour un nœud définition

fonctionnelle deux règles de réécriture sauf pour le cas particulier de la forme fonctionnelle composition où le nœud forme fonctionnelle admet, quel que soit le cas, un successeur.
Pour certaines valeurs de nœud par exemple l'*apply to all*, l'application de la règle d'exploration permet aux nœuds racines des sous arborescences fonctionnelles représentant les paramètres de récupérer plusieurs DAE à la fois. Ceci signifie que chacune de ces sous arborescences sera explorée simultanément m fois, m étant le nombre de DAE. Un groupe de DAE envoyé simultanément à un nœud fonctionnel est représenté par la figure 4.35.

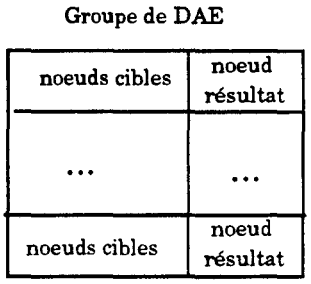


Figure 4.35 : Représentation d'un groupe de DAE

Les règles d'exploration pour les nœuds forme fonctionnelle sont illustrées par les figures 4.36,4.37, 4.38 et 4.39 respectivement, pour la forme fonctionnelle {}, la forme fonctionnelle α, la forme fonctionnelle *cste* et la forme fonctionnelle *binul*.

8.2.2 Les règles de réduction

- En s'appliquant à une suite d'arguments munis d'un DAR, une règle de réduction permet l'obtention de la forme réduite de l'objet résultat de l'application de la fonction spécifiée dans le DAR.

- Chaque suite d'arguments possédant un DAR est donc candidate à l'application d'une règle de réduction.

- La règle à choisir dépend :

  — du symbole de fonction contenu dans le DAR.
  — des objets arguments, si pour deux types d'objets différents le traitement à effectuer n'est pas le même.
  — de l'état de création des nœuds racines des arbres de réduction représentant les arguments.

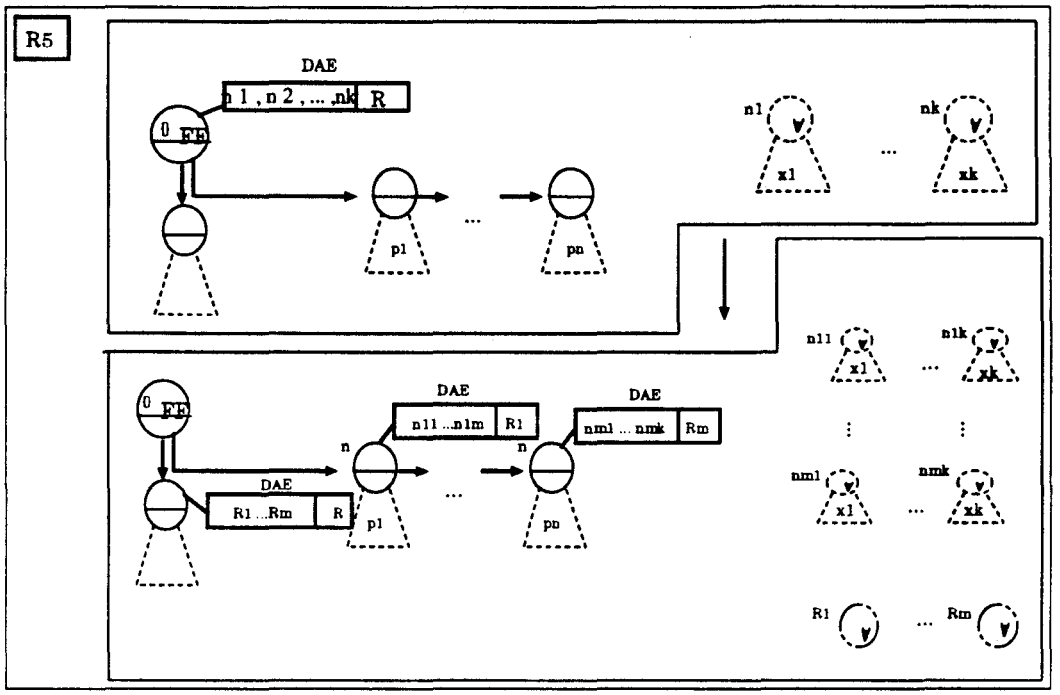


Figure 4.36 : Règle d'exploration pour un nœud de valeur {}

— de l'existence ou non du nœud résultat spécifié dans le DAR. Ce nœud, s'il n'existe pas déjà, est créé par l'application de la règle de réduction. Dans le cas contraire, il change d'état à la suite de l'application de la règle de réduction puisqu'il a désormais une valeur, donc il devient dans l'état de création effective.

- Le DAR à traiter par une règle de réduction disparait à la suite de l'application de celle-ci.

Chaque groupe de règles où le symbole de fonction contenu dans le DAR est identique traduit la sémantique d'application de la fonction spécifiée. Dans le système de réécriture proposé, pour chaque fonction, il existe un groupe de règles de réduction associé. Dans cet annexe, nous en donnons quelques exemples.
La figure 4.40 illustre les règles de réduction associées à la fonction primitive *add*.
La figure 4.41 illustre les règles de réduction associées à la fonction primitive *list* et au combinateur *if*.

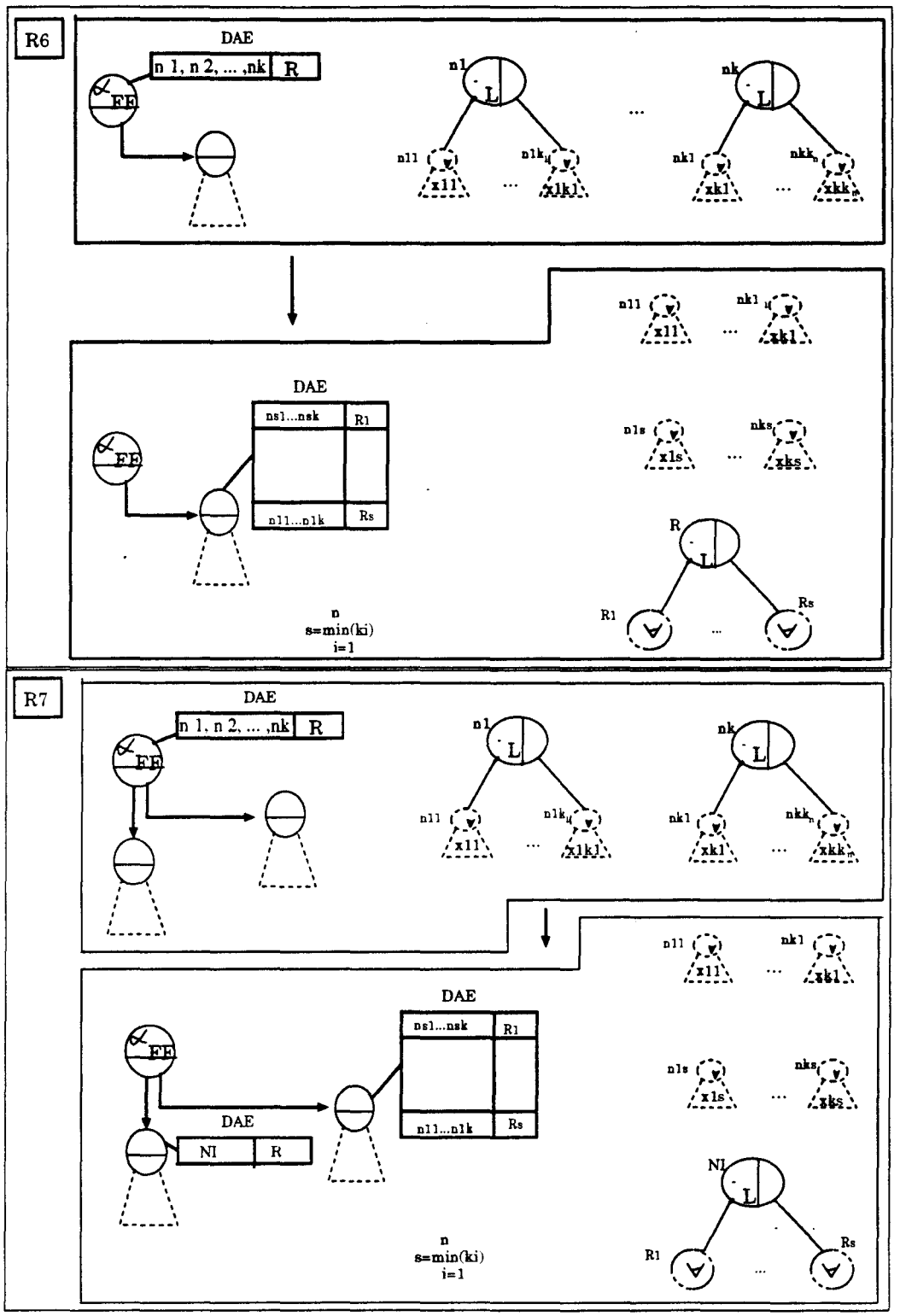


Figure 4.37 : Règles d'exploration pour un nœud de valeur α

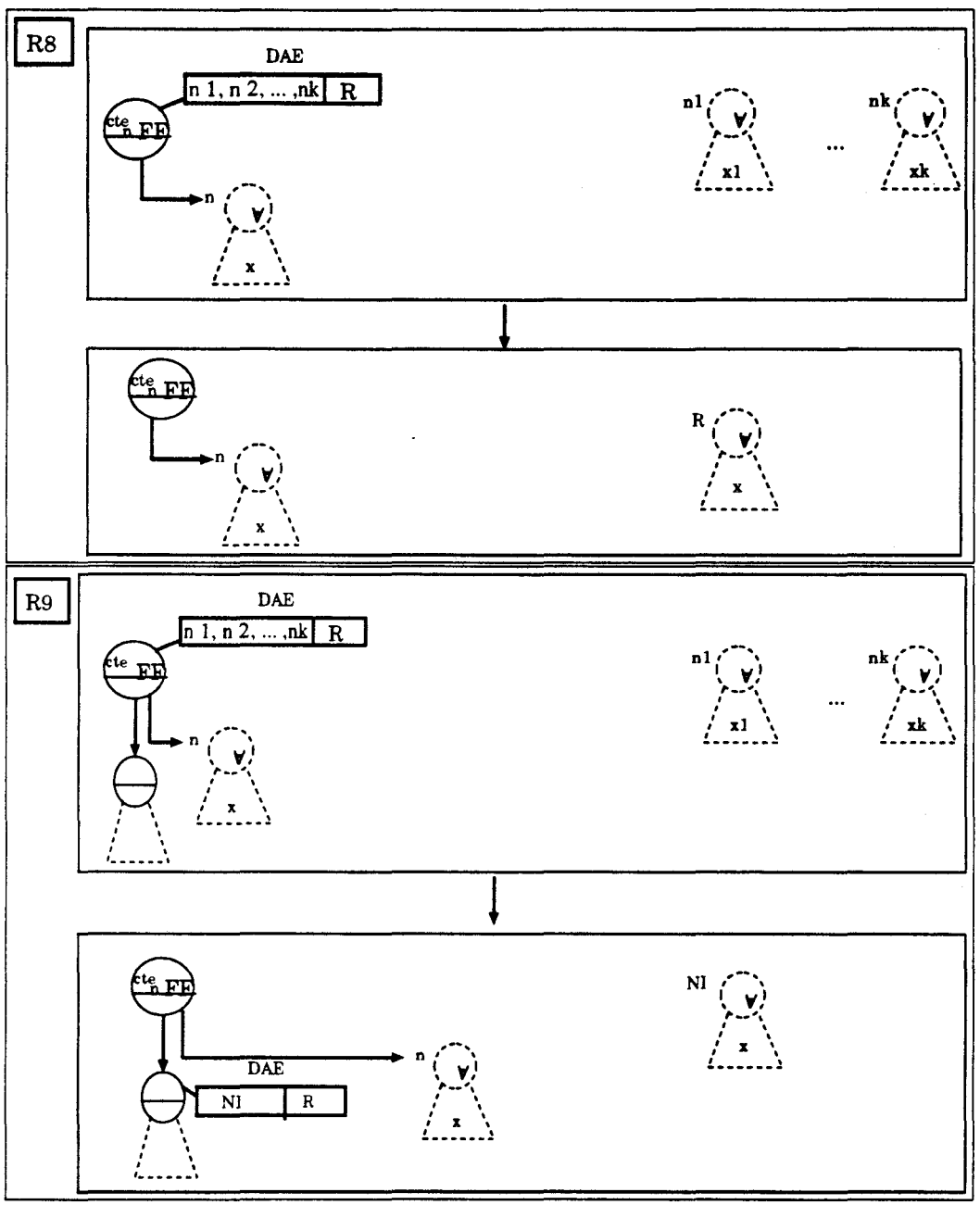


Figure 4.38 : Règles d'exploration pour un nœud de valeur *cste*

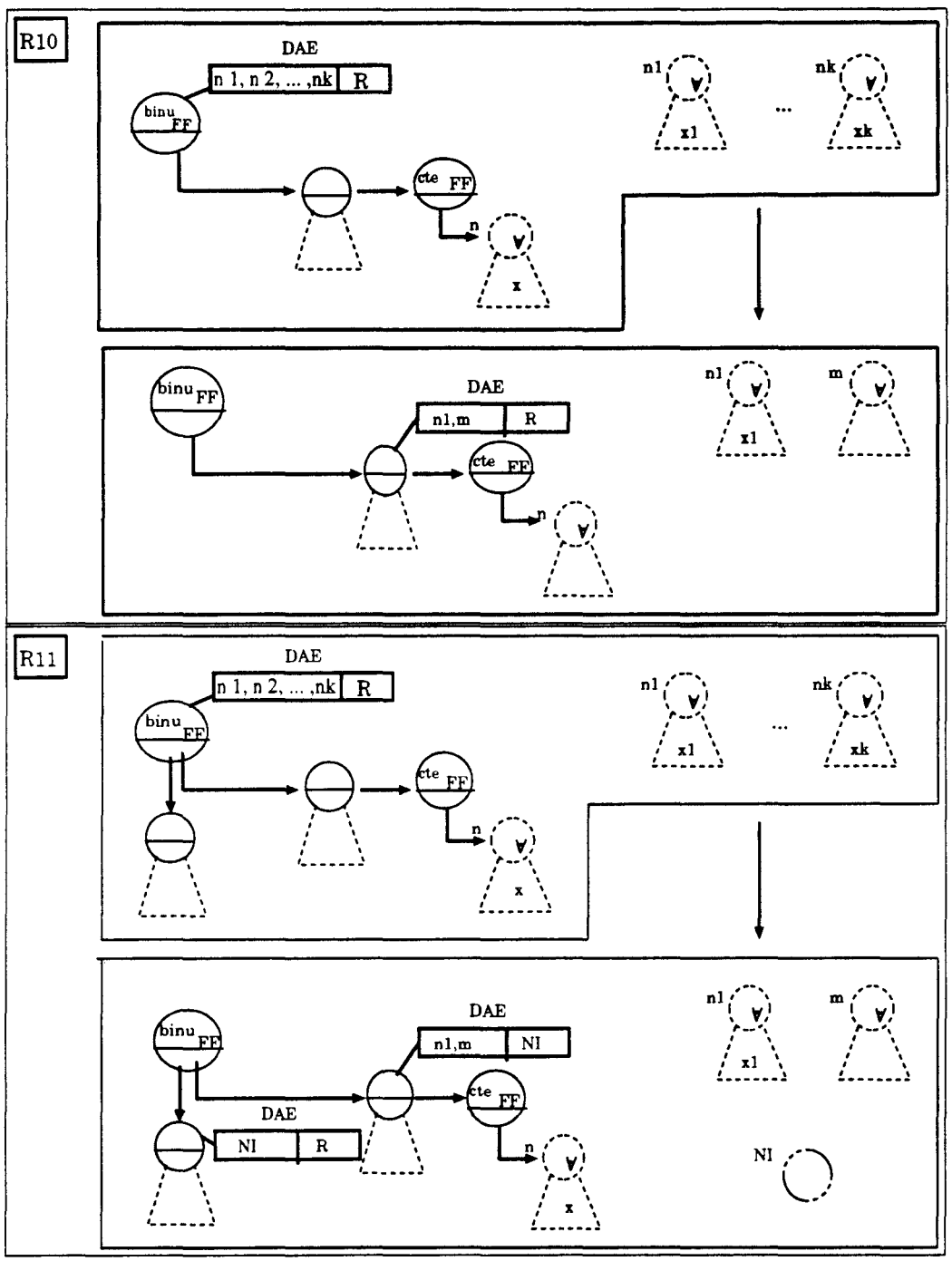


Figure 4.39 : Règles d'exploration pour un nœud de valeur *binu*

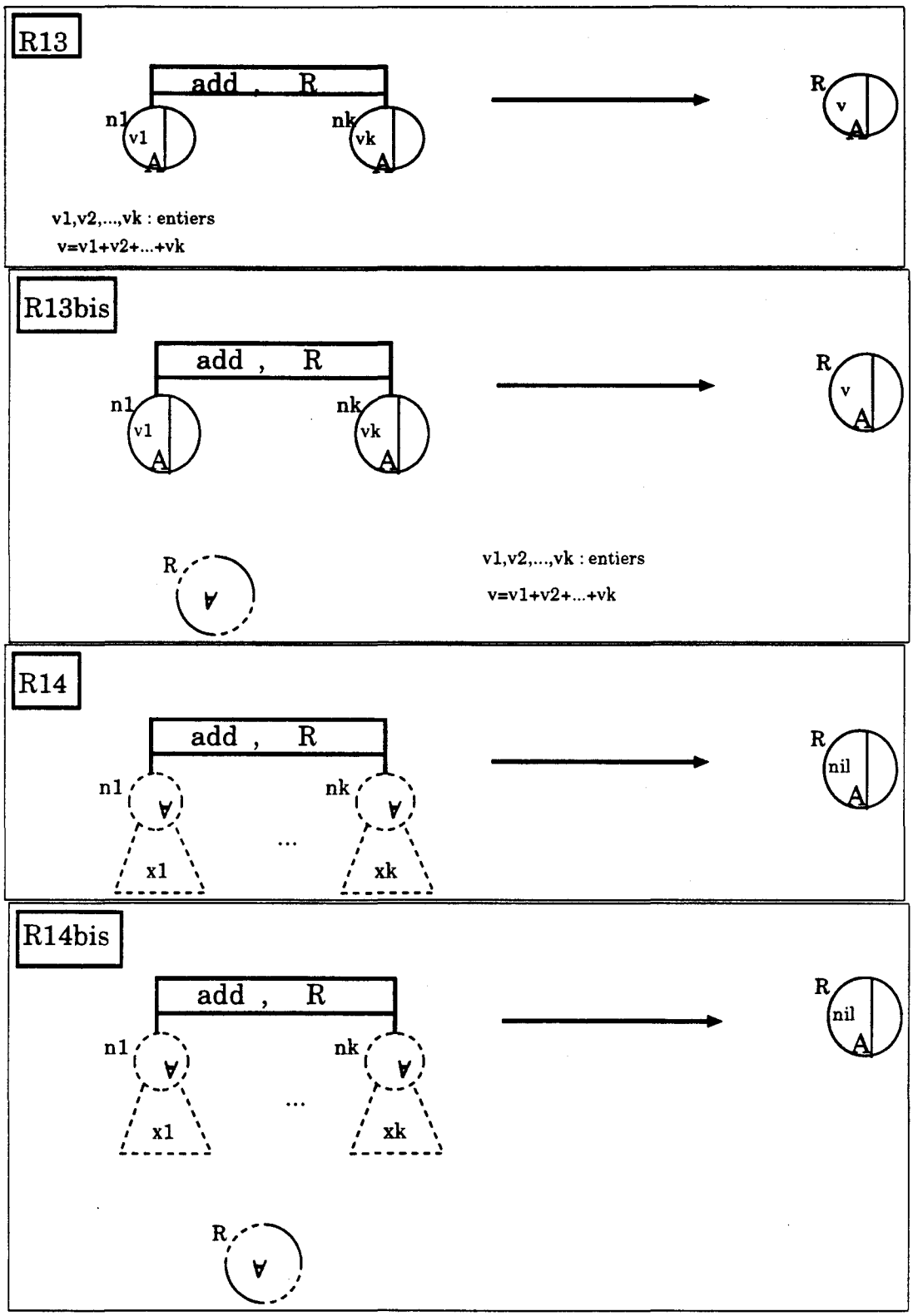


Figure 4.40 : Règles de réduction

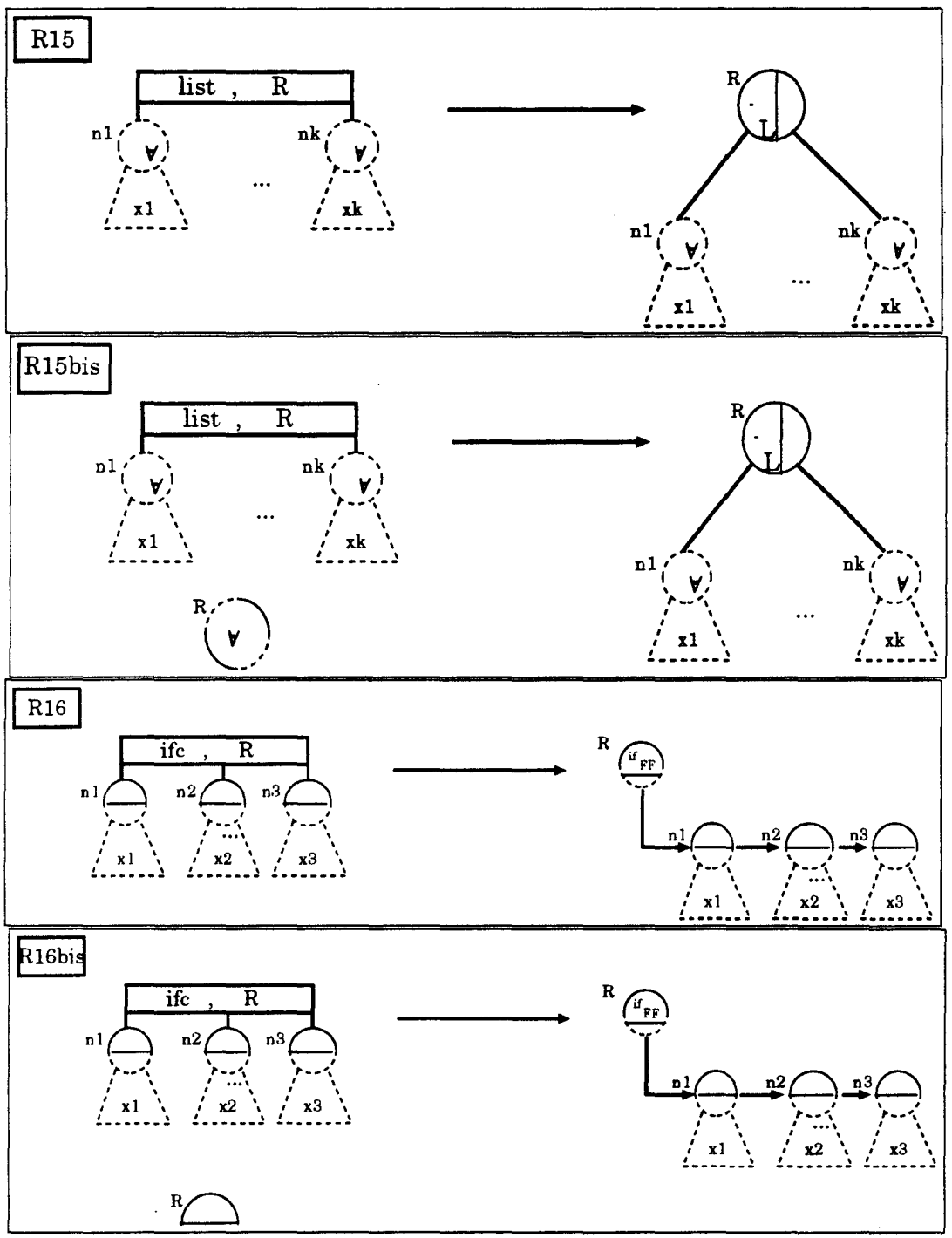


Figure 4.41 : Règles de réduction

Chapitre 5

Optimisation du système de réécriture

Dans ce chapitre, nous présentons une optimisation du modèle P^3. Cette optimisation s'impose suite à la constatation suivante : les DAR créés suite à l'application d'une règle d'exploration peuvent être triés en deux classes :

1. Les DAR qui peuvent être immédiatement traités par une règle de réduction dès leur création. Ce type de DAR illustre l'application de fonctions non strictes.

2. Les DAR qui ne peuvent être traités par une règle de réduction que si les nœuds cibles auxquels ils sont associés sont dans l'état de création effective. Ce deuxième type de DAR illustre l'application de fonctions qui sont strictes sur tous leurs arguments. Un ensemble de DAR de ce type, générés par des nœuds fonctionnels consécutifs dans un chemin séquentiel, sont obligatoirement pris en compte séquentiellement.

Dans le modèle proposé dans le chapitre 4, l'anticipation de l'exploration se fait au prix de la création anticipée des nœuds cibles des nœuds fonctionnels explorés. Cette création anticipée permet l'anticipation de la réduction dans le cas des DAR de type 1 seulement.
L'idée de l'optimisation consiste à limiter la création des nœuds cibles aux nœuds

générant des DAR de type 1. Cette optimisation est réalisée grâce à un découpage en tronçons, des chemins séquentiels composant l'arborescence fonctionnelle. Chaque tronçon se compose de nœuds fonctionnels consécutifs produisant des DAR de type 2. Ces DAR constituent une *séquence de DAR.*
En revanche, afin de ne pas empêcher l'anticipation de l'exploration, nous introduirons la notion de *fenêtre de réduction*, qui sera considérée comme un méta-nœud cible pour tous les nœuds fonctionnels d'un même tronçon.
L'association entre un tronçon et une fenêtre de réduction n'est pas bijective i.e à une même fenêtre de réduction peuvent être associés plusieurs tronçons. Par conséquent, il est indispensable d'établir un ordre partiel de ces tronçons. Cet ordre est établi grâce à une numérotation dynamique des tronçons qui s'opère au cours de l'exploration des nœuds fonctionnels.

Ce chapitre est donc organisé comme suit : la section 1 met en évidence le problème des créations inutiles de nœuds cibles. La section 2 définit formellement les notions de tronçon et de séquence de DAR correspondante, décrit le découpage en tronçons des arborescences fonctionnelles et aborde la numérotation statique des nœuds fonctionnels. La section 3 définit formellement la notion de fenêtre de réduction. Dans la section 4, nous établissons quelques propriétés des tronçons qui nous permettent de déterminer l'ensemble des tronçons associés à une fenêtre de réduction et par conséquent la numérotation dynamique des tronçons. Cette dernière sera donc décrite en section 5. Enfin, la section 6 décrit les modifications à apporter au système de réécriture pour tenir compte des notions et des mécanismes nouvellement introduits à savoir les fenêtres de réduction, la numérotation dynamique des tronçons et l'ordonnancement des DAR.

1 Mise en évidence du problème

Notations

Soit n_i un nœud fonctionnel appartenant au chemin séquentiel $\mathrm{CS}(n_1 \mapsto n_k)$.

1. On notera $DAE_{n_i}, i \in [2 \ldots k]$, le DAE qui sera transmis au nœud n_i par son prédécesseur le nœud n_{i-1}.
2. On notera DAR_{n_i}, le DAR créé à la suite de l'application d'une règle d'exploration au DAE_{n_i}.

Soit le chemin séquentiel $\mathrm{CS}(n_1 \mapsto n_m)$ composé des nœuds $n_1, n_2, \ldots, n_m$ où l'indice i désigne le rang du nœud n_i dans le chemin (voir figure 5.1). On suppose que tous les nœuds n_i sont dans l'état création effective.

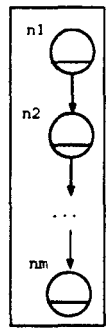


Figure 5.1 : Illustration du chemin séquentiel CS($n_1 \mapsto n_m$)

L'application d'une règle d'exploration r au DAE_{n_i} pour chaque nœud n_i du chemin séquentiel CS($n_1 \mapsto n_m$) permet la création du $DAE_{n_{i+1}}$. Ce DAE est complet i.e contient l'ensemble des nœuds cibles et le nœud résultat du nœud n_{i+1}. Par conséquent, on peut le traiter immédiatement à son tour par une règle d'exploration sans attendre le traitement du DAR_{n_i} (qui permet l'obtention de la valeur des nœuds cibles n_{i+1}) et des DAR qui le précèdent. Ceci veut dire que le modèle permet **l'anticipation au niveau de la propagation des DAE** et donc l'anticipation de l'exploration.

Le modèle permet également **l'anticipation au niveau des réductions**, dans le cas des fonctions non strictes. Nous illustrons cette possibilité par l'exemple de la figure 5.2. Dans cet exemple, les DAR : DAR_{n_1} et DAR_{n_2} ont été créés suite à l'exploration respective des nœuds fonctionnels n_1 et n_2. Dans le cas où la fonction contenue dans le DAR_{n_2} est une fonction non stricte, le DAR_{n_2} peut être traité éventuellement avant le traitement du DAR_{n_1}. C'est le cas de la fonction *list* dont la règle de réduction à appliquer est rappelée ci-après (figure 5.3). Pour cette fonction, la connaissance des valeurs des nœuds de donnée concernés par ce DAR n'est pas indispensable à l'application de la règle de réduction de la figure 5.3. Pour une fonction stricte, la fonction *null* par exemple, le DAR_{n_2} ne **peut** être traité avant le DAR_{n_1} puisque la connaissance du type et de la valeur de l'argument sont indispensables pour le choix et l'application d'une règle de réduction.

Ces deux formes d'anticipation (de l'exploration et de la réduction) sont réalisées au prix de la création de nœuds résultat intermédiaires. Ces nœuds créés permettent la propagation de DAE complets et donc entraînent la possibilité d'anticipation de l'exploration. Ces créations ne sont effectivement utiles que dans le cas où on peut anticiper au niveau de la réduction des DAR créés comme c'est le cas de la fonction *list*.

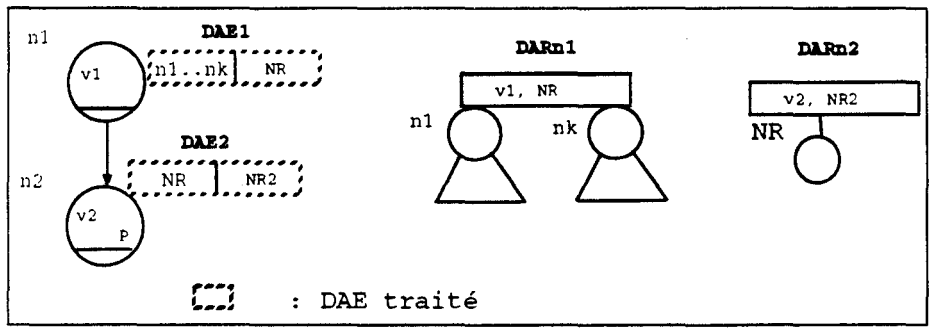


Figure 5.2 :

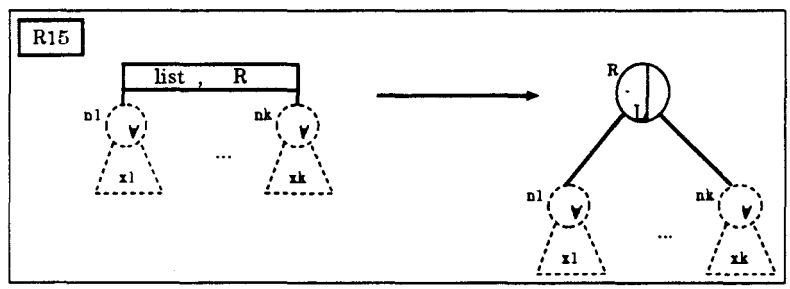


Figure 5.3 : Règle de réduction pour la fonction primitive list

Le but de ce chapitre est de proposer une optimisation du modèle visant à supprimer la création inutile de nœuds résultats intermédiaires tout en conservant la possibilité d'anticiper, et sur l'exploration et sur les réductions. Ceci est rendu possible grâce à l'introduction de deux notions nouvelles : **les tronçons** où sont regroupés tous les nœuds fonctionnels d'un même chemin séquentiel engendrant des réductions de nature séquentielle, et **les fenêtres de réduction** qui permettent l'anticipation sur l'exploration sans création de nœuds intermédiaires. Ces deux notions seront définies respectivement aux sections 2 et 3.

2 Notion de tronçon

2.1 Définition d'un tronçon

Un tronçon se compose d'un ensemble de nœuds fonctionnels vérifiant les **conditions** suivantes :

1. Ce sont des nœuds consécutifs d'un **même** chemin séquentiel i.e chaque nœud du tronçon sauf le dernier admet pour successeur dans le tronçon,

son successeur dans le chemin séquentiel.

2. Les DAR générés par chacun de ces nœuds sont traités obligatoirement de façon séquentielle et constituent une *séquence* de DAR.

3. Ni le prédécesseur du premier nœud du tronçon, ni le successeur du dernier nœud du tronçon ne vérifie la condition 2. Le regroupement en tronçons est donc **maximal**.

2.2 Définition d'un nœud frontière

Le dernier nœud d'un tronçon est appelé **nœud frontière**.

Un nœud frontière est soit :

- Un nœud forme fonctionnelle de valeur {}, binu ou binul... etc. Ces nœuds une fois explorés, engendrent des DAR pouvant être traités immédiatement.
- Un nœud fonctionnel précédant un nœud fonction primitive dont la valeur est le symbole d'une fonction non stricte. Ex: *list,const.*
- Un nœud définition : en effet, le successeur d'un nœud définition ne vérifie pas la condition 2; reprenons la règle d'exploration R4 (figure 5.4) :

  Cette règle montre bien que le nœud cible de Succ(n) - où n est un nœud définition- est le nœud résultat du dernier nœud du chemin séquentiel principal de l'arborescence fonctionnelle représentant la définition. Ce qui signifie que les DAR_n et $DAR_{Succ(n)}$ ne sont pas traités séquentiellement, pour peu que le $DAR_{Succ(n)}$ contient une fonction non stricte. Ceci explique pourquoi la condition 2 d'appartenance à un tronçon n'est pas vérifiée pour Succ(n).

Remarque :
Exception faite du dernier tronçon d'un chemin séquentiel, **tous** les tronçons d'un chemin séquentiel se terminent **nécessairement** par un nœud frontière.

2.3 Découpage en tronçons

Le découpage d'un chemin séquentiel en tronçons consiste simplement à repérer les nœuds frontières contenus dans le chemin.

A l'issue du découpage d'un chemin en tronçons, nous pouvons structurer les noms des nœuds fonctionnels (cf. section 2.4):

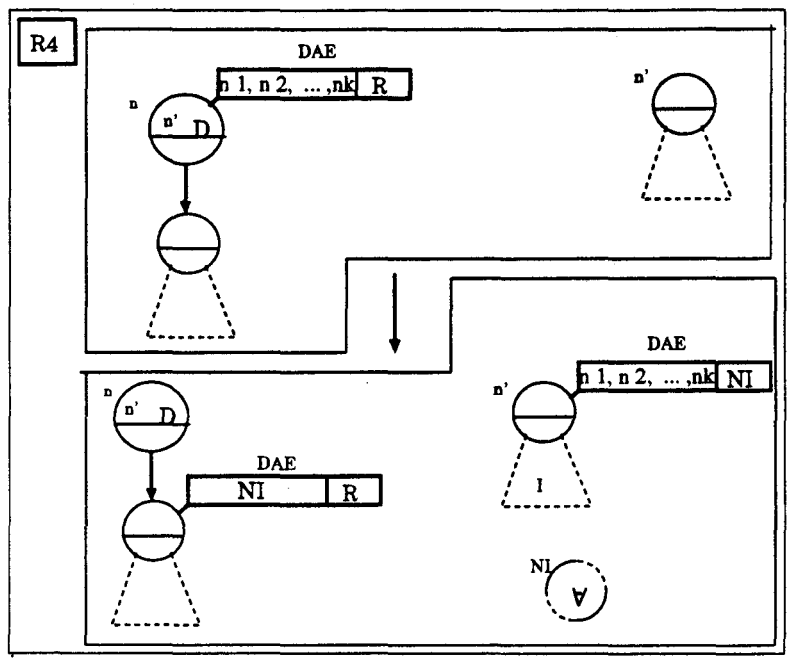


Figure 5.4 : Règle d'exploration pour un nœud définition

2.4 Structuration des noms des nœuds fonctionnels

Le nom d'un nœud fonctionnel n se compose désormais des informations suivantes :

- Le numéro de l'arborescence fonctionnelle à laquelle il appartient. Tous les nœuds fonctionnels apppartenant à une arborescence fonctionnelle ont le même numéro d'arborescence NARB.
- Le numéro du chemin séquentiel auquel appartient le nœud n. (NCH)
- Le numéro du tronçon (NTR) auquel appartient le nœud n. Ce numéro est compris entre 0 et le nombre de tronçons maximum du chemin NCH-1.
- Le numéro d'ordre du nœud n dans le tronçon NTR. (NORD)
- Le nombre maximum de tronçons dans le chemin séquentiel NCH. (NTr-Max)
- La marque de fin du tronçon (IFT). Si le nœud n est le dernier nœud d'un tronçon, IFT vaut 1. Dans le cas contraire, IFT est égale à 0.

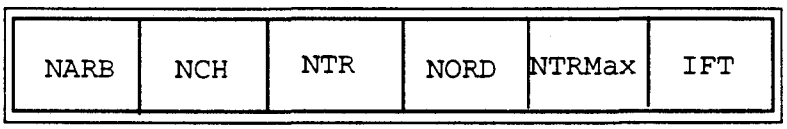


Figure 5.5 : Structure des noms des nœuds fonctionnels

La figure 5.5 résume la structure des noms de nœuds fonctionnels.

Cette numérotation permet de tester si deux nœuds fonctionnels donnés n_1 et n_2 appartiennent au même tronçon. Cette condition est vérifiée si les champs NARB, NCH et NTR sont identiques pour les deux nœuds. De plus le numéro d'ordre des nœuds permet de savoir lequel des deux nœuds est le plus proche de la racine du tronçon : un nœud fonctionnel est plus proche de la racine du tronçon si son numéro d'ordre NORD, dans le tronçon, est le plus petit. Cette information nous sera utile pour l'ordonnancement des DAR issus d'un même tronçon.

2.5 Exemple de découpage

Soit l'arborescence fonctionnelle de la figure 5.6 où les noeuds fonction primitive de valeur f désignent des fonctions primitives strictes sur tous leurs arguments :

Cette arborescence est découpée en tronçons de la façon suivante :

- Les nœuds de valeur *list* et $\{\}$ dans le chemin séquentiel 0 sont des nœuds frontières. Par conséquent, le nombre de tronçons dans ce chemin est égal à 3.
- Le chemin séquentiel 1 comporte un seul nœud frontière qui est le dernier nœud du chemin. Le chemin se compose d'un seul tronçon uniquement.
- Les chemins séquentiels 2 et 3 contiennent respectivement 2 et 1 tronçons.

Pour pouvoir anticiper sur l'exploration des nœuds fonctionnels d'un même tronçon, il faudrait que tout DAE affecté à un nœud fonctionnel soit complet i.e pour chaque nœud fonctionnel, connaître son (ou ses) nœud(s) cibles ainsi que son nœud résultat. Or, si on s'interdit la création de nœuds résultats intermédiaires pour les nœuds d'un même tronçon n_1,n_2,...,n_k, pour les raisons évoquées dans la section 1, les nœuds cibles d'un nœud n_i ne sont connus que si le $DAR_{n_{i-1}}$ a déjà été traité par une règle de réduction. Pour rendre de nouveau indépendantes les activités d'exploration et de réduction, on introduit une nouvelle notion appelée **fenêtre de réduction**. Une fenêtre de réduction se substitue aux nœuds cibles

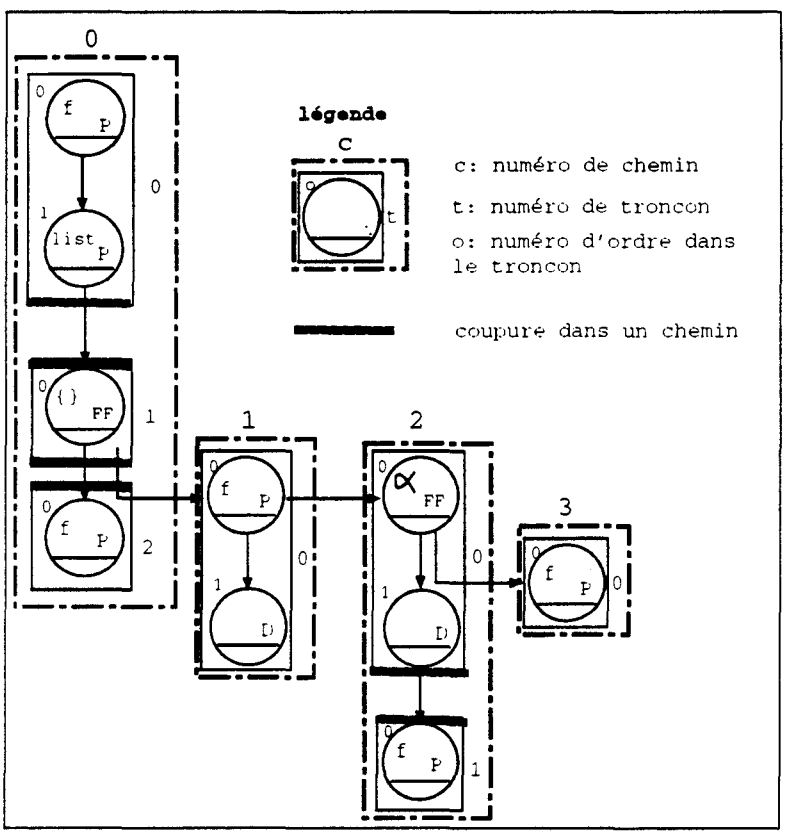


Figure 5.6 : Exemple de découpage en tronçons

d'un nœud fonctionnel. Elle est affectée au tronçon dès que son premier nœud est exploré. L'exploration pourra ainsi continuer puisque chaque nœud du tronçon connaît sa cible : il s'agit de la fenêtre de réduction associée.
La section suivante définit la notion de fenêtre de réduction.

3 Les fenêtres de réduction

Une fenêtre de réduction est définie par :

- Une suite d_1,d_2,...,d_n de DAR. Cette suite est dite *associée* à la fenêtre de réduction. Elle se compose de k séquences de DAR dont le traitement est séquentiel. Les tronçons générant ces séquences de DAR et l'ordre partiel de ces séquences sera précisé dans la section 4.

- Un ensemble d'arbres de réduction a_1,a_2,...,a_m.

Une fenêtre de réduction est une entité dynamique. En effet:

- La suite de DAR associée à une fenêtre de réduction n'est pas construite entièrement à la création, les DAR étant générés au rythme de l'exploration des arborescences fonctionnelles.

- De même, les éléments de la suite de DAR associée sont traités sans attendre la création de tous les éléments de la suite.

- Enfin, la suite d'arbres de réduction représentent à chaque instant, les arguments du premier DAR non traité de la suite de DAR associée.

Une fenêtre de réduction peut donc être illustrée à chaque instant t par un état E= (i,k,< $a_{i-1\ 1}$ ... $a_{i-1\ n_i}$ >) où

- $d_i, d_{i+1}, \ldots, d_{k-1}, d_k$ est une sous suite croissante de la suite de DAR associée à la fenêtre. d_i étant le premier élément non traité de la suite et d_k le dernier DAR généré à l'instant t de la suite associée de DAR.
- < $a_{i-1\ 1}$ ... $a_{i-1\ n_i}$ > est la suite d'arbres de réduction produits en résultat du traitement du DAR d_{i-1} i.e du dernier DAR traité.

Dans la suite de ce chapitre, nous illustrerons un état de la fenêtre de réduction par :

- Une *file d'attente* pour représenter la sous suite de DAR associée, généré mais non traitée.
- Un *espace d'action* où sont représentés les arbres de réductions à l'instant t.
- un ensemble d'informations décrivant l'état de la fenêtre . Ces informations servent à l'ordonnancement des DAR pour assurer l'ordre correct de traitement. Ces informations seront précisées dans la section 5.5 de ce chapitre.

3.1 Les fenêtres associées à un chemin

Le nombre de fenêtres de réduction associées à un chemin séquentiel est égal au nombre de tronçons dans le chemin.

Cas 1 : c'est le cas général : le chemin séquentiel se compose de deux tronçons ou plus. La transition des arguments d'une fenêtre de réduction à une autre se fait de la façon suivante :
Pour un chemin séquentiel c composé des tronçons $t_1,t_2,\ldots,t_n$ auxquels sont associées respectivement n fenêtres $f_1,f_2,\ldots,f_n$, les arguments initiaux du chemin se trouvent dans l'espace d'action de la fenêtre de réduction f_1, appelée **fenêtre de réduction initiale**.
Une fois le dernier DAR issu d'un tronçon t_i, est traité dans l'espace d'action de la fenêtre de réduction f_i, le résultat obtenu, appelé *résultat du tronçon*, représente l'état initial de l'espace d'action de la fenêtre f_{i+1}. Ainsi le résultat du dernier tronçon se retrouve dans l'espace d'action de la fenêtre f_n. La fenêtre de réduction f_n est alors appelée **fenêtre résultat** du chemin c.

Cas 2 : C'est un cas particulier du précédent : le chemin séquentiel se compose d'un seul tronçon.
Dans ce cas on affecte une seule fenêtre de réduction au chemin séquentiel. Cette fenêtre est la fenêtre de réduction initiale mais aussi la fenêtre résutat du chemin séquentiel.

Ces 2 cas sont illustrés par la figure 5.7.

Une fenêtre de réduction joue le rôle d'interface entre l'exploration et la réduction : les DAR produits par l'exploration transitent par la file d'attente d'une fenêtre de réduction avant d'être pris en compte dans l'espace d'action de la fenêtre de réduction en question. La fenêtre de réduction a donc deux rôles : recueillir un ensemble de DAR dans sa file d'attente et les traiter dans un ordre précis permettant de garder la cohérence du résultat. L'intérêt des fenêtres de réduction est aussi de conserver l'anticipation de l'exploration et de la réduction.

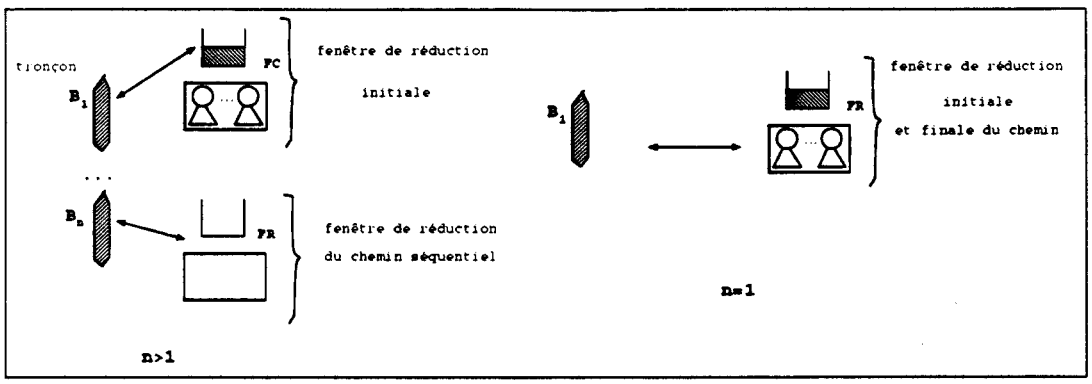


Figure 5.7 : Les fenêtres associées à un chemin séquentiel : cas de figures

Les paragraphes suivants précisent le rôle des fenêtres de réduction dans ces deux formes d'anticipation.

3.2 Anticipation de l'exploration

Une fenêtre de réduction est associée à un ensemble de tronçons. En effet, la suite de DAR associée se compose de séquences de DAR produites chacune par un tronçon. L'association entre une fenêtre de réduction et un tronçon se fait dynamiquement, **une seule fois**, quand le premier nœud du tronçon est exploré. Cette association signifie que chaque nœud du tronçon transmet son DAR vers la fenêtre de réduction associée au tronçon. Il découle de cette association que chaque nœud du tronçon connait sa cible et **peut** donc être exploré **sans création** d'une nouvelle destination.
Les nœuds cibles, au sens défini dans le chapitre 4, pour les nœuds explorés ne sont par contre pas connus à ce stade de l'évaluation. Ils le seront après la prise en compte des DAR générés par l'exploration des nœuds précedents dans le même tronçon. L'anticipation est donc rendue possible en substituant la notion de fenêtre cible à celle de nœuds cibles. Le contenu des DAE change en conséquence; un DAE_n contient donc les champs suivants :

FC — Fenêtre cible. C'est la fenêtre associée au tronçon auquel appartient le nœud fonctionnel n.

FR — Fenêtre résultat. C'est la fenêtre vers laquelle le résultat de l'application du $DAR_{n_{max}}$ est destiné, n_{max} étant le dernier nœud du tronçon.

3.3 Anticipation de la réduction

L'anticipation de la réduction est permise d'abord par le découpage en tronçons des chemins séquentiels, qui isole les fonctions non strictes des fonctions précédentes dans le chemin séquentiel; mais surtout grâce à l'attribution de fenêtres de réduction distinctes à deux tronçons consécutifs, ce qui permet aux fonctions non strictes d'être traitées dans un ordre indépendamment de celui de la séquence de DAR précédente.

Une fenêtre de réduction est associée à plusieurs tronçons. Cette association s'effectue dynamiquement en cours d'exploration. Du fait de l'exploration parallèle des arborescences fonctionnelles, plusieurs tronçons associés à la même fenêtre de réduction peuvent être explorés simultanément. Il s'agit dans ce cas, d'établir un ordre partiel entre les tronçons associés à une même fenêtre qui permettra par conséquent d'établir l'ordre d'évaluation des séquences de DAR constituant la suite de DAR associée. Létablissement de l'ordre partiel des tronçons est basé sur une numérotation dynamique des tronçons, effectuée en cours d'exploration. Avant de détailler le processus d'attribution des numéros dynamiques, la section suivante détermine, en établissant quelques propriétés sur les tronçons, l'ensemble des tronçons pouvant être simultanément associés à une même fenêtre de réduction.

4 Tronçons associés à une fenêtre de réduction

Afin de pouvoir identifier l'ensemble des tronçons associés à une même fenêtre de réduction, nous établissons les trois propriétés suivantes que nous énoncerons et démontrerons après quelques définitions.

4.1 Définitions

- **Définition 1**
  Soit a une arborescence fonctionnelle illustrant la représentation principale d'une définition d. Soient t_1 et t_2 deux tronçons tels que t_2 est le premier tronçon du chemin séquentiel principal de l'arborescence fonctionnelle a et tel que le dernier nœud du tronçon t_1 est un nœud définition représentant une occurrence d'utilisation de la définition d. Le tronçon t_2 est appelé le *descendant* du tronçon t_1.

- **Définition 2**
  Soit a une arborescence fonctionnelle illustrant la représentation principale d'une définition d. Soient t_1 et t_2 deux tronçons tels que t_2 est le dernier tronçon du chemin séquentiel principal de l'arborescence fonctionnelle a et tel que le dernier nœud du tronçon précédant le tronçon t_1 est un nœud définition représentant une occurrence d'utilisation de la définition d. Le tronçon t_2 est appelé l' *ascendant* du tronçon t_1.

- **Définition 3**
  Soient t_1 et t_2 deux tronçons associés à la même fenêtre de réduction. Le tronçon t_1 est *prioritaire* par rapport au tronçon t_2 si les DAR issus de t_1 **doivent tous** être traités **avant** ceux de t_2 dans l'espace d'action de la fenêtre de réduction commune.

- **Définition 4**
  Soient $d_1,d_2,\ldots,d_n$, une suite de tronçons tels que $\forall\ i\ \in [2,n]$, d_i est le descendant du tronçon d_{i-1}. $d_1,d_2,\ldots,d_n$ constituent une **suite croissante de tronçons** de profondeur n.

- **Définition 5**
  Soient $d_1,d_2,\ldots,d_n$, une suite de tronçons tels que $\forall\ i\ \in [2,n]$, d_i est l'ascendant du tronçon d_{i-1}. $d_1,d_2,\ldots,d_n$ constituent une **suite décroissante de tronçons** de profondeur n.

Nous allons démontrer les propriétés suivantes :

- **Propriété 1**
  Etant donnée $(t)_n$, une suite croissante de tronçons de profondeur n, tout tronçon $t_i\ \forall\ i\ \in [1,n-1]$ est *plus prioritaire* que son successeur t_{i+1}.

- **Propriété 2**
  Etant donnée $(t)_n$, une suite décroissante de tronçons de profondeur n, tout tronçon $t_i\ \forall\ i\ \in [1,n-1]$ est *plus prioritaire* que son successeur t_{i+1}.

- **Propriété 3**
  Soit FR une fenêtre de réduction. Il existe au plus :

  - Une suite **sc** croissante de tronçons.
  - Une suite **sd** décroissante de tronçons,

  qui destinent leurs DAR à FR. De plus, tous les éléments de **sd** sont plus prioritaires que ceux de **sc**.

Les propriétés 1 et 2 seront démontrées par récurrence. La propriété 3 est démontrée par l'absurde. La figure 5.8 suivante montre l'ensemble des illustrations des tronçons utilisées dans les démonstrations.

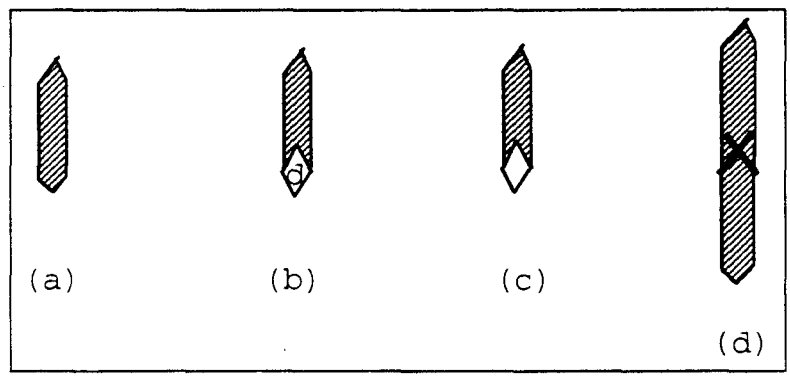


Figure 5.8 : Illustration des tronçons : (a).Tronçon quelconque.(b) Tronçon ayant comme nœud frontière un nœud définition. (c). Tronçon privé de son dernier nœud. (d). Fusion de deux tronçons.

4.2 Démonstration de la propriété 1

Nous commençons par vérifier la propriété pour une profondeur de 2 tronçons. Puis nous démontrons le cas général en supposant la propriété vraie à l'ordre *n-1*.

Cas de n=2

La propriété 1 dans le cas de deux tronçons peut être formulée de la manière suivante : soient deux tronçons consécutifs *s* et *s+1* d'un chemin séquentiel *c* et soit une arborescence fonctionnelle dont le chemin séquentiel principal se compose de m_d tronçons notés comme suit : d/0,d/1,...,d/m_d (voir figure 5.9.a) tel que le tronçon d/0 est le *descendant* du tronçon *c/s*. Les tronçons *c/s* et d/0 ont la même fenêtre de réduction et c/s est plus prioritaire que d/0.

Remplaçons le nœud *d* dans le chemin séquentiel *c* par le corps de l'arborescence fonctionnelle principale représentant la définition correspondante. Nous obtenons donc un chemin séquentiel *c'* illustré par la figure 5.9.b. Dans ce chemin séquentiel, le tronçon *c'/s* est équivalent au tronçon *c/s* privé de son dernier nœud fonctionnel. Le découpage en tronçons obtenu n'est pas maximal : en effet, le dernier nœud du tronçon *c'/s* n'est pas un nœud frontière puisqu'il n'était pas le dernier du tronçon *c/s*. Le redécoupage du chemin *c'* en tronçons obtenu est

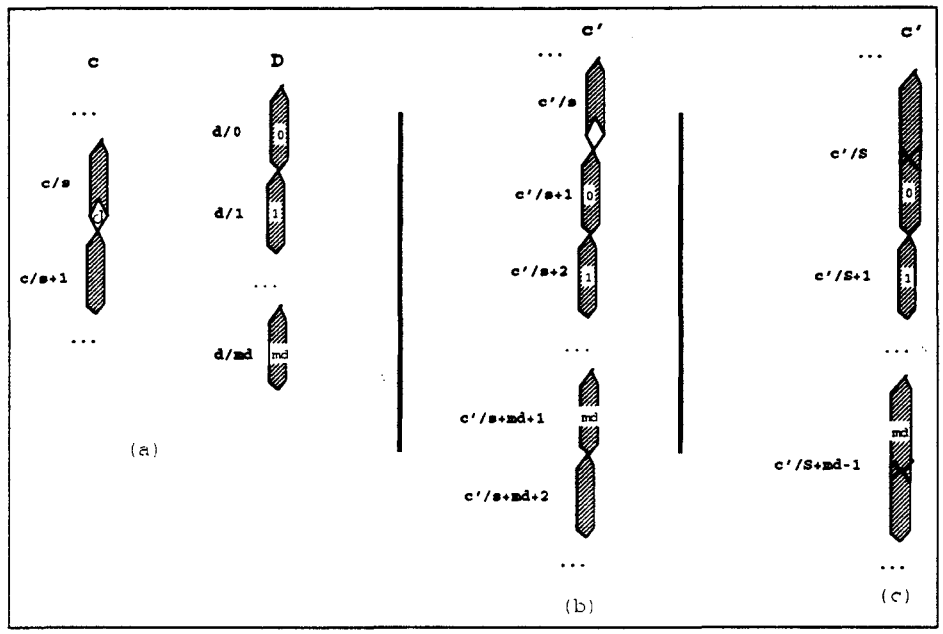


Figure 5.9 : a.Illustration des hypothèse de la propriété 1.(b) Insertion du chemin séquentiel principal de l'arborescence d dans le chemin c. (c) Correction du découpage

illustré par la figure 5.9.c où le tronçon c'/s est obtenu par fusion des tronçons c'/s et d/0.

On peut donc en conclure que:

- les tronçons c'/s et d/0 ont la même cible.
- Dans le tronçon c'/s , les "anciens" nœuds fonctionnels de d/0 se retrouvent à la suite de ceux de c'/s anciennement. Ils sont donc moins prioritaires. Par conséquent, le tronçon c'/s est moins prioritaire que le tronçon d/0. CQFD.

Généralisation

On peut facilement démontrer cette propriété dans le cas général. Supposons que cette propriété est vraie à l'ordre n i.e étant donné un tronçon s d'un chemin séquentiel c tel que le dernier nœud du tronçon s est un nœud définition d_1 de valeur la racine d'une arborescence fonctionnelle D_1, et étant donné $D_1/0.D_2/0,\ldots D_n/0$, n tronçons tels que $D_i/0$ est le premier tronçon du chemin

séquentiel principal de l'arborescence fonctionnelle D_i et tel que le dernier nœud du tronçon $D_i/0$ est un nœud définition de valeur $Rac(D_{i+1})$. Les tronçons :

$$c/s, D_1/0, D_2/0, \ldots, D_{n+1}/0$$

ont dans cet ordre la même cible.

Cette propriété est supposée être vraie pour une profondeur n i.e pour les tronçons $c/s, D_1/0, D_2/0, \ldots, D_n/0$. D'après la démonstration de la propriété 1 au rang 2, les tronçons $D_n/0$ et $D_{n+1}/0$ vérifient la propriété 1. Le résultat final de la démonstration générale s'obtient donc par transitivité.

4.3 Démonstration de la propriété 2

Cas de n=2
Comme dans le cas de la propriété 1, on reformulera la propriété 2 de la manière suivante : soient deux tronçons consécutifs s et $s+1$ d'un chemin séquentiel c et soit une arborescence fonctionnelle dont le chemin séquentiel principal se compose de m_d tronçons notés comme suit : d/0,d/1,...,d/m_d (voir figure 5.10.a) tel que le tronçon $c/s+1$ est l'*ascendant* du tronçon d/m_d. Les tronçons d/m_d et $c/s+1$ ont la même fenêtre de réduction et d/m_d est plus prioritaire que $c/s+1$. On suppose, dans un premier temps, que le dernier nœud du tronçon d/m_d n'est pas un nœud frontière (cf. section 2.2).

Remplaçons comme dans le cas précédent le nœud d dans le chemin séquentiel c par le corps de l'arborescence fonctionnelle principale représentant la définition correspondante. Nous obtenons donc un chemin séquentiel c' illustré par la figure 5.10.b. Dans ce chemin c', le tronçon c'/s+m_d+2 est l'ancien tronçon $c/s+1$. Le découpage en tronçons obtenu n'est pas maximal : en effet par hypothèse, le dernier nœud du tronçon d/m_d n'est pas un nœud frontière. Il en résulte que ce dernier a donc la même cible que les nœuds du tronçon $c'/s+m_d+2$. La fusion de ces deux tronçons aboutit au découpage illustré par la figure 5.10.c. Nous en concluons que :

- les tronçons d/m_d et $s+1$ ont la même cible.
- Dans le tronçon $c'/s+m_d-1$, les "anciens" nœuds du tronçon d/m_d précèdent ceux du tronçon $c/s+1$. Il en résulte que le tronçon d/m_d est plus prioritaire que le tronçon $c/s+1$, d'après la numérotation statique des nœuds. CQFD

Cas particulier : Cas où le dernier nœud du tronçon d/m_d est un nœud frontière (un nœud définition par exemple) (figure 5.11.a).

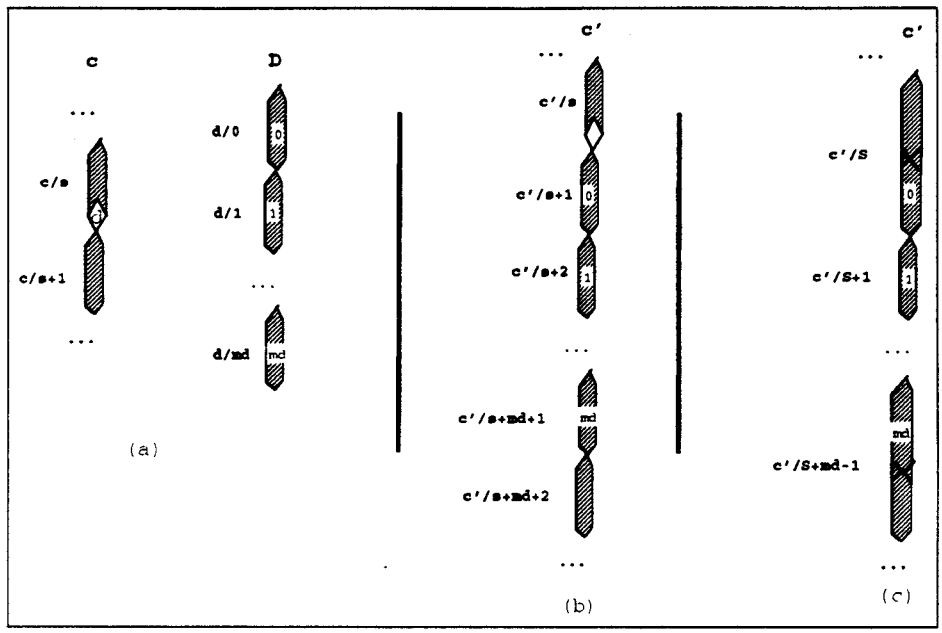


Figure 5.10 : a.Illustration des hypothèse de la propriété 1.(b) Insertion du chemin séquentiel principal de l'arborescence d dans le chemin c. (c) Correction du découpage

Ce cas pose un problème que nous illustrons par la situation de la figure 5.11.a. Si de la même façon que précédemment, nous remplaçons le nœud définition du tronçon d_1/m_{d1} par le chemin séquentiel principal de l'arborescence D_2, nous obtenons le chemin $d1'$ (figure 5.11.b). Il est évident que les tronçons $d1/m_{d1}$ (d'_1/md_1 dans le chemin d'_1) et le tronçon *c/s+1* n'ont pas la même cible. Par contre, les tronçons *c/s+1* et $d1'/m_{d1}+m_{d2}-1$ (le tronçon $d2/m_{d2}$ anciennement) ont la même cible.
Pour prendre en compte ce cas particulier, on modifie les arborescences fonctionnelles initiales en ajoutant à chaque chemin séquentiel se terminant par un nœud frontière, un nouveau tronçon. Ce tronçon se compose d'un nœud fonctionnel unique de valeur le symbole *nop*. Ce nœud fait partie des nœuds fonction primitive. Le symbole de fonction *nop* ne provoque aucune modification au niveau des arguments. Dans la situation précédente, on obtiendrait à partir des arborescences de la figure 5.12.a , le chemin séquentiel d'_1 de la figure 5.12.b, en remplaçant le nœud définition dans l'arborescence fonctionnelle D_2.

L'ajout des tronçons *nop* permet de se ramener au cas général. La propriété 2 est donc de ce fait vraie dans tous les cas.

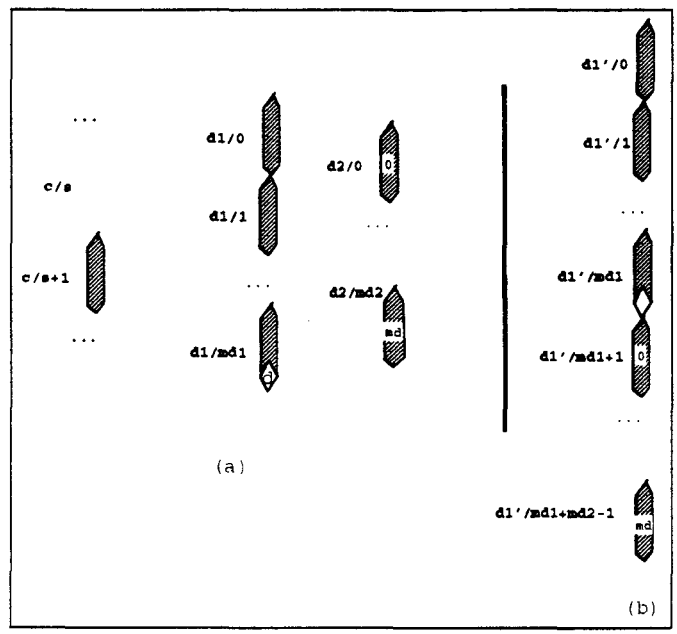


Figure 5.11 : Illustration du cas particulier

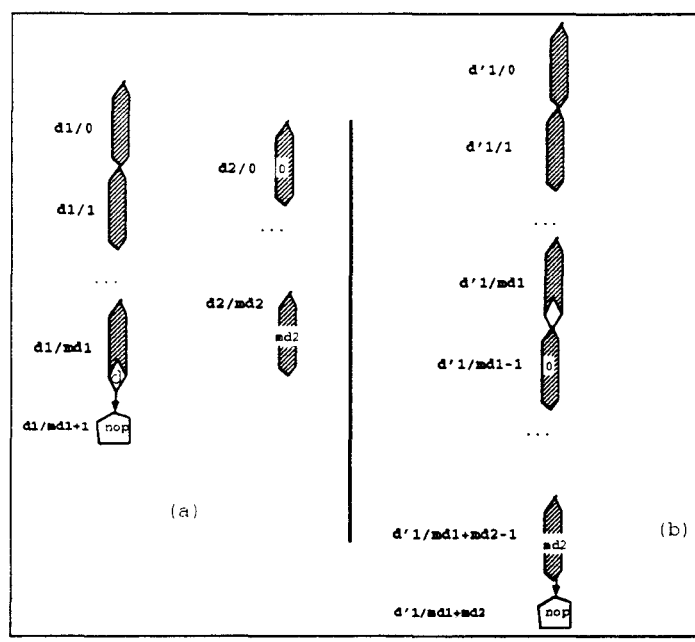


Figure 5.12 : Introduction des nœuds fonctionnels de valeur nop

Généralisation

De la même façon que nous l'avons fait pour la propriété 1, nous pouvons généraliser la propriété 2 à une profondeur d'appel quelconque. La propriété généralisée s'énonce comme suit :
Soient s et $s+1$ deux tronçons consécutifs d'un chemin séquentiel c tel que s se termine par un nœud définition d_1. Soient $D_1, D_2, ...D_n$ les arborescences fonctionnelles principales représentant respectivement les définitions d_1,d_2,...,d_n. La notation D_i/max désigne le dernier tronçon du chemin principal de D_i. Par hypothèse, chaque tronçon D_i/max est précédé d'un nœud définition dont la valeur est la racine de l'arborescence fonctionnelle D_{i+1}.
Les tronçons $D_n/max, D_{n-1}/max, ..., D_1/max, s+1$ dans cet ordre ont la même cible.

4.4 Démonstration de la propriété 3

Pour démontrer la propriété 3, nous distinguerons deux types de tronçons :

- des tronçons ayant pour nœud frontières un nœud définition et que l'on nommera des *tronçons de type 1.* Ce type de tronçon transmet sa cible à un autre tronçon.
- des tronçons se terminant par un nœud autre qu'un nœud définition que l'on nommera des *tronçons de type 2.*

Pour démontrer cette propriété, nous supposerons la propriété vraie i.e qu'à chaque fenêtre de réduction FR, sont associées une suite croissante et une suite décroissante de tronçons. Puis, nous démontrerons que, quel que soit B, un tronçon associé à une fenêtre de réduction FR, il appartient, nécéssairement, à la suite croissante ou à la suite décroissante associée à la fenêtre.

Dans la suite de ce paragraphe nous nous intéressons d'abord aux tronçons de type 1 quelle que soit leur position dans un chemin séquentiel, la nature du chemin séquentiel -principal, secondaire- auquel ils appartiennent et le type du tronçon précédent -type 1 ou type 2-. Nous procèderons de même pour les tronçons de type 2.

4.4.1 Les tronçons de type 1

- **B est le premier tronçon.** (figure 5.13.a)
  Si B appartient à un chemin séquentiel principal, il peut hériter sa cible d'un

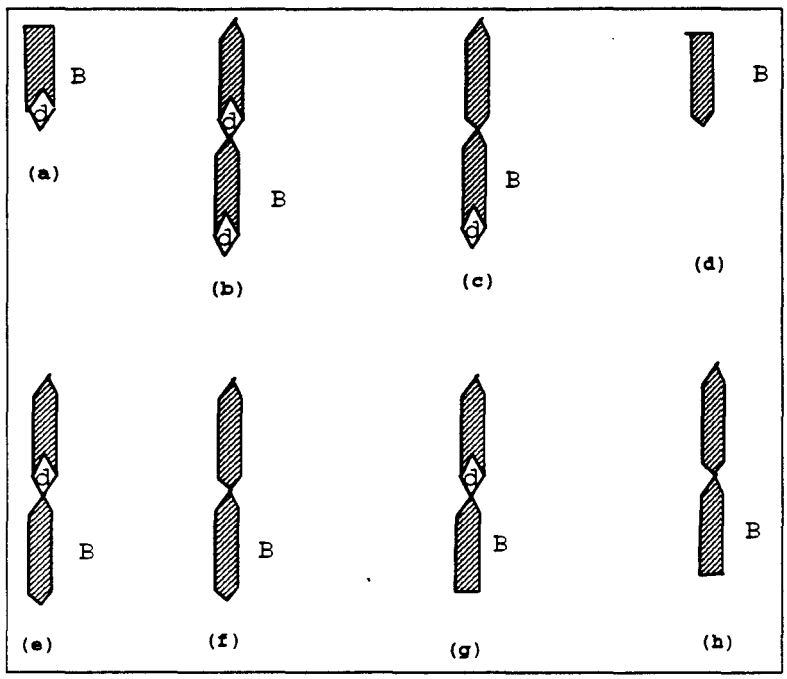


Figure 5.13 : Exemple d'arborescence fonctionnelle

tronçon ascendant. Dans tous les cas, il transmet sa cible à un descendant, premier du chemin séquentiel principal de la définition. B appartient donc à une suite croissante seulement.

- **B est un tronçon intermédiaire.**
  1er cas:
  Le tronçon B-1 est de type 1 également. (figure 5.13.b)
  Dans ce cas, le tronçon B est le dernier d'une suite décroissante. Il est également le premier d'une suite croissante puisqu'il est de type 1. Le tronçon B relie donc une suite croissante et une suite décroissante telle que cette dernière est plus prioritaire.
  2ème cas:
  Le tronçon B-1 est de type 2 (figure 5.13.c).
  Dans ce cas, le tronçon B ne peut avoir d'ascendant. Il admet cependant un descendant et appartient donc à une suite croissante.

4.4.2 Les tronçons de type 2

- **B est le premier tronçon.** (figure 5.13.d)
  Si B appartient à un chemin séquentiel principal, il est le dernier d'une

suite croissante. Dans le cas contraire, il est l'unique tronçon d'une suite croissante.

- **B est un tronçon intermédiaire.**
  1er cas:
  Le tronçon B-1 est de type 1 . (figure 5.13.e)
  B est alors le dernier tronçon d'une suite décroissante. Il ne peut appartenir à une suite croissante.
  2ème cas:
  Le tronçon B-1 est de type 2 (figure 5.13.f).
  B est l'unique tronçon associé à une fenêtre de réduction.

- **B est le dernier tronçon**
  1er cas:
  Le tronçon B-1 est de type 1 . (figure 5.13.g)
  Le tronçon B fait partie d'une suite décroissante. Il n'est ni le premier tronçon d'un chemin séquentiel principal, ni de type 1 donc il ne peut appartenir à une suite croissante.
  2ème cas:
  Le tronçon B-1 est de type 2 (figure 5.13.h).
  Si B appartient à un chemin séquentiel principal, il est donc le premier tronçons d'une suite décroissante. Dans le cas contraire, B appartient à une suite croissante de profondeur 1.

Cas particulier
Le tronçon B est le seul tronçon d'un chemin séquentiel principal. Dans ce cas, il peut appartenir à la fois à une suite croissante et à une suite décroissante telle que la suite croissante est plus prioritaire.

En résumé, une fenêtre de réduction admet au maximum une alternance de suites décroissantes et croissantes de tronçons. Ces suites sont liées les unes aux autres par des tronçons de type 1, intermédiaires, précédés par un tronçon de type 1 (cas de la figure 5.13.b).
Une fenêtre de réduction admet au minimum une suite croissante (ou une suite décroissante) d'au moins un tronçon.

La numérotation dynamique, que nous proposons dans le paragraphe suivant, permet de numéroter sans collision et sans synchronisation, une suite décroissante et une suite croissante dans cet ordre pour chaque fenêtre de réduction.

5 Les numéros de parcours dynamiques

Les numéros des nœuds fonctionnels ne suffisent pas à eux seuls à ordonner les DAR dans la file d'attente de la fenêtre de réduction cible. Seuls les DAR créés par les nœuds d'un même tronçon peuvent être classés. Aussi pour permettre le classement des DAR créés par des nœuds appartenant à différents tronçons, on introduit une nouvelle numérotation des tronçons. Cette numérotation se fait dynamiquement en cours d'exploration.

5.1 Conditions à satisfaire

La numérotation dynamique proposée doit satisfaire les conditions suivantes:

- **Condition 1**
  Tous les tronçons ayant une même fenêtre cible doivent être numérotées sans **redondance.**

- **Condition 2**
  Les numéros doivent traduire la priorité du tronçon : le tronçon le plus prioritaire doit avoir le plus petit numéro de parcours.

- **Condition 3**
  La numérotation des tronçons ayant la même fenêtre cible doit pouvoir se faire en parallèle si ces derniers sont explorés en parallèle et ne doit pas engendrer de synchronisation supplémentaire.

5.2 Numérotation adoptée

Pour satisfaire ces conditions, la numérotation dynamique se fera de la manière suivante :

- Pour toute suite croissante de tronçons (B_n) de profondeur n, on affecte les numéros 0,1,...,n-1 respectivement aux tronçons B_1 B_2 ... B_n.

- Pour toute suite décroissante de tronçons (B_n) de profondeur n, on affecte les numéros -1,-2,...,-n respectivement aux tronçons B_1 B_2 ... B_n.

Cette numérotation vérifie les conditions énoncées ci dessus. En effet :

- **Condition 1**
  Chaque fenêtre de réduction est la cible d'au plus deux suites de tronçons : une suite croissante et une suite décroissante. Les numérotations proposées pour ces deux types de suites sont disjointes donc non redondantes.

- **Condition 2**
  La numérotation est conforme à la priorité des tronçons comme le montrent les 3 cas de figure suivants :
  cas 1 : Si une fenêtre de réduction est cible d'une suite croissante de tronçons uniquement, le plus petit numéro (0) est affecté au tronçon B_0 de la suite. B_0 est l'élément le plus prioritaire de la suite (cf propriété 1).
  cas 2 : Si une fenêtre de réduction est cible d'une suite décroissante de tronçons uniquement, le plus petit numéro est le numéro -n attribué au tronçon B_n. Ce dernier est le tronçon le plus prioritaire (cf. propriété 2).
  cas 3 : la fenêtre est la cible à la fois d'une suite croissante de tronçons de profondeur n_1 et d'une suite décroissante de tronçons de profondeur n_2. Les numéros affectés à la suite croissante de tronçons sont respectivement 0,1,2,...,n_1. Les numéros affectés aux éléments de la suite décroissante sont -1,-2,-3,...,-n_2. Les éléments de la suite croissante étant tous moins prioritaires que le premier élément de la suite décroissante de tronçons (cf. propriété 3), les numéros de parcours affectés traduisent bien la priorité des tronçons.

- **Condition 3**
  Les tronçons appartenant à la suite croissante et ceux appartenant à la suite décroissante associées à une même fenêtre, peuvent être numérotés en parallèle puisque la numérotation se fait dans les deux sens, l'un positif et l'autre négatif. Ceci implique que la numérotation n'a pas besoin de synchronisation.

Pour assurer l'ordonnancement, les tronçons sont donc numérotés en cours d'exploration. Il en résulte que :

- les DAE doivent contenir des informations pour assurer la numérotation en parallèle des tronçons et se transforment donc en DAEE ou DAE étendus.
- les DAR créés suite au traitement d'un DAE doivent véhiculer des informations supplémentaires pour permettre l'ordonnancement local au niveau d'une fenêtre. Ils se transforment en DAER ou DAR Etendus.
- Enfin, la fenêtre de réduction doit disposer d'un ensemble d'informations lui permettant de désigner à chaque instant le prochain DAR à traiter.

Le contenu des DAEE, DEAR et les informations nécessaires à l'ordonnancement dans une fenêtre de réduction sont précisés dans les trois paragraphes suivants.

5.3 Les DAE Etendus (DAEE)

Un DAEE associé à un nœud fonctionnel n, contient, en plus des informations spécifiées au paragraphe 3.2, les informations suivantes :

NP le numéro de parcours dynamique du tronçon auquel appartient le nœud n. Ce numéro permet de numéroter les prochains éléments de la suite croissante de numéro dynamique np.

ND Dans le cas où le chemin séquentiel auquel appartient le noeud n contient au moins un tronçon appartenant à une suite décroissante, nd est le numéro de parcours dynamique à lui associer.

5.4 Les DAR Etendus (DEAR)

Le traitement d'un DAEE entraîne la création d'un DEAR (Drapeau Etendu d'Activation de la Réduction) qui sera mis dans la file d'attente de la fenêtre cible correspondant au tronçon. Une fois le DEAR choisi pour être traité dans l'espace d'action de la fenêtre, un DAR - au sens défini au chapitre précédent- est construit à partir du DEAR en ne gardant que les informations utiles à la réduction proprement dite. Le contenu d'un DEAR est :

- La fonction à appliquer et
- la fenêtre résultat qui reçoit le résultat de la réduction.

NP: Le numéro de parcours dynamique affecté au tronçon émetteur du DEAR.

NORD Le numéro d'ordre dans le tronçon du nœud fonctionnel.

IFT Information fin du tronçon. Cette information vaut 1 si le nœud émetteur du DEAR est le dernier du tronçon. Elle vaut 0 autrement.

5.5 Informations associées à une fenêtre de réduction

Les informations suivantes permettent de gérer le fonctionnement d'une fenêtre de réduction :

NP : le numéro de parcours dynamique du dernier tronçon traité dans la fenêtre de réduction.

NORD : le numéro d'ordre du nœud dans le tronçon ayant généré le DEAR.

IFT : fin du tronçon.

Ces informations permettent le choix du prochain DEAR qui transitera de la file d'attente vers l'espace d'action de la fenêtre.

P : le numéro de parcours le plus petit affecté à un tronçon appartenant à une suite décroissante associée à la fenêtre de réduction.

G : le numéro de parcours le plus grand affecté à un tronçon appartenant à une suite croissante associée à la fenêtre de réduction.

La figure 5.14 illustre le contenu des DAEE, des DEAR et du "registre d'état" d'une fenêtre de réduction.

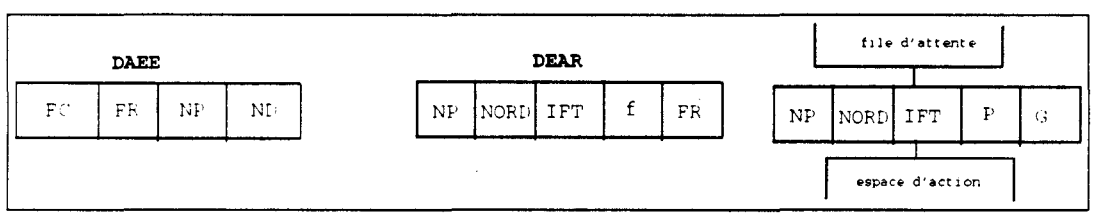


Figure 5.14 : (a). Contenu d'un DAEE (b). Contenu d'un DEAR (c). Informations associées à la fenêtre de réduction

5.6 Numérotation en cours d'exploration

La numérotation des tronçons se fait en cours d'exploration **lorsqu'un nœud frontière est exploré**. La numérotation des tronçons dépend de la nature du nœud frontière (nœud définition ou autre nœud frontière) et de la situation du tronçon suivant (s'il existe) dans le chemin. La numérotation se fait en se basant uniquement sur le contenu du DAEE traité. Nous avons donc distingué 4 cas

possibles : Soit le DAEE = (FC,FR,NP,NR) à traiter. Ce DAEE est affecté au dernier nœud n d'un tronçon B.

- **Cas1** : n est un nœud définition et le tronçon B+1 est le dernier du chemin séquentiel. Dans ce cas deux tronçons seront numérotés : le tronçon B+1 et le tronçon 0 du chemin séquentiel principal de la définition. Le tronçon B+1 appartient à une suite décroissante (cf section 4.4). Il admet comme numéro de parcours dynamique le numéro *NR*. Le premier tronçon de l'arborescence définition appartient à la même suite croissante que le tronçon B. Il aura donc comme numéro de parcours *NP+1*.

- **Cas2** : n est un nœud définition et le tronçon B+1 n'est pas le dernier du chemin séquentiel. Dans ce cas deux tronçons seront numérotés : le tronçon B+1 et le tronçon 0 [1] du chemin séquentiel principal de la définition. Le tronçon B+1 est le dernier d'une suite décroissante (cf section 4.4). Il admet comme numéro de parcours dynamique le numéro *-1*. Le premier tronçon de l'arborescence définition appartient à la même suite croissante que le tronçon B. Il aura donc comme numéro de parcours dynamique, le numéro *NP+1*.

- **Cas3** : n n'est pas un nœud définition et le tronçon B+1 est le dernier du chemin séquentiel. Le tronçon B+1 est le seul tronçon à numéroter. Il est le premier tronçon d'une suite décroissante. Son numéro de parcours est égal à *NR*.

- **Cas4** : n n'est pas un nœud définition et le tronçon B+1 n'est pas le dernier du chemin séquentiel. Le tronçon B+1 est le seul tronçon à numéroter. Il est l'unique tronçon associé à une suite croissante (cf. section 4.4). Son numéro de parcours croissant est égal à 0.

  Le schéma de la figure 5.15 montre dans les 4 cas de figure, le contenu des DAEE créés. Le symbole "-" dans le champ ND symbolise l'absence de suite décroissante en cours. Dans les cas 1 et 2, le champ ND est initialisé au dernier numéro attribué d'une suite décroissante.

A chaque fois qu'un nouveau tronçon est atteint par l'exploration, il sera numéroté en utilisant les informations contenues dans le DAEE. Il en résulte que les règles d'exploration doivent être modifiées pour prendre en compte d'une part, la numérotation dynamique des tronçons et d'autre part, pour distinguer les cas où la création d'une fenêtre de réduction est nécéssaire du cas où elle ne l'est plus. Ces modifications seront discutées dans la section suivante.

[1] Il s'agit du numéro statique

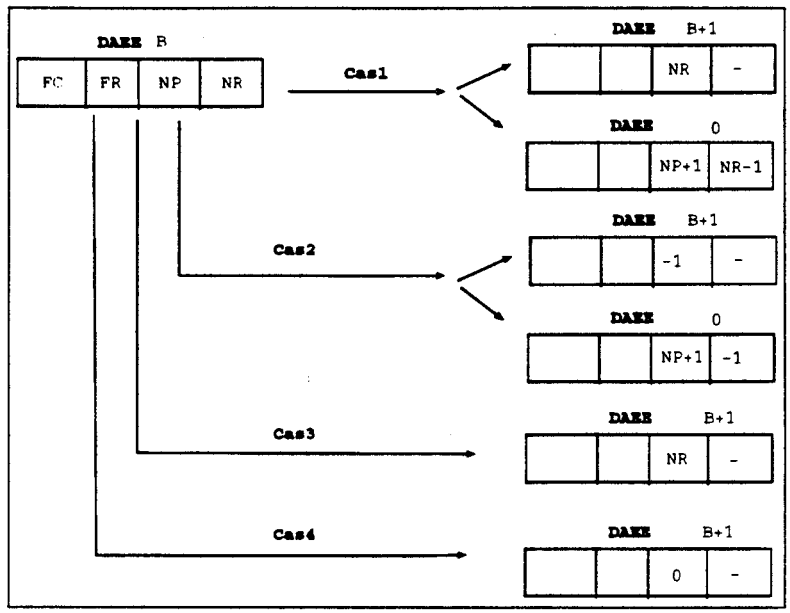


Figure 5.15 : Illustration de la numérotation dynamique en cours d'exploration

6 Modification du système de réécriture

Comparé au système de réécriture proposé au chapitre 4, le système de réécriture optimisé présente les différences suivantes :

- Les règles d'exploration ont été modifiées pour tenir compte de la numérotation dynamique et de la création sélective des fenêtres de réduction.

- Les règles de transition constituent une nouvelle catégorie de règles permettant le passage d'un DAR (correspondant au DEAR sélectionné dans la file d'attente) vers l'espace d'action de la fenêtre.

Remarque
Les règles de réduction ne subissent aucune modification.

Dans cette section, nous discutons d'abord des règles d'exploration puis des règles de transition.

7 Les règles d'exploration

7.1 Aspect général des règles

Trois principaux changements sont à noter au niveau de la représentation des règles dans le système de réécriture optimisé par rapport au système de réécriture présenté au chapitre 4 :

- Les DAE sont remplacés par les DAEE.
- La notion de fenêtre cible se substitue à la notion de nœuds cibles.
- Le traitement d'un DAEE crée un DEAR dans la file d'attente de la fenêtre cible au lieu d'un DAR affecté aux nœuds cibles.

7.2 Exemples de règles d'exploration

- **Exemple1** : (figure 5.16) Cette règle montre l'exploration d'un nœud fonctionnel non frontière.

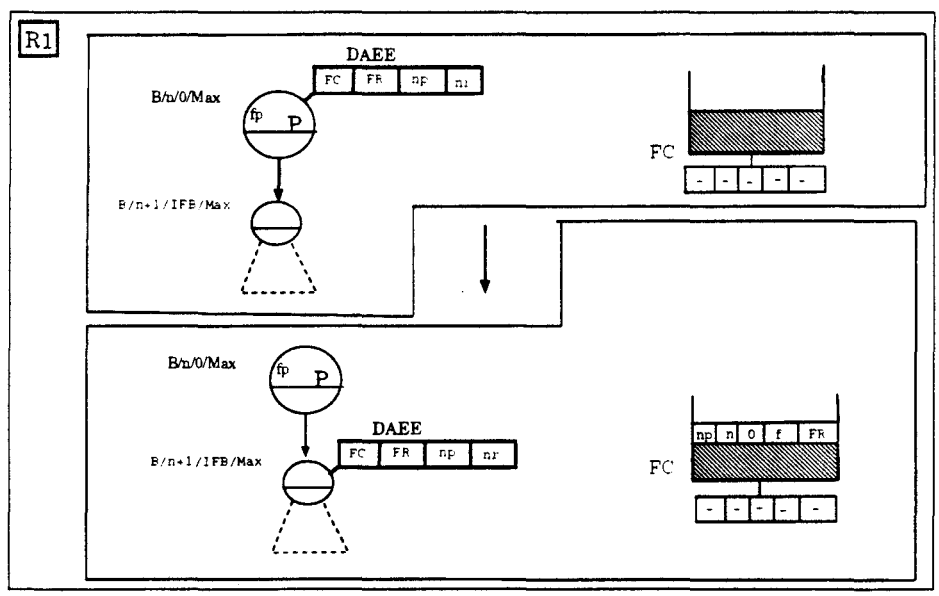


Figure 5.16 : Exemple 1 de règle d'exploration

Le contenu du DAEE construit dans cette règle est identique à celui traité par la règle. A noter également que la fenêtre cible est identique pour tous les nœuds du tronçon.

- **Exemple2** : (figure 5.17) Règle d'exploration d'un nœud définition dans le cas 1 (cf. section 5.6)

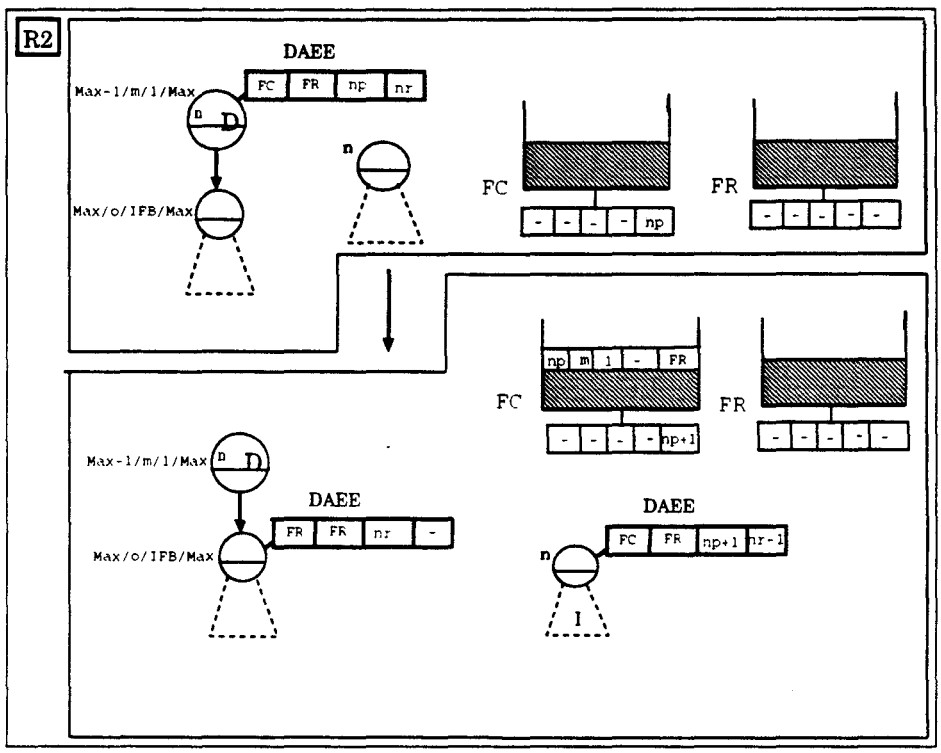


Figure 5.17 : Exemple 2 de règle d'exploration

Cette règle permet aussi bien l'exploration d'un nœud que la numérotation des tronçons dont l'exploration commence. Aucune fenêtre de réduction n'est créée. Le champ G de la fenêtre FC est mis à jour.

- **Exemple3** : (figure 5.18) Règle d'exploration d'un nœud définition dans le cas 2 (cf. section 5.6)

La création d'une fenêtre de réduction intermédiaire est la seule différence avec la règle précédente.

- **Exemple4** : (figure 5.19) Règle d'exploration d'un nœud frontière ayant pour valeur une fonction primitive non stricte (cas 3 cf. section 5.6)

Le tronçon B+1 dans cet exemple est le premier d'une suite décroissante. On notera que cette règle permet d'initialiser le champ P de la fenêtre FI. Ceci déclenche le début des réductions dans cette fenêtre (cf. section 8).

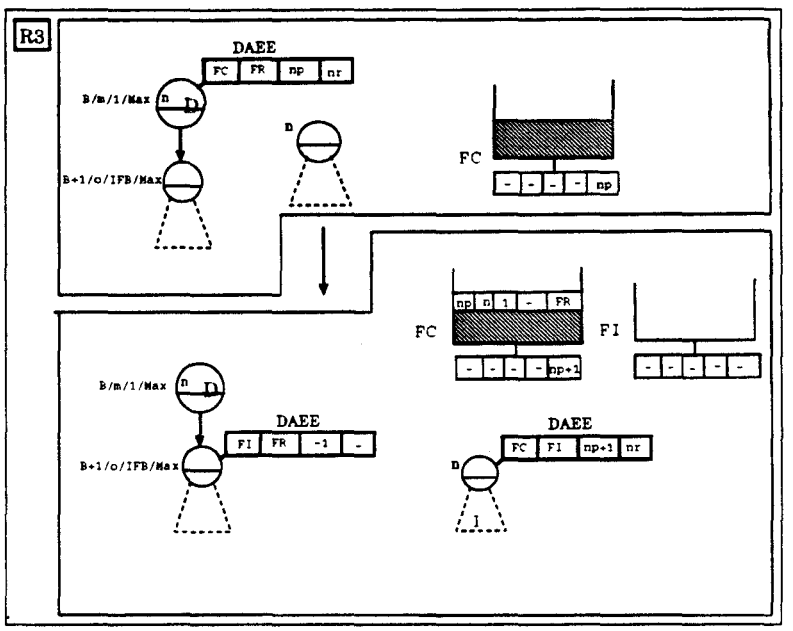


Figure 5.18 : Exemple 3 de règle d'exploration

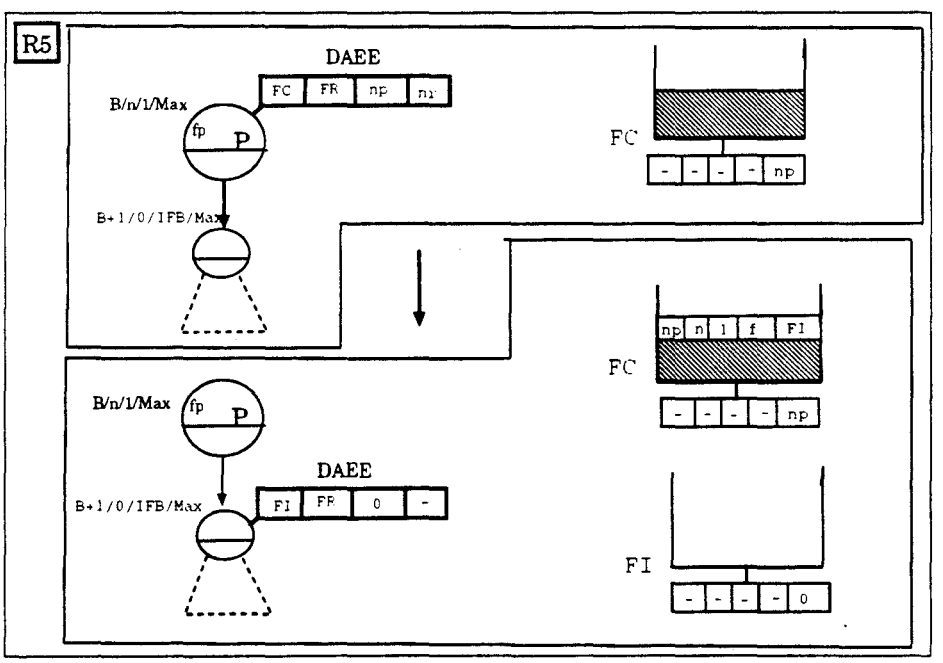


Figure 5.19 : Exemple 4 de règle d'exploration

- **Exemple5** : (figure 5.20) Règle d'exploration d'un nœud frontière ayant pour valeur une fonction primitive non stricte (cas 4 cf. section 5.6)

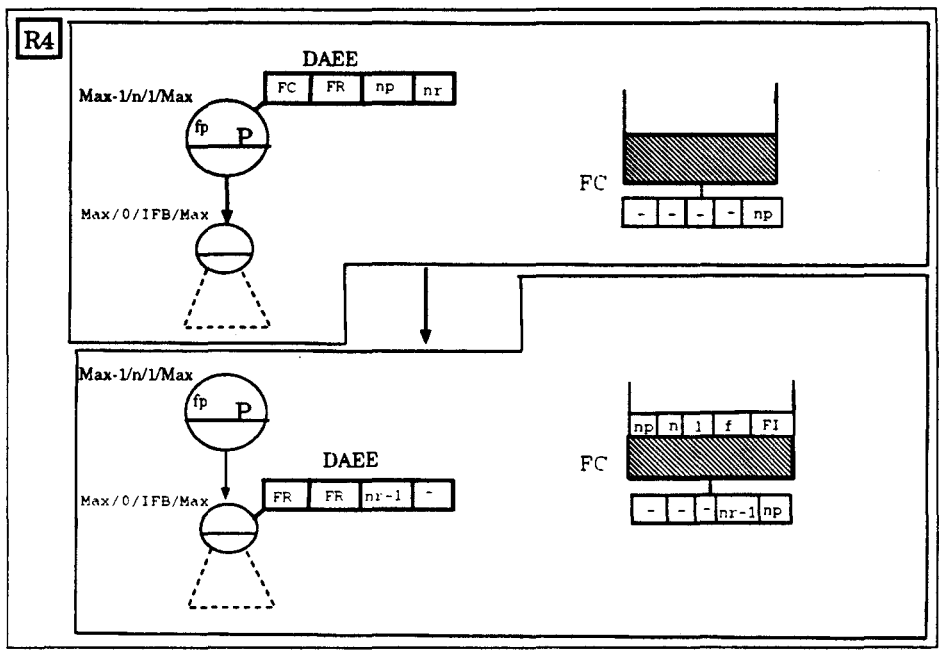


Figure 5.20 : Exemple 5 de règle d'exploration

Le tronçon B+1 étant un tronçon intermédiaire, une nouvelle fenêtre de réduction est alors créée.

8 Règles de transition

Notation :
Un DAR créé suite au traitement d'un DAE_{n_i} où n_i est le ième nœud d'un tronçon B ayant un numéro de parcours dynamique **p** est noté $(DAR^B_{n_i})_p$.

Il existe 3 règles de transition qui permettent la sélection d'un DAR dans la file d'attente d'une fenêtre de réduction. Quelle que soit la règle à appliquer, les conditions suivantes doivent être réunies.

8.1 Conditions générales d'application

Pour pouvoir appliquer une règle de transition, il faudrait réunir les conditions suivantes :

- La fenêtre de réduction n'est pas active i.e le dernier DAR introduit dans l'espace d'action a déjà été traité par une règle de réduction.
- La file d'attente associée n'est pas vide.

8.2 Conditions spécifiques

- Quand il s'agit de sélectionner le premier DAR qui doit être traité dans la fenêtre de réduction, la règle de transition 1 est applicable.
- La règle de transition 2 permet de sélectionner un nouveau tronçon quand le dernier DAR traité est issu du dernier nœud d'un tronçon.
- La règle de transition 3 permet de sélectionner le DAR suivant dans le même tronçon.

8.3 La règle 1 de transition

Cette règle est illustrée par la figure 5.21. La condition spécifique d'application de cette règle peut être formulée comme suit :

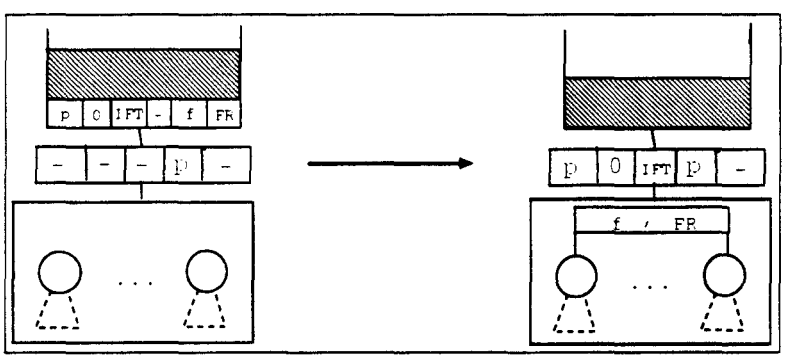


Figure 5.21 : Règle de transition 1

- Aucun DAR n'a jusque là transité de la file d'attente vers l'espace d'action (les champs NP, NORD et IFT ne sont pas initialisés).
- Les $(DAR^B_{n_i})_p$ créés par l'application des règles d'exploration aux $(DAE^B_{n_i})_p$ où :

  - p est le plus petit numéro de parcours dynamique associé au tronçon ayant pour fenêtre cible la fenêtre de réduction FR.
  - Le tronçon en question est le tronçon B.

sont déjà dans la file d'attente associée à la fenêtre de réduction.

Dans ce cas, le DAR choisi est : $(DAR^B_{n_0})_p$.

8.4 La règle 2 de transition

Cette règle est illustrée par la figure 5.22.

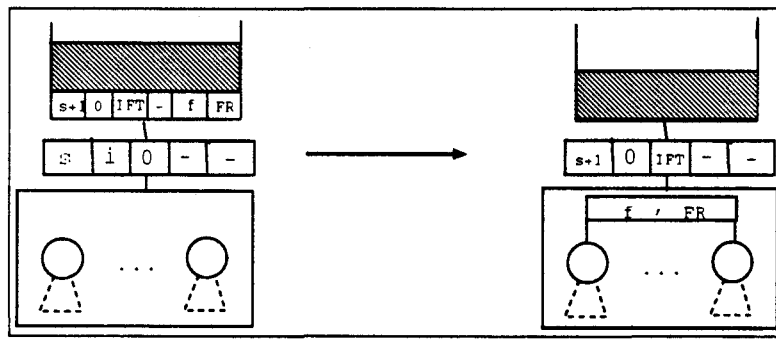


Figure 5.22 : Règle de transition 2

La condition spécifique d'application de cette règle se traduit comme suit :

- Le $(DAR^B_{n_i})_s$ est le dernier DAR choisi.
- Le nœud fonctionnel n_i est le dernier nœud fonctionnel du tronçon B.
- s n'est pas le plus grand numéro de parcours dynamique affecté (la valeur du champ G de la fenêtre de réduction est supérieur à s).

Dans ce cas le DAR choisi est : $(DAR^B_{n_0})_{s+1}$

8.5 La règle 3 de transition

Cette règle est illustrée par la figure 5.23.

La condition spécifique d'application de cette règle se traduit comme suit :

- $(DAR^B_{n_i})_m$ tel que le nœud fonctionnel n_i du tronçon B n'est pas le dernier nœud fonctionnel de B est le dernier DAR choisi.

Dans ce cas, le DAR choisi est : $(DAR^B_{n_{i+1}})_m$

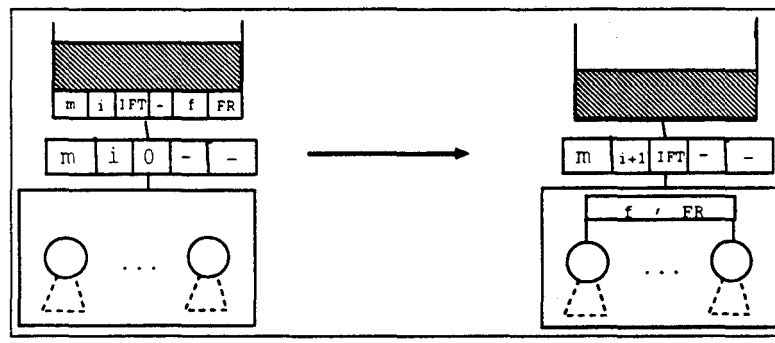


Figure 5.23 : Règle de transition 3

9 Conclusion

Nous avons proposé une optimisation du système de réécriture du modèle P^3 qui tend à minimiser les créations de nœuds résultats intermédiaires tout en permettant d'anticiper au maximum sur l'exploration des nœuds fonctionnels et sur la réduction. Cette optimisation a été rendue possible grâce à l'introduction de la notion de fenêtre cible et au regroupement des nœuds d'un même chemin en tronçons possédant la même cible. Cette optimisation permet également de conserver le parallélisme de l'exploration et celui de la réduction ce qui assure, de plus, l'indépendance de ces deux activités. Pour tenir compte de cette amélioration, le système de réécriture a subi des modifications : les règles d'exploration permettent de transmettre les fenêtres cible et résultat à chaque nœud de l'arborescence fonctionnelle au lieu des nœuds cibles et des nœuds résultats. Ces derniers sont complètement masqués par la présence des fenêtres de réduction. Les règles de transition permettent le choix du DAR dans la file d'attente d'une fenêtre de réduction. Les règles de réduction quant à elles ne subissent aucune modification.

Ceci termine les travaux d'extension du modèle P^3. Le chapitre suivant compare le modèle P^3 étendu au modèle de réduction de graphe.

Partie III

Evaluation du modèle P^3

Chapitre 6

Comparaison de P^3 au modèle de réduction de graphe

L'étude, dans le chapitre 3, des modèles d'évaluation existants a permis de situer le modèle P^3 tel qu'il a été défini par N. Devesa [Dev90] par rapport aux autres modèles de réduction. Cette étude a permis également de faire le point sur les limites du modèle P^3. L'extension du modèle P^3 , présentée aux chapitres 4 et 5 pallie quelques unes des limites de ce modèle notamment la nature des expressions traitées, l'utilisation des fonctions d'ordre supérieur et le parallélisme des langages fonctionnels exploité.
L'objectif de ce chapitre est d'établir une évaluation théorique du modèle P^3 étendu. Dans cette optique nous avons choisi de le comparer au modèle de réduction de graphe. Ce choix se justifie par le fait que le modèle P^3 s'apparente plus surtout au modèle de réduction de graphe (cf. chapitre 3). De plus, la comparaison au modèle de réduction de graphe est valorisante pour le modèle P^3 puisque la puissance et l'efficacité du modèle de réduction de graphe sont reconnues.
Cette comparaison portera sur deux aspects différents : le premier aspect concerne le coût de gestion des deux modèles. Le deuxième aspect concerne la puissance de ces deux modèles i.e le type d'expression traité, le parallélisme exploité, les modes d'évaluation pouvant être utilisés et le partage de données.
Ce chapitre est donc structuré de la manière suivante. La section 1 permet d'établir le coût d'évaluation du modèle de réduction de graphe. La section 2 en fait autant pour le modèle P^3. La section 3 établit une étude des coûts présentés

dans les sections 1 et 2. La section 4 compare les deux modèles en se basant sur les possibilités offertes par chacun.

1 Le coût d'évaluation du modèle de réduction de graphe

L'évaluation d'une expression fonctionnelle dans le modèle de réduction de graphe occasionne plusieurs coûts que nous détaillerons dans la suite de cette section. Auparavant, nous rappelons les principes d'évaluation du modèle de réduction de graphe.

1.1 Analyse du fonctionnement du modèle

Une expression est représentée dans le modèle de réduction de graphe par un graphe dont les nœuds intermédiaires sont des nœuds application. Chaque graphe, de racine un noeud application symbolise un redex. L'évaluation consiste donc à réécrire tous les sous graphes ayant pour racine un nœud application. L'ordre de réécriture doit simplement respecter la dépendance entre les redex et privilégier les redex qui peuvent être traités indépendamment des autres redex (afin de permettre l'anticipation). Cette contrainte impose une synchronisation naturelle entre les différents redex. De par la représentation du graphe, les redex dont l'évaluation ne dépend d'aucun autre redex se retrouvent en profondeur dans le graphe, et de ce fait, ils sont atteints grâce à un parcours en profondeur du graphe. Chaque sous graphe réécrit permet aux redex qui le référencent d'être à leur tour traités. A la fin, le graphe obtenu est un graphe irréductible qui représente le résultat de l'expression évaluée. Nous pouvons distinguer deux types de redex : les redex où la fonction appliquée est une fonction primitive. La prise en compte de ce type de redex consiste simplement à appliquer la fonction à ses arguments, en respectant sa sémantique. Le graphe résultat équivalent, est forcément réduit. Le deuxième type de redex possède une fonction appliquée plus complexe. Ce type de redex, une fois pris en compte ne donne pas forcément un résultat irréductible : en effet, l'application d'une règle de réécriture dans ce cas, permet de réorganiser le graphe en introduisant d'autres redex et donc d'autres nœuds application. Nous illustrons ce type de redex par celui de la figure 6.1 :

Dans cet exemple, la réécriture du seul redex présent sur le graphe de gauche, entraîne l'introduction, en cours d'évaluation, de n nouveaux redex (à droite sur la figure). Les nœuds application 1, 2, ..., n sur la figure sont créés et le graphe est organisé de façon à retrouver une structure du graphe analogue à celle construite

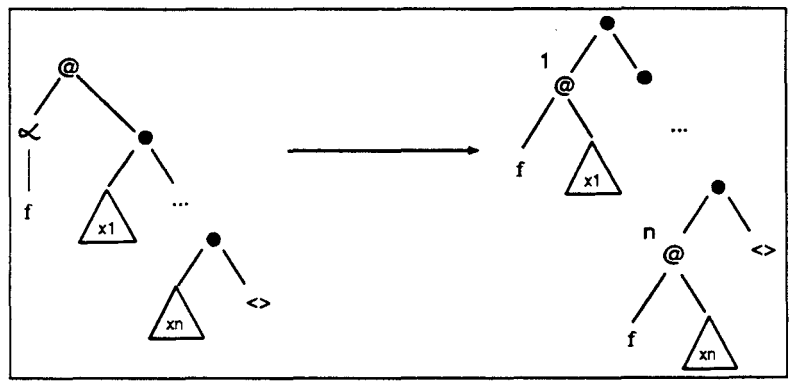


Figure 6.1 : Exemple de redex impliquant une fonction complexe

au départ i.e où les redex sont tous représentés.

Remarque

En réalité, la fonction f n'est pas dupliquée n fois : elle est partagée par les n sous graphes de racines les nœuds application 1, 2,..., n. Nous reviendrons sur le partage de sous graphes dans la section 4.4 de ce chapitre.

1.2 Analyse des coûts

L'évaluation d'une expression dans le modèle de réduction de graphe engendre donc les coûts suivants :

1. **Un coût de stockage des nœuds application**
   On retrouve ce coût à la représentation initiale de l'expression puisque dans le modèle de réduction de graphe, les nœuds application symbolisent les redex qui doivent être traités et donc font partie intégrante de la représentation de l'expression. D'autres redex sont construits dynamiquement, lors du traitement d'un redex impliquant une fonction complexe (cf. figure 6.1). Systématiquement, chaque redex présent dans la représentation implique la présence d'un nœud application. De plus, la représentation décurryfiée des fonctions polyadiques dans le modèle de réduction de graphe introduit un nombre de nœuds application, égal au nombre d'arguments de chaque fonction.

2. **Un coût de parcours du graphe** à la recherche de redex prêts à être traités.
   Ce coût est imposé par la structure même du graphe : les redex réécrits

en premier sont ceux qui se trouvent en profondeur.Ce coût existe pour le graphe initial, mais également, pour chaque nouveau sous graphe réductible, construit dynamiquement.

3. **Un coût de réorganisation du graphe** généré par la prise en compte de redex où la fonction appliquée n'est pas une fonction primitive. Ce coût ne comprend pas le coût de création des nœuds application que nous avons isolé au préalable. Il comprend uniquement le coût de construction du nouveau graphe à partir de l'ancien.

4. **Un coût de préparation des données** incluant la duplication au sens large des arguments i.e l'activation du partage ou de la copie.

5. **Un coût d'application d'une fonction primitive** désignant le coût de calcul engendré par l'application d'une fonction à ses arguments.

6. **Un coût de synchronisation des réductions** s'opérant sur le même graphe.

2 Le coût d'évaluation du modèle P^3

Le modèle P^3 engendre aussi un coût de gestion lors de l'évaluation d'une expression. Nous allons tenter de la même manière que pour le modèle de réduction de graphe de détailler le coût d'évaluation du modèle P^3, afin d'établir une comparaison des coûts des deux modèles mais auparavant analysons le fonctionnement du modèle P^3 en se plaçant au même niveau que pour le modèle de réduction de graphe.

2.1 Analyse du fonctionnement du modèle P^3

Une expression fonctionnelle est représentée dans le modèle P^3 par une forêt d'arborescences fonctionnelles et un ensemble d'arbres de réduction. Au stade initial de la représentation, seuls les redex symbolisant les applications (cf. chapitre 4) sont présents. Ces redex sont symbolisés par les drapeaux d'activation de l'exploration, présents sur les racines des représentations principales des fonctions appliquées. Grâce au mécanisme d'exploration, les redex sont constitués au fur et à mesure du fait de la propagation de l'activation des nœuds, au cours de l'évaluation. Un redex est construit dynamiquement lors de l'apparition d'un DAE (ou DAEE) sur un nœud fonctionnel. La transformation du DAE

(resp. DAEE) en DAR (resp. DARE) par l'application d'une règle d'exploration peut être considérée comme une "mutation" du redex qui est "affecté" donc à l'ensemble de ses arguments. Le nombre de redex créés à un instant donné est égal au nombre de drapeaux existant dans la représentation à cet instant i.e qui ne sont pas encore traités. Dans le modèle P^3, Il n'y a donc pas besoin de parcours préalable, les fonctions étant placées dans l'arborescence de façon à ce que les premiers redex constitués soient ceux qui peuvent logiquement s'exécuter sans attente d'un quelconque événement. D'autre part, de la même manière que dans le modèle de réduction de graphe, nous distinguons deux types de redex. Le premier type de redex, une fois traité entraîne l'application de la fonction à ses arguments. Le deuxième type de redex est celui liant un nœud forme fonctionnelle à ses arguments. En effet, il suffit d'examiner les règles d'exploration où le nœud fonctionnel concerné est un nœud forme fonctionnelle (cf. chapitre 4) pour remarquer que ce type de redex n'engendre pas de calcul mais plutôt une préparation des arguments des fonctions composant chacune des sous arborescences fonctionnelles paramètres. Cette préparation est comparable sémantiquement à la phase de préparation dans le modèle de réduction de graphe, à la seule différence qu'elle ne nécessite aucune réorganisation de graphe. Par ailleurs, chaque drapeau une fois traité est remplacé par le résultat du redex. Les redex sont traités dans un ordre précis grâce à un algorithme d'ordonnancement décrit dans le chapitre 5.

2.2 Analyse des coûts engendrés

Analysons les coûts occasionnés par l'exécution d'une expression dans le modèle P^3. Nous distinguons les coûts suivants :

1. **Le coût d'exploration des arborescences fonctionnelles.**

2. **Le coût de stockage des drapeaux** . En effet, l'exploration des arborescences fonctionnelles comportant moins de synchronisation que la version originelle du modèle P^3, la vitesse de production des redex est plus élevée. Pour peu que la vitesse de prise en compte des redex soit inférieure à cette dernière, il y a une accumulation de drapeaux dans les files d'attente des fenêtres de réduction et donc le nombre de redex non traités augmente engendrant le coût de stockage.

3. **Le coût d'ordonnancement des redex** dans les files d'attente des fenêtres de réduction pour retrouver l'ordre cohérent de traitement des redex.

4. **Le coût de préparation des arguments** dans le cas de traitement des redex impliquant une forme fonctionnelle.

5. **Le coût d'application d'une fonction** à ses arguments dans le cas de traitement de redex impliquant une fonction primitive.

3 Etude des coûts d'évaluation

Le tableau suivant présente un récapitulatif des coûts de gestion des deux modèles :

Le modèle de réduction de graphe	*Le modèle P^3*
1) Coût de stockage des nœuds application	1) Coût d'exploration des arborescences fonctionnelles
2) Coût de parcours du graphe	2) Coût de stockage des drapeaux
3) Coût de réorganisation du graphe	3) Coût d'ordonnancement
4) Coût de préparation des arguments	4) Coût de préparation des arguments
5) Coût d'application d'une fonction	5) Coût d'application d'une fonction

La numérotation des coûts aussi bien pour le modèle P^3 que pour le modèle de réduction de graphe reflète l'ordre chronologique de génération des coûts. Dans la suite de cette section, nous tentons de comparer les coûts occasionnés par les deux modèles.

Avant d'étudier les coûts engendrés par les deux modèles, nous pouvons faire la classification suivante : dans l'ensemble des coûts engendrés, nous pouvons distinguer les coûts qui proviennent de la sémantique d'application des fonctions. Ces coûts existent aussi bien dans le modèle de réduction de graphe que dans le modèle P^3. Il s'agit des coûts de préparation, de création de redex et d'application des fonctions. Ces coûts sont nécessairement identiques dans les deux modèles.
Le deuxième type de coût est lié à la représentation de l'expression choisie et/ou aux mécanismes d'évaluation de chaque modèle. Cette deuxième famille de coûts sera étudiée plus en détail.
Pour le modèle P^3, les coûts propres au modèle et qui sont : le coût d'exploration des arborescences fonctionnelle, le coût d'ordonnancement et le coût de stockage des redex caractérisent plus les mécanismes de réduction que la représentation choisie. Pour le modèle de réduction de graphe, les coûts propres au modèle sont plus fonction de la représentation choisie.

Dans la suite de cette section, nous effectuons la comparaison des coûts suivants :

- le coût de parcours des arborescences fonctionnelles et celui des graphes syntaxiques. Cette comparaison se justifie par la nature même du coût : il s'agit d'un parcours de graphes dans les deux cas, mais également parce que ce coût met en évidence l'avantage de la représentation de l'expression de P^3 par rapport au modèle de réduction de graphe.

- Le coût de stockage des nœuds appliaction et celui des drapeaux. Ces deux entités représentent tout deux les redex à traiter dans les deux modèles. Cette comparaison met en avant l'intérêt de lasynchronisme des deux activités de P^3 par rapport à l'évaluation dans le modèle de réduction de graphe.

Puis nous abordons les coûts existants dans un des deux modèles uniquement.

- **Coût de parcours du graphe syntaxique vs. coût d'exploration des arborescences fonctionnelles**

  Nous établissons une étude de ces coûts en se basant d'abord sur le nombre de nœuds parcourus puis sur les conséquences de ces parcours sur l'évaluation.

  Ces parcours s'effectuent du haut vers le bas aussi bien pour les arborescences fonctionnelles que pour les graphes syntaxiques. Le parcours du graphe syntaxique dans le modèle de réduction de graphe concerne les nœuds application. Dans le modèle P^3, il concerne les nœuds fonctionnels. Comparons maintenant ces parcours en termes de nœuds parcourus. Le graphe syntaxique parcouru est obtenu par combinaison de sous graphes de deux types représentés respectivement par la figure 6.2.a et 6.2.b.

  Le premier représente une composition de n fonctions $f_1 \circ \ldots f_n$ - où chaque f_i est une fonction simple- appliquée à un argument x. Le deuxième représente l'application d'une fonction f à n arguments $a_1,a_2,\ldots,a_n$. Dans le premier cas, le nombre de nœuds parcourus est égal à n. La même expression sera représentée dans le modèle P^3 comme l'illustre la figure 6.3.

  Le parcours des nœuds fonctionnels racines des représentations respectives des fonctions f_i s'effectue également en n étapes. Donc pour ce type d'expression, les parcours en termes de nœuds parcourus sont similaires.
  Le coût unitaire de parcours est égal à p_u, le coût de parcours d'un graphe de ce type est de

  $n.p_u$

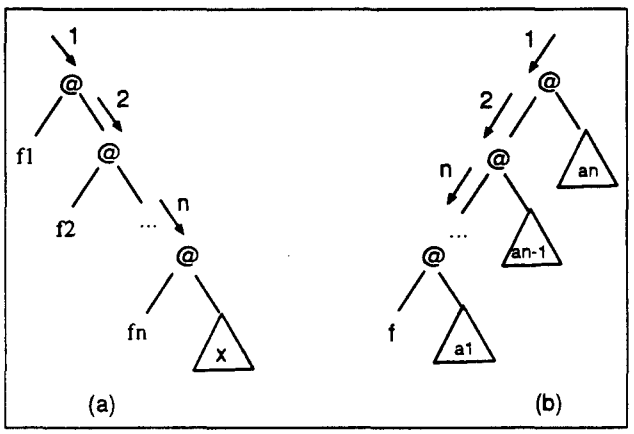


Figure 6.2 :

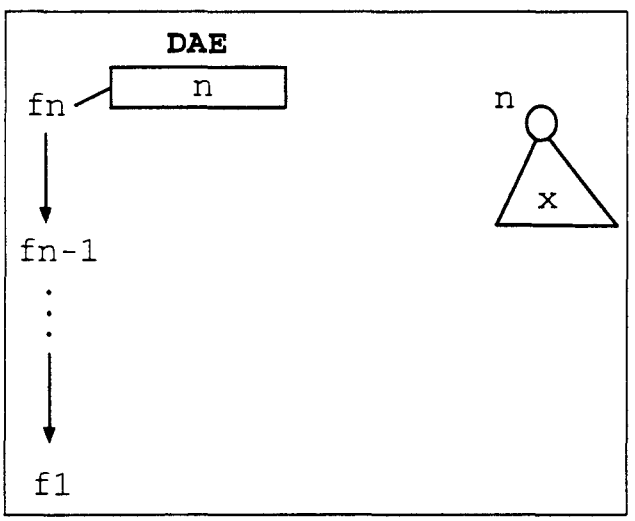


Figure 6.3 :

Considérons maintenant le deuxième cas (figure 6.2.b). Ce graphe serait représenté de la manière suivante dans le modèle P^3 (figure 6.4) :

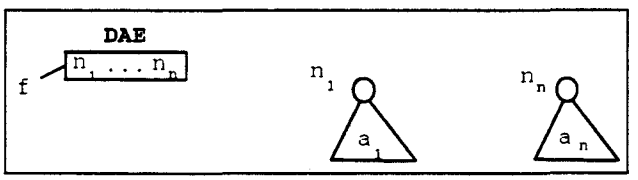


Figure 6.4 :

Pour ce deuxième cas de figure, le nombre de nœuds parcourus dans le modèle de réduction de graphe pour atteindre la fonction f est égal à n. Dans le modèle P^3, du fait que la polyadicité est gérée sans avoir recours à la curryfication des fonctions, un seul nœud est parcouru pour atteindre la fonction f.
Le graphe syntaxique est une combinaison de ces deux types de graphes.

Pour déterminer le coût de parcours dans chacun des deux modèles, nous allons tenter de formaliser le problème. Nous supposons que l'expression traitée contient c compositions de fonctions. En effet, la fonction appliquée est une composition de fonctions dont certaines sont des formes fonctionnelles admettant pour paramètres un ensemble de composition de fonctions. Nous supposons dans un premier temps que chaque composition de fonctions ne contient que des fonctions monadiques. Soit n_f, le nombre moyen de fonctions dans chaque composition de fonctions. Dans le modèle de réduction de graphe, ces compositions de fonctions sont toutes représentées par un graphe ayant la forme illustrée par la figure 6.2.a. Dans P^3, chacune de ces composition est représentée par un chemin séquentiel de longueur moyenne n_f. Le coût de parcours pour les deux modèles est :

$$c.n_f.p_u$$

Si maintenant, nous autorisons l'existence de fonctions polyadiques dans une compositions de fonctions. Le nombre d'arguments en moyenne n'est plus égal à 1. Soit n_{arg} ce nombre moyen, le coût de parcours dans le modèle de réduction de graphe est donc égal à :

$$c.(n_f.n_{arg}).p_u$$

Dans le modèle P^3, l'existence des fonctions polyadiques ne change rien au coût de parcours.

Si de plus, nous prenons en compte le coût de parcours des sous graphes construits dynamiquement, engendré par la réorganisation et qui est égal à $R.n_R.p_u$ (cf. section 3), nous obtenons donc, dans le modèle de réduction de graphe, un coût de parcours général égal à :

$$(c.n_f.n_{arg} + R.n_R).p_u$$

Dans le modèle P^3, le coût de parcours demeure inchangé; il est égal à :

$$c.n_f.p_u$$

puisqu'il n'y a pas de phase de réorganisation dans P^3.
Globalement, le coût de parcours est donc plus important dans le modèle de réduction de graphe.

Une autre manière d'envisager une comparaison de ces deux parcours concerne l'impact de chacun sur l'évaluation d'une expression; Dans le modèle de réduction de graphe, les redex pouvant être traités en premier étant en profondeur dans le graphe, les réductions commencent après un délai qui correspond au temps de parcours du graphe. En effet, aucune réduction ne sera possible avant d'atteindre les redex prioritaires. Dans le modèle P^3, ce parcours n'est pas aussi pénalisant du fait de la création des redex les plus prioritaires à proximité de la racine des arborescences fonctionnelles. Ainsi, les premiers redex construits pourront être traités simultanément avec l'avancée de l'exploration. Il en résulte que le parcours de l'arborescence fonctionnelle est entièrement recouvert par les réductions et ne pénalise pas le déroulement de l'évaluation. Concrètement, ceci permet un gain de temps au niveau de l'évaluation de l'expression. Ce gain est exprimé de la manière suivante :
Pour simplifier, prenons un graphe ayant la forme illustrée par la figure 6.2.a contenant initialement n redex. Soit p_u le temps unitaire d'accès à un nœud durant le parcours et t_u le temps moyen de prise en compte d'un redex. Dans le modèle de réduction de graphe, l'expression représentée par un tel graphe sera évaluée en un temps égal à

$$c_f.p_u + n.t_u$$

en posant

$$c_f = k.c.n_f$$

k étant une constante qui permet de convertir le coût de parcours en temps de parcours. En effet, le traitement des redex contenus dans un sous graphe ayant la forme illustrée par la figure 6.2.a commence après la fin du parcours du graphe. Dans le modèle P^3, le temps de parcours de l'arborescence fonctionnelle correspondante est identique i.e égal à

$$c_f.p_u$$

Par contre, à la différence du modèle de réduction de graphe, ce parcours est entièrement recouvert par le traitement des redex. De ce fait le temps d'évaluation de la même expression est plus court dans le modèle P^3. En effet, soit n' le nombre de redex qui a pu être traité durant le parcours de l'arborescence fonctionnelle qui a permis de générer les n redex. Le temps nécessaire à l'évaluation de l'expression, dans le modèle P^3, est égal à

$$c_f.p_u + (n - n')t_u$$

Dans le cas d'expressions comportant des fonctions polyadiques et engendrant des transformations du graphe, en posant

$$C_f = k.(c.n_f.n_{arg} + R.n_R)$$

, le temps de traitement incluant le parcours du graphe dans le modèle de réduction de graphe, est de

$$C_f.p_u + N.t_u$$

Dans le modèle P^3, le temps de traitement incluant le parcours est de

$$c_f.p_u + (N - N').t_u$$

Par conséquent, le temps de traitement d'une expression incluant le temps de parcours est plus important dans le modèle de réduction de graphe.

- **Coût de stockage des nœuds application vs. coût de stockage des drapeaux**

Les nœuds application dans le modèle de réduction de graphe symbolisent l'existence de redex dans le graphe. Il en est de même pour les drapeaux dans le modèle P^3. La présence de ces nœuds (resp. ces drapeaux) représente un coût de stockage que l'on retrouve aussi bien dans le modèle

de réduction de graphe que dans le modèle P^3. Comparons donc ces coûts dans les deux modèles. Pour illustrer la différence entre ces deux coûts, supposons, pour simplifier, qu'on ait à évaluer une expression de la forme $F : x$ où F est une composition de n fonctions monadiques. La représentation initiale dans le modèle de réduction de graphe comprend donc n redex. Par conséquent, le nombre de nœuds application est égal à n. Compte tenu de ces hypothèses, le graphe de réduction a la forme illustrée par la figure 6.2.a. Par conséquent, tous les redex sont traités en séquentiel.

Initialement, le nombre de nœuds application présents est égal à n dans le modèle de réduction de graphe. Dans le modèle P^3, pour cette même expression, un seul drapeau représente le seul redex présent initialement dans la représentation de l'expression. Donc, initialement, le nombre de nœuds application présents dans la représentation de l'expression dans le modèle de réduction de graphe est plus important que le nombre de drapeaux dans P^3.

Essayons maintenant de comparer l'évolution du nombre de redex présents dans la représentation de l'expression tout au long de l'évaluation dans les deux modèles. Dans le modèle de réduction de graphe, le nombre de redex reste stable un instant, le temps du parcours du graphe puis il diminue par la suite au fur et à mesure que les redex sont traités. Le coût de stockage devient nul à la fin de l'évaluation.
Dans le modèle P^3, la "production" des redex par le mécanisme d'exploration et le traitement des redex sont des activités complètement asynchrones. A vitesses égales de traitement des redex et de parcours dans les deux modèles, pour une même expression respectant les hypothèses émises plus haut, le nombre de nœuds application dans le modèle de réduction de graphe est globalement supérieur au nombre de drapeaux présents dans le modèle P^3. Pour illustrer la quantité de drapeaux (resp. de noeuds application) présents simultanément tout au long de l'évaluation, nous visualisons les courbes de production et de consommation des redex tout au long de l'évaluation pour les deux modèles (figure 6.5). Ces courbes ont été tracées de manière empirique i.e elles ne se basent sur aucune étude quantitative exacte et permettent uniquement de comprendre facilement la comparaison faite de ces coûts. Sur cette figure, le coût de stockage des drapeaux et des nœuds application est représenté chacun à l'aide d'une surface. Chaque surface est délimitée par deux courbes : la courbe de production des nœuds application (resp. des drapeaux) et la courbe de traitement des redex symbolisés par les nœuds application (resp. des drapeaux). En effet chaque nœud application (resp. drapeau), une fois traité disparait.

Supposons que p représente le temps nécessaire pour le parcours du graphe

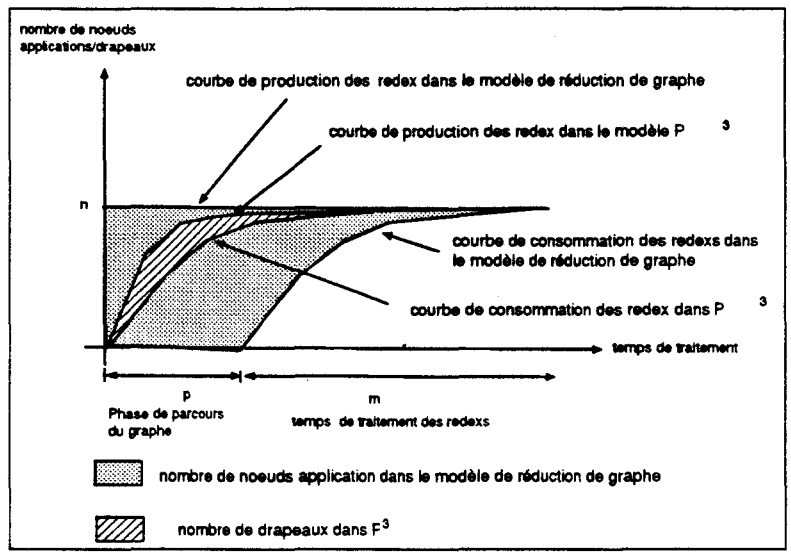


Figure 6.5 : Evolution du nombre de redex dans le modèle P^3 et dans le modèle de réduction de graphe dans un cas simple

syntaxique afin d'atteindre le premier redex réductible. En tenant compte des hypothèses sur l'expression traitée, formulées ci-haut, la courbe de production des redex est une constante n, ce qui explique l'aspect horizontal de la courbe. La courbe de consommation est nulle durant le temps du parcours du graphe p. Pendant m unités de temps, temps nécessaire au traitement de tous les redex, la courbe de consommation des redex croît et rencontre finalement la courbe de production des redex.

Pour le modèle P^3, la courbe de production des redex croît et se stabilise à n après le temps p, temps nécessaire pour la production des redex suite à l'exploration de l'arborescence fonctionnelle. Contrairement au modèle de réduction de graphe la courbe de consommation pourra croître immédiatement et mettra un temps m pour atteindre la valeur n i.e le maximum. Le temps m de consommation des redex est identique dans les deux modèles puisque nous avons supposé égales les vitesses de traitement. Le nombre de drapeaux présents globalement tout au long de l'évaluation dans le modèle P^3, est inférieur à celui des nœuds application présents globalement en cours d'évaluation dans le modèle de réduction de graphe.

Ces coûts pour les deux modèles peuvent être exprimés de la manière suivante : Soit $f_p(t)$ la fonction de production des drapeaux dans le modèle P^3. Soit $f_c(t)$ la fonction de traitement des redex au cours du temps. Le

coût de stockage des drapeaux durant l'évaluation est égale à :

$$\int_0^m f_p(t) - f_c(t)\,dt$$

Soit p_{gr}(t) la fonction de production des redex dans le modèle de réduction de graphe. p_{gr}(t)=n quel que soit t. n étant le nombre de redex initialement présents dans le graphe. Soit c_{gr}(t) la fonction de traitement des redex dans le modèle de réduction de graphe. Nous avons l'égalité suivante :

$$\forall t, c_{gr}(t) = f_c(t-p)$$

Le coût de stockage des drapeaux durant l'évaluation est égale à :

$$n(p+m) - \int_p^{m+p} c_{gr}(t)\,dt$$

qui est égal à

$$n(p+m) - \int_0^m f_c(t)\,dt$$

après simplification.

$\int_0^m f_p(t)\,dt$ est inférieure à l'aire $n(p+m)$ puiqu'elle s'inscrit dans cette dernière. Par conséquent, le coût de stockage des nœuds application est plus important que celui des drapeaux dans le modèle P^3, pour le cas spécifié par l'hypothèse.

Si la fonction F contient des fonctions complexes, le graphe subit alors des réorganisations durant l'évaluation. Par conséquent, les redex présents au départ ne constituent pas la totalité : il y en aura d'autres créés par la réorganisation du graphe. Soit n le nombre final de nœuds application créés. Toujours à vitesse égale de traitement, les courbes de traitement des redex sont identiques dans les deux modèles avec un décalage p_1, p_1 est le temps à partir duquel, le premier redex dans le modèle de réduction de graphe pourra être traité. La courbe de traitement des redex $c'_{gr}(t)$, dans le modèle de réduction de graphe est définie de la façon suivante :

$$c'_{gr}(t) = 0 \quad si \quad t \quad < \quad p_1$$

$$c'_{gr}(t) = f'_c(t-p_1) \quad sinon$$

Dans le modèle P^3, nous supposons qu'il faut un temps p_2 pour que l'exploration génère la totalité des redex.
Pour démontrer l'importance du coût de stockage des redex dans le modèle de réduction de graphe par rapport au modèle P^3, il suffit de démontrer que :

$$\int_{-\infty}^{+\infty} f'_c(t) - c'_{gr}(t)\,dt > 0 \tag{6.1}$$

et

$$\int_{-\infty}^{+\infty} p'_{gr}(t) - f'_p(t)\,dt > 0 \tag{6.2}$$

L'inégalité (1) est vrai puisqu'à chaque instant de l'évaluation, le nombre de redex traités dans le modèle P^3 est supérieur au nombre de redex traités dans le modèle de réduction de graphe, du fait que l'exploration et la réduction sont asynchrones.
L'inégalité (2) est vraie. en effet, au départ, le nombre de redex dans le modèle de réduction de graphe est plus important (construction du graphe initial). De plus, à chaque réorganisation du graphe, un ensemble de redex s'ajoute au nombre initial. Les mêmes redex seront construits au fur et à mesure dans le modèle P^3.

Il en résulte que le nombre de noeuds application stockés dans le modèle de réduction de graphe est plus important que dans le modèle P^3, entraînant ainsi une gestion mémoire plus coûteuse en espace et en récupération. Cette différence de coût est illustrée de la même façon que dans le premier cas, par les courbes de production et de consommation des redex dans les deux modèles (figure 6.6) permettant simplement d'illustrer ce raisonnement.

- **Coût de réorganisation**

La réorganisation du graphe est une transformation du "squelette" du graphe. Durant cette phase, de nouveaux nœuds application sont créés et replacés dans le nouveau graphe. Le graphe ainsi construit pourra de nouveau être parcouru pour traiter les nouveaux redex.
Notre interprétation de ce coût est que le contrôle dans le modèle de réduction de graphe est assuré par le graphe lui-même. En effet, le graphe doit représenter à tout instant l'expression en cours d'évaluation. Ces transformations sont donc nécessaires pour restructurer la forme du graphe et réintroduire les nœuds application qui symbolisent les traitements à effectuer sur le graphe. Ce qui justifie d'ailleurs le fait qu'initialement pratiquement,

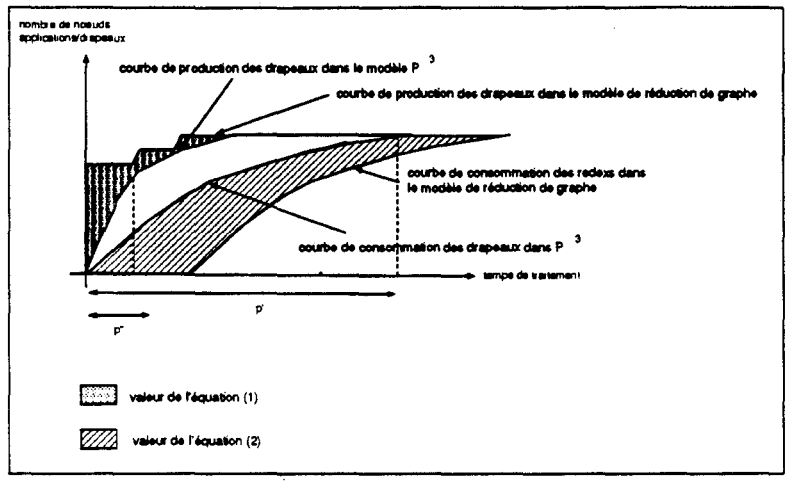


Figure 6.6 : Evolution du nombre de redex dans le modèle P^3 et dans le modèle de réduction de graphe dans le cas général

tous les redex prévisibles au moment de la représentation de l'expression figurent dans le graphe initial.
Dans le modèle P^3, cet état global de l'expression existe implicitement. Il est géré par les mécanismes d'exploration et d'ordonnancement qui assurent à eux deux l'obtention du résultat et la cohérence du traitement des redex. En effet, le contenu des drapeaux affectés à chaque nœud exploré suffit à établir le lien entre les différents redex. Le coût de réorganisation, dans le modèle de réduction de graphe, est donc égal à

$$R.C_{org}$$

où R est le nombre de transformations effectuées en cours d'évaluation et C_{org} est le coût unitaire moyen de chaque transformation.

De plus, la réorganisation du graphe entraîne indirectement d'autres coûts. Les coûts générés sont :

- un coût supplémentaire de création de nœuds application qui symbolisent les redex du nouveau graphe. Ce coût est proportionnel au nombre de réorganisations effectuées. Soit n_R, le nombre de nœuds application moyen créés à chaque réorganisation, le coût supplémentaire de création est de $n_R.R$. - un coût de parcours de ce même graphe pour atteindre les redex prêts à être traités, sur le même modèle que le parcours initial à partir de la

racine du sous graphe. Ce coût est proportionnel au nombre de réorganisations effectuées. Il est donc égal à

$$n_R.R.p_u$$

où p_u est le temps unitaire d'accès à un nœud.

- **Le coût d'ordonnancement des drapeaux vs. coût de synchronisation des réductions** Dans le modèle P^3, la création et le traitement des redex, étant deux activités complètement asynchrones, un algorithme d'ordonnancement permet de retrouver l'ordre de création des drapeaux. Cet algorithme s'active pour chaque redex, il est donc proportionnel au nombre de redex à traiter. Dans le modèle de réduction de graphe, la synchronisation des réductions au sein joue le même rôle.

En conclusion à cette étude des coûts, nous pouvons classer les coûts générés par les deux modèles en 3 classes :

- d'abord les coûts identiques pour les deux modèles : dans cette catégorie de coûts, nous retrouvons les coûts de traitement des redex et les coûts de préparation des données et le coût de création des nœuds application. Ces coûts sont liés à l'expression évaluée et non au modèle mis en oeuvre ce qui explique le fait qu'ils soient identiques dans les deux modèles.

- Les coûts qui sont supérieurs dans le modèle de réduction de graphe : dans cette catégorie, nous retrouvons le coût de parcours et le coût de stockage des noeuds application (resp. des drapeaux) représentant les redex. L'importance de ces coûts dans le modèle de réduction de graphe, est étroitement liée à la représentation de l'expression, qui occasionne un stockage plus important des nœuds application du fait de leur présence dans la représentation initiale, et un coût de parcours pour atteindre les fonctions applicables en premier.

- Les coûts propres à chacun des deux modèles. Le coût de réorganisation que l'on trouve dans le modèle de réduction de graphe uniquement pour retrouver une forme réductible du graphe syntaxique. Le coût d'ordonnancement des redex qui est proportionnel au nombre de redex traités. Il correspond au coût de synchronisation des réductions dans le modèle de réduction de graphe.

Après avoir comparé les coûts de gestion du modèle de réduction de graphe à ceux du modèle P^3, il convient également de comparer leurs puissances respectives.

Nous entendons par puissance d'un modèle, l'ensemble des possibilités offertes par ce modèle. Les aspects suivants : le type d'expressions traité, le parallélisme exploité, les modes d'évaluation utilisés et le partage. Dans le chapitre 3, ces aspects ont déjà été abordés concernant le modèle de réduction de graphe et la première version du modèle P^3. L'intérêt de cette section est de revoir certains de ces aspects qui étaient considérés comme des limitations de la première version du modèle P^3 auxquels, nous avons tenté d'apporter une réponse grâce à l'extension du modèle.

4 Comparaison des puissances des deux modèles

4.1 Les expressions traitées

Les expressions traitées aussi bien dans le modèle P^3, à la suite de son extension, que dans le modèle de réduction de graphe, sont de la forme :

$$Exp_f \ : \ Exp_{a_1} \ Exp_{a_2} \ \ldots \ Exp_{a_n}$$

où Exp_f est soit une composition de fonctions soit une expression quelconque et où pour chaque i, Exp_{a_i} est soit une composition de fonctions, une donnée ou une expression quelconque. De plus, pour chaque composition de fonction de la forme $f_1 \circ \ldots \circ f_n$, chaque fonction f_i peut également être une expression quelconque. Dans le modèle P^3 étendu, ceci a été amplement détaillé dans le chapitre 4.
Dans le modèle de réduction de graphe, ces expressions seront traités sur le modèle décrit antérieurement. Nous en concluons que le même type d'expression sans variables peut être traité aussi bien dans le modèle P^3 que dans le modèle de réduction de graphe.

4.2 Le parallélisme exploité

Le parallélisme des langages fonctionnels est exploité dans le modèle de réduction de graphe. Nous en avons apporté la preuve au chapitre 3. Par conséquent, nous n'y reviendrons pas dans ce chapitre. Par ailleurs, il est intéressant de montrer que le parallélisme horizontal est conservé dans la version étendue du modèle P^3 et que le parallélisme vertical y est exploité contrairement au modèle P^3 originel. L'exploitation du parallélisme horizontal dans le modèle P^3 originel est réalisée grâce à l'exploration parallèle des sous arborescences fonctionnelles paramètres

d'une même arborescence fonctionnelle. L'exploration parallèle des arborescences fonctionnelles paramètres étant conservée dans le modèle étendu, ce type de parallélisme demeure donc exploité.
Par ailleurs, le parallélisme vertical est également exploité : une fonction quelconque peut s'appliquer à un ou plusieurs arguments qui sont complètement ou partiellement calculés. En effet, un argument en cours d'évaluation est une expression dont les redex ne sont pas encore pris en compte. Dans le modèle P^3 étendu, nous avons introduit la notion d'*état de création partielle* des nœuds. Ces nœuds assurent entre autres la liaison entre plusieurs expressions évaluées simultanément (cf. chapitre 4). Dans le cas qui nous intéresse i.e celui où une fonction F s'applique à n arguments tels qu'au moins un est une expression, un nœud résultat dans l'état de création partielle représente cet argument dans l'expression appliquant la fonction F à ses arguments. L'exploration de l'arborescence fonctionnelle représentant la fonction F pourra alors se faire même si l'argument représenté par le nœud dont la création est partielle, est en cours d'évaluation puisque "tous" les arguments de la fonction F sont présents. Les redex issus de cette exploration pourront alors être pris en compte si la valeur de l'argument en cours d'évaluation n'est pas indispensable.
Le parallélisme vertical est également exploité lors de l'exploration d'une même arborescence fonctionnelle. En effet, imaginons l'arborescence fonctionnelle représentant une composition de fonctions de la forme $\ldots \circ f \circ (pp\ f_1\ f_2\ \ldots\ f_n) \circ \ldots$ où pp est une forme fonctionnelle de n paramètres. Cette arborescence est illustrée par la figure 6.7 suivante :

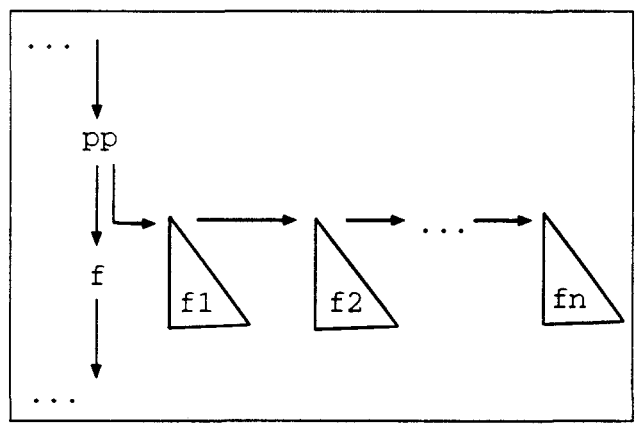


Figure 6.7 :

Les arguments de la fonction f sont les résultats obtenus après traitement de tous les redex construits suite aux explorations respectives des sous arborescences fonctionnelles paramètres de la forme fonctionnelle pp. Contrairement au modèle P^3 originel, dans le modèle P^3 étendu, l'exploration du chemin séquentiel principal

continue après la rencontre d'un nœud forme fonctionnel (cf. chapitre 4) sans attendre la fin de l'exploration des sous arborescences paramètres. De ce fait, le redex faisant intervenir la fonction f est alors construit et même traité si la fonction f est non stricte. Là aussi, chaque argument de la fonction f est représenté par un nœud résultat en état de création partielle en attendant la fin du calcul de ces arguments.
Nous en concluons qu'aussi bien le parallélisme vertical que le parallélisme horizontal sont exploités dans le modèle P^3 étendu et ce en partie, grâce à la possibilité de représenter un argument par un nœud en état de création partielle quand la valeur de l'argument qu'il représente, n'est pas disponible.

Nous pouvons en conclure que les deux modèles permettent l'exploitation du parallélisme des langages fonctionnels. Par ailleurs, le modèle P^3 évalue une expression grâce à ses deux mécanismes l'exploration et la réduction. Ces deux mécanismes en étant asynchrones, génèrent du parallélisme même dans le cas de traitement d'une expression séquentielle (l'application d'une composition de fonction à sés arguments), ce qui n'est pas le cas dans le modèle de réduction de graphe où le graphe est d'abord parcouru avant que les réductions puissent avoir lieu.

4.3 Les modes d'évaluation utilisés

Dans le modèle de réduction de graphe, aussi, bien l'appel par valeur que l'appel par nécessité sont utilisés et ce en effectuant le parcours du graphe de la même façon.
Dans le cas où l'appel par valeur est utilisé, l'évaluation des arguments peut être déclenchée en cours de parcours du graphe, même avant d'atteindre la fonction appliquée. Dans le cas d'une fonction non stricte, l'évaluation de tels arguments est inutile mais ceci est un inconvénient de l'appel par valeur et non pas de la manière dont ce mode est mis en oeuvre dans le modèle de réduction de graphe.
Dans le cas où l'appel par nécessité est utilisé, le parcours d'un sous graphe ayant la forme décrite par la figure 6.2.a s'arrête dès qu'une fonction non stricte est détectée sur le parcours, la fonction appliquée n'ayant pas besoin du résultat de l'évaluation du sous graphe non parcouru. Dans le cas de sous graphes ayant la forme décrite par la figure 6.2.b, l'évaluation des arguments est différée jusqu'à ce que la fonction appliquée est atteinte par le parcours. Le parcours de ce deuxième type de graphe révèle l'inconvénient de l'appel par nécessité dans le cas où l'évaluation d'un des arguments est nécessaire. En effet, l'évaluation d'un tel argument ne pourra pas se faire lors du parcours du graphe mais seulement quand

la fonction appliquée est atteinte par le parcours.

Dans le modèle P^3, le mode d'évaluation implicitement mis en oeuvre est l'appel par valeur. En effet, les redex sont créés par le mécanisme d'exploration dans l'ordre dans lequel ils devraient être pris en compte si toutes les fonctions contenues dans l'arborescence fonctionnelle sont strictes. De plus l'ordre de traitement des redex est similaire à l'ordre de construction des redex grâce à l'algorithme d'ordonnancement.

Pour pouvoir mettre en oeuvre l'appel par nécessité dans le modèle P^3, la première possibilité serait de modifier l'ordre de prise en compte des redex. Ce qui signifie que pour prendre en compte un redex quelconque issu d'un chemin séquentiel, il faut attendre que **tous** les redex issus de ce chemin soient construits, puisque les fonctions succeptibles d'utiliser un résultat se trouvent à la fin du chemin.

Cette solution est pénalisante pour deux raisons : Dans le cas où l'évaluation des redex générés est indispensable, ces redex auraient été générés grâce au parcours de l'arborescence fonctionnelle, mais traités beaucoup plus tard, jusqu'à ce que la construction du dernier redex issu du chemin séquentiel soit effectuée. Ceci augmente le coût de stockage des drapeaux et complique l'algorithme d'ordonnancement.
Dans le cas où l'évaluation de ces redex est inutile, le coût de stockage des redex augmente inutilement puisque ces redex ne seront pas traités.

Une deuxième tentative d'utiliser l'appel par nécessité consiste à modifier l'ordre de construction et d'évaluation des redex. En effet, il s'agirait de construire d'abord, les redex impliquant les fonctions se trouvant en bas d'un chemin séquentiel, en commençant l'exploration dans le sens inverse. Le but de cette démarche est d'éviter l'exploration des nœuds fonctionnels générant des redex dont l'évaluation est inutile.
L'exploration des arborescences fonctionnelles s'effectuerait alors de la manière suivante :

- L'exploration d'une arborescence fonctionnelle devrait commencer au dernier nœud du chemin séquentiel principal.

- L'exploration d'un nœud fonctionnel n entraîne l'exploration de son prédecesseur dans le chemin **uniquement** si le nœud n contient une fonction stricte et donc qui a besoin de ses arguments ce qui justifie la poursuite de

l'exploration. Dans le cas contraire, l'exploration du chemin s'achève sur l'exploration du nœud n.

- L'exploration d'un nœud forme fonctionnelle entraîne l'exploration des chemins séquentiels paramètres en commençant par le dernier nœud de chaque chemin. A la différence du mécanisme d'exploration décrit dans le chapitre 4, seuls les chemins séquentiels générant des redex dont le traitement est indispensable, sont explorés. En effet, l'évaluation des redex construits à partir des fonctions contenues dans ces chemins séquentiels permettent l'obtention des arguments de la fonction succédant dans le même chemin, au nœud forme fonctionelle exploré. Soit f cette fonction. Le nœud contenant la fonction f étant exploré antérieurement dans le parcours que nous proposons dans ce paragraphe, les arguments sur lesquels la fonction f est stricte sont identifiés préalablement. Dans ce cas seuls les chemins correspondant aux arguments sur lesquels la fonction f est stricte sont explorés.

Les redex sont constitués selon la description faite dans le chapitre 4 i.e chaque nœud exploré génère un redex. Un redex impliquant une fonction stricte est stocké en attente d'évaluer la ou les arguments impliqués et dont les valeurs sont le résultat du traitement des redex construits à partir de fonctions se trouvant plus haut dans le chemin séquentiel. Si par contre la fonction impliquée est une fonction non stricte, le redex peut être pris en compte immédiatement. Le résultat de l'application d'une telle fonction à ses arguments est un arbre de réduction dans lequel chaque argument réutilisé mais non évalué est représenté à l'aide d'un nœud en état de création partielle. En effet, la valeur de ces arguments étant inutile pour l'application de la fonction, les redex qui permettent de les calculer n'ont pas à être générés. Cependant, à la suite de l'application d'autres fonctions à l' arbre de réduction résultat, la valeur de ces arguments peut s'avérer nécessaire. Afin de permettre une évaluation par nécessité de la valeur de ce type d'arbre de réduction sans pour autant garder les redex permettant de les générer, un nœud en état de création partielle représente un argument dont l'évaluation n'est pas en cours et contiendra la référence du premier nœud fonctionnel non exploré dans l'arborescence fonctionnelle (cf. figure 6.8), ce qui permettra la reprise de l'exploration d'une arborescence fonctionnelle, si nécessaire.

Dans le cas de l'exemple, le dernier noeud non exploré du chemin séquentiel est le noeud fonctionnel de valeur i. L'argument obtenu par application de la fonction $i \circ j$ est représenté dans l'arbre de réduction par un noeud de donnée n, en état de création partielle. Dès que la valeur du sous arbre de réduction de racine le noeud n est sollicitée par l'application d'une fonction, l'exploration de la sous arborescence fonctionnelle de racine, le noeud de valeur i est redéclenchée de façon paresseuse.

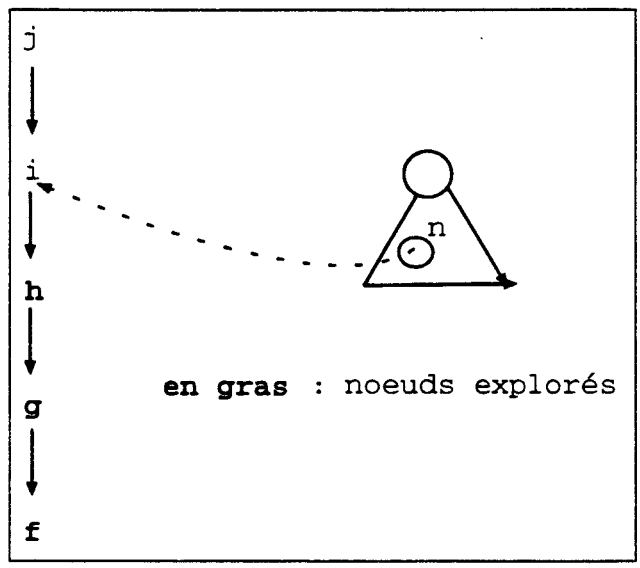


Figure 6.8 :

Nous retiendrons donc cette méthode pour la mise en oeuvre de l'appel par nécessité, dans le modèle P^3. Comparons maintenant la mise en oeuvre de ce mode d'évaluation dans les deux modèles, en reprenant le détail des coûts de la même manière que dans la section 3 :

- **Le coût de parcours**
  Dans les deux modèles, le coût de parcours est moins élevé comparé au cas où l'appel par valeur est utilisé . En effet, dans le modèle P^3, seuls les nœuds fonctionnels "utiles" sont explorés. Dans le modèle de réduction de graphe, le parcours d'un graphe est interrompu dès la rencontre d'une fonction non stricte.

- **Le coût de stockage de nœuds application vs. coût de stockage des drapeaux**
  Initialement, le nombre de redex présents dans la représentation de l'expression dans le modèle de réduction de graphe est supérieur au nombre de redex présents dans la représentation de la même expression dans le modèle P^3. Ceci a déjà été justifié dans la comparaison des coûts proposés dans la section 1 de ce chapitre. Si l'appel par nécessité est utilisé, le nombre de redex non traités dans P^3 croît, en cours d'évaluation, jusqu'à la rencontre d'une fonction non stricte, auquel cas les redex sont alors traités les uns après les autres. Dans le modèle de réduction de graphe, le nombre de redex diminue dès la rencontre de la première fonction pouvant être appliquée. A la différence du modèle P^3 où les redex inutiles ne sont pas

générés, dans le modèle de réduction de graphe, ces mêmes redex faisant partie de la représentation restent présents durant tout le temps nécessaire à l'évaluation. Le coût de stockage des drapeaux reste donc majoré par le coût de stockage des nœuds application.

- **Le coût de réorganisation du graphe**
  La réorganisation de graphe fait partie du processus d'évaluation et est imposée par la représentation de l'expression, dans le modèle de réduction de graphe. Ce coût existe donc quel que soit le mode d'évaluation adopté. Il est peut-être moins important dans le cas de l'appel par nécessité, si les graphes non traités génèraient une réorganisation.

- En revanche, **Le coût de l'ordonnancement dans le modèle P^3** est nécessairement plus important que dans le cas où l'appel par valeur est utilisé. En effet, le fait que certains redex ne sont pas constitués implique que l'ordonnancement ne peut plus se baser sur les mêmes informations pour classer les redex ce qui complique nécessairement l'algorithme d'ordonnacement.

En conclusion de ce paragraphe, l'appel par valeur et l'appel par nécessité peuvent tous deux être utilisés aussi bien dans le modèle de réduction de graphe que dans le modèle P^3. La classification des coûts que nous avons présenté en conclusion de la section 1 reste valable dans le cas où l'appel par nécessité est utilisé.

4.4 Le partage d'expression

Le partage d'expressions dans le modèle de réduction de graphe consiste à référencer plusieurs fois un même sous graphe par plusieurs nœuds du graphe. Le partage est une des plus importantes caractéristiques du modèle de réduction de graphe puisqu'il permet d'éviter des calculs redondants et optimise la copie d'arguments qui n'aura lieu qu'au cas où une des fonctions référençant le sous graphe vise à modifier la structure de ce graphe. La durée du partage dépend donc des traitements qui sollicitent simultanément le sous graphe partagé. Le partage de graphes occasionne cependant, un coût supplémentaire. Ce coût provient de la gestion des compteurs de référence qui empêche la modification des graphes partagés afin d'assurer la cohérence du résultat.

Dans le modèle P^3, il s'agit plus d'un partage de donnée dans le cas où l'appel par valeur est utilisé et d'un partage d'expression dans le cas où l'appel par nécessité est utilisé : si l'appel par valeur est utilisé, chaque fonction s'applique à

son argument quand l'évaluation de celui-ci est terminée ou tout au moins entamée; chaque fonction f située dans un chemin séquentiel s'applique à l'argument obtenu par application des redex issus des nœuds ancêtres dans le chemin. Ces redex une fois générés sont obligatoirement traités, bien que le traitement de la fonction f n'en utilise pas le résultat. Dans ce cas, la différence avec le modèle de réduction de graphe est qu'une expression partagée dans P^3 ne peut se faire sur des expressions dont l'évaluation n'est pas déclenchée. De ce fait, la duplication de calcul est évitée.
Si par contre l'appel par nécessité est utilisé, le partage d'expressions non évaluées est effectué de manière similaire au modèle de réduction de graphe. La seule différence est que dans le modèle de réduction de graphe, une expression partagée est représentée par un graphe syntaxique. Dans le modèle P^3 cette même expression est représentée par un noeud en état de création partielle référençant la sous arborescence fonctionnelle à explorer si l'expression est non évaluée, ou un arbre de réduction si la donnée est déjà évaluée.

Remarque

Dans le modèle P^3, nous pouvons citer une autre forme de partage : le partage de code; plusieurs expressions peuvent se partager, le cas échéant, le même code par une multi-exploration des arborescences fonctionnelles le représentant.
Dans le modèle de réduction de graphe, le code n'est pas réellement partagé puisqu'il est destructuré par les réorganisations du graphe et disparît dès qu'aucun redex ne le référence, ce qui n'est pas le cas dans le modèle P^3.

5 Conclusion

Dans ce chapitre, nous avons tenté d'établir une comparaison théorique entre le modèle P^3 et le modèle de réduction de graphe. Cette comparaison a tenu compte de deux critères complémentaires : le coût de gestion des deux modèles et les possibilités offertes par chacun.

En conclusion à l'étude des coûts de gestion, le coût de parcours et le coût de stockage des noeuds application sont plus importants dans le modèle de réduction de graphe que dans le modèle P^3 et ce, quel que soit le mode d'évaluation choisi. Le gain apporté par le modèle P^3 pour ces deux coûts est fortement lié à la représentation de l'expression proposée.

Cependant, pour pouvoir comparer globalement les coûts de gestion des deux modèles, il faut procéder à une réelle implantation des deux modèles et comparer

leur efficacité.
Les deux modèles offrent quasiment les mêmes possibilités. De plus, le modèle P^3 permet un réel partage du code qui n'engendre aucun surcoût. Ses mécanismes fonctionnant de façon asynchrone permettent un gain au niveau du temps d'exécution même dans le cas d'expressions séquentielles.

Le chapitre suivant propose une validation du modèle P^3 par une simulation. Celle-ci a permis également d'évaluer le parallélisme potentiel du modèle P^3. Le chapitre suivant aborde également les problèmes de répartition de données qui risquent de se poser dans une réelle implantation et propose quelques solutions pouvant contribuer à résoudre partiellement le problème.

Chapitre 7

Simulation du modèle P^3

Parallèlement aux travaux d'extension du modèle P^3, nous avons réalisé une simulation de l'implantation du modèle P^3 sur un multiprocesseur à mémoire distribuée. Les objectifs d'une telle simulation sont : tout d'abord la validation du modèle P^3. En effet, cette simulation permet de vérifier le fonctionnement de l'algorithme d'exploration et l'algorithme d'ordonnancement des requêtes de réduction. Le deuxième objectif de cette simulation est de mesurer le parallélisme potentiel du modèle P^3 indépendamment de toute contrainte sur la communication ou sur le nombre de processeurs utilisé. Enfin le troisième objectif est d'introduire les problèmes de répartition qui se posent pour les données dans les langages fonctionnels. En effet, les données d'un programme fonctionnel se caractérisent par un fort degré de dynamicité. Le placement adéquat dans ce cas est de type dynamique. A l'issue de cette étude préalable des problèmes de placement, nous proposons une méthode de répartition des copies qui contribue à l'amélioration du placement dynamique des données.

Ce chapitre est donc structuré de la manière suivante : La section 1 présente une description du simulateur que nous avons développé. La section 2 présente les résultats théoriques qui permettent la validation du modèle P^3 et montrent son parallélisme potentiel. Ces résultats sont qualifiés de théoriques parce qu'aucune contrainte ni sur la communication ni sur le nombre de processeurs impliqués n'a été imposée. La section 3, après étude préalable du type de placement adapté au modèle P^3, présente d'abord une étude de l'influence de la répartition initiale puis propose une méthode dynamique de placement des copies.

1 Description de la simulation

Dans cette section, nous décrirons successivement les choix d'implantation que nous avons retenus dans le modèle, la structure des sites, le fonctionnement global du simulateur ainsi que la description des caractéristiques du simulateur.

1.1 Choix d'implantation effectués

En décrivant le modèle P^3 de façon formelle, dans les chapitres 4 et 5, nous avons fait volontairement abstention de tout détail d'implantation. Cependant, la simulation du modèle nécessite de faire quelques choix d'implantation concernant la topologie simulée mais aussi concernant le déroulement de l'évaluation dans le modèle.
Ces choix sont décrits dans cette section. Ils concernent l'implantation des mécanismes du modèle, la réalisation et l'optimisation des copies ainsi que la mise en œuvre de la notion de fenêtre de réduction définie dans le chapitre 5.

1.1.1 Description du modèle en terme de messages

Dans la simulation, les deux activités du modèle P^3 sont exprimées en termes de messages. Ce choix d'implantation présente des avantages et des inconvénients. Le principal inconvénient est la finesse du grain de parallélisme. En effet, le traitement d'un message consiste la plupart du temps en la création et l'envoi d'autres messages. Ceci implique un taux de communication massif. Par ailleurs, cette décomposition présente les avantages suivants :

- Elle met plus en évidence le parallélisme implicite exploité dans le modèle P^3.

- Le deuxième but de la simulation étant d'étudier quelques aspects de la répartition des arborescences fonctionnelles et des arbres de données, la decomposition en termes de messages assure un traitement similaire et donc plus simple à mettre en oeuvre quelle que soit la répartition des noeuds fonctionnels composant une arborescence (resp. des noeuds de données composant les arbres de données)

La simulation du modèle P^3 a été réalisée parallèlement à l'extension du modèle. Aussi, l'algorithme d'exploration mis en oeuvre est celui proposé dans la thèse de N. Devesa [Dev90]. Par rapport à l'algorithme d'exploration décrit au chapitre

4, l'algorithme d'exploration mis en oeuvre dans la simulation se caractérise par davantage de synchronisation au cours de l'exploration. Ainsi :

- L'exploration du successeur d'un noeud forme fonctionnelle ne peut se faire que si les sous arborescences fonctionnelles paramètres sont explorées.
- L'exploration du successeur d'un noeud définition, ne peut se faire que si l'arborescence fonctionnelle représentant la définition est entièrement explorées.

Par ailleurs, les noeuds fonctionnels ont un seul noeud cible et plusieurs noeuds fonctionnels peuvent avoir le même noeud cible. Aussi, les requêtes de réduction issues des noeuds fonctionnels doivent être ordonnées. Dans ce but, les chemins séquentiels ont été découpé en *branches séquentielles* [1]. A la différence de l'algorithme d'ordonnancement proposé dans le chapitre 5, l'algorithme proposé ne permet pas d'interclasser les ordres de réduction en provenance de plusieurs branches séquentielles. Aussi, l'exploration d'une branche séquentielle ne peut commencer que si le premier DAR de la branche précédente est créé.

La décomposition du mécanisme d'exploration en termes de messages est la suivante :

- les messages d'exploration qui ont pour destination des noeuds fonctionnels et correspondent aux DAE dans la description formelle du modèle. Un message d'activation à destination d'un noeud fonctionnel successeur d'un noeud définition ou d'un noeud forme fonctionnelle est créé bloqué en attente d'un nombre de messages de fin d'exploration.
- les messages de fin d'exploration à destination de messages d'activation bloqués : Ces messages sont générés à la fin de l'exploration d'une arborescence fonctionnelle secondaire représentant une définition ou à la fin de l'exploration d'une sous arborescence fonctionnelle paramètre d'une forme fonctionnelle.
- les messages d'acquittement : un message d'acquittement débloque l'exploration d'une nouvelle branche séquentielle bloquée afin d'assurer l'ordonnancement.

La réduction des arguments occasionnée par les fonctions appliquées se décompose en deux types de messages :

[1] Il s'agit d'une décomposition des chemins séquentiels sur des critères différents que celle proposée dans le chapitre 5 (cf. [Dev90])

- les messages NENR, (Noeud Exploré vers Noeud à Réduire) correspondant aux DAR dans la description formelle. Ils sont envoyés aux noeuds cibles de la fonction à la suite du traitement d'un message d'activation.

- L'application d'une règle de réduction se décompose en un ensemble de messages NRNR (Noeud à Réduire vers Noeud à Réduire). Ces messages sont propres à chaque règle de réduction. Les nœuds émetteur et destinataire d'un message NRNR appartiennent à un ou plusieurs arbres de données, concernés par la même requête de réduction.

Exemple

La règle de réduction applicable à un DAR contenant la fonction +, représentée par la figure 7.1.a, se décompose en 3 messages NRNR : m_1, m_2 et m_3 figure 7.1.b.

- Le message m_1, contenant la fonction +, est adressé au noeud de donnée représentant le premier argument de valeur 5.

- Le message m_2 généré après réception du message m_1, et adressé au deuxième argument, véhicule la valeur du noeud de donnée représentant le premier argument.

- Le message m_3 généré après réception du message m_2 par le deuxième argument, permet de communiquer la somme des deux arguments au noeud résultat intermédiaire R.

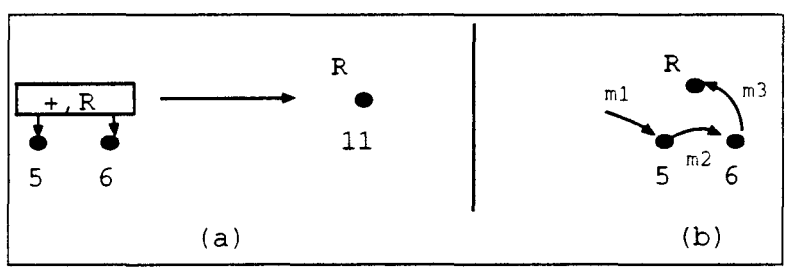


Figure 7.1 : (a)règle de réduction de la fonction +.(b)Description de la règle de réduction en terme de messages de réduction

Remarque : La polyadicité des fonctions est prise en compte dans la simulation.

1.1.2 Réalisation des copies

Dans le simulateur nous avons choisi de mettre en oeuvre la copie d'arguments en effectuant une décomposition en termes de messages NRNR. L'avantage de ce procédé est que la copie comme toute autre opération sur les arguments s'exprime en termes de messages. D'autre part, cette approche implique qu'une telle copie s'effectue entièrement en parallèle pour peu que la donnée copiée ne se trouve pas sur le même processeur. Le choix de simuler le fonctionnement du modèle P^3 sur un multiprocesseur à mémoire distribuée et l'évolution constante des données, occasionnée par les réductions successives implique qu'on se trouve constamment confronté à cette situation. L'intérêt d'effectuer une copie en parallèle est de permettre la reprise le plus vite possible des réductions sur l'argument copié et sur la copie, puisqu'elle se fera plus rapidement.
La copie d'un argument se décompose en trois phases :

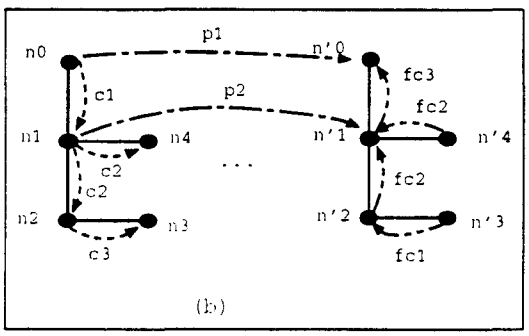


Figure 7.2 : Phases d'une copie

- La *phase de propagation des demandes de copies* illustrées par les arcs de type c (figure 7.2). Le numéro d'un arc de type c indique la chronologie de propagation des demandes de copies. Ainsi la propagation des copies peut se faire de façon parallèle (arcs *c2* dans l'exemple).

- La *phase de placement des nœuds* illustrée par les arcs étiquetés p_i. En effet chaque nœud ayant reçu une demande de copie se duplique. Le nouveau nœud doit être placé sur un processeur a priori quelconque.

- La *phase de remontée des fins de copie* illustrée par les arcs de type *fc* sur la figure 7.2. Cette phase est nécessaire à la reconstitution de la structure du nouvel arbre de données. Elle s'effectue également en parallèle mais nécessite une synchronisation supplémentaire. Ainsi le message de fin de copie *fc3* ne peut être envoyé au nœud n'_0 que si les deux messages de fin de copie *fc2* issus des nœuds n'_2 et n'_4 ont été reçus par le nœud de donnée n'_1.

Autre caractéristique de cette méthode en plus du fait qu'elle permet la copie en parallèle, est sa simplicité. En effet, quel que soit le placement de la donnée copiée, le traitement effectué est similaire pour la copie de chacun des nœuds. Cette méthode pourra être optimisée en faisant un placement groupé des données i.e en ne faisant qu'une seule communication pour un groupe de nœuds de donnée adjacents placés sur un même processeur.

1.1.3 Optimisation des duplications

Dans le simulateur, nous avons choisi la copie d'arguments comme forme de duplication quand plusieurs fonctions s'appliquent simultanément à une même donnée. Cependant, dans le cas où plusieurs arguments d'une même fonction sont dupliqués, il peut arriver que seul un sous ensemble d'arguments soit réellement utile. On retrouve ce problème notamment dans le cas d'une arborescence fonctionnelle $(pp \quad f_1 \quad f_2 \quad ...f_n)$ où chaque paramètre f_i de la forme fonctionnelle est une composition de fonctions de la forme $f_{i_1} \circ f_{i_2} \circ \ldots \ f_{ij_i}$. Une telle forme fonctionnelle est représentée par l'arborescence fonctionnelle 7.3 :

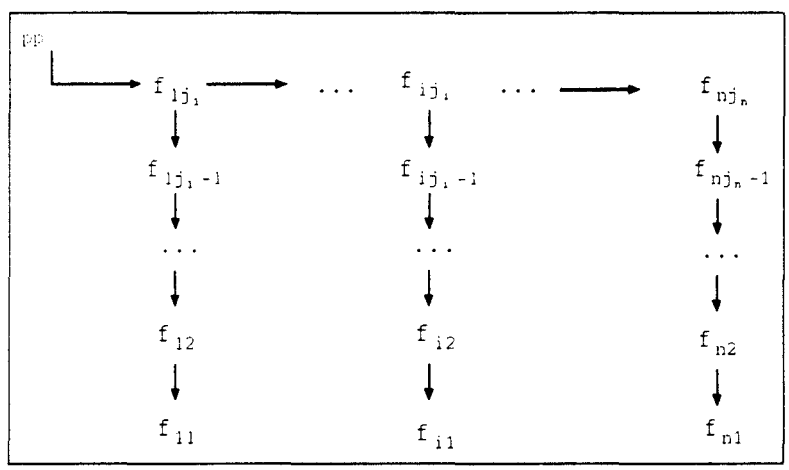


Figure 7.3 :

L'optimisation consiste à adapter la copie d'arguments en fonction des besoins des paramètres f_i. Par exemple, si f_{ij_i} est un sélecteur s_k, ceci signifie que seule la copie du kième argument est indispensable. Si f_{ij_i} est une fonction n'utilisant aucun des arguments, la copie d'arguments n'est alors pas utile. L'optimisation de la copie consiste dans ces cas à ne copier que les arguments utiles. Ceci permet d'abréger le temps de copie puisque le volume des nœuds copiés est moins important ce qui permet d'achever plus rapidement la duplication et donc de reprendre plus rapidement les réductions, et sur les données copiées et sur les

données originelles. De plus, il y aura moins de nœuds de donnée à placer et de ce fait moins de communication.
La phase de préparation des données qui permet d'initialiser la copie de donnée est activée à l'exploration d'un nœud forme fonctionnelle. Or à ce stade, le contenu des nœuds racines des chemins séquentiels paramètres n'est pas connu. Le seul moyen d'appliquer donc l'optimisation décrite plus haut est de retarder la copie d'arguments jusqu'à l'activation des racines des chemins séquentiels paramètres. Nous avons donc opté pour cette solution.

1.1.4 Implantation des fenêtres de réduction

La notion de fenêtre de réduction est une entité fictive que nous avons définie au chapitre 5 pour permettre l'ordonnancement des requêtes de réduction séquentielles. Dans la simulation, nous choisissons d'implanter une fenêtre de réduction comme un nœud de donnée quelconque. L'intérêt d'un tel choix est de normaliser les entités pouvant recevoir des messages. En effet dans le simulateur, chaque nœud est doté d'une file d'attente des messages dont il est le destinataire. La sélection des messages dans la file d'attente d'un nœud de donnée représentant une fenêtre de réduction s'effectue en appliquant l'algorithme d'ordonnancement, énoncé dans le chapitre 5, à la file d'attente du nœud de donnée en question. Une fenêtre de réduction contenant n arguments est donc représentée comme le montre la figure 7.4, par un nœud de donnée ayant n fils chacun étant la racine d'un argument. Ce choix d'implantation des fenêtres de réduction pose le problème d'accès aux données qui n'est de ce fait pas direct. Cependant, cette solution est moins pénalisante que la solution permettant l'accès direct et qui consiste à implanter la fenêtre de réduction comme le montre la figure 7.5 suivante :

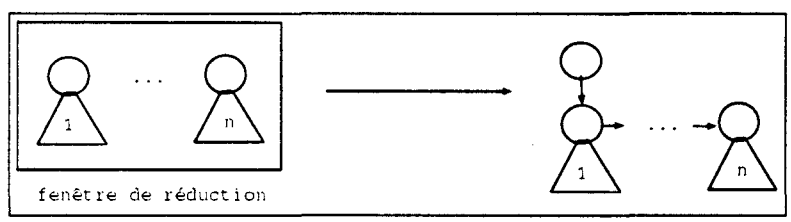


Figure 7.4 : Implantation d'une fenêtre de réduction

i.e la fenêtre de réduction est mise en oeuvre par un noeud de taille variable. En effet, la décomposition des fonctions engendrerait plus de messages du fait que les arguments ne sont pas liés : l'accès aux arguments nécessite des accès multiples au noeud racine.

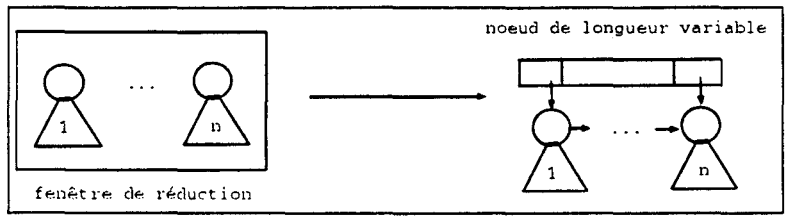


Figure 7.5 : Implantation d'une fenêtre de réduction

1.2 La topologie simulée

La simulation réalisée du modèle P^3 permet l'évaluation d'une expression fonctionnelle de façon distribuée sur un ensemble de processeurs reliés par un réseau d'interconnexion (cf figure 7.6). La composition de chaque site est décrite dans cette section.

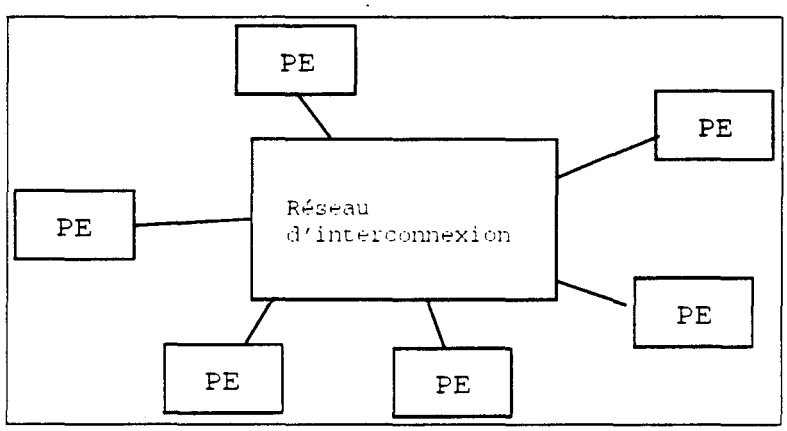


Figure 7.6 : topologie simulée

La figure 7.7 schématise la composition d'un site.

Un site comprend:

- Une unité d'exploration : UE qui traite les messages d'exploration.
- Une unité de réduction : UR qui traite les messages NENR de la réduction et les messages NRNR.
- Une file d'attente Fmin des messages reçus d'autres sites
- Un répartiteur local qui distribue les messages en provenance des autres sites selon leur type sur les files d'attente associées.
- Un espace mémoire local où sont stockés les nœuds fonctionnels traités par l'UE locale et les nœuds de donnée traités par l'UR locale. Cet espace est partagé entre l'UR et l'UE en ce sens qu'il n'existe

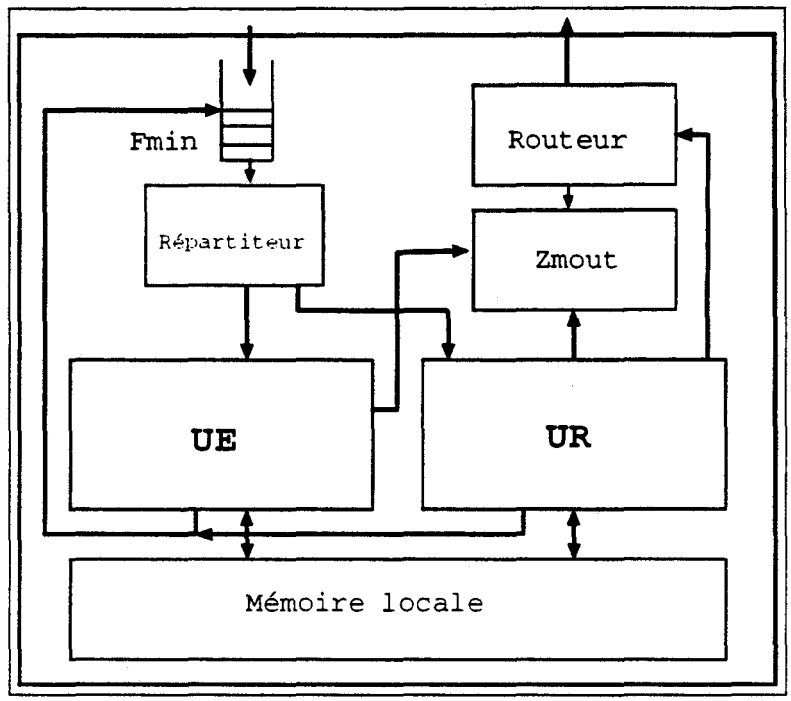


Figure 7.7 : Description d'un site

pas un espace d'allocation privé pour l'UR et un autre pour l'UE : chaque unité peut allouer autant d'espace mémoire que le permet la taille mémoire locale toute entière. Cet espace mémoire est réellement partagé entre l'UE et l'UR lors du traitement des fonctions d'ordre supérieur. En effet une arborescence fonctionnelle résultat est construite par l'UR. Elle peut être explorée par l'UE si elle s'applique elle-même à des arguments. – Une zone tampon Zmout où sont déposés les messages, générés par les unités du site, à destination d'autres sites. – Un routeur qui s'occupe de l'envoi des messages non locaux aux sites destinataires.

1.2.1 Structure d'une unité d'exploration

Une unité d'exploration se compose de 4 files d'attente FMACT, FMFE, FMACK et FMB et d'un processus de traitement voir figure 7.8.

FMACT File d'attente des messages d'activation destinés à un nœud fonctionnel local.

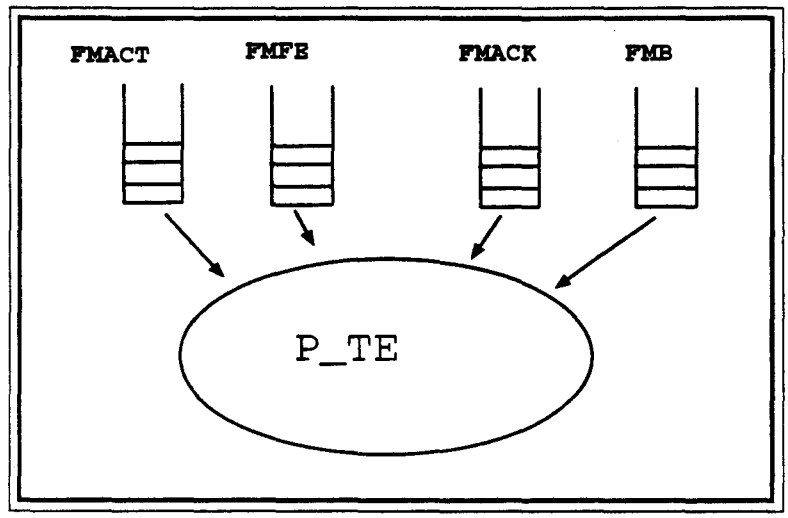


Figure 7.8 : L'unité d'exploration

FMB File d'attente des messages d'activation ou de fin d'exploration émis à la suite du traitement d'un message d'activation par un nœud fonctionnel local. Ces messages sont en attente d'un ou de plusieurs messages de fin d'exploration et/ou d'acquittement.

FMFE File d'attente des messages de fin d'exploration émis à destination de messages d'activation ou de fin d'exploration bloqués dans la file d'attente FMB locale.

FMACK File d'attente des messages d'acquittement émis à destination de messages d'activation ou de fin d'exploration bloqués dans la file d'attente FMB locale.

P_TE Processus de traitement des messages d'activation, de fin d'exploration et d'acquittement.

1.2.2 Structure d'une Unité de réduction

Une unité de réduction se compose de 3 files d'attente : FMNENR, FMNRNR, FMSERV et d'un processus de traitement P_TR (voir figure 7.9).

FMNENR File d'attente des messages NENR dont les destinataires sont des nœuds de donnée locaux.

FMNRNR File d'attente des messages NRNR dont les destinataires sont des nœuds de donnée locaux.

FMSERV File d'attente des messages de service. Ces messages sont des messages NRNR particuliers : ce sont des demandes de placement et de copie de nœuds.

P_TR Processus de traitement des messages NENR, NRNR.

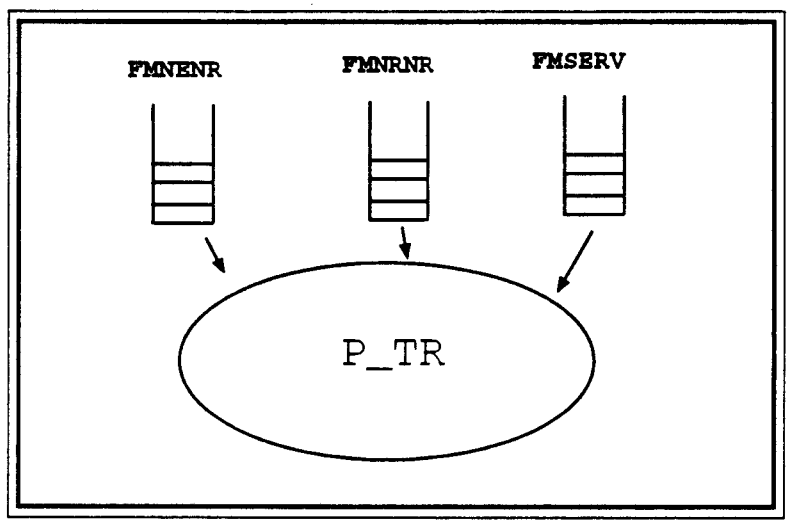


Figure 7.9 : L'unité de réduction

1.2.3 Le routeur

Le routeur est un processus qui établit la communication entre le processeur auquel il est affecté et les autres processeurs. Dans le simulateur, l'algorithme de routage a été considérablement simplifié, le but de la simulation n'étant pas de simuler le fonctionnement du réseau ni celui d'un routeur réel mais uniquement la simulation de la communication.

1.2.4 Le répartiteur

Le répartiteur extrait les messages qui se trouvent dans la file d'entrée du processeur et les range selon leur type dans une des files d'attente propres à l'UE ou à l'UR (cf. figure 7.10).

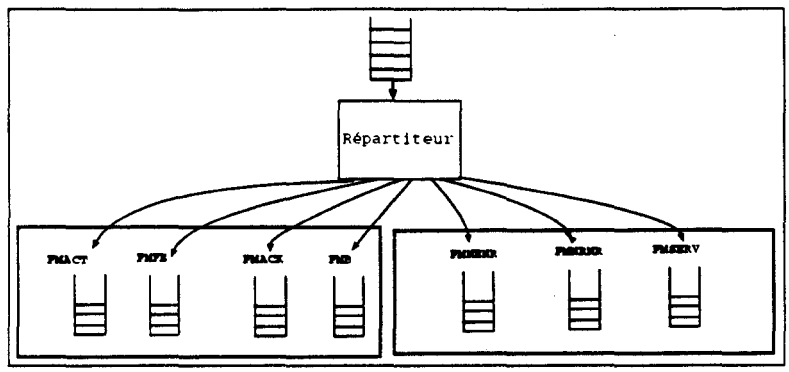


Figure 7.10 : Le répartiteur

1.3 Fonctionnement du simulateur

- La simulation est initialisée par un placement initial des noeuds fonctionnels et des noeuds de données, composant respectivement les arborescences fonctionnelles et les arbres de données représentant l'expression évaluée, dans les mémoires locales des processeurs.

- La simulation débute par la création d'un premier message d'activation adressé au noeud fonctionnel racine de l'arborescence fonctionnelle principale de chaque fonction appliquée.

- Durant chaque unité de temps, sur chaque site se déroulent les activités suivantes :

  * Le répartiteur scrute la file d'attente en entrée Fmin. Les messages qui s'y trouvent sont répartis selon leur type sur les files d'attente de l'UR et de l'UE : un message d'activation est placé dans la file d'attente FMACT, un message de fin d'exploration dans la file d'attente FMFE et ainsi de suite.
  * Un message est traité durant une unité de temps logique. La priorité est donnée aux messages liés à l'exploration i.e les messages d'activation, de fin d'exploration et d'acquittement. Le traitement d'un message génère un ou plusieurs messages qui sont placés selon leurs destinataires soit dans la file d'attente Fmin si les noeuds destinataires sont locaux ou dans la zone tampon Zmout dans le cas où au contraire les noeuds destinataires se trouvent sur d'autres sites.
  * Le routeur achemine les messages qui ont été générés et placés dans la zone tampon Zmout si la capacité du réseau le permet, en les plaçant dans les files Fmin des processeurs destinataires.

- La simulation se termine quand il ne reste plus aucun message à traiter.

1.4 Le traitement des messages

Les messages reçus sur un site sont traités selon leur type de la manière suivante :

- Le traitement d'un message d'activation consiste à appliquer une règle d'exploration en fonction des critères précisés au chapitre 4. Un message NENR est alors généré et d'autres messages d'activation à destination d'autres noeuds fonctionnels sont alors créés.
- Le traitement d'un message de fin d'exploration (resp. d'acquittement) consiste à décrémenter le nombre de messages de fin d'exploration (resp. d'acquittement) attendus par un message d'activation ou de fin d'exploration, dans la file d'attente FMB.
- Un message NENR représente un ordre d'application d'une fonction à ses arguments. L'application des fonctions a été décomposée en un ensemble de messages NRNR (cf. section 1.1.1). Le traitement des messages NENR consiste à générer un sous ensemble de ces messages.
- Le traitement d'un message NRNR faisant partie de la décomposition en messages NRNR de l'application d'une fonction, et destiné à un noeud de donnée, consiste à effectuer un calcul local au noeud destinataire et à générer d'autres messages NRNR faisant partie du même ensemble vers d'autres nœuds de données.

1.5 Caractéristiques du simulateur

Dans cette section, nous abordons quelques choix établis dans la réalisation du simulateur. Ces choix concernent la simulation de la communication et du routage, le choix du rapport entre le temps de communication et le temps de traitement, et l'estimation de la charge de travail des processeurs. Ce dernier point est abordé car, outre la validation du modèle, nous nous sommes intéressés aussi à certains problèmes de placement (cf. section 3).

1.5.1 Simulation de la communication

Le programme que nous avons développé simule l'implantation du modèle P^3 sur un multiprocesseur à mémoire distribuée. Nous nous sommes inspiré pour

cela d'un cas réel de machine à mémoire distribuée : la machine Multicluster II de Parsytec qui est un multiprocesseur dont le réseau reliant les différents processeurs est reconfigurable.
Les choix d'implantation adoptés ne sont pas nécessairement optimaux mais permettent l'obtention de résultats approximant le cas réel.

Dans un multiprocesseur à mémoire distribuée, la communication entre les processeurs s'effectue grâce à un réseau d'interconnexion. La simulation d'un vrai réseau nécessiterait la gestion du routage des messages i.e le choix de l'itinéraire des messages et la gestion des collisions dans le cas où un ou plusieurs liens choisis pour acheminer le messages sont occupés. Etant donnés nos objectifs, nous avons simplifié au plus la simulation du réseau d'interconnexion en s'inspirant des caractéristiques d'un réseau de type Mesh. Nous avons donc procédé de la manière suivante :

- Nous supposons constante, la distance en nombre de liens parcourus entre toute paire de processeurs désirant communiquer. Cette distance est égale à 2 liens. Elle a été judicieusement choisie : une première simulation simplifiée -sans tenir compte des collisions- d'un réseau de type Mesh, ont permis le calcul de la distance moyenne parcourue par un message lors d'une communication. Cette distance a été retenue comme une distance constante pour chaque communication.

- Nous approximons la prise en compte de la collision dans le réseau en se fixant une borne de saturation du réseau. Au delà de cette borne, un message ne peut être immédiatement acheminé vers son destinataire. Il devra attendre que le réseau soit moins saturé. La borne de saturation choisie est égale à $2p$, p étant le nombre de processeurs concernés par la simulation.
  Le réseau devient moins saturé quand le nombre de messages "dans le réseau" est inférieur à la borne de saturation. D'autres messages en attente peuvent alors être envoyés. Afin de ne pas privilégier l'envoi de messages émis par un processeur particulier, le processeur qui émettra le prochain message est choisi de façon cyclique.

1.5.2 Rapport temps de traitement/ temps de communication

L'unité de temps de simulation est une unité logique. Elle correspond au temps nécessaire au traitement d'un message de type quelconque. Comme rapport entre le temps de traitement d'un message et le temps de communication d'un message,

nous avons pris le rapport suivant

$$\frac{Temps\ de\ communication}{Temps\ de\ traitement} = 2$$

i.e le temps de communication d'un message d'un processeur à un autre correspond au temps de traitement de deux messages sur un processeur.

Ce rapport a été adopté après des essais expérimentaux réalisés sur un multiprocesseur réel [2], avec le temps moyen d'exécution d'une procédure de traitement d'un message et le temps de communication moyen entre 2 processeur sur la même machine.

1.5.3 Estimation de la charge de travail

Nous avons choisi de quantifier le travail que chaque processeur doit effectuer en termes de messages à traiter tous types confondus. En effet, chaque message envoyé vers un processeur constitue une charge supplémentaire de travail puisqu'il doit être traité par le processeur qui l'a reçu et génère d'autres messages qu'il devra adresser au routeur local afin qu'ils puissent être acheminés vers leurs destinataires respectifs.

1.6 Mesures effectuées par le simulateur

Afin de suivre le comportement du programme en cours d'exécution, nous nous basons sur 3 courbes de simulation qui permettent de comprendre les étapes de l'évaluation.

- **Courbe du taux de parallélisme**

  Ce premier type de courbes exprime le taux de parallélisme, i.e le nombre de processeurs actifs simultanément, tout au long de la simulation.

- **Courbe des communications**

  Ce type de courbes indique le taux de communication tout au long de la simulation. Le taux de communication est exprimé en nombre de messages

[2] Multicluster II de Parsytec

simultanément acheminés ou en attente d'être acheminés vers leurs destinataires. Sur le même graphique, nous représentons la courbe de la fonction $f(x) = 2p^a$

[a] p est le nombre de processeurs impliqués dans la simulation

Cette courbe indique la capacité maximale du réseau (cf. section 1.5.1) et permet de constater la saturation éventuelle du réseau : si la courbe de communication est au dessus de la courbe constante $f(x) = 2p$, le réseau est saturé. La surface entre la courbe de communication et cette dernière indique alors le volume de messages en attente d'être transmis à leurs destinataires. Par opposition, si la courbe de communication est en dessous de la courbe $f(x) = 2p$, le réseau n'est pas saturé.

- **Courbe d'activité moyenne des processeurs actifs**
  Cette courbe montre tout au long de l'évaluation, la quantité de travail moyenne, en nombre de messages restant à traiter par chaque processeur actif.

Ces 3 courbes pour une même simulation se complètent pour fournir les informations suivantes : un faible taux de parallélisme, un faible taux de communication et une courbe d'activité élevée traduisent un regroupement excessif des données sur un nombre réduit de processeurs. En effet, les messages dans le simulateur sont destinés à des nœuds - fonctionnels ou de données - et le nombre de messages destinés à un processeur est donc proportionnel au nombre de nœuds présents sur un processeur. Le regroupement des nœuds d'une même structure i.e d'une arborescence fonctionnelle ou d'un arbre de données permet respectivement l'exploration ou la réduction, sans communication ce qui explique le taux faible de communication.
Inversement, un taux de communication élevé, un taux de parallélisme élevé et une activité peu importante des processeurs actifs traduisent un grain de parallélisme très fin. Ces 3 types de courbes permettent, en somme, de vérifier si la charge de travail est bien répartie. Une bonne répartition des données doit permettre l'obtention :

- d'une courbe de communication faible,
- d' une courbe de parallélisme importante.
- une activité des processeurs actifs, proportionnelle à la quantité de travail requise pour l'évaluation de l'expression.

2 Résultats théoriques maximaux

La simulation théorique se caractérise par deux facteurs importants : le nombre de processeurs considéré est potentiellement infini : le nombre de processeurs utilisés s'adapte à la quantité de messages à traiter à un instant donné et la communication n'est pas prise en compte i.e. on considère que la durée de communication entre deux processeurs est nulle et qu'il n'y a pas de saturation de réseau. L'intérêt d'une telle simulation est avant tout, de valider le fonctionnement du modèle. Le parallélisme obtenu est maximal puisqu'aucune contrainte de localité des traitements, de communication ou de limitation du nombre de processeurs n'est imposée. Les résultats obtenus permettent d'apprécier ou non l'exploitation du parallélisme implicite des langages fonctionnels dans le modèle P^3. Avant de donner les résultats, nous présentons les programmes qui ont servi de test pour valider la simulation.

2.1 Programmes utilisés

Les programmes testés par la simulation sont de deux types : des programmes fortement récursifs où le parallélisme est introduit par le style de programmation "divide and conquer". Comme exemples de programmes de ce type, nous avons testé le calcul de la suite de Fibonacci et le quicksort. Le deuxième type de programmes testé est non récursif et manipule des données de grande taille, où le parallélisme provient d'un traitement répétitif sur plusieurs parties de la donnée. Comme exemples de ce type de programme, nous avons testé la multiplication de matrices et la multiplication de polynômes.

Les sources de ces programmes en GRAAL sont les suivants :

- **le quicksort**

$$
\begin{array}{l}
quick \equiv \\
\quad (if\ \ (binu\ \ =\ \ 1) \circ length \\
\qquad id \\
\qquad (if\ \ null \\
\qquad\quad id \\
\qquad\quad \{append\ \ quick \circ gauche\ \ \{cons\ \ car\ \ quick \circ droite\}\}
\end{array}
$$

$$
\begin{array}{lll}
gauche & \equiv & \{inferieur\ \ cdr \circ id\ \ car\} \\
inferieur & \equiv & gather \circ (distr\ \ (if\ \ le\ \ 1\ \ nil))
\end{array}
$$

$$
\begin{array}{lll}
droite & \equiv & \{superieur\ \ cdr \circ id\ \ car\} \\
superieur & \equiv & gather \circ (distr\ (if\ \ gt\ \ 1\ \ nil)) \\
gather & \equiv & (if\ \ null \\
 & & \qquad nil \\
 & & \qquad (if\ \ null \circ car \\
 & & \qquad\qquad gather \circ cdr \\
 & & \qquad\qquad \{cons\ \ car\ \ gather \circ cdr\}))
\end{array}
$$

- **La suite de Fibonacci**

$$
\begin{array}{lll}
fibo & \equiv & (if\ \ (binu < 3) \\
 & & \qquad 1 \\
 & & \qquad \{+\ \ fibo \circ sub1\ \ fibo \circ (binul\ \ -\ \ 2)\})
\end{array}
$$

- **La multiplication de matrices**

$$
\begin{array}{lll}
pm & \equiv & \{(distr\ \ (distl\ \ ps))\ \ 1\ \ (binul\ \ apply\ \ \alpha list) \circ 2\} \\
ps & \equiv & (binul\ \ apply\ \ +) \circ \alpha *
\end{array}
$$

- **La multiplication de polynômes**

$$
\begin{array}{lll}
mulpoly & \equiv & (distl\ \ distr\ \ mulmonom) \\
mulmonom & \equiv & \{list\ \ mulcoeff\ \ mulexpo\} \\
mulcoeff & \equiv & (binul\ \ apply\ \ mul) \circ \alpha car \\
mulexpo & \equiv & (binul\ \ apply\ \ \alpha add) \circ \alpha(car \circ cdr)
\end{array}
$$

2.2 Résultats

La table suivante décrit les résultats obtenus pour l'évaluation des expressions suivantes :

- *quick*: :L, L étant une liste de 30 entiers choisis aléatoirement,
- *fibo* : 14,
- *ps* : M_1 M_2, M_1 et M_2 étant deux matrices carrées de taille 10 et
- *mulpoly* : P_1 P_2, P_1 et P_2 étant deux polynômes de taille 10.

programmes / *résultats*	*Quicksort*	*Fibonacci*	*mul. matrices*	*mul. polynômes*
temps de simulation	11862 Ut	234 Ut	272 Ut	188 Ut
nb. total de messages	61007 Mes.	36127 Mes	20026 Mes	36135 Mes.
taux de parallélisme	5.14	154.39	73.62	192.21
messages traités / Ut	5.14 Mes.	154Mes.	73 Mes.	192 Mes.
nb. de noeuds créés	12368 nds	6398 nds	3636 nds	6859 nds
nb. de noeuds détruits	12178 nds	6331 nds	386 nds	110 nds
nb. de copies [a]	1796	2635	100	199
nb. max de noeuds copiés	31 nds	1 nds	223 nds	131 nds
nb. moyen de noeuds copies	12 nds	1 nds	11 nds	25 nds
Messages ACT traités	5218 Mes.	6022 Mes.	340 Mes.	1443 Mes.
Messages FE traités	2088 Mes.	3385 Mes.	105 Mes.	1100 Mes.
Messages ACK traités	1352 Mes.	2634 Mes.	115 Mes.	911 Mes.
Messages NENR traités	5725 Mes.	6398 Mes.	1529 Mes.	3112 Mes.
Messages NRNR traités	12617 Mes.	4891 Mes.	9787 Mes.	12504 Mes.
Messages de service traités	34007 Mes.	12797 Mes.	8150 Mes.	17065 Mes.

[a] Le nombre maximum de noeuds copiés, donné dans ce tableau est mesuré par copie

commentaires des résultats

- La première constatation que nous pouvons faire est que le degré de parallélisme obtenu est important (nombre de messages traités par unité de temps) pour tous les programmes testés. Ceci traduit l'importance du parallélisme implicite exploité dans le modèle P^3, compte tenu de l'absence de contraintes matérielles.

- Ces simulations révèlent l'importance du nombre de copies et du volume de noeuds copiés pour les programmes manipulant des données de taille importante, i.e la multiplication de polynômes et de matrices, ce qui explique l'importance du nombre de messages de service pour ces deux programmes. Dans les programmes récursifs testés, le volume de noeuds copiés à chaque fois est négligeable mais le nombre de copies est plus important que pour les deux programmes déjà cités vu la nature récursive de ces problèmes ce qui explique pourquoi le nombre de messages de service est également élevé.

- Le pourcentage de messages liés à l'exploration des arborescences fonctionnelles [3] est très important pour les programmes récursifs (33 % du nombre total de messages pour le programme *fibo* et 14 % pour le *quicksort*). En

[3] les messages d'activation, de fin d'exploration et d'acquittement

effet, les programmes étant récursifs, les arborescences fonctionnelles correspondantes sont explorées plusieurs fois. Le nombre de messages NENR est important également puisque plus il y a de noeuds fonctionnels explorés et plus ce type de messages est généré.

- Le pourcentage de messages liés à la réduction est très important pour la multiplication de polynôme (90 %) et la multiplication de matrices (97 %). Ce pourcentage croît proportionnellement à la taille de la donnée traitée. En effet, ce type de programme consiste à traiter de façon identique plusieurs parties de la donnée. Dans la multiplication de matrices, le produit scalaire est répété autant de fois qu'il y a de composantes dans la matrice résultat; dans la multiplication de polynômes, le nombre de fois où on effectue la multiplication de monômes augmente proportionnellement aux nombres de monômes initialement dans chaque polynôme. Le nombre de messages d'exploration est identique quelle que soit la taille de la donnée traitée. Le deuxième facteur qui explique le nombre de messages NRNR est le nombre de copies effectuées ainsi que la taille des données copiées.

- Le nombre de messages de fin d'exploration et d'acquittement est important surtout pour les programmes récursifs. Dans la version étendue du modèle, ce type de messages n'existe plus. L'exploration est donc moins synchronisée. Les messages d'activation et les messages NENR correspondant sont générés plus tôt entraînant l'augmentation du nombre de messages générés puis traités par unité de temps. Le degré de parallélisme maximal est donc plus important.

Ces résultats ont révélé donc l'importance du parallélisme potentiel du modèle P^3. Ce parallélisme constitue une borne majoratrice pour les cas d'évaluation réels i.e où le nombre de processeurs est borné et où la communication est prise en compte. Dans ce cas, bien souvent, l'efficacité d'évaluation et le parallélisme obtenus sont étroitement liés à la répartition des données sur les sites. La section suivante introduit donc une étude préliminaire de la répartition des données et des programmes dans le modèle P^3.

3 Etude du placement dans le modèle P^3

Dans cette section, nous tentons de déterminer le type de placement adéquat pour les structures permettant la représentation d'une expression i.e les arborescences fonctionnelles et les arbres de données. Nous distinguons deux types de

placement : le placement statique et le placement dynamique. Dans la suite de cette section nous présentons un bref aperçu de ces deux types de placement.

3.1 Le placement statique

Le placement statique des données ou des processus consiste à trouver une fonction d'allocation des données (resp. des processus) sur les processeurs et ce, au chargement du programme ou à la phase de compilation. Le choix de ce style de placement nécessite une connaissance de l'évolution des données (resp. des processus) lors de l'exécution. Il existe plusieurs méthodes pour mettre en oeuvre le placement statique :

- La théorie des graphes [Sto77] qui modélise le problème du placement par un graphe regroupant processus et processeurs et qui tente de résoudre le problème en cherchant une fonction, associant à chaque processus un processeur, qui minimise le coût total de communication.

- L'utilisation de la programmation mathématique i.e la programmation linéaire, l'optimisation combinatoire et la programmation dynamique. Le principe de ces méthodes consiste à exprimer le problème du placement par une équation non linéaire avec contraintes dont la résolution est une optimisation du coût du placement. Cette méthode permet d'énumérer toutes les solutions intéressantes et détermine la solution optimale. Le coût de mise en oeuvre de telles méthodes augmente exponentiellement avec l'augmentation de la taille du problème à résoudre.

- Le troisième type d'approche utilisé est l'utilisation d'heuristiques qui par des approximations, permet l'obtention d'une solution optimale. Parmi les heuristiques utilisées, nous distinguons le *groupement de processus* [SE86] qui consiste à déterminer un groupement initial et à l'optimiser en effectuant des échanges de processus entre groupes; *la limitation du routage* en plaçant les processus qui communiquent fréquemment sur des processeurs liés physiquement dans la machine [Bok81, AP88] et la *méthode du recuit simulé* ou méthode de *Monte Carlo* qui fût utilisée initialement en mécanique statistique, pour simuler l'évolution de l'état d'un solide en fonction de la température.

  Une synthèse des méthodes utilisées pour le placement statique se trouve dans [TM90]

3.2 Le placement dynamique

Le placement dynamique des données (resp. des processus) est utilisé dans le cas d'applications parallèles dont le comportement est difficilement prévisible, voire même impossible à prévoir, même en plaçant de façon optimale les données (resp. les processus) initialement.
Le placement dynamique par opposition au placement statique s'effectue en cours d'exécution. Il vise à améliorer l'efficacité d'exécution des programmes en optimisant l'utilisation des processeurs. Ce type de placement, pour être efficace, doit satisfaire les objectifs suivants :

- Utiliser les processeurs au maximum pour améliorer le temps d'exécution en effectuant des migrations de données ou de processus afin d'équilibrer la charge des processeurs.
- Respecter la localité des traitements en minimisant la distance entre les processeurs qui échangent fréquemment de l'information.

L'équilibrage de la charge de travail des processeurs nécessite une estimation de la charge des processeurs. L'efficacité de la migration dépend de l'exactitude des informations de charge recueillies : plus les informations de charge recueillies sont récentes et plus la chance de placer correctement les processus ou les données migrés augmente. L'estimation de la charge de travail crée un surcoût dû à l'échange de messages d'estimation de la charge. Une bonne méthode de répartition dynamique doit permettre un compromis entre la fiabilité des informations de charge et le surcoût engendré.

Il existe des variantes quant aux méthodes d'estimation de la charge : des méthodes basées sur la charge mémoire [XH, LK87a], calculée en nombre de demandes d'accès ou en taux d'occupation, des méthodes qui tiennent compte de la charge de travail [Dow91, KW91] et des méthodes mixtes qui combinent les deux critères.

La fréquence d'estimation de la charge des processeurs est également une variante du placement dynamique. Il existe plusieurs variantes dont voici quelques exemples : un processeur peut communiquer sa charge :

- périodiquement après chaque intervalle de temps [XH].
- périodiquement à chaque modification de charge.
- à la demande d'un autre processeur [Dow91, ELZ86].

- ou lorsque sa charge dépasse un seuil.

3.3 Le placement dans le cas des langages fonctionnels

Les implantations parallèles des langages fonctionnels sont, pour la plupart, basées sur la réduction de graphe parallèle. Nous pouvons citer comme exemples, les machines suivantes : MARS [CCC+87, CDL87], FLAGSHIP [Sar86, WW87], REDIFLOW [KL84, LK87b] et GRIP [PCSH87].

Dans la réduction de graphe, il s'agit de placer le graphe syntaxique sur l'ensemble des processeurs. A la différence d'autres applications, il n'y a quasiment pas de distinction entre le placement de tâches et le placement des données. En effet, les données à placer dans ce cas sont les graphes syntaxiques. Les tâches sont les processus de réduction associés aux graphes syntaxiques.

La mise en oeuvre de l'exécution des langages fonctionnels fait partie de ces applications où l'évolution des données est difficilement prévisible au stade de la compilation. En effet, la succession des applications de fonction entraîne une constante modification des données. Typiquement les problèmes de dynamicité des données peut être illustré par la figure 7.11 où est représentée une donnée en double exemplaire, par une structure arborescente quelconque. Le placement du graphe dans les deux cas est indique par les numeros p_i sur les sous arbres. Les numéros de réduction R_i, indiquent la chronologie de prise en compte des réductions. Deux cas sont possibles :

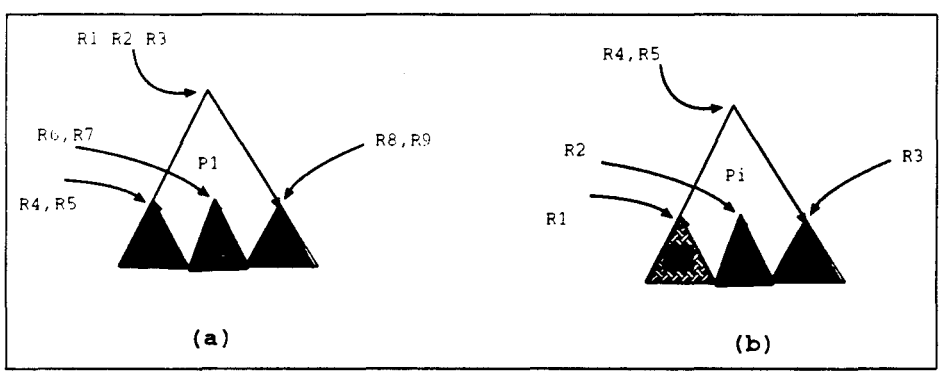


Figure 7.11 : Illustration de la remise en cause permanente du placement des données en cours d'évaluation

- **Cas 1** illustré par la figure 7.11.a
  Dans ce cas les premières requêtes de réduction, R1, R2 et R3, concernent

l' arbre de données tout entier. Le placement de tout l'arbre sur un même processeur P_1 est idéal initialement. Si un tel placement est adopté, les traitements des requêtes R4, R5 ... etc R9 se fera en séquentiel alors que pour ces réductions, l'idéal aurait été d'avoir les sous arbres concernés par les mêmes requêtes de réduction sur des processeurs distincts.

- **Cas 2** illustré par la figure 7.11.b
  Les premières requêtes de réduction (R1,R2 et R3) concernent 3 sous arbres distincts. Le placement de ces 3 sous arbres respectivement sur les processeurs P1, P2 et P3, permet de conserver sur chaque site les noeuds concernés par les mêmes réductions. Ces réductions peuvent donc s'effectuer en parallèle **sans communication**, les nœuds étant sur 3 sites distincts. Le problème survient pour le traitement des requêtes R4 et R5 qui concernent l'arbre de données tout entier. Pour ces dernières, l'idéal serait d'avoir l'arbre de données tout entier sur un même processeur pour ne pas générer de communication.

Il en résulte qu'un placement statique aussi optimal qu'il soit est compromis. Le type de placement adéquat pour les langages fonctionnels est donc un placement dynamique. L'objectif d'un tel placement est de faire en sorte que le plus possible, tout graphe concerné par une requête de réduction soit entièrement présent sur un même processeur. Ceci se fait par des migrations de sous graphes.

Dans le modèle P^3, il s'agit de placer deux types de structures : les arborescences fonctionnelles et les arbres de données.

- Les arborescences fonctionnelles sont statiques i.e une fois construites - même en cours d'exécution- elles ne sont plus modifiées. Un placement statique convient dans ce cas.

- Les arbres de données sont les structures dynamiques du modèle P^3 et subissent les réductions successives commandées par l'exploration des arborescences fonctionnelles. Selon la position des fonctions dans une arborescence fonctionnelle, les réductions concernent tout ou partie de l'ensemble des arbres de données. Nous retrouvons donc les mêmes phénomènes de dynamicité décrits plus haut pour les graphes syntaxiques. Pour permettre une exécution efficace des programmes, il faut assurer d'une part la proximité des arbres de données et des fonctions qui leurs sont appliquées et d'autre part la présence sur un même processeur, de tous les noeuds, d'un arbre de données, concernés par les mêmes réductions. Les liens entre les fonctions

et les arguments se créent dynamiquement par le mécanisme d'exploration, il est donc impossible de concevoir un placement statique efficace.

Dans le cadre de la simulation, nous nous sommes intéressés à deux points précis liés à l'étude du placement :

- L'influence du placement initial des arborescences fonctionnelles et des arbres de données. Cette étude a permis d'une part d'expérimenter plusieurs placements initiaux des données et des fonctions et d'autres part, de confirmer qu'un placement dynamique pour les données est le style de placement adéquat.

- Le placement des copies : dans le modèle P^3, quand deux fonctions s'appliquent à un même argument, représenté initialement par un arbre de données, celui-ci est dupliqué. Selon le mode de duplication mis en oeuvre, il le sera au moment où les deux fonctions sont explorées, si la copie est systématique, ou au plus tard dès que l'une des fonctions appliquées peut modifier l' arbre de données qui le représente, si la copie se fait de façon paresseuse. Une copie de donnée est donc associée à un ensemble de réductions. Une copie de donnée, vue sous cet angle peut être utilisée comme une forme de migration de tâches. Dans ce cas, un placement dynamique efficace des copies peut contribuer à l'amélioration du placement dynamique des données.

  Remarque : Tous les résultats présentés dans la suite de ce chapitre sont obtenus en simulant le modèle P^3 sur 16 processeurs.

Remarque
Ces expérimentations seront menées sur les programmes de la multiplication de matrices et de polynômes. En effet, pour observer l'influence du placement des copies sur l'évaluation, les données copiées doivent être de taille importante.

3.4 Influence de la répartition initiale

Pour les arborescences fonctionnelles, nous avons choisi un placement statique par chemins séquentiels. Ce placement se justifie par le fait que l'exploration d'un chemin est effectuée de manière séquentielle et donc un placement d'un même chemin sur plusieurs processeurs engendrerait plus de communication qui nuierait aux performances d'évaluation de l'expression fonctionnelle.

Pour les arbres de données, nous avons expérimenté 3 types de placement :

- Un placement groupé des arbres de données initiaux. Ce placement présente deux variantes :

  - Un placement de tous les arguments du même programme sur un seul processeur (RGI1)
  - Un placement des arguments à raison d'un argument par processeur (RGI2)

- Un placement des arbres de données par quantum : Chaque argument est réparti sur plusieurs processeurs. Le nombre de processeurs concernés dépend de la taille de la donnée. Plusieurs cas sont possibles :

  - Si la donnée est de petite taille (moins de 10 nœuds), la donnée est placée sur un seul processeur.
  - Si par contre la taille de la donnée est grande, la donnée est répartie sur plusieurs processeurs à raison d'un quantum par processeur. La valeur du quantum dépend d'un pourcentage de répartition P, fixé dès le départ. La valeur de P peut changer d'une simulation à une autre.

  Le découpage d'un argument en quanta se fait en privilégiant un regroupement en profondeur d'abord. Si le pourcentage de découpage est bien choisi, le regroupement se fait par lignes, comme le montre la figure 7.12 i.e il se fait par sous arbres entiers.

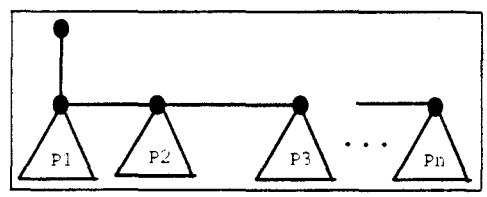


Figure 7.12 : Répartition par quantum

- Un placement complètement dispersé où chaque paire de noeuds de données voisins sont sur deux sites différents. Ce placement, certes inefficace, permet simplement d'avoir une borne de comparaison des placements cités plus haut.

Pour toutes les simulations effectuées, aucun algorithme de migration dynamique des données n'est mis en oeuvre. En revanche, le même algorithme de placement des copies est adopté afin de ne pas perturber l'analyse des résultats obtenus.

Dans un premier temps, nous comparons les résultats obtenus avec les deux répartitions initiales groupées RGI1 et RGI2. Les figures 7.13.a et 7.13.b comparent les taux de parallélisme obtenus, en fonction du temps, pour les deux répartitions groupées RGI1 et RGI2. Pour les deux programmes utilisés, la répartition RGI2 donne des résultats meilleurs. Ceci s'explique essentiellement par le temps de copie qui est moins important dans le cas de la répartition RGI2 puisque les données sont plus distribuées et les copies peuvent être parallélisées et se font donc plus vite.

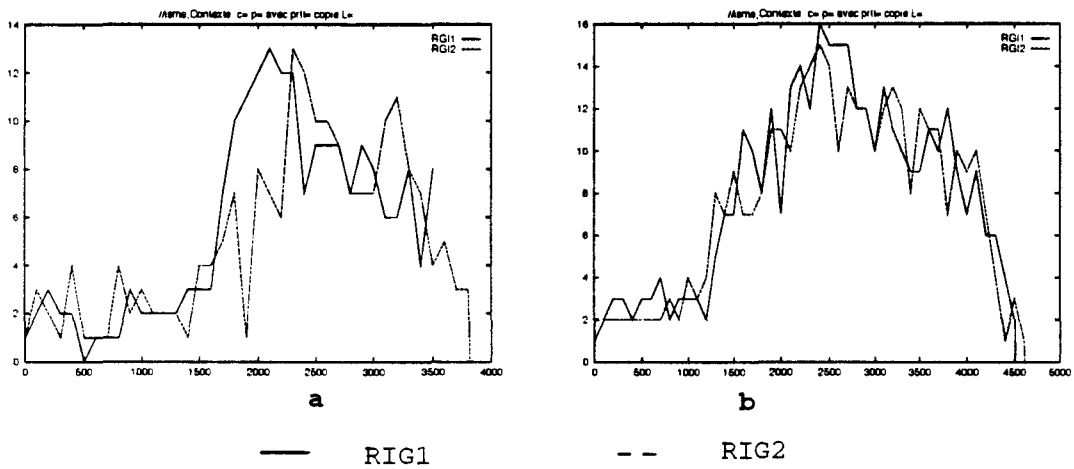
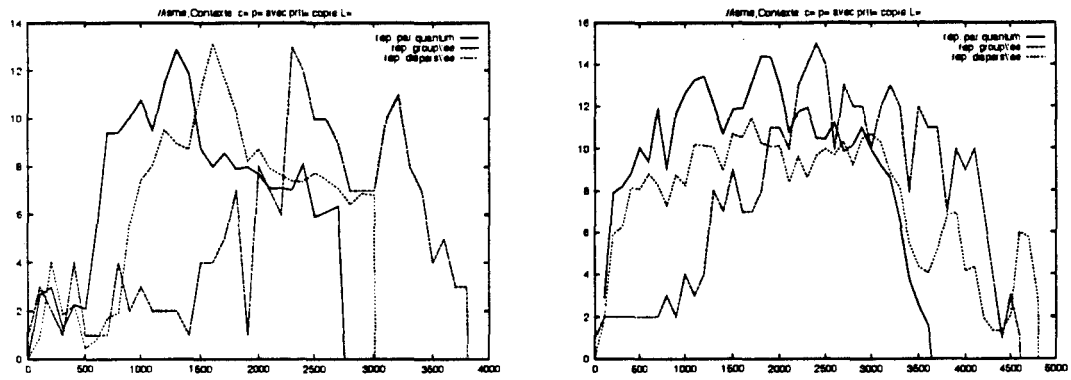


Figure 7.13 : Comparaison du taux de parallélisme, en fonction du temps, obtenu pour les répartitions initiales groupées RGI1 et RGI2 pour (a). la multiplication de matrices et pour (b). la multiplication de polynômes

Les figures 7.14.a et 7.14.b donnent respectivement les résultats obtenus pour les programmes de la multiplication de matrices et de la multiplication de polynômes avec les répartitions RGI2, la répartition par quantum et la répartiton dispersée. Nous remarquons que la répartition par quanta permet d'obtenir un temps de simulation meilleur et un taux de parallélisme plus élevé dès le début de l'exécution. Ce taux de parallélisme s'explique par le parallélisme de copie possible grâce à la répartition par quanta.

La conclusion de cette première expérience est que la répartition par quantum fait la différence par rapport aux autres méthodes de répartition initiales. Cependant, l'aspect dynamique des données fait que l'effet d'une bonne répartition initiale par rapport à une répartition initiale choisie arbitrairement est vite estompé si la répartition dynamique se fait de façon arbitraire.
Dans la suite de ce chapitre, nous retiendrons la répartition par quanta comme répartition initiale. Le paragraphe suivant étudie l'effet de la variation du quantum.



Répartition par quantum

Répartition groupée (RIG2)

Répartition dispersée

Figure 7.14 : Comparaison du taux de parallélisme obtenu avec 3 répartitions initiales différentes pour la multiplication de matrices (a) et la multiplication de polynômes (b)

3.4.1 Variation du quantum

Afin de déterminer la taille idéale des quanta, nous avons simulé l'exécution des deux programmes avec des pourcentages de découpage de 33 %, 20 % et 6 % donnant un quantum de 1/3, 1/5, et 1/16 de la donnée sur chaque processeur, pour une matrice carrée 10*10 et pour des polynômes de 10 monômes de 10 variables chacun.
Globalement, les courbes illustrant le taux de parallélisme sont quasi identiques. On arrive donc à la même conclusion que précédemment. Cependant, sur les 1000 premières unités de temps, le comportement des simulations diffèrent. En observant simultanément les courbes de parallélisme (figure 7.15), de communication (figure 7.16) et d'activités des processeurs actifs (figure 7.17), nous pouvons faire les constatations suivantes :
la répartition avec un pourcentage de 6 % comparé aux autres pourcentages donne le plus grand taux de communication. Le taux de parallélisme est assez élevé quoique irrégulier (chute brutale que nous pouvons expliquer par une copie distante d'un sous arbre vu le taux de communication). La courbe d'activité des processeurs est faible. Ces observations sont caractéristiques d'un grain de parallélisme trop fin.

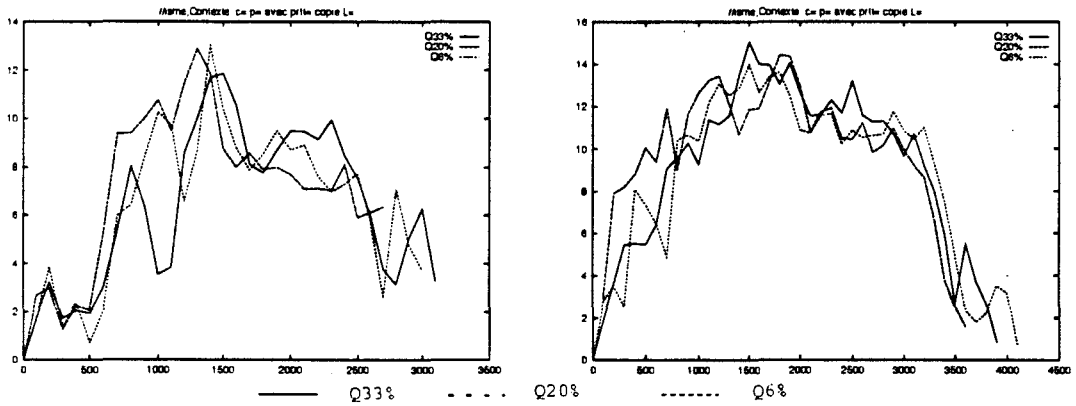


Figure 7.15 : Variation du quantum : Courbes illustrant le taux de parallélisme

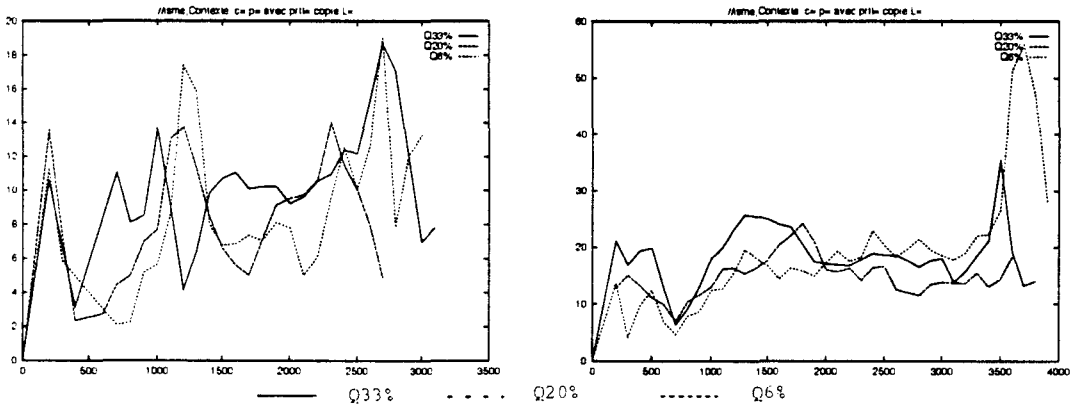


Figure 7.16 : Variation du quantum : Courbes illustrant le taux d'activité des processeurs actifs

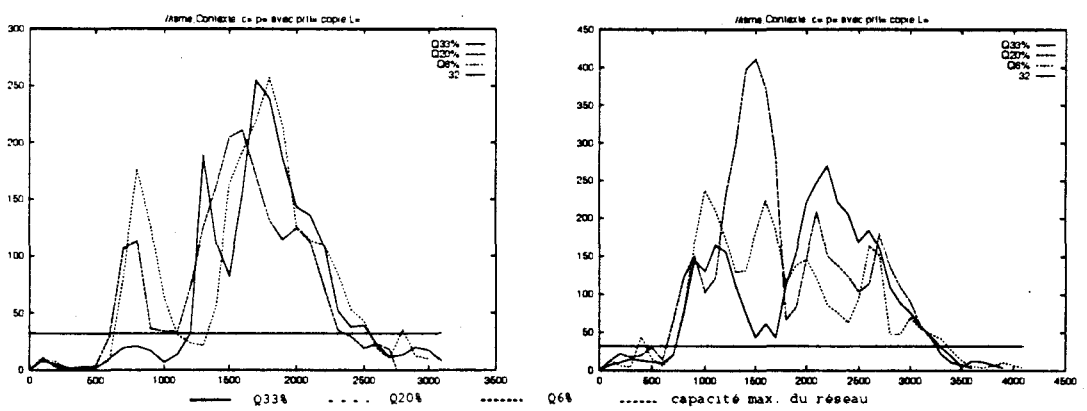


Figure 7.17 : Variation du quantum : Courbes illustrant le taux d'activité des processeurs actifs

La répartition avec un pourcentage de 33 % engendre le moins de communication comparée aux autres simulations. Le parallélisme est moins élevé au départ et le taux d'activité des processeurs actifs est le plus élevé. Ces observations révèlent un regroupement excessif qui limite le taux de parallélisme en début d'exécution.

La répartition initiale avec un pourcentage de 20 % donne le taux de parallélisme le plus élevé dès le début de l'exécution avec un taux de communication comparable à celui du cas précédent. De plus l'activité des processeurs est relativement stable et assez élevée. Cette taille de quantum est donc un compromis entre les deux cas précédents. Nous pouvons expliquer ce résultat par le fait qu'une répartition de 20% équivaut à un regroupement "par ligne" i.e pour la multiplication de matrices, la matrice finale est répartie par ligne sur les processeurs. Pour la multiplication de polynômes, le polynôme résultat est réparti à raison d'un monôme par processeur. Cette répartition favorise, au départ, la réduction en parallèle. De plus, les premières copies réalisées peuvent se faire en parallèle du fait que les données sont initialement distribuées.

Conclusion

L'étude de l'influence de la répartition initiale sur l'évaluation d'une expression dans P^3 a montré qu'une répartition initiale optimale n'a pas beaucoup d'impact sur le déroulement de l'évaluation si ce n'est sur la phase d'initialisation de l'évaluation i.e la phase d'application des premières fonctions. Ceci se voit sur les courbes de comparaison du taux de parallélisme dans le cas de répartitions initiales l'une optimale et l'autre entièrement groupée. Ces résultats confirment qu'une méthode de répartition statique des données n'est pas adéquate pour le modèle P^3.

Nous abordons dans le paragraphe suivant un deuxième aspect de l'étude de la répartition dans le cadre de P^3 et qui concerne la répartition dynamique des copies.

3.5 Gestion des copies

Dans cette section, nous étudions le placement dynamique des copies. L'idée de la méthode proposée est de copier les données de telle façon que les réductions effectuées sur les copies se fassent en parallèle en engendrant le moins de communication possible.
Dans cette section, nous présentons deux méthodes de placement des copies : la première méthode est basée sur le regroupement entier des copies sur les processeurs. La deuxième est une optimisation de la première méthode. Elle permet de copier les données avec un regroupement contrôlé par un seuil.

Afin d'apprécier les résultats obtenus avec les deux méthodes, nous avons choisi une méthode de placement des copies que l'on appelera *méthode de placement témoin* et qui ne vise pas particulièrement à optimiser le placement des copies.
Dans la méthode témoin, l' arbre de données est copié et réparti sur les processeurs de la même manière que l'est l'arbre de données originel. Le seul atout d'une telle méthode est que la phase de placement de la copie s'effectue sans communication. L'inconvénient de cette méthode est que le placement des copies est très dépendant du placement des données copiées. Si la donnée au moment de la copie est bien placée, la copie le sera également et vice versa.
Dans le cas des deux méthodes proposées, le choix du processeur est important. On choisira toujours le processeur le moins chargé. Dans le simulateur, la charge des processeurs est mesurée en termes de messages, -tous types confondus- que celui-ci doit traiter.
Le choix d'un processeur en se basant sur la charge de travail suppose une estimation de la charge des processeurs est effectuée. Dans le simulateur, nous supposons que chaque processeur détient des informations sur la charge qui se rapprochent de la réalité.

3.6 Placement entièrement groupé

3.6.1 Description

Le placement entièrement groupé des copies (PEGC) consiste à recopier entièrement la copie sur un seul processeur quelle que soit la répartition de l'original. On peut illustrer ce procédé par la figure 7.18. Comparé à la répartition témoin, nous obtenons moins de communications mais moins de parallélisme pour les programmes pris comme exemple comme le montre la figure 7.19. Le temps de simulation est 2 fois plus long.

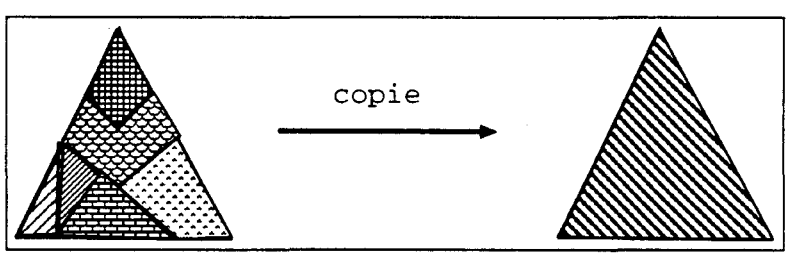


Figure 7.18 : Placement entièrement groupé

Constatation En observant la courbe de parallélisme et celle de l'activité moyenne des processeurs actifs (cf. figure 7.19), nous constatons que le peu de processeurs qui travaillent sont surchargés.

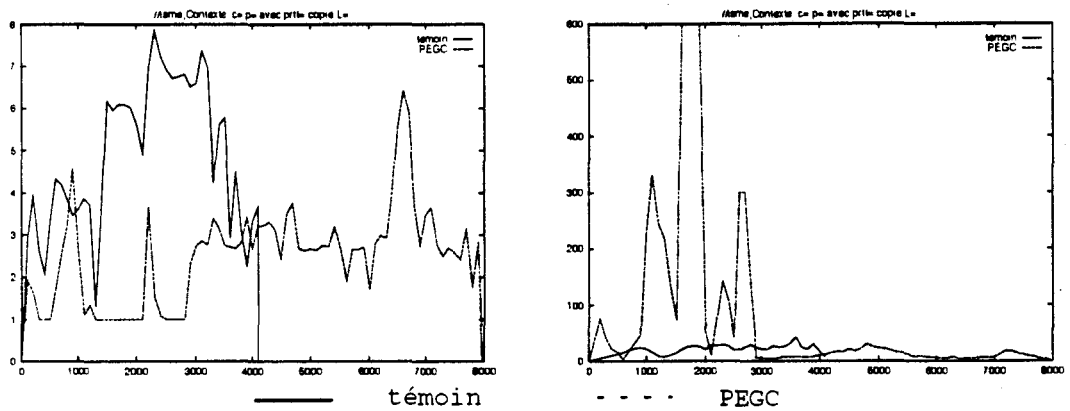


Figure 7.19 : La multilication de matrices : comparaison des taux de parallélisme et des activités des processeurs actifs, obtenus avec la répartition témoin et la répartition PEGC

3.6.2 Analyse des résultats

- Le temps d'exécution est deux fois plus long avec la méthode proposée. Le temps écoulé à effectuer des copies est à l'origine de ces résultats. En effet, les copies de nœuds dirigés vers le même processeur ralentissent le temps global de copie (étapes 2 et 3 de la copie cf. section 1.1.2). Tant que la copie n'est pas terminée, aucune réduction ni sur la copie ni sur l'original n'a lieu ce qui retarde les réductions.

- Les processeurs actifs sont surchargés et le parallélisme est faible. Ceci révèle un excès de regroupement. En effet, nous nous retrouvons dans la situation décrite par la figure 7.11.a où des réductions qui doivent en principe être exécutées en parallèle, sont effectuées en séquentiel surchargeant ainsi, les processeurs actifs (ceux qui ont reçu les copies tout au long de l'exécution) et réduisant le degré de parallélisme.

La répartition témoin est effectivement meilleure. Nous avançons à cela les raisons suivantes : la copie se fait de manière répartie, localement sur chaque processeur. De ce fait, elle s'effectue plus vite, sans communication, et permet donc la reprise des réductions rapidement. De plus bien que naïve, et peut-être pas optimale, cette méthode entraîne une copie sur plusieurs sites si tel est le cas pour la donnée originelle, ce qui permet l'obtention de granules de taille plus réduits favorisant ainsi le parallélisme. Cette méthode ne régularise cependant pas la taille des granules. En effet, une donnée entièrement regroupée sur un même processeur entrainerait les mêmes conséquences que la méthode PEGC. Ceci est illustré

par la figure 7.20 où l'évaluation de la même expression a été simulée avec une répartition initiale groupée RGI2 (cf. 3.4) et en appliquant la répartition témoin et la répartition PEGC.

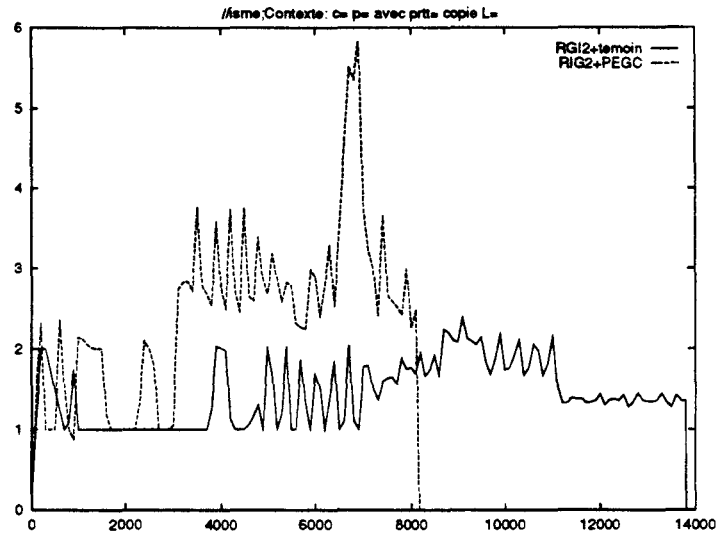


Figure 7.20 : La multiplication de matrices : comparaison des taux de parallélisme obtenus avec la répartition PEGC et la répartition témoin avec une répartition initiale groupée

3.6.3 Conclusion

Cette première expérience a révélé la nécessité de réduire le temps de copie pour gagner en efficacité. Elle nous permet de déduire que le regroupement entier des copies est excessif. L'analyse de la relative efficacité de la méthode témoin nous amène à envisager une répartition des copies avec seuil. Ce seuil aura pour effet de contrôler la granularité des copies et par conséquent celle du parallélisme, et permettra ainsi d'arriver à un compromis.

3.7 Placement groupé avec seuil

3.7.1 Description de la méthode

Le placement groupé avec seuil des copies (PGSC) consiste à placer sur des sites distincts, la copie d'un argument par sous arbres entièrement groupés et ce indépendamment de la répartition de l'argument copié (figure 7.21).

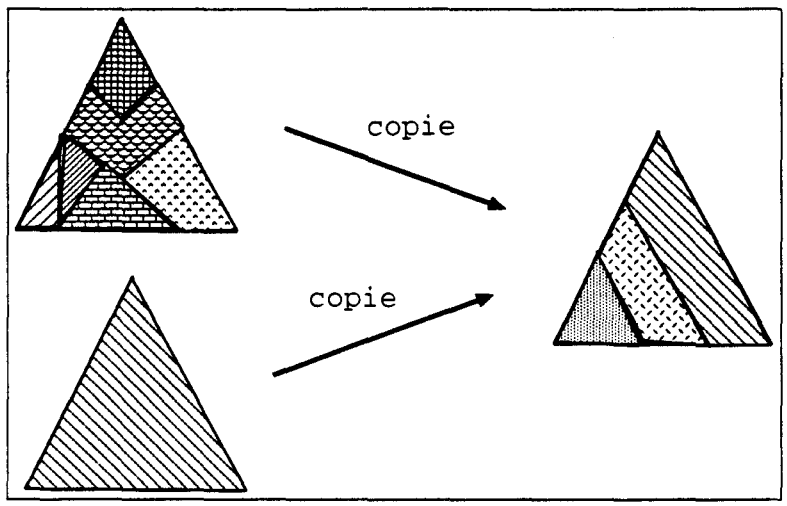


Figure 7.21 : Placement groupé avec seuil

A cet effet, un seuil de distribution - fixé au début d'une simulation- détermine la taille des sous arbres copiés chacun sur un processeur. En effet, plus la valeur du seuil s est grande et plus la taille des granules est grande et inversement. Un choix convenable du seuil de distribution conduirait à un regroupement idéal pour satisfaire les deux objectifs que l'on projette d'atteindre. En effet, la donnée sera suffisamment groupée par sous arbres pour effectuer des réductions sans communications. Elle est en revanche assez distribuée pour permettre des réductions en parallèlle.

En comparant les taux de parallélisme obtenus avec les méthodes PGSC et PEGC, pour les deux programmes expérimentaux, nous pouvons émettre les constatations suivantes (figure 7.22) :

- Le taux de parallélisme est plus important dès le début de la simulation (cf. figure 7.22).
- Un taux de communication moins important (cf. figure 7.23)

3.7.2 Analyse des résultats

L'amélioration du taux de parallélisme par rapport à celui obtenu avec la méthode précédente s'explique d'abord par le fait que les copies distribuées prennent moins de temps à faire et bloquent donc moins longtemps la reprise des réductions. Ces copies s'effectuent en parallèle sur les processeurs et provoquent elles mêmes une hausse du taux d'activité moyen des processeurs et donc une hausse

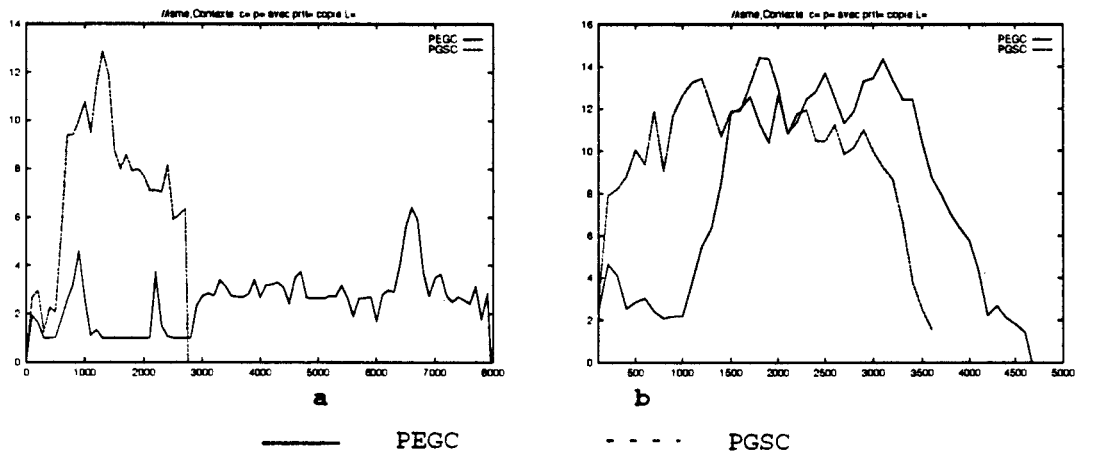


Figure 7.22 : Courbes de parallélisme : Comparaison du placement groupé des copies avec et sans seuil pour la multiplication de matrices(a). et de polynômes (b).

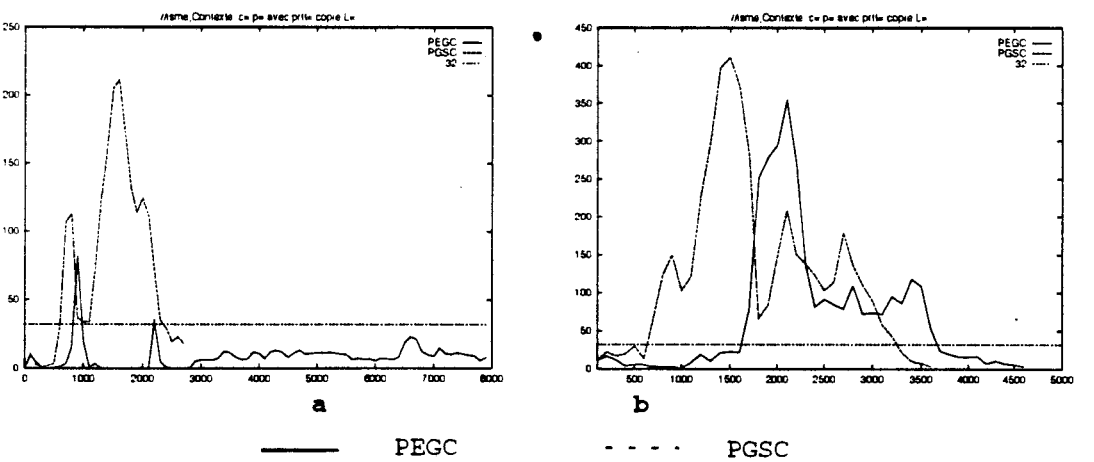


Figure 7.23 : Courbes des taux de communication : Comparaison du placement groupé des copies avec et sans seuil pour la multiplication de matrices(a). et de polynômes (b).

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R1	D	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0
	A	0	0	10	0	10	0	30	10	10	0	10	0	10	0	10	0
R2	D	10	10	0	0	0	0	10	0	10	0	10	10	10	10	10	10
	A	0	10	0	10	0	10	10	20	0	10	0	0	0	20	0	10
R3	D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	A	10	0	10	0	10	10	0	10	0	10	0	10	0	10	0	10

Figure 7.24 : Effet régulateur du seuil : la multiplication de matrice

du taux de parallélisme.

La copie s'effectue par des regroupements partiels de sous arbres. Le parallélisme obtenu se justifie alors aussi par le fait que plusieurs réductions ont pu se faire simultanément du fait de la distribution des copies par groupes de nœuds adjacents.

3.7.3 Effet régulateur du seuil

L'introduction d'un seuil permet le contrôle permanent du grain de parallélisme qui s'active lors de la réalisation de la copie. Cet effet régulateur du seuil est visible par l'observation de la répartition du résultat de l'évaluation sur les processeurs. En partant initialement de trois répartitions initiales différentes :

- la répartition RGI1 (R1),
- la répartition par quanta avec un quantum égal au 1/5 de la donnée (R2),
- et la répartition dispersée (R3)

qui sont représentatives des cas extrêmes (R1 et R3) et du cas idéal (R2), et en pratiquant un placement groupé avec seuil des copies (PGSC), nous obtenons les tableaux 7.24 et 3.7.3 pour les programmes expérimentaux, qui illustrent la répartition des lignes, en pourcentage, sur 16 processeurs, au début (lignes D des tableaux) et à la fin de la simulation (lignes A des tableaux), pour les 3 répartitions initiales et pour les deux programmes expérimentaux.

Analyse des résultats

Pour la multiplication de matrices, le pourcentage de lignes entièrement groupées sur les processeurs est de 100%, 100% et 90% respectivement pour les répartitions

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R1	D	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0
	A	4	4	14	2	5	5	6	3	6	3	7	8	1	5	12	4
R2	D	30	0	0	0	20	0	30	0	30	0	0	20	0	20	20	30
	A	7	8	7	5	1	5	5	12	4	2	7	6	6	9	2	10
R3	D	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	A	0	7	4	10	2	6	12	6	12	3	4	7	5	4	5	1

Figure 7.25 : Effet régulateur du seuil : la multiplication de polynômes

initiales R1, R2 et R3.

Pour la multiplication de polynômes, le pourcentage de lignes entièrement groupées sur les processeurs est de 89%, 96% et 88% respectivement pour les répartitions initiales R1, R2 et R3.

Dans le cas de la répartition par quanta, ceci reflète bien l'effet régulateur du seuil : le seuil permet le contrôle de la granularité.
Les résultats obtenus pour les cas extrêmes R1 et R3 prouvent qu'en plus de l'effet régulateur, le seuil a un effet correcteur et permet donc de rattraper une répartition initiale non adaptée.

Pour illustrer cet effet correcteur, nous avons comparé les taux de parallélisme obtenus pour la méthode PGSC et pour la méthode témoin -qui ne corrige pas la répartititon initiale- pour une répartition initiale entièrement groupée, pour les deux programmes expérimentaux. La figure 7.26 montre ce résultat.

La figure 7.27 illustre également l'effet correcteur du seuil. Les résultats obtenus avec une répartition initiale trop groupée ou trop dispersée comparés à ceux obtenus avec une répartition initiale par quantum sont nettement moins bons (figure 7.27) du fait qu'il faut un temps minimum pour corriger les répartitions initiales, temps durant lequel les imperfections de la répartition initiale influent sur le fonctionnement. Nous constatons néanmoins que, sur les 1200 premières unités de temps unitaires de ces 3 simulations (cf. figure 7.27) , l'aspect global des courbes de parallélisme est quasiment identique- avec un décalage dans le temps bien sûr-. Ceci nous conforte dans notre analyse.

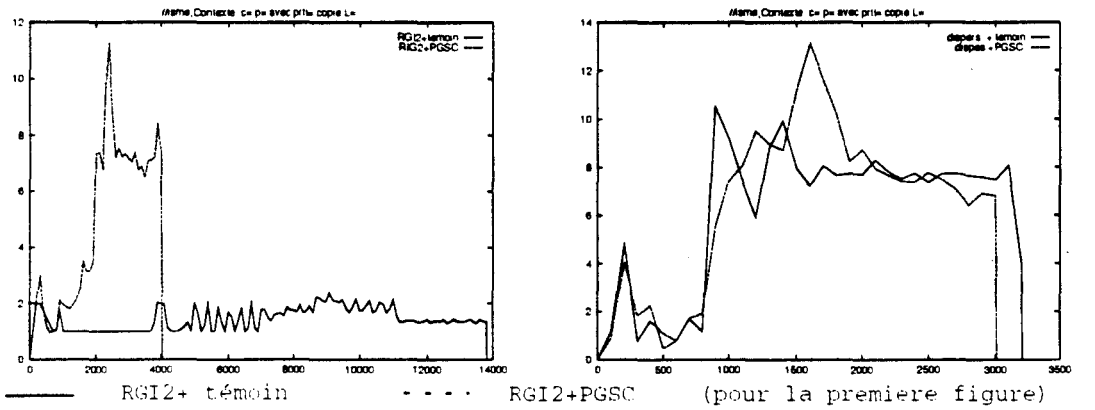


Figure 7.26 : Effet correcteur du seuil : Comparaison du taux de parallélisme obtenu avec les répartitions témoin et PGSC pour (a). la multiplication de matrices et (b).la multiplication de polynômes

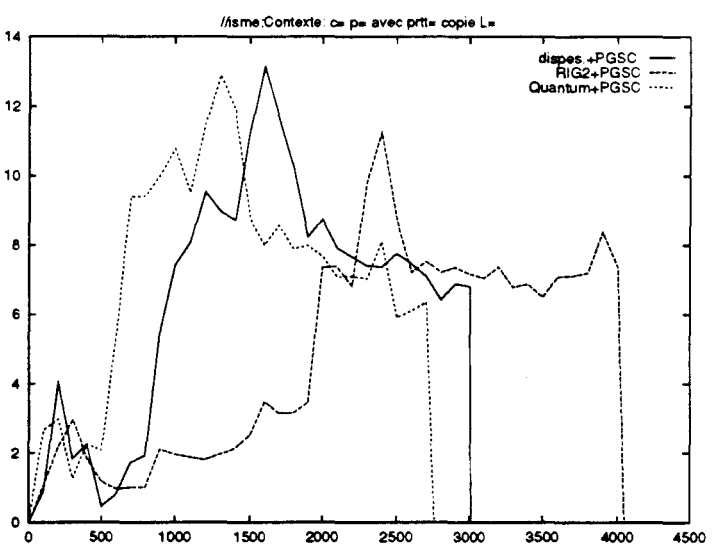


Figure 7.27 : Multiplication de polynômes : Comparaison du taux de parallélisme obtenu avec 3 répartitions initiales différentes et la répartition des copies avec seuil

3.7.4 Variation du seuil dynamique

La valeur du seuil détermine la taille des granules sur les processeurs. Pour un problème particulier, avec des données d'une certaine taille, la valeur idéale du seuil est celle qui permet une répartition logique des copies i.e telle que chaque sous arbre de données concerné par un ensemble de réductions se retrouve entièrement sur un processeur. Pour la multiplication de matrices et la multiplication de polynômes, nous avons expérimenté plusieurs valeurs du seuil. Pour les tailles de données choisies (matrices de dimension 10 pour le premier et des polynômes de 10 monômes chacun ayant 10 variables), la taille idéale du seuil est égale à 100. Pour des valeurs du seuil comprise dans un intervalle autour de la valeur 100, les courbes du taux de parallélisme obtenues ne présentent pas de différences fondamentales mais la correction de la répartition initiale n'est pas aussi efficace. Ceci se voit à la diminution du pourcentage de lignes entièrement groupées sur les processeurs. Pour tenter de comprendre l'importance du choix de la valeur du seuil, nous avons choisi de commenter les courbes de simulation obtenues pour les valeurs suivantes du seuil : 1 (pas de regroupement), 100 (valeur idéale du seuil étant donnée la taille des données) et ∞ (regroupement total).
Les courbes décrivant le degré de parallélisme obtenu, la charge moyenne des processeurs actifs et la charge du réseau avec les choix des valeurs suivantes pour le seuil sont données par la figure 7.28, 7.29 et 7.30.

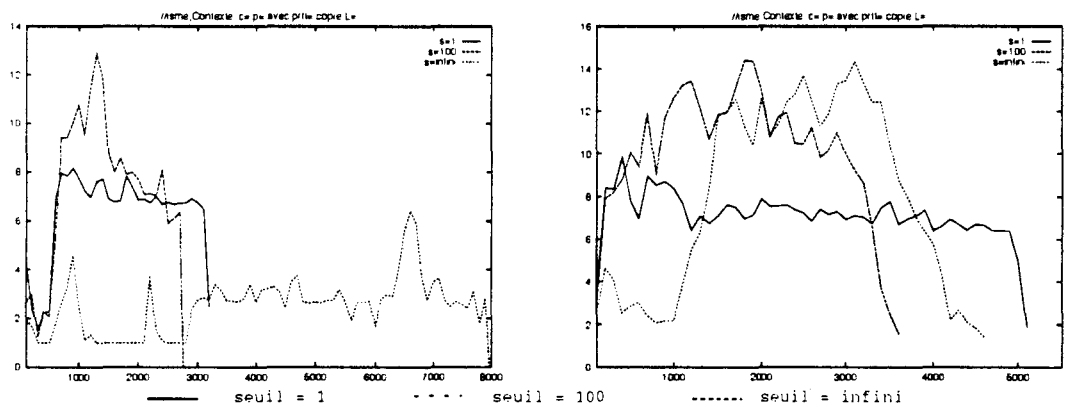


Figure 7.28 : Variation du seuil dynamique de répartition(taux de parallélisme) (a) pour la multiplication de matrices et (b) pour la multiplication de polynômes

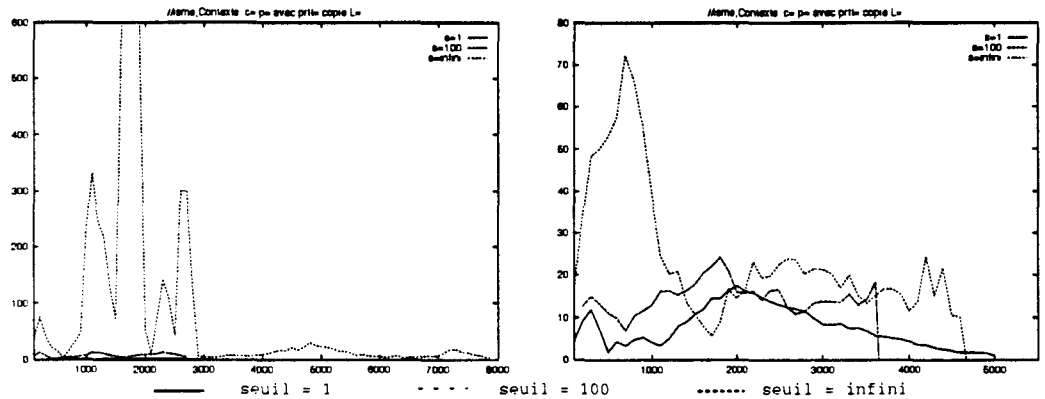


Figure 7.29 : Variation du seuil dynamique de répartition (charge de travail des processeurs actifs) (a) pour la multiplication de matrices et (b) pour la multiplication de polynômes

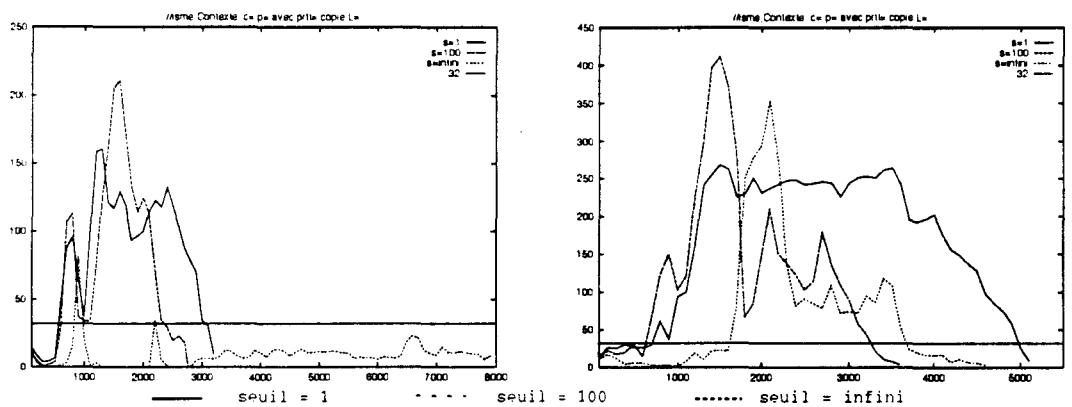


Figure 7.30 : Variation du seuil dynamique de répartition (état du réseau) (a) pour la multiplication de matrices et (b) pour la multiplication de polynômes

Analyse des résultats

- seuil = 1
  Avec un seuil de 1, une copie est complèment distribuée sur les processeurs de telle façon que deux nœuds voisins ne se trouvent pas sur le même processeur. Ce seuil de répartition des copies permet l'obtention d'un bon taux de parallélisme dû à la réalisation, en parallèle, de chacune des 3 phases de copie. Le taux de communication est le plus élevé, indiquant la finesse du grain.

- seuil = ∞

  Le comportement des deux programmes est globalement similaire pour un seuil infini. On observe la faiblesse du taux de parallélisme par rapport à d'autres simulations expérimentant d'autres seuils. Cette faiblesse s'explique comme le montrent les courbes d'activité des processeurs et les courbes des taux de communication par un regroupement excessif traduit par la quantité de travail associé à peu de processeurs actifs et par la faible quantité de communication. Pour la multiplication de polynômes, le parallélisme est faible surtout au départ . Ceci s'explique par le nombre de copies à effectuer qui est plus élevé que celui effectué dans le programme de la multiplication de matrices. Ces copies sont faites sur des processeurs différents et engendrent par conséquent des réductions en parallèle sur les copies.

- seuil = 100
  Le seuil de valeur 100 est un compromis pour les tailles de données choisies. Le taux de communication est ponctuellement plus élevé. Nous pouvons expliquer cela par le placement à distance de plusieurs copies simultanément.

Conclusion

Les résultats de simulation ont révélé qu'un seuil de 100 donne des résultats optimaux dans le cas de la multiplication de matrices de taille 10 et la multiplication des polynômes de taille 10. Ce résultat s'explique par le fait qu'un seuil de 100 permet une répartition logique des données. La valeur optimale du seuil dépend énormément de la taille des données mises en jeu. Ce seuil devra alors être adapté en fonction de la taille des données.

3.7.5 Importance du choix du processeur

Afin d'évaluer l'importance du choix du processeur, nous avons comparé le taux de parallélisme obtenu, avec un choix du processeur recevant la copie qui tient compte de la charge de travail et un choix de processeur aléatoire, tout en adoptant la même méthode de répartition des copies. Les courbes de la figure 7.31 illustrent les résultats obtenus pour les deux programmes expérimentaux [4].

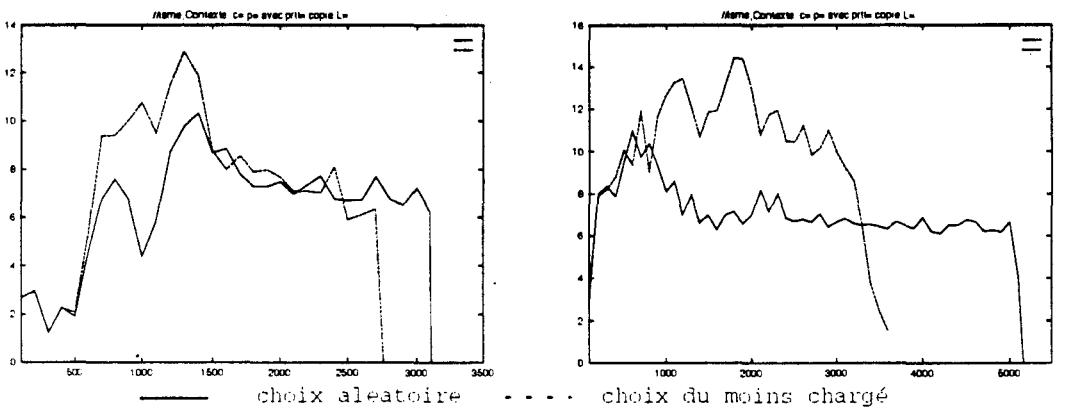


Figure 7.31 : Courbes du taux de parallélisme : Influence du choix du processeur pour les programmes de la multiplication de matrices (a). et la multiplication de polynomes(b).

Nous pouvons constater que le taux de parallélisme obtenu est moins élevé quand le choix du processeur est aléatoire ce qui peut être expliqué assez facilement puisque si des copies consécutives ou simultanées sont placées sur le même processeur. alors la charge de travail du ou des processeurs concerné(s) est trop importante par rapport aux autres processeurs. Les messages mettront donc beaucoup plus de temps à s'éxécuter. D'autre part, ces messages non traités risquent de bloquer d'autres traitements sur d'autres processeurs.

4 Conclusion

La simulation que nous avons décrite dans ce chapitre a donc permis la validation du modèle P^3 et a révélé ses potentialités en matière de parallélisme. Cette simulation a permis de mettre en évidence l'aspect dynamique des données et la nécessité de proposer des méthodes de répartition dynamiques pour le placement des données dans le modèle P^3, et ce dans un cadre plus général que celui auquel

[4]multiplication de matrices et multiplication de polynômes

nous nous sommes intéressés dans ce chapitre i.e qui ne concerne que les programmes manipulant des données de grandes taille.
Nous pouvons néanmoins ajouter que le placement des copies dans le cas de cette catégorie de programmes contribue à l'amélioration du placement dynamique des données. En effet, quelle que soit la solution retenue pour l'implantation de la duplication, la copie d'argument devra tôt ou tard se faire si une réduction devant modifier l'argument doit être effectuée. La méthode de la copie groupée avec seuil , proposée dans ce chapitre, permet de plus de relancer, à chaque copie , le contrôle de la granularité et du taux de parallélisme également.
Enfin, l'application de cette méthode dans une implantation réelle du modèle P^3 dans un contexte distribué nécessiterait l'élaboration d'un processus d'estimation de la charge des processeurs qui se rapprocherait de l'estimation de la charge recueillie de façon simplifiée dans le simulateur.

Conclusion

Pour conclure, nous allons tenter dans un premier temps, de dégager les caractéristiques du modèle P^3 étendu puis, nous aborderons dans un deuxième temps les perspectives de ce travail.

Nous avons donc présenté un modèle d'évaluation parallèle des langages fonctionnels sans variable prenant en compte l'évaluation d'expressions fonctionnelles sans variable et exploitant le parallélisme des langages fonctionnels.

Nous sommes partis au départ de ce travail, d'une première version du modèle d'évaluation P^3, dans laquelle seules les expressions fonctionnelles du langage de J.W. Backus, FP, sont évaluées. Ces expressions ne comprennent ni les fonctions polyadiques, ni les fonctions d'ordre supérieur.

L'extension du modèle P^3 a nécessité les modifications suivantes :

- Au niveau du mécanisme d'exploration, il n'y a plus de synchronisation entre l'exploration des chemins séquentiels d'une même arborescence fonctionnelle, voire même de plusieurs arborescences fonctionnelles.
- L'algorithme d'ordonnancement permet, quel que soit l'ordre d'exploration des noeuds fonctionnels, de reclasser les requêtes de réduction sans avoir recours à une quelconque synchronisation.
- Du fait de la prise en compte des fonctions d'ordre supérieur, les fonctions peuvent être des arguments mais aussi des résultats d'évaluation. Par conséquent, les arborescences fonctionnelles peuvent être dynamiquement construites. Cependant, une fois construites, elles ne sont plus modifiées.
- L'introduction d'un nouvel état pour les noeuds fonctionnels ou de données : l'*état de création partielle* qui permet aussi bien l'anticipation de l'exploration et de la réduction que l'établissement de liens entre plusieurs expressions interdépendantes évaluées simultanément.

Nous avons montré comment, par la représentation de l'expression proposée, le modèle P^3 occasionne un coût de parcours et une occupation mémoire plus réduits et offre les mêmes possibilités pour l'évaluation d'une expression fonctionnelle que

le modèle de réduction de graphe.

Grâce à la simulation du modèle P^3, nous avons pu montrer le potentiel de parallélisme du modèle. Le parallélisme potentiel est en partie expliqué par le parallélisme exploité des langages fonctionnels mais aussi par l'asynchronisme entre les deux activités du modèle : l'exploration et la réduction du modèle.
Nous avons pu déterminer le type de placement qui convient aux structures du modèle : vu la dynamicité des données, un placement dynamique des données est indispensable.
Dans les modèles de réduction, les tâches de réduction sont associées aux données auquelles elles s'appliquent, ce qui permet de conclure qu'il n'y a pas de différence entre le placement des données et celui des tâches de réductions associées. Ceci justifie l'intérêt du placement dynamique des copies et sa contribution à améliorer le placement dynamique des données.
Nous avons donc proposé une stratégie de placement des copies groupées avec seuil qui permet de contrôler le placement dynamique et de corriger le placement initial des données.

Nous avons montré que dans le modèle P^3, l'appel par nécessité et le partage d'expression sont tout à fait réalisables. A court terme, nous envisageons une étude plus détaillée de leurs mise en oeuvre.

Parallèlement aux travaux de simulation et des conclusions que nous avons tirées concernant le placement de données, N. Melab actuellement en préparation de thèse étudie plus en détail, la répartition dynamique mais aussi les optimisations statiques visant à améliorer la répartition dynamiques des données. Cette étude est orientée selon deux axes complémentaires :

- L'optimisation statique du placement consiste à étudier la portée des fonctions constituant les arborescences fonctionnelles i.e des "parties" de la donnée qu'elles "utilisent". Son but est de favoriser la mise en présence de paquets de fonctions et de paquets de nœuds de donnée de façon a ce que les réductions des arguments par ces fonctions puissent se faire sans communication (en augmentant la granularité).

- L'élaboration d'une stratégie dynamique de placement adaptée aux besoins du modèle P^3 en réutilisant la méthode de placement des copies avec seuil.

A moyen terme, une implantation du modèle P^3 sur une machine parallèle est envisageable. Cette implantation permettrait :

- de mettre en oeuvre les méthodes de placement retenues et tester leurs efficacité.
- d'effectuer une réelle comparaison des performances obtenues pour l'évaluation d'une expression dans le modèle P^3 et dans le modèle de réduction de graphe.
- Une optimisation de la gestion mémoire par l'utilisation des algorithmes tels ceux de Bevan [Bev87] et de Lester [Les89], adaptés à la réduction parallèle de graphe et qui sont des algorithmes de récupération mémoire à compteur de références utilisés dans un contexte distribué.

Enfin, afin d'augmenter l'efficacité d'exécution des langages fonctionnels en utilisant comme schéma d'évaluation, le modèle P^3, l'étude de l'utilisation de la compilation des activités du modèle en un programme de bas niveau fait partie de nos préoccupations ...

Références

[Abd76] S.K. Abdali. An abstract algorithme for combinatory logic. *Journal of Symbolic Logic*, 41(1):222–224, March 1976.

[AJ89] L. Augustsson and T. Johnsson. Parallel graph reduction with the <v,g>-machine. In *ACM 1989*, pages 202–212, 1989.

[AP88] A. Andre and J.L. Pazat. Le placement de tâches sur des architectures parallèles. *Technique et Science Informatique*, 7(4):385–401, 1988.

[Bac78] J.W. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of program. *CACM 21*, 8, 1978.

[Bac86] Backus, J.W. Fl language manual (preliminary version). *Research Report RJ 5339 (IBM Almaden Research)*, 1986.

[BB92] K. Benabadji and C. Bey Benyelles. Machine à réduction matricielle pour le langage vaal. In *JFLA92*, Fevrier 1992.

[Bel85] Bellot, P. Jym : un langage de programmation sans variables et ses réalisations. *Actes des journées AFCET-GROPLAN 1985, Bulletin BIGRE+GLOBULE, J.ANDRE ed.*, Juillet 1985.

[Bel86] B. Bellot . *Sur les sentiers du GRAAL, étude, conception et réalisation d'un langage de programmation sans variable.* PhD thesis, Université Pierre et Marie Curie -Paris IV, 1986.

[Bev87] D.I. Bevan. Distributed garbage collection using reference counting. In *PARLE 89*, volume 2, pages 176–187, 1987.

[BJ87] Bellot, P. and Jay, V. A theory for natural modelisation and implementation of functions with variable arity. In *Functional Programming Languages and Computer Architecture*, pages 212–233, Portland, Orego,USA, Septembre 1987.

[BJ88] P. Bellot and V. Jay. Uniformly applicative structures, a theory of computability and polyadic functions. *FST- TCS 88*, December 1988.

[BMS80] Burstall, R., Mc Queen, and Sanella, D. Hope : an experimental applicative language. In *LISP Conference*, pages 136–143, Stanford California, 1980.

[Boh64] C. Bohm. *The CUCH as a formal and description language.* T. STEELE Eds, 1964.

[Bok81] S.H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C–30(No.3):207–214, Mar 1981.

[Bur82] F.W. Burton. A linear space translation of functional programs to turner combinators. *Inf. Procces. Letter*, 14(5):201–204, 1982.

[BVP+87] Barendregt, H.P., Van Eekelen, M.C.J.D., Plasmeijer, M.J., Hartel, P.H., Hertzberger, L.O., and Vree, W.G. The dutch parallel reduction machine project. *North-Holland Future Generations Computers System*, 3:261–270, 1987.

[CCC+87] Castan, M., Contessa, A., Cousin, E., Durrieu, G., Lecussan, B., Lemaitre, M., and Ng, P. Le processeur de réduction de mars, machine à réductions symbolique. In *Journées* c^3, pages 2–3, Juillet 1987.

[CDL87] M. Castan, M.H. Durand, and M. Lemaitre. A set of combinators for abstraction in linear space. *Information Processing Letters*, 24:183–188, 1987.

[CDL87] Castan, M., Durand, M.H., and Lemaitre, M. Le processeur de réduction de mars, machine à réductions symbolique. In *Information Processing Letters*, pages 183–188, North-Holland, 87.

[CF58] Curry, H.B and Feys, R. *Combinatory Logic*, volume 1. North Holland, 1958.

[Chu41] A. Church. *The calculi of Lambda conversions.* Princeton University Press, 1941.

[Cou91] E. Cousin. *Compilation optimisée d'un langage fonctionnel pour une machine parallèle à réduction de graphe.* PhD thesis, Onera-CERT Toulouse, 1991.

[Den90] Dennis. *Proceedings of the fourth conference of the NORTH America Transputers Users Group*, 4:153–163, Octobre 1990.

[Dev90] N. Devesa. *Proposition d'un Schema d'Evaluation Parallèle du Langage Fonctionnel FP sur un Réseau de Processus.* PhD thesis, Université des Sciences et Technologies de Lille, Janvier 90 1990.

[Dow91] S. Dowaji. Répartition dynamique de travail pour le calcul formel. *Rapport de DEA, LMC/IMAG*, Sep 1991.

[ELZ86] D. Eager, E. Lazowsk, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engeneering*, SE–12(No.5):662–675, 1986.

[GMW79] Gordon, M., Milner, R., and Wadsworth, C. Edinbourg lcf. *Lecture Notes in Computer Science*, 1979.

[GRC89] R. Goldman, Gabriel R.P., and Sexton C. Qlisp : an interim report. In *Parallel Lisp : languages and systems*, pages 161–181, January 1989.

[Hal85] Halstead, R.H. Multilisp : a language for concurrent symbolic computation. In *ACM TOPLAS*, volume 7-4, pages 501–538, October 1985.

[HF92] P. Hudak and J.H. Fasel. A gentle introduction to haskell. *ACM SIGPLAN Notices*, 27(5):1 - 53, May 92 1992.

[HMM86] Harper, R., Macqueen, D., and Milner, R. Standard ml. *Edinbourg University, Internal Report ECS-LFGS-86-2*, 1986.

[Hug82] J. Hughes. Supercombinators : a new implementation method for applicative languages. In Prentice Hall International Series in Computer Science, editor, *ACM Conference on LISP and Functional Programming*, pages 1–10, August 1982.

[Kie85] R.B. Kieburtz. A fast graph reduction evaluator. *LNCS 201*, pages 400–413, 1985.

[Kie87] Kieburtz. A maj????? *A RECTIFIER*, 1987.

[KL84] R.M. Keller and F.C.H. Lin. Simulated performance of a reduction-based multiprocessor. *ieeetse*, 17(7), 1984.

[KW91] H. Kuchen and A. Wagener. Comparison of dynamic load balancing strategies. *x*, pages 303–314, Apr 1991.

[LC90] Lemaitre, M. and Coustet, C. Evaluation de mars sur une application de traitement symbolique parallèle. In *Actes du 2eme symposium ANM*, pages 12–14, Toulouse - France, Septembre 1990.

[Le 88] J. Le Maître. Le langage fonctionnel de manipulation de base de données griffon. In *J.I.S.I.'88*, Ecole Nationale des Sciences de l'Informatique. Tunis, Avril 1988.

[Les89] D. Lester. An efficient distributed garbage collection algorithm. In *PARLE89*, volume 1, pages 207–223, Juin 1989.

[LK87a] F.C.H. Lin and R.M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engeneering*, SE–13(No.1):32–38, Jan 1987.

[LK87b] Lin, F.C.H. and Keller, R.M. The gradient model load balancing method. In *IEEE Transactions on Software Engineering*, volume 1, pages 32–38, 1987.

[LKID88] R. Loogen, H. Kuchen, K. Indermark, and W. Damm. Distributed implementation of programmed graph reduction. In *Workshop on the implementation of lazy functional languages*, Aspenas- Goteborg- Sweden, 5–8 September 1988.

[LS89] J. Lemaitre and Y. Scolan. Le langage fonctionnel de manipulation de bases de données : griffon (manuel de référence). Technical Report 345, GRTC- Marseille, Avril 1989.

[MAE+62] McCARTHY, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. Lisp 1.5 programmer's manual. *MIT Press*, 1962.

[Mag79] G.A. Mago. Making parallel computations simple : the ffp machine. *Springer COMPCON*, 1979.

[NH85] Noshita, K. and Hikita, T. The bc-chain method for representing combinators in linear space. *New Generation Computers*, 3:131–144, 1985.

[OB92] R. Ortega and D. Bourget. Vers un langage fonctionnel sur armen. In *Deuxièmes journées Armen*, ENSTB- Brest - France, 1992.

[PCSH87] Peyton Jones,S.L., Clack, C., Salkild, J., and Hardie, M. Grip-a high performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture*, pages 98–112, Portland, Oregon,USA, September 1987.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science, 1987.

[Rab93] F. Rabhi. Functional languages and parallelism : towards a paradigm oriented solution. In *Journées du parallélisme*, Mons, Belgium, October 1993.

[Rob87] A. Robinson. *A functional programming interpreter.* PhD thesis, University of Illinois- Department of Computer Science, March 1987.

[Sar86] J. Sargeant. Load balancing, locality, and parallelism control in fine-grain parallel machines. Technical Report UMCS-86-11-5, University of Manchester, November 1986.

[Sch24] Schonfinkel, M. Uber dis bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924.

[Sch86] M. Scheevel. Norma : a graph reduction processor. In *ACM Conference on LISP and functional programming*, pages 212–219, Cambridge, Massachussets, August 1986.

[SE86] Saddayapan, P. and Ercal, F. Processor scheduling for linearly connected parallel processors. In *IEEE Transactions on Computers*, volume C-35-7, pages 632–638, july 1986.

[Sto77] H.S. Stone. Program assignment in three-processor systems and tricutset partitionning of graphs. Technical Report ECE-CS-77-7, University of Massachussets,amherst, Dep. of Elect. and Computer Eng, 1977.

[TM90] E-G. Talbi and T. Muntean. Placement statique de processus sur une architecture parallèle. Technical Report RR-833-I-, LGI-IMAG Grenoble, 1990.

[Tur79] D.A. Turner. A new implementation technique for applicative language. *Software Practise and Experience*, 9, Janvier 1979.

[Tur82] D.A. Turner. *Recursion equations as a programming langage.* University Press, J. DARLINGTON, P. HENDERSON, D.A. TURNER (Eds), 1982.

[WG88] Y.H. Wei and J.L. Gaudiot. Demand-driven interpretation of fp programs on a dataflow multiprocessor. *IEEE Transactions on Computers*, 37–8, August 1988.

[WS87] Wirsin,M. and Sanella, D. Une introduction à la programmation fonctionnelle : hope et ml. *Techniques et Science Informatique*, 6(16), 1987.

[WW87] P. Watson and I. Watson. Evaluating functional programs on the flagship machine. *Functional Programing Lanaguages and Computer Architecture*, Springer-Verlag LNCS(274):80–97, Sep 1987.

[XH] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing supercomputer. *IEEE*, pages 1–20, Apr.