Systèmes de processus communicants et interprétation parallèle de langages fonctionnels

Boubakar Gamatié

INRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (3) 954 90 20

Rapports de Recherche

Collection RR

N° 320

SYSTÈMES DE PROCESSUS COMMUNICANTS ET INTERPRÉTATION PARALLÈLE DE LANGAGES FONCTIONNELS

Boubakar GAMATIE

Juillet 1984

IRISA — PUBLICATIONS INTERNES -

SYSTEMES DE PROCESSUS COMMUNICANTS ET INTERPRETATION PARALLELE DE LANGAGES FONCTIONNELS

Boubakar GAMATIE

Publication n° 227 - Juin 1984

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

SYSTEMES DE PROCESSUS COMMUNICANTS ET INTERPRETATION PARALLELE DE LANGAGES FONCTIONNELS

Boubakar GAMATIE

Publication Interne n° 227
30 pages
Juin 1984

résumé:nous pésentons une algèbre de processus dont les termes permettent d'exprimer de façon parallèle et efficace les deux schémas fondamentaux d'interprétation parallèle de langages fonctionnels: "data-flow" et "par réduction". Cette algèbre est munie d'une relation d'equivalence observationnelle qui permet de décider de l'equivalence de deux implantations différentes, et d'une relation d'ordre sur les classes d'equivalences,qui permet de juger de l'efficacité d'une implantation par rapport à une autre.

abstract:a communicating processes algebra is presented,whose terms allow a parallel and efficient implementation of the data-flow and reduction schemes.This algebra comes with an observational equivalence relation so that equivalence of different implementations of the same function can be decided;an order relation over the equivalence classes is olso provided so that efficiency of implementation can be apreciated.

remerciemments:Je tiens à remercier ici,tout particulièrement Darondeau Ph,pour toutes les discussions très fructueuses que nous avons eues,dans le cadre de ce travail.

SYSTEMES DE PROCESSUS COMMUNICANTS ET INTERPRETATION PARALLELE DE LANGAGES FONCTIONNELS

(Gamatié,B.)

I)INTRODUCTION

La caractéristique des langages fonctionnels est qu'ils permettent d'exprimer les algorithmes en n'utilisant que des objets mathématiquement définis (dans l'algebre des fonctions) :les fonctions et combinateurs de fonctions.

Cette caractéristique se traduit par un très haut niveau d'abstraction dans la programmation,ce qui dispense l'utilisateur de la spécification du flot de contrôle entre les diverses parties de son programme et de la réservation des ressources physiques nécéssaires à l'exécution de son programme. En fait,l'expression de l'algorithme s'effectue indépendamment de tout schéma concret de machine;en conséquence,tous les choix de mise en oeuvre pour une exécution efficace du programme,doivent être réalisés par l'interpreteur.

Certains de ces choix peuvent conduire,si des précautions ne sont pas prises,à des abérations au niveau de l'occupation de la place mémoire, de la gestion des processus,ou à un nombre considérable de calculs redondants,d'où une perte de temps et une inefficacite non négligeable. Prenons un exemple :soit à réaliser la fonction de Fibonacci. Dans un langage qui utilise des équations (à la HOPE [3]),cette fonction peut être définie de la façon suivante:

```
Fib(1) = Fib(0) = 1
Fib(n+2) = Fib(n) + Fib(n+1)
```

Pour une interprétation "naive",qui consisterait à évaluer systématiquement tous les appels à la fonction,on voit ,pour n=4 par exemple,que Fib(2) est calculé deux fois et Fib(1) trois fois,alors qu'une seule évaluation (pour chacune des expressions) serait suffisante.

Les problèmes de réduction du nombre de calcul redondants,d'optimisation du nombre de ressources et de gain de temps,en vue d'une exécution efficace se posent pour toute interprétation d'un langage de programmation quelconque;cependant ils sont rendus plus aigus dans le cas de la programmation fonctionnelle,du fait de la non intervention du programmeur dans les choix qu'ils supposent.

Plusieurs methodes ont été proposées afin de résoudre ces problèmes. Ce sont par exemple:

- la méthode des annotations où il est proposé d'effectuer une extension du langage de programmation en y introduisant des annotations ([23],[20])qui sont des indications fournies par l'utilisateur à la machine,en vue d'une exécution optimale de son programme.

- la **méthode de transformation de programmes** qui se propose de transformer le programme utilisateur,généralement de façon semi-automatique ([20],[4]) , afin d'obtenir un code mieux adapté à un schéma particulier d'exécution.

-la **méthode de transformation du mode de transfert de données** où il s'agit d'adopter chaque fois que cela est possible,le mode d'appel par nécéssité au lieu du mode d'appel par valeur([27]),afin d'éviter la duplication inutile de calcul; mais aussi d'appliquer le principe de l'évaluation paresseuse([11]),afin de permettre la manipulation d'objets dont l'évaluation serait infinie.

Ces méthodes qui ont pour objet des **problèmes inhérents au style de la programmation fonctionnelle**,ont permis d'envisager la mise en oeuvre d'interpréteurs efficaces notamment sur les machines "conventionnelles".

Une autre méthode qui consiste à utiliser des systèmes de processus parallèles est de plus en plus envisageable.Cette méthode tient essentiellement du developpement de la technologie,en particulier dans le domaine VLSI,et qui a conduit à proposer et réaliser de nouveaux types de machines,basées sur des principes différents de ceux jusqu'a présent utilisés dans la conception de machines.

Ces nouvelles machines peuvent être classées en deux types,suivant leur mode d'organisation physique(c-a-d la configuration et l'algorithme d'allocation des ressources physiques)[25]:

i)**les machines à communication par paquets**,par exemple:la machine "data-flow" du MIT [9],ou le système AMPS [14].

ii)**les machines à manipulation d'expressions**(machines vectorielles ou arborescentes), ce sont par exemple :la machine cellulaire et arborescente de MAGO[16],ou la machine DDM1[7].

Toutes ces machines utilisent le parallélisme pour effectuer plusieurs tâches simultanément,de façon à réduire le temps global d'exécution.Dans le cas des langages fonctionnels,une interprétation parallèle est à priori bien adaptée, du fait de l'inexistence d'effets de bord lors de l'évaluation d'expressions fonctionnelles;cependant elle pose deux types de problèmes:d'une part ceux inhérents au parallélisme,(c-a-d gestion et allocation de processus) et d'autre part,ceux concernant la suppression de calculs redondants et la manipulation d'objets infinis.

Il s'agit donc pour l'interprétation de langages fonctionnels,d'adopter des structures de langage qui permettent d'exprimer à la fois le parallélisme et les solutions de ces problèmes.De telles structures doivent permettre d'une part d'exprimer les applications en termes de processus communicants et d'autre part de les exécuter sur l'un ou l'autre type de machine physique. Elles doivent aussi utiliser un schéma de communication explicite et symétrique et qui permette la communication à la fois,de signaux purs et de valeurs significatives(pour le calcul),ce qui faciliterait l'expression des divers schémas de communication utilisés dans les différents types de machine. Nous les voudrions de plus, d'une part intégrant la notion de non-déterminisme pour permettre une certaine abstraction de la machine physique,dans l'expression des schémas d'interprétation et d'autre part,utilisant exclusivement des constructions "bien gardées" afin de permettre un meilleur contrôle des synchronisations entre les parties d'un programme . Enfin, ces structures doivent exclure la notion de récursivité au profit de celle d'ittération, qui (c'est bien connu), correspond à un schéma d'interprétation plus efficace.

Plusieurs modèles(et calculs) de processus existent dans la littérature [1,2,5,8,12,13,17,18,19] , dont quelques uns ont même servi de base pour une traduction d'expressions fonctionnelles sous forme de réseaux de processus (voir par exemple[24]);cependant aucun de ces modèles ne présente tous les désiratas exprimés plus haut ,à la fois.

Ce rapport exhibe une algèbre de processus construite à partir de ces désiratas.Elle présente des agents (de calculs) parallèles qui coopèrent (par communication explicite) afin d'assurer une exécution efficace des programmes.L'utilisation de ces agents permet d'exprimer les solutions des problèmes que nous évoquions quant à l'exécution de programmes fonctionnels,tout en permettant une interprétation parallèle de façon a réduire le temps d'exécution.Ces agents sont munis d'une relation d'équivalence qui permet de décider de l'équivalence de deux implantations différentes d'une même fonction .

Dans une première partie nous exposons brièvement les principes généraux de l'interprétation de langages et discutons de l'adéquation des schémas proposés dans la littérature,au cas des langages fonctionnels.

La deuxième partie expose les structures du langage C.F.A et en donne une sémantique opérationnelle, en même temps qu'une définition de l'implantation parallèle d'une fonction,avec des relations qui permettent de juger de l'efficacité d'une telle implantation.

La dernière partie du rapport est consacrée à l'expression ,en termes d'agents C.F.A,des différents schémas fondamentaux d'interprétation parallèle de langages fonctionnels.

II)SCHEMAS D' INTERPRETATION PARALLELE DE LANGAGES FONCTIONNELS

1)PRINCIPES DE BASE D' INTERPRETATION

Un interpréteur de langage,quelle que soit la machine qui le supporte, est (conceptuellement) constitué de deux parties: l'une définissant le mécanisme de transfert de contrôle,l'autre celui de transfert de données.

Le **mécanisme de transfert de contrôle** définit l'ordonnancement dans le temps,des exécutions des différentes étapes du calcul.Il existe fondamentalement trois types de transfert de contrôle[22].Excluant le premier: le contrôle séquentiel-qui est classique et bien connu-nous nous attardons sur les deux autres qui se prètent mieux à une interprétation parallèle .

i)**Transfert de contrôle dirigé par la donnée** (data-driven): toute expression dont les arguments sont disponibles(c-a-d dont les valeurs sont connues) est exécutée.Il s'agit du type de contrôle qui permet,par essence,de minimiser le temps d'exécution en favorisant le parallélisme, moyennant certains problèmes de gestions des processus.

ii)Transfert de contrôle dirigé par la demande (demand-driven): une expression n'est exécutée que si le résultat qu'elle élabore est utilisé comme argument de l'expression à exécuter.On peut de cette façon obtenir une cascade de demandes d'exécution ,chacune attendant les résultats des demandes postérieures.Il faut noter aussi que dans ce type de transfert de contrôle,une expression peut "demander" les évaluations,en même temps,de tous ses arguments et donc activer plusieurs exécutions en parallèle.L'utilisation de ce contrôle permet de minimiser le nombre de calculs inutiles.

Le mécanisme de transfert de données définit la façon dont les arguments d'une expression sont accédés et utilisés.Il existe fondamentalement deux types de transfert de données qui peuvent être modifiés ou aménagés pour s'adapter au modèle opérationnel ou à l'architecture d'une machine specifique:

i)L'appel par valeur: l'argument est évalué et une copie de sa valeur placée dans chaque expression utilisatrice.L'appel par valeur,dans le cas d'un transfert de contrôle data-driven,devient l'appel par valeur parallèle dans lequel tous les arguments sont évalués en parallèle puis leurs valeurs placées dans les instructions utilisatrices.

ii)L'appel par nom: l'argument est sous forme d'expression dont une copie est placée dans chaque environnement(ou expression)d'utilisation. L'évaluation de l'argument peut dans ce cas s'effectuer plusieurs fois (une par exécution d'expression où il apparait).L'appel par nom peut être aménagé pour minimiser ce nombre d'évaluations des arguments; on obtient alors l'appel par nécéssite[27], où il n'existe qu'une seule copie de l'argument pour toutes les expressions qui utilisent sa valeur;il n'est alors évalué qu'à la "demande" et une seule fois au maximum.Un autre aménagement est encore possible; on obtient l'évaluation paresseuse[11] où l'argument n'est évalué qu'une seule fois au maximum,à la demande,et peut l'être partiellement,ce qui permet de manipuler des arguments dont l'évaluation complète serait infinie.

2) SCHEMAS D'INTERPRETATION

Les schémas d'interprétation des langages sont obtenus par combinaison des différents types de transfert de contrôle et de données.Ils déterminent l'organisation,dans la machine,des instructions du programme et la façon dont elles sont exécutées c-a-d le modèle opérationnel de la machine.
Dans le cas des langages fonctionnels,les schémas proposés pour leur interprétation reposent principalement sur les deux schémas de base suivants:

i)Le schéma "data-flow" qui présente fondamentalement un transfert de données de type appel par valeur et un transfert de contrôle de type data-driven.Toute expression dont les arguments sont disponibles doit être exécutée et les données sont transmises d'instructions(productrices) en instructions (consommatrices). La seule relation d'ordre entre les

instants d'exécution des instructions(relation de causalité) est celle qui relie un consommateur à un producteur.Elle explique la possibilité de parallélisme massif puisque par exemple deux instructions qui évaluent deux arguments différents d'une troisième instruction sont généralement non reliées par cette relation et peuvent donc être exécutées en parallèle.

ii)Le schéma par réduction qui présente un transfert de contrôle de type demand-driven.Le transfert de données,par contre,peut être de deux types différents:

- par valeur il s'agit alors de la réduction par chaine qui a l'avantage de permettre une mise en oeuvre distribuée relativement aisément.
- par nécéssite il s'agit dans ce cas de la réduction de graphe.Ce schéma, contrairement au précédent, possède l'avantage de minimiser le nombre de calculs. Dans le schéma par réduction,l'exécution d'un programme se résume à une succession de réductions qui lui sont appliquées,d'où le nom du schéma.

3) DISCUSSION

Les interpréteurs parallèles de langages fonctionnels généralement proposés mettent en oeuvre l'un des schémas data-flow ou par réduction sur une architecture qui est soit de type à communication par paquets soit de type à manipulation d'expressions.Alice[6] par exemple,met en oeuvre un schéma par réduction(de graphes) sur une architecture à communication par paquets.Cet exemple montre bien la possibilité de mettre en oeuvre l'un et l'autre des schémas sur l'une quelconque des architectures, bien que de façon générale,les architectures à communication par paquets soient mieux adaptées au schéma data-flow et celles à manipulation d'expressions au schéma par réduction.

La remarque précédente nous autorise donc à nous restreindre ,dans notre discussion,au modèle opérationnel de la machine de mise en oeuvre,indépendamment de son architecture physique.

A priori le schéma data-flow semble le mieux indiqué pour favoriser le parallélisme;cependant il présente certains inconvénients que nous relevons ici: conceptuellement,ce schéma suppose l'exécution de toute instruction dont les arguments sont élaborés.En fait,une mise en oeuvre réaliste du schéma nécéssite d'imposer des "priorités" entre les instructions pour tenir compte des disponibilités effectives de calcul de la machine. Cela amène généralement à introduire des "autorisations de s'exécuter" ou des "suspensions"[6] qui sont transmises d'instructions en instructions.Cet aspect "demand-driven" de la mise en oeuvre du schéma est rendue encore plus nécéssaire dans le cas de la conditionnelle où la notion d'exclusion entre les exécutions des différents alternants fait partie de la sémantique de l'opération.Une autre insuffisance du schéma data-flow "pur"est qu'il suppose le "principe d'activation atomique" c-a-d la connaissance de tous les arguments d'une fonction avant son exécution. Or il existe plusieurs cas où le calcul peut être initialisé sans attendre l'évaluation complète de tous les arguments.Dans ces cas,l'évaluation atomique entraine une dégradation des performances à l'exécution. Dans tous les cas,l'activation

atomique exclut l'utilisation d'arguments dont l'évaluation complète pourrait être infinie.

Le schéma par réduction permet de contourner ces insuffisances,d'une part parce qu'il est basé sur le type de transfert de contrôle demand-driven, d'autre part parce qu'il intègre une forme d'appel par nom qui peut être amenagée en l'évaluation paresseuse qui exclut le principe d'activation atomique et permet de manipuler des objets infinis .

Cependant ce schéma ne favorise pas le parallélisme,notamment dans le cas de fonctions récursives(auto-référentiables).En effet pour ce cas,les différents appels de la même fonction sont forcément réduits sequentiellement,puisqu'ils s'activent les uns les autres.

Prenons un exemple:soit à réaliser un additionneur de nombres entiers positifs.En supposant que l'on ne dispose que d'additionneurs élémentaires qui réalisent les opérations +1 et -1,la fonction addition,dans un langage qui utilise des équations récursives,s'écrit:

$$Add(0,y) = y$$
$$Add(x+1,y) = Add(x,y+1)$$

On voit bien ,pour l'évaluation de Add(n,m),que les n+1 appels de la fonction Add vont s'exécuter séquentiellement et qu'il faudra 2n+1 étapes pour réaliser la somme n+m.Une idée pour minimiser ce nombre d'étapes,est d'exécuter le plus tôt possible et en parallèle avec les soustractions de 1 à x,les additions de 1 à y.Cela revient à activer en parallèle deux processus,l'un effectuant les additions élémentaires,l'autre les soustractions élémentaires;ces deux processus pouvant bien entendu communiquer, pour décider de "la fin du calcul".Il s'agit donc d'une part de contourner le principe d'activation atomique et partant d'abandonner un des principes du schéma data-flow "pur",d'autre part d'exécuter en parallèle des appels récursifs d'une même fonction,c-a-d de contourner un des principes du schéma par réduction.

On peut dire,en conclusion de cette discussion,qu'une interprétation réaliste et efficace de langages fonctionnels doit intégrer divers aspects des deux schémas data-flow et par réduction à la fois.

Ce point de vue est à certains égards similaire à celui developpé dans [26],où il est noté que le schéma data-flow est insuffisant lorsqu'il est utilisé de façon exclusive.

Le langage de description d'un interpréteur parallèle de programmes fonctionnels doit permettre l'expression de réseaux de processus dont les interactions obeissent aux schémas "data-flow" et "par réduction". C'est là l'objet de notre proposition qui repose sur le constat suivant:
Pour une interprétation parallèle d'un langage de programmation,exhiber un parallélisme massif n'est pas suffisant,encore faut-il pouvoir synchroniser et gérer l'allocation des processus,de façon à prendre en compte les potentialités de calcul de la machine.En effet lorsque la politique de gestion des processus est par trop complexe,le temps passé à allouer et désallouer les ressources physiques peut devenir très considérable, entrainant du coup ,une dégradation générale des performances.

La démarche que nous considérons ici,vise à réduire le nombre de calculs exécutés en établissant des communications entre les agents qui effectuent ces calculs,ce qui entraine une diminution du nombre d'agents nécéssaires, d'où une simplification de la politique de gestion des agents. En l'adoptant,nous allons dans le même sens que [20],[21], où il est exprimé que l'efficacité d'un schéma d'interprétation parallèle passe par une

complémentarité entre parallélisme et communication.Cependant nous adoptons un langage différent,basé sur des principes différents de ceux utilisés dans [21], ce qui nous permet notamment,de mieu rendre compte de la vision intuitive que l'on a de l'exécution d'une fonction: vue de son environnement,une fonction est une "boite noire"qui importe des arguments et délivre,si elle est définie pour ces arguments,un résultat dans un temps fini.

III) LE LANGAGE D' EXPRESSION DES AGENTS

Nous exposons dans ce paragraphe les éléments du langage C.F.A(Concurent Functional Agents),qui nous permet d'exprimer l'évaluation d'expressions fonctionnelles,en utilisant des processus parallèles. Un programme C.F.A. est un système(clos) de processus asynchrones muni d' un protocole de rendez-vous multiple pour la synchronisation et la communication inter-processus.

Le mécanisme de rendez-vous est particulièrement important ici d'une part parce qu'il est symétrique,d'autre part parce qu'il nous permet d'exprimer les différents transferts de contrôle utilisés dans les interpréteurs parallèles de langages fonctionnels.Il permet en outre,lorsqu'il est généralisé(mécanisme de rendez-vous multiple ou diffusion),à un agent de diffuser des informations(signaux purs ou valeurs)vers un ensemble de processus,ce qui facilite le contrôle de l'activation en parallèle de plusieurs processus et la suppression de calculs redondants.

Les actions(de communications)effectuées par les processus constituent un ensemble qui possède une structure de monoide abélien,muni d'une loi de composition qui rend compte du parallélisme(c'est l'idée fondamentale du modèle de Milner):deux actions a et b effectuées simultanément par deux processus p et q (respectivement) constituent,par leur composition, une action a.b du système formé par les processus p et q.

1) SYNTAXE

Nous supposons donnés les ensembles suivants:

- -T un ensemble dénombrable (fini ou non)de types;
  exemple:entiers,booléens,caractères,...
- -X un ensemble d'identificateurs.
- -V un domaine de valeurs que l'on supposera être la réunion de sous-domaines de valeurs de même type;
  ie: $V = U_{t\in T} V_t$ où V_t content les valeurs de type t.
- -Λ^oun ensemble dénombrable indexé par T et Λ l'ensemble des ports de communication tq $\Lambda = <\Lambda^o \times Z >$ avec $\Lambda = <\Lambda^o \times \{0\}>$.

1) SYNTAXE

Les termes du langage sont définis récursivement de la façon suivante :

si $t, t_1, \ldots, t_n$ *sont des termes,* $v \in V$, $b \in V_{bool}$, *et* $\forall i, \alpha, \alpha_1 \ldots, \alpha_n \in \Lambda$

$x, x_1, \ldots, x_n \in X, \Psi \in [\Lambda \to \Lambda \cup \{\varepsilon\}]$

alors un terme est de l'une des formes suivantes :

i)	nil	*inaction*
ii)	$\alpha!v:t$	*exportation*
iii)	$b \to t_1 ; t_2$	*conditionnelle*
iv)	$t_1 \& t_2$	*composition*
v)	$(\alpha_1?x_1:t_1, \ldots, \alpha_n?x_n:t_n)$ *notée* $\sum_{i=1,n} (\alpha_i?x_i:x_i:t_i)$	*multigrade*
vi)	$\alpha?x \to t$	*horloge*
vii)	$t[\Psi]$	*renommage*

Avant de donner une sémantique formelle des opérateurs que nous utilisons, illustrons sur des exemples le sens intuitif de ces constructions.

i)L'**inaction** décrit un processus(ou agent)qui ne peut effectuer aucune action.Dans toute la suite,bien que ce processus fasse partie intégrante du langage,nous prendrons parfois la liberté de l'omettre de nos programmes dans un souci de clarté; bien entendu le contexte supprimera toute confusion. Par exemple nous noterons $\alpha!v$ pour $\alpha!v:nil$.

ii)L'**exportation** permet de décrire un agent qui exporte une valeur et se comporte de la façon définie par le terme qui suit le caractere":".
Le schéma de communication que nous avons choisi étant le rendez-vous, les exportations tout comme les importations dont on parlera plus tard, ne peuvent s'effectuer que s'il existe une proposition de communication complémentaire dans l'environnement du processus.
Par exemple s'il existe une proposition d'importation de valeur via le port α ,le processus $\alpha!0:nil$ exporte la valeur 0 sur le port α puis devient inactif.

iii)**La conditionnelle** possède son sens habituel ; si b est un booléen, $b \to \alpha!0 \ ; \ \alpha!1$ exporte la valeur 0 ou la valeur 1 sur le port α ,suivant que b soit vrai ou faux .

iv)**La composition** permet de décrire des réseaux de processus:deux processus t1 et t2 forment losqu'il sont composés,un processus(t1&t2)qui peut effectuer soit des actions de t1 (toutes seules),soit des actions de t2, soit simultanément des actions compatibles de t1 et t2 (deux actions sont compatibles si elles ne peuvent pas entrer en conflit pour l'usage d'un même port lors d'exportations de valeurs).
Exemple $\alpha!0$ et $\alpha!1$ sont incompatibles;le processus $\alpha!0:nil \ \& \ \alpha!1:nil$ ne peut qu'exporter soit 0 soit 1 sur le port α .

Par contre le processus $\alpha!0:nil \;\&\; \beta!1:nil$ peut soit exporter 0 via α ,soit exporter 1 via β ,soit exporter 0 via α et 1 via β simultanément.

v)La multigarde est un opérateur qui permet d'exprimer les importations de valeurs;il introduit en même temps un aspect non-déterministe dans le comportement des agents.Une multigarde $\sum_i(\alpha_i?x_i:t_i)$ représente un processus qui attend des informations sur n ports $(\alpha_1,\alpha_2,..,\alpha_n)$ à la fois.Après réception de valeurs sur les ports(par exemple) $\alpha_j,\ldots,\alpha_\ell$ (avec $1<j<\ell<n$), le processus se comporte comme un(seul)processus t_k. choisi arbitrairement (avec $j<k<\ell$),dont toutes les occurrences de x_k sont remplacées par la valeur recue via le port α_k .L'importation de valeur est une opération liante.

Par exemple le processus $(\alpha?x:res!y,\delta?z:\beta!y+z)$ attend de recevoir la valeur de x via α ,ou la valeur de z via δ ;dans le premier cas il exporte la valeur y via res,dans le second il exporte y+z via β . Le comportement d'un tel processus est d'une part conditionné par l'environnement:s'il n'arrive jamais de valeurs sur le port α ,le processus ne peut au maximum qu'exporter y+z sur β ,s'il reçoit z sur δ ;d'autre part, le processus décide (arbitrairement),en cas de conflit, de son comportement: si les valeurs de x et de y sont simultanément présentées sur α et δ , le processus décide unilatéralement de l'envoie de y via *res* ou de y+z via β .

vi)L'horloge est un opérateur qui permet d'activer des exemplaires d'un même processus,moyennant le passage d'une garde(c-a-d sous le contrôle d'un autre processus).Les différents exemplaires engendrés sont identifiés par introduction d'un renommage implicite de tous leurs ports(ceux du ième exemplaire activé sont tous numérotés i).Noter qu'un exemplaire peut lui même passer la garde qui permet d'engendrer de nouveaux processus.L'horloge permet de cette façon,d'exprimer la récursivité en utilisant la communication via les ports de la garde; après le passage de la garde $\alpha?x$,le terme $t=(\alpha?x \rightarrow t')$se comporte comme la composition t&t' avec, en plus, un renommage (incrémentation du numéro)de tous les ports des processus t et t'.

Exemples

1) Le processus suivant:
$(\alpha?x\rightarrow\alpha!x+1:nil) \;\&\; (\alpha!0:nil)$
permet de générer,via les ports $<\alpha,i>$,l'ensemble des entiers naturels.

2)La décomposition d'un nombre entier positif en deux suites de nombres pairs et impairs(respectivement)qui lui sont inférieurs s'écrit:
$(\alpha?x\rightarrow x=0\rightarrow nil \;;\; \beta!x-1:nil) \;\&\; (\beta?y\rightarrow y=0\rightarrow nil \;;\; \alpha!y-1:nil) \;\&\; (\alpha!n)$
Si n est pair(resp. impair), la suite des nombres inférieurs à n et pairs (resp. impairs)apparait sur les ports $<\alpha,i>$.

3)En reprenant l'exemple précédant,on réalise le shuffle de la décomposition de deux nombres avec le processus suivant:
$(\alpha?x\rightarrow x=0\rightarrow nil \;;\; \beta!x-1) \;\&\; (\beta?y\rightarrow y=0\rightarrow nil \;;\; \alpha!y-1) \;\&\; (\alpha!n \;\&\; \beta!m)$
Pour n=3 et m=5 par exemple,le résultat peut être schématisé comme suit:

$ports<\alpha,i>$:	3	4	1	2	0
$ports<\beta,i>$:	5	2	3	0	1

vii)Le renommage exprime la possibilite de changer le nom d'un port,ce qui permet de "masquer" certains ports d'un processus c-a-d interdire toute communication via ces ports.

Exemples 1) Soit le processus suivant:

$((\alpha?x \rightarrow x=0 \rightarrow nil\ ;\ (\beta!x-1)[\varphi]\ )\ \&\ (\alpha!n)$

Si $\varphi(\beta)=\alpha$ et $\varphi(\alpha)=\alpha$ alors ce processus génère sur les ports $<\alpha,i>$,l'ensemble des nombres entiers inférieurs à n.

2) Nous considérons ici,la version "numérotée" de Λ et le renommage Ψ^1 utilisé implicitement dans l'opération horloge et qui défini par:

$$\Psi^1(<\alpha,n>) = <\alpha,n+1>,\ \forall n\in Z,\ \forall \alpha\in\Lambda^o$$

Le processus suivant:

$(\alpha?x \rightarrow x=0 \rightarrow nil\ ;\ \alpha!x-1)\ \&\ (\alpha!n:nil)$

(où on note α pour $<\alpha,0>$) génère la séquence des nombres inférieurs à n sur n ports différents $<\alpha,i>$,i=1,n tels que la valeur m<n apparaisse sur le port $<\alpha,n-m>$.

Pour terminer cette présentation informelle du langage nous donnons deux exemples de processus qui réalisent des fonctions bien connues.Le lecteur pourra se convaincre que ces processus sont "corrects",nous le montrerons formellement plus tard.

Dans toute la suite nous utilisons l'application Ψ^i définie par: $\Psi^i(<\alpha,n>)=<\alpha,n+i>$ Nous notons α pour $<\alpha,0>$ et considérons qu'il existe un type particulier(noté SIGNAL) dans T tq les valeurs de ce type(ie contenues dans VSIGNAL) soient non significatives pour le calcul;nous notons $\alpha!$ pour l'exportation de valeurs de ce type via le port α et $\alpha?$ pour les importations de valeurs du même type.

Exemple 1: Fonction factorielle:

$(\ (ent?x \rightarrow x<=1 \rightarrow ok!;ent!x-1\ \&\ \delta!x)$
$\&(E?x \rightarrow (\delta?y:E!y*x:nil,\ ok?:sort!x:nil))$
$\&(E!1:nil)$
$)[\varphi]$ et $\varphi(E) = \Psi(\delta) = \Psi(ok) = \varepsilon$

Ce processus importe une valeur entière(n)via le port "ent" et délivre,sur le port "sort",la valeur n!.

Exemple 2 : Addition de deux nombres entiers en utilisant les opérations +1 et -1

$(\alpha?x \rightarrow x=0 \rightarrow \aleph!:nil;\alpha!x-1\ \&\ \delta!1)$
$\&\ (\beta?x \rightarrow (\delta?z:\beta!x+z:nil,\aleph?:res!x:nil))$

Ce processus réalise l'addition de deux nombres qu'il importe via les ports α et β . Remarquer que losqu'une seule des valeurs est présentée, le processus s'exécute qu'en même:il réalise la fonction currifiée:

Addc(x)(y)=x+y. En effet le premier processus génère x valeurs égales à 1 sur x ports différents $<\delta,i>$ $(1<i<x)$.Le deuxième processus effectue la somme de y avec ces valeurs.Nous avons réalisé une version parallèle de l'addition de deux nombres entiers en contournant le principe de l'activation atomique.

Nous donnons à présent la sémantique opérationnelle des termes C.F.A.

2) SEMANTIQUE OPERATIONNELLE

Définissons au préalable certaines notions qui sont utilisées dans la suite.

Définition 1 : L'ensemble des actions que peut effectuer un terme (ou agent) C.F.A. est noté $\mathcal{A}$; il est constitué par le monoïde abélien engendré par $A \cup \overline{A} \cup \tilde{A} \cup \{1\}$, quotienté par la relation $\lceil\ \rceil$ avec :

$$A = \{\alpha?V_t \mid \alpha\in\Lambda,\ V_t \subseteq V \text{ et } t \text{ est le type de } \alpha\}$$

$$\overline{A} = \{\alpha!v \mid \alpha\in\Lambda,\ v\in V_t \text{ et } t \text{ est le type de } \alpha\}$$

$$\tilde{A} = \{\tilde{\alpha}v \mid \alpha\in\Lambda,\ v\in V_t \text{ et } t \text{ est le type de } \alpha\}$$

et $\lceil\ \rceil$ *est la fermeture réflexive transitive de la relation définie par :*

$$\alpha?V_\delta\alpha!v = \tilde{\alpha}v.$$

Note : L'action 1 est une action comme une autre et ne joue pas, dans notre cas, le rôle d'élément neutre, qui lui est notée ε et représente l'absence d'action.

Définition 2 : On appelle multi-ensemble d'éléments d'un ensemble D , une application $M : D \to \mathbb{N}^+$ où $\mathbb{N}^+$ est le treillis complet des entiers naturels.

L'ensemble des multi-ensembles d'éléments d'un ensemble D est noté $\mathcal{P}_m(D)$. Plusieurs opérations peuvent être définies sur un tel ensemble ; nous définissons ci-dessous celles qui nous seront utiles dans la suite :

1) union : notée $\oplus$ et définie par : $\oplus \in \mathcal{P}_m(D) \times \mathcal{P}_m(D) \to \mathcal{P}_m(D)$ avec :

$$(M_1 \oplus M_2)(x) = M_1(x) + M_2(x)$$

2) intersection : notée $\ominus$ et définie par : $\ominus \in \mathcal{P}_m(D) \times \mathcal{P}_m(D) \to \mathcal{P}_m(D)$ avec :

$$(M_1 \ominus M_2)(x) = M_1(x) - M_2(x) \text{ si } M_1(x) \geq M_2(x),\ 0 \text{ sinon}$$

3) inclusion notée $\subseteq$ et définie par : $\subseteq \in \mathcal{P}_m(D) \times \mathcal{P}_m(D)$ avec :

$$M_1 \subseteq M_2 \iff \forall x \in D,\ M_1(x) \leq M_2(x) \text{ si } M_1(x) \text{ est définie.}$$

4) ensemble engendré par un multi-ensemble noté S et défini par :

$$S \in \mathcal{P}_m(D) \to \mathcal{P}(D) \text{ avec } S(M) = \{x\in D \mid M(x) \geq 1\}$$

Note : Le multi-ensemble vide est noté $\emptyset$ et est défini par :

$$\forall x\ D\ \emptyset(x) = 0.$$

Ce multi-ensemble possède plusieurs propriétés ; citons entres autres : pour tout multi-ensemble $M \in \mathcal{P}_m(D)$

i) $\emptyset \oplus M = M \oplus \emptyset = \emptyset$

ii) $M \ominus \emptyset = M$

iii) $\emptyset \subseteq M$

iv) $S(\emptyset) = \emptyset$ *où* $\emptyset$ *dénote aussi l'ensemble vide* $\in \mathcal{P}(D)$.

Définition 3 : On appelle Support-Import et Support-Export deux applications (notées respectivement "$\underline{\ }$" et "$\overline{\ }$") de $\mathcal{A} \to \mathcal{P}_m(\Lambda)$ qui définissent les ports utilisés en importation et en exportation (respectivement) dans une action. Ces applications sont définies par induction structurelle comme suit :

$$\underline{1} = \overline{1} = \emptyset$$

$$\underline{\tilde{\alpha}v} = \alpha!v = \emptyset \qquad \overline{\tilde{\alpha}v} = \overline{\alpha!v} = \{\alpha\}$$

$$\underline{\alpha?V_t} = \{\alpha\} \qquad \overline{\alpha?V_t} = \emptyset$$

$$\underline{ab} = \underline{a} \oplus \underline{b} \qquad \overline{ab} = \overline{a} \oplus \overline{b}$$

Note : Nous utiliserons souvent le terme support pour signifier l'union des support-import et support-export d'une action.

Définition 4 : On dit de deux actions a et b qu'elles sont compatibles et on note $a \# b$ *ssi* $\overline{a} \ominus \overline{b} = \emptyset = \overline{b} \ominus \overline{a}$.

Définition 5 : On dit que deux actions a et b sont complémentaires et on note $a \sim b$ *ssi* $S(\underline{ab}) = \overline{ab}$.

Définition 6 :

i) si a est une action et α un port tq $\{\alpha\} \subseteq \overline{a}$, on note $\vartheta_a(\alpha)$, la valeur exportée via α, pendant l'action a.

ii) si α est un port, x un identificateur et t un terme, on dit que x est lié par α dans t, si t possède un sous terme de la forme $\alpha?x:t'$ ou $\alpha?x \to t'$.

L'ensemble des identificateurs liés par un port α dans un terme t est noté $\chi_t(\alpha)$.

Règles de réécriture.

Etant donné les définitions précédentes, la sémantique des agents C.F.A. est définie (à la Plotkin [19]) de façon structurelle en utilisant des règles de réécriture conditionnelles. Ces règles utilisent des relations notées $\xrightarrow{a}$ pour $a \in \mathcal{A}$ qui définissent les transitions possibles d'un terme : $t \xrightarrow{a} t'$ signifie que le terme t peut effectuer l'action a et ce faisant se transformer en t'. Les relations $\xrightarrow{a}$ sont les plus petites satisfaisant :
($t\langle v/x\rangle$ représente la substitution par v, des occurrences libres de x dans t).

1) $\alpha!v:t \xrightarrow{\alpha!v} t$ 2) $\sum_i \alpha_i?x_i:t_i \xrightarrow{\alpha_J?V\delta_J} t_J$ *où* δ_J *est le type de* α_J.

3) $t_1 \xrightarrow{a} t_1' \Rightarrow vrai \to t_1$; $t_2 \xrightarrow{a} t_1'$, $faux \to t_2$; $t_1 \xrightarrow{a} t_1'$

4) $t_1 \xrightarrow{a} t_1' \Rightarrow t_1 \,\&\, t_2 \xrightarrow{a} t_1' \,\&\, t_2$, $t_2 \,\&\, t_1 \xrightarrow{a} t_2 \,\&\, t_1'$

5) $t_1 \xrightarrow{ab} t_1'$, $t_2 \xrightarrow{cd} t_2'$, $a \# c$, $b \sim d \Rightarrow$
$t_1 \,\&\, t_2 \xrightarrow{abcd} t_1'' \,\&\, t_2''$, $t_2 \,\&\, t_1 \xrightarrow{abcd} t_2'' \,\&\, t_1''$ *avec*

$t_1'' \equiv t_1' \langle \vartheta_a(\lambda)/\chi_{t_1}(\lambda)\rangle$ *pour tout* $\lambda \in \underline{b}$ *et*

$t_2'' = t_2' \langle \vartheta_b(\lambda)/\chi_{t_2}(\lambda)\rangle$ *pour tout* $\lambda \in \underline{d}$.

6) $\alpha?x \to t \xrightarrow{\alpha?V\delta} ((\alpha?x \to t)\&t)\,[\varphi^1]$ *si* δ *est le type de* α *et* $\varphi^1(\langle\lambda,n\rangle) = \langle\lambda,n+1\rangle$.

7) $t \xrightarrow{a} t' \Rightarrow t\,[\varphi] \xrightarrow{\tilde{\varphi}(a)} t'\,[\varphi]$ *si* $\tilde{\varphi}(a) \neq \varepsilon$ *où* $\tilde{\varphi}$ *est l'extension de* φ *à* $\mathcal{A}$ *défini par :*

$si\ \Psi(\alpha) = \varepsilon\ alors\ \vec{\Psi}(\tilde{\alpha}v)=1,\ \vec{\Psi}(\alpha!v)=\vec{\Psi}(\alpha?V_\delta) = \varepsilon$

$sinon\ \vec{\Psi}(\tilde{\alpha}v)= \tilde{\Psi}(\alpha)v,\ \vec{\Psi}(\alpha!v)= \Psi(\alpha)!v\ ,\ \vec{\Psi}(\alpha?V_\delta)= \Psi(\alpha)?V_\delta$

$\vec{\Psi}(ab) = \vec{\Psi}(a)\ .\ \vec{\Psi}(b)\quad \forall a,b\in \mathcal{A}$

3) EQUIVALENCE ET PREORDRE OBSERVATIONNELS

Définition 1 : On appelle interface une application (notée "I") de $C.F.A. \to \mathcal{P}_m(\mathcal{A})$ définie par induction structurelle comme suit :

1) $I(nil) = \emptyset$

2) $I(\alpha!v:t) = \{\alpha!v\}$

3) $I(\sum_i \alpha_i?x_i:t_i) = \{\alpha_i?V_{\delta_i} \mid \delta_i \text{ est le type de } \alpha_i,\ i=1,n\}$

4) $I(t_1 \,\&\, t_2) = I(t_1) \oplus I(t_2) \oplus \{ab \mid a\in I(t_1),\ b\in I(t_2) \text{ et } a \neq b\}$

5) $I(\alpha?x\to t) = \{\alpha?V_\delta\}$ où δ est le type de α.

6) $I(t[\Psi]) = \{\vec{\Psi}(a) \mid a\in I(t) \text{ et } \vec{\Psi}(a) \neq \varepsilon\}$

7) $I(vrai\to t_1, t_2) = I(faux\to t_2; t_1) = I(t_1)$

Proposition : Pour tout agent C.F.A. p, $\{x\in\mathcal{A} : I(p)(x) \neq 0\}$ est fini.

Preuve : Par induction structurelle sur p, en utilisant le fait que l'union de deux multi-ensembles finis est finie.

Définition 2 : L'extension des relations $\xrightarrow{a}$ aux séquences d'actions, est notée $\overset{a}{\Longrightarrow}$, pour $s\in\mathcal{A}^*$. Elle est définie par :

1) $p \overset{\varepsilon}{\Longrightarrow} p\ \forall p$, où ε est la séquence vide.

2) $p \overset{as}{\Longrightarrow} p' \iff p \xrightarrow{1^*} p_1 \xrightarrow{a} p_2 \overset{s}{\Longrightarrow} p'$ où la notation $p \xrightarrow{1^*} p'$ signifie qu'il existe n fini (éventuellement nul) tq :

$p \xrightarrow{1} p_1 \xrightarrow{1} p_2 \xrightarrow{1} \dots \xrightarrow{1} p_{n-1} \xrightarrow{1} p'$

Note : 1) si n est nul on note $p \xrightarrow{\varepsilon} p'$ et dans ce cas $p' \equiv p$, où ε est l'élément neutre de $\mathcal{A}$, noté (de façon ambigue) de la même façon que la séquence vide.

2) $p \overset{1}{\Longrightarrow} p' \iff p \xrightarrow{1} p_1 \xrightarrow{1^*} p'$; on note aussi $p \xrightarrow{1^+} p'$.

Définition 3 : si p et q sont deux agent C.F.A., alors :

i) $p \underset{\sim}{\sqsubseteq} q \iff \forall s\in(\mathcal{A}-\{1\})^*,\ p \overset{s}{\Longrightarrow} p' \implies (q \overset{s}{\Longrightarrow} q' \wedge S(I(p')) \subseteq S(I(q')))$

ii) $p \simeq q \iff p \underset{\sim}{\sqsubseteq} q \wedge q \underset{\sim}{\sqsubseteq} p$

Corollaire : $\forall p,p'$, si $p \xrightarrow{1^*} p'$ alors $p \underset{\sim}{\sqsubseteq} p'$.

Preuve : Immédiate.

Lemme : $\forall p,\ p\&nil \simeq nil\&p \simeq p$.

Preuve évidente d'après les définitions de $\xrightarrow{a}$ et $\overset{s}{\Longrightarrow}$.

Exemples :

1) si $\Psi(\alpha) = \varepsilon$ *alors* $(\alpha?:\beta!:nil \;\&\; \alpha!:nil)[\Psi] \;\tilde{\subset}\; \beta!:nil$
la preuve est immédiate en remarquant que :

$(\alpha?:\beta!:nil \;\&\; \alpha!:nil)[\Psi] \xrightarrow{1} (\beta!:nil \;\&\; nil)[\Psi]$, *et en utilisant le lemme et le corollaire ci-dessus.*

2) $((\alpha? \longrightarrow ((\delta?:\alpha!, \delta?:\beta!) \;\&\; \delta!)[\Psi_\delta] \;\&\; \alpha!:nil)[\Psi_\alpha])[\Psi_o]) \;\tilde{\supset}\; \beta!:nil$
si $\Psi_\delta(\delta) = \varepsilon$ *et* $\Psi_\alpha(\alpha) = \varepsilon$ *et* $\forall <\lambda,n> \in \Lambda \;\; \Psi_o(<\lambda,n>) = <\lambda,0>$

En effet : I(β!:nil) = {β!} et si on note p le membre gauche du préordre, on a $\forall n$ *fini* $p \xrightarrow{1^n} p'$ *tq* $I(p') = \{1, \beta!\} \not\subseteq \{\beta!\}$

par contre $\beta!:nil \xrightarrow{\beta!} nil$ *et* $p \xrightarrow{\beta!} ((\alpha? \rightarrow ((\delta?:\alpha!, \delta?:\beta!) \& \delta!)[\Psi_\delta] \& nil)[\Psi_a])[\Psi_o]$
que l'on note p' avec $\emptyset = I(p') \supseteq I(nil) = \emptyset$

Remarque : L'équivalence ainsi définie sur les agents C.F.A. se différencie des équivalences proposées dans les calculs de processus sus-cités, elle n'est pas récursive (et ne repose pas sur la notion de bissimulation) comme celles de [17] ou [2], et elle n'utilise ni la notion de tests, comme l'équivalence des tests de [8], ni la notion de traces comme dans [12].

Il est certainement très intéressant d'étudier les propriétés de cette équivalence (qui se révèle être en fait une congruence). Cependant nous la remettons à un cadre plus approprié, nous contentant dans la suite de poursuivre notre but premier, qui est d'exprimer, à l'aide d'agents C.F.A. les principaux schémas d'interprétation exposés au chapitre II.

IV) EXPRESSION DE SCHEMAS D'INTERPRETATION PARALLELE

1) PROCESSUS REALISANT UNE FONCTION

Définition 1 : Un agent (ou processus) F est dit muet ssi

$\forall a \varepsilon I(F), \; a = 1$

par exemple, le processus $(\alpha!v \;\&\; \alpha?x:t)[\Psi]$ *où* $\Psi(\alpha) = \varepsilon$, est muet.

Définition 2 : Un agent t est dit bloqué ssi $I(t) = \emptyset$
(en particulier nil est bloqué).

Définition 3 : Un agent t diverge s'il est indéfiniment muet et non bloqué
ie : t diverge ssi :

i) t muet
ii) $\forall n, \; \exists t' : t \xrightarrow{1^n} t'$ *et t' muet*

(où $\xrightarrow{s}{}^*$ *pour* $s \varepsilon \Sigma^*$ *est la fermeture réflexive transitive des relations* $\xrightarrow{a}$:

$p \xrightarrow{\varepsilon}{}^* p \; \forall p$ *et* $p \xrightarrow{as}{}^* p' \Leftrightarrow p \xrightarrow{a} p_1 \xrightarrow{s}{}^* p'$)

par exemple le processus $((\alpha?x \rightarrow \alpha!x+1:nil) \;\&\; (\alpha!0:nil))[\Psi]$
où $\Psi(\alpha) = \varepsilon$ *diverge.*

Définition 4 : Un agent F réalise une fonction f, si après importation d'une valeur v pour laquelle f est définie, il présente sur l'un (au moins) des ports, au bout d'un nombre fini de transitions, la valeur de la fonction f au point v.

F réalise f ssi $\forall v$ *tq f(v) défini,* $\exists \beta \in \Lambda$ *et* $\varphi \in [\Lambda \to \Lambda \cup \{\varepsilon\}]$ *tq*

$(F \,\&\, \alpha!v{:}nil)[\varphi] \xrightarrow{1^*} F'$ *et* $\{\beta!f(v)\} \subseteq I(F')$

si de plus $\forall F'' : F' \xrightarrow{w}{}^* F''$ *pour* $w \in \mathcal{V}^*$ *F" est bloqué ou diverge, on dit que F réalise exactement f.*

Proposition : $\forall p,q$ *deux agents si p réalise f et* $p \sqsubseteq_{\sim} q$ *alors q réalise f.*
la preuve est immédiate.

Exemple : Soient $p \equiv (\alpha?{:}\alpha!x)$ *et* $q \equiv (\beta?{:}(\alpha?x{:}\alpha!x)\ \&\ \beta!{:}nil)[\varphi_\beta]$

avec $\varphi_\beta(\beta) = \varepsilon$ *et* $\varphi_\beta(\alpha) = \alpha$.

p réalise la fonction f, qui appliquée à toute valeur, rend cette valeur. Il est évident que $p \sqsubseteq_{\sim} q$ puisque $q \xrightarrow{1} p[\varphi_\beta]$ et de façon triviale, q réalise aussi la fonction f.

2) IMPLANTATION EFFICACE

Définition : si p et q sont deux agents, alors :

$p \preccurlyeq q \iff \forall s \in \mathcal{V}^*,\ p \xrightarrow{s}{}^* p' \implies (q \overset{s}{\Longrightarrow} q' \wedge S(I(p')) \subseteq S(I(q')))$

Proposition : $\preccurlyeq$ est une relation de préordre sur les classes d'équivalences modulo $\sim$.

preuve $\preccurlyeq$ *est réflexive de façon évidente :*

montrons $p_1 \preccurlyeq p_2$ *et* $p_2 \preccurlyeq p_3 \Rightarrow p_1 \preccurlyeq p_3$ *avec* $p_1 \sim p_2 \sim p_3$.

soit $p_1 \xrightarrow{s}{}^* p_1'$ *alors* $p_2 \overset{s}{\Longrightarrow} p_2'$ *et donc (puisque* $p_2 \sim p_3$*)*

$p_3 \overset{s}{\Longrightarrow} p_3'$ *avec* $S(I(p_2')) \subseteq S(I(p_3'))$ *d'où*

$S(I(p_1')) \subseteq S(I(p_2')) \subseteq S(I(p_3'))$ *cqfd*

Proposition : $\preccurlyeq \subseteq \sqsubseteq_{\sim}$

preuve immédiate en remarquant que $\xrightarrow{s}{}^* \subseteq \overset{s}{\Longrightarrow}$

Définition :
i) On dit d'un agent q qu'il est une implantation plus efficace d'une fonction f, qu'un autre agent p ssi q réalise f et $p \sim q$ avec $q \preccurlyeq p$.

ii) On appelle implantation la plus efficace d'une fonction, le plus petit élément (au sens de la relation $\preccurlyeq$) de la classe d'équivalence (au sens de la relation $\sim$) d'un agent qui réalise cette fonction.

Exemple : En reprenant les agents p et q de l'exemple du 1), on a $p \lesssim q$ puisque $q \xrightarrow{1} p[\varphi_\beta]$ et que p ne peut pas effectuer d'actions de communication via le port β, donc

$\forall s \in \mathcal{A}^* : p \xrightarrow{s}{}^* p',\ q' \xrightarrow{1} p[\varphi_\beta] \xrightarrow{s}{}^* p'$ *c-a-d* $q \stackrel{s}{\Longrightarrow} p'$.

Intuitivement p est une implantation plus efficace que q d'une fonction, si q effectue les mêmes actions que p, à la différence peut-être d'un nombre plus grand d'actions 1 (de silences) et en présentant à chaque transition un choix au moins aussi grand d'actions possibles.

Deux questions se posent immédiatement, au sujet de l'implantation la plus efficace d'une fonction ; d'abord celle de son existence : étant donné une fonction réalisable par un agent C.F.A., existe-t-il une implantation plus efficace que toutes autres ? Ensuite si une telle implantation existe, comment peut-elle être construite ?

La réponse à la première semble être négative ; en effet les classes d'équivalences de la relation $\simeq$ ne constituent pas des ensembles totalement ordonnés pour la relation $\lesssim$. Citons pour contre exemple le cas suivant :

$p \equiv (\alpha?x:(\beta?:\alpha!x)\ \&\ \beta!)[\varphi_\beta]$ *avec* $\varphi_\beta(\beta) = \varepsilon$ *et* $\varphi_\beta(\alpha) = \alpha$.

$q \equiv (\beta?:(\alpha?x:\alpha!x)\ \&\ \beta!)[\varphi_\beta]$

dans ce cas on a bien $p \simeq q$, *mais ni*

$p \lesssim q$ *puisque* $p \xrightarrow{\alpha?V1}{}^* \alpha!x$, *mais* $\nexists q' : q \stackrel{\alpha?V1}{\Longrightarrow} q'$ *ni*

$q \lesssim p$ *puisque* $q \xrightarrow{1\alpha?V}{}^* \alpha!x$ *mais* $\nexists p' : p \stackrel{1\alpha?V}{\Longrightarrow} p'$.

La seconde question possède une réponse plus favorable. En effet, si une implantation plus efficace que toutes les autres existe, l'agent qui la réalise possède une forme caractéristique. Pour le construire nous définissons la notion suivante :

Définition : Soit p un agent C.F.A., alors

$w(p) = \{s \in \mathcal{A}^* ;\ \exists p' \text{ et } p \xrightarrow{s}{}^* p'\}$

Proposition : soit p un agent C.F.A. ; notons $||p||$ la classe d'équivalence de p pour la relation $\simeq$; alors

si $w(p) \in (\mathcal{A}-\{1\})^*$, *p est minimal pour la relation* $\lesssim$ *dans* $||p||$.

preuve soit p tq $w(p) \in (\mathcal{A}-\{1\})^*$, *et soit q tq* $p \simeq q$.

supposons $p \xrightarrow{s}{}^* p'$ *pour* $s \in \mathcal{A}^*$, *par hypothèse* $s \in (\mathcal{A}-\{1\})^+$ *et puisque* $\longrightarrow^* \subseteq \Longrightarrow$, *on a*

$p \stackrel{s}{\Longrightarrow} p' \Longrightarrow q \stackrel{s}{\Longrightarrow} q'$ *et* $S(I(p')) \subseteq S(I(q'))$, *puisque* $p \simeq q$.

cqfd.

L'implantation la plus efficace est donc réalisée par un agent qui ne peut pas effectuer d'actions 1 ; or la seule façon d'obtenir des actions 1 est d'appliquer un renommage φ à une action de diffusion a (appartenant à $\mathcal{A}$) avec $\varphi(\underline{a}) = \varepsilon$. On obtient la proposition suivante :

Proposition : si un agent p ne contient pas de sous-terme de la forme $t[\varphi]$ avec $\varphi(\underline{a}) = \varepsilon$ pour $a \in I(t)$, alors p réalise l'implantation la plus efficace de la fonction réalisée par les agents de sa classe.

3) EXEMPLES DE PROCESSUS REALISANT UNE FONCTION

i) Fonction constante : f définie par f(x)=v ∀x

$F \equiv ((\alpha?x \to \alpha!x:nil) \& (\alpha!v:nil))\ \lceil\varphi\rceil$ *où*

$\varphi(<\alpha,i>)=\alpha=<\alpha,0>,\ \forall i$

Ce processus représente une constante dont la valeur v est toujours disponible sur le port α. Il est initialisé par le terme α!v:nil.
Pour qu'un processus ait accès à la constante, il suffit donc qu'il exécute une importation de valeur via le port α:α?x:...
En effet examinons les transitions possibles du processus F :

$\alpha!v:nil \xrightarrow{\alpha!v} nil$ *(R1)*

$\alpha?x \to \alpha!x:nil \xrightarrow{\alpha?v} ((\alpha?x \to \alpha!x:nil) \& (\alpha!v:nil))\ \lceil\psi^1\rceil$ *(R5)*

$\Rightarrow F \xrightarrow{\widetilde{\alpha!v}} (((\alpha?x \to \alpha!v:nil) \& (\alpha!v:nil))\ \lceil\psi^1\rceil \& nil)\ \lceil\varphi\rceil$ *(R6, R3)*

$\simeq (((\alpha?x \to \alpha!x:nil) \& (\alpha!v:nil))\ \lceil\psi^1\rceil)\ \lceil\varphi\rceil \equiv F_1$ *(t&nil ≃ t)*

$\xrightarrow{\widetilde{\alpha!v}} (((t \& (\alpha!v:nil))\lceil\psi^1\rceil \& nil)\ \lceil\psi^1\rceil)\ \lceil\varphi\rceil$ *(où* $t \equiv (\alpha?w \to \alpha!x)$*)*

$\simeq (((t \& (\alpha!v:nil))\ \lceil\psi^1\rceil)\ \lceil\psi^1\rceil)\ \lceil\varphi\rceil \equiv F_2$

$\longrightarrow \ \ldots$

on a : $\forall n\ F \xrightarrow{(\widetilde{\alpha!v})^n} F_n$ *où*

$F_n \equiv (\ldots (t \& (\alpha!v:nil))\ \lceil\psi^1\rceil)\ \lceil\psi^1\rceil)\ldots)\ \lceil\psi^1\rceil)\ \lceil\varphi\rceil$

et
puisque $\forall i\ \varphi(<\alpha,i>) = \alpha \Rightarrow$

$(F_n \& \alpha?x:nil) \xrightarrow{\widetilde{\alpha!v}} F_{n+1},\ \forall n=0,1,\ldots$

Remarquer que le processus F ne s'arrête jamais et est toujours prêt à expédier à son environnement la valeur v, via le port α. Une version plus "économe" en transitions peut être réalisée : il suffit de garder l'émission de la valeur v, par la réception d'un signal :

$F' \equiv ((\alpha?x \to \delta?:\alpha!x) \& (\alpha!v:nil \& \delta!:nil))\ \lceil\varphi\rceil$ *où*

$\varphi(<\alpha,i>) = \alpha$ *et* $\varphi(<\delta,i>) = \delta\ \ \forall i$

Dans ce cas, l'accès à la valeur de la constante doit être précédé de l'émission d'un signal sur le port δ. ie :

$\delta!:\alpha?x:\ \ldots$

Les transitions du processus F' sont conditionnées par son environnement : une valeur v ne peut être expédiée sur le port α, que si l'environnement propose l'exportation d'un signal sur le port δ.

ii) Fonction factorielle : cas d'un appel de la fonction avec 2 comme argument :

$FACT \equiv ((ent?x \rightarrow x<1 \rightarrow ok!:nil;ent!x-1:nil \;\&\; \delta!x:nil)\;\&$
$((\varepsilon?x \rightarrow (\delta?y:\varepsilon!y*x:nil, ok?:sort!x:nil))\;\&\;(\varepsilon!1:nil))$
$)[\varphi] \quad \&\; (ent!2:nil)$

où φ *n'est défini ni pour ok ni pour* ε, *ni pour* δ *et tq :*

$\Psi(<sort,i>) = sort, \forall i,$ *et* $\Psi(<ent,0) = ent$ *et* $\Psi(<ent,i>)$ *non défini pour* $i \neq 0$.

Les transitions de ce processus sont les suivantes :

$FACT \xrightarrow{ent!2} FACT' \equiv (((t_{11}\&t_{12})[\psi^1])\&((t_{21}\&t_{22})[\psi^1]\;\&\;nil))[\varphi] \;\&\; nil$
$\simeq (((t_{11}\&t_{12})[\psi^1])\;\&\;((t_{21}\&t_{22})[\psi^1])[\varphi] \equiv FACT_1$

avec

$t_{11} \equiv (ent?x \rightarrow x<1 \rightarrow ok!:nil;ent!x-1:nil \;\&\; \delta!x:nil)$

$t_{12} \equiv (2<1 \rightarrow ok!:nil;ent!1:nil \;\&\; \delta!2:nil)$

$t_{21} \equiv (\varepsilon?x \rightarrow (\delta?y:\varepsilon!y*x:nil, ok?:sort!x:nil))$

$t_{22} \equiv (\delta?y:\varepsilon!y*1:nil, ok?:sort!1:nil)$

$FACT_1 \xrightarrow{1} FACT_2 \equiv (((t_{11}\&t_{13})[\psi^1])[\psi^1]\&\;(t_{21}\&\varepsilon!z:nil)[\psi^1])[\varphi]$

avec

$t_{13} \equiv (1 \leqslant 1 \rightarrow ok!:nil;ent!0;nil\&\delta!1:nil)$

$FACT_2 \xrightarrow{1} ((((t_{11}\&vrai \rightarrow ok!:nil;ent!0;nil\&\delta!1:nil)[\psi^1])[\psi^1]) \;\&$
$((t_{21}\&t_{23})[\psi^1])[\psi^1])[\varphi]$

avec

$t_{23} \equiv (\delta?y:\varepsilon!z*y:nil, ok?:sort!z:nil)$

$FACT_2 \xrightarrow{1} (((t_{11}\&nil)[\psi^1])[\psi^1]\;\&\;((t_{21}\&sort!z:nil)[\psi^1])[\psi^1])[\varphi]$
$\xrightarrow{sort!2} ((t_{11}[\psi^1])[\psi^1]\;\&\;(t_{21}[\psi^1])[\psi^1])[\varphi]$

La valeur fact(2) = 2 est bien exportée, après un nombre fini de transitions via le port sort= $\Psi(<sort,2>)$.

iii) *Fonction addition définie par :*

$Add(0,y) = y$, $Add(x+1,y) = Add(x,y+1)$

cas de $x = 2$ *et* $y = 0$

$$Add \equiv ((\alpha?x \to (x=0 \to \gamma!:nil; \alpha!x-1:nil \,\&\, \delta!:nil)) \;\&\; (\beta?x \to (\delta?y:\beta!x+y:nil, \gamma?:\rho!x)))[\varphi] \;\&\; (\alpha!2:nil \;\&\; \beta!0:nil)$$

où $\varphi(\gamma) = \varphi(\delta) = \varepsilon$ *et tq* $\varphi(<\rho,i>) = \rho, \forall i$ *et* $\varphi(<\alpha,0>) = <\alpha,0>$, $\varphi(<\beta,0>) = <\beta,0>$ *et* $\varphi(<\alpha,i>) = \varphi(<\beta,i>) = \varepsilon$ *pour* $i \neq 0$.

$$Add \equiv (t_\alpha \& t_\beta)[\varphi] \;\&\; (\alpha!2:nil \& \beta!0:nil)$$

$$Add \xrightarrow{\widetilde{\alpha!2}\;\widetilde{\beta!0}} ((t_\alpha \& t(\alpha!1:nil) \& (\delta!1:nil))[\psi^1] \& (t_\beta \& (\delta?y+0:nil, \gamma?\rho!0))[\psi^1])[\varphi]$$

$$\xrightarrow{1} ((t_\alpha \& (\alpha!0:nil) \& (\delta!1:nil))[\psi^1])[\psi^1] \& (t_\beta \& \beta!1:nil)[\psi^1])[\varphi]$$

$$\xrightarrow{1} ((((t_\alpha \& \gamma!:nil)[\psi^1] \;\&\; \delta!1:nil)[\psi^1])[\psi^1]) \;\&\; ((t_\beta \& (\delta?y:\beta!1+y:nil, \gamma?:\beta!1))[\psi^1])[\psi^1])[\varphi]$$

$$\xrightarrow{1} ((((t_\alpha \& \gamma!:nil)[\psi^1])[\psi^1] \;\&\; ((t_\beta \& \beta!1+1:nil)[\psi^1])[\psi^1])[\varphi]$$

$$\xrightarrow{1} ((((t_\alpha \& \gamma!:nil)[\psi^1])[\psi^1])[\psi^1] \;\&\; (((t_\beta \& (\delta?y:\beta!2+y:nil, \gamma?:l!2:nil))[\psi^1])[\psi^1])[\varphi]$$

$$\xrightarrow{1} ((((t_\alpha)[\psi^1])[\psi^1])[\psi^1] \;\&\; (((t_\beta \& \rho!2:nil)[\psi^1])[\psi^1])[\psi^1])[\varphi]$$

$$\xrightarrow{\widetilde{\rho!2}} ((((t_\alpha)[\psi^1])[\psi^1])[\psi^1] \;\&\; (((t_\beta)[\psi^1])[\psi^1])[\psi^1])[\varphi]$$

La valeur $2=0+2=x+y$ *est bien exportée, après un nombre fini de transitions sur le port* ρ.

Nous illustrons à present,sur des exemples,les divers schémas d'interprétation,exprimés en termes d'agents C.F.A.

4) DATA-FLOW

Ce schéma utilise un transfert de contrôle de type data-driven et un transfert de données par valeurs.Il est largement utilisé dans les applications où l'on a besoin d'exhiber un parallélisme massif.

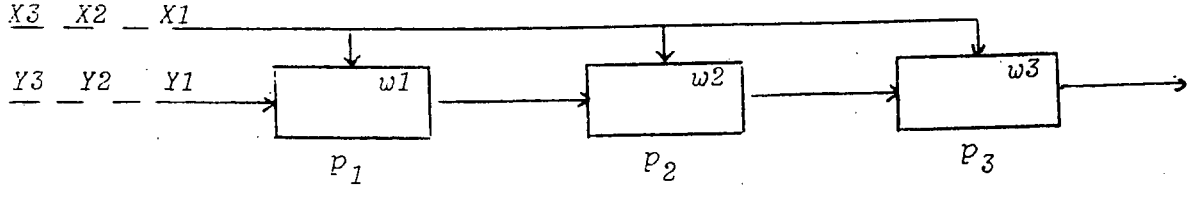
Un processus qui fonctionne suivant ce type de schéma doit s'exécuter chaque fois qu'une valeur est présentée sur l'un de ses ports d'entrée. L'utilisation de l'opérateur horloge permet d'exprimer immédiatement le comportement de ce genre de processus; par exemple le processus:

$$(\alpha?x \rightarrow t)$$

exprime un serveur qui est activé pour toute valeur présentée sur le port α ;pour chaque activation,il rend le "service" caracterisé par le terme t.La composition(à l'aide de &)de plusieurs processus "serveurs" définit un "multiserveur" qui est ré-activé par chaque présentation de valeur sur l'un quelconque des ports d'entrée des horloges qui le composent.Noter qu'un "multiserveur" peut rendre plusieurs services en parallèle .

Un exemple caractéristique de l'utilisation de ce schéma est fourni par les applications systoliques [15],[20], qui allient "pipeline" et synchronisme. Une application systolique est un ensemble de processus connectés de façon régulière (linéaire, polygonale, arborescente, ...), synchrones, et qui effectuent des calculs (suffisamment simples) suivant le schéma data-driven.

Un exemple d'application systolique pour le calcul du produit de convolution de deux vecteurs X et Y avec les poids W,est donné dans [15].La solution peut être schématisée comme suit:(on considère des vecteurs de dimension trois)



Elle propose l'utilisation de trois processus connectés linéairement,qui fonctionnent de la façon suivante:

- -les valeurs wi sont résidentes(chaque wi dans le processus pi correspondant)
- -les valeurs xi sont diffusées les unes après les autres à tous les pro cessus,

-les valeurs yi circulent de gauche à droite entre les processus,de façon systolique

A chaque pas de calcul,les processus effectuent les actions suivantes:

- réception de la valeur x
- réception de la valeur y
- multiplication de x par w(qui est résidant)
- addition à y de x*w
- émission de y.

Après le dernier pas de calcul(toutes les valeurs de X ont eté reçues),les valeurs yi présentes sur les sorties des processus,constituent le vecteur solution.(Rappelons que le produit de convolution de X par Y, avec les poids W s'écrit:

$$Y_J = \sum_{i=1}^{J} x_i * w_J \quad , \qquad \forall J$$

).

En utilisant les agents C.F.A,la solution de ce problème s'écrit:

$(t \,\hat{a}\, Ew \,\hat{a}\, Ex \,\hat{a}\, Ey)$ où

$t = (\beta?y \to \alpha?x:\gamma?w:\beta!w*x+y:nil)[\varphi]$ et $\varphi(<\alpha,i>)=><\alpha,0>,\varphi(<\gamma,i>)=<\gamma,0>, \forall i$

$Ew = (\delta?\to entw?w:\gamma!w:nil)[\varphi_w]$ et $\varphi_w(<entw,i>)=<entw,0>,\varphi_w(<\gamma,i>)=<\gamma,0>, \forall i$

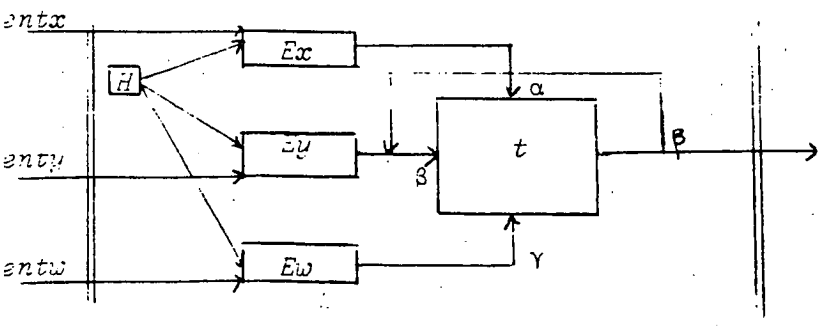
$Ex = (\delta?\to entx?x:\alpha!x:nil)[\varphi_x]$ et $\varphi_x(<entx,i>)=<entx,0>,\varphi_x(<\alpha,i>)=<\alpha,0>, \forall i$

$Ey = (\delta?\to enty?y:\beta!y:nil)[\varphi_y]$ et $\varphi_y(<enty,i>)=<enty,0>,\varphi_y(<\beta,i>)=<\beta,i>, \forall i$

(On se rappelle que $<\lambda,0> = \lambda \quad , \forall \lambda \in \Lambda$)

Nous supposons dans un premier temps l'existance d'une horloge(de processeur) H (qui n'est pas décrite ici),qui émet des signaux sur le port δ à chaque top d'horloge.

Cette solution propose un(réseau de)processus qui peut être schématisé de la façon suivante:



(Les doubles lignes verticales représentent la "frontière" d'avec l'environnement du processus).

Après chaque étape de calcul,deux nouveaux exemplaires du processus t sont activés;l'un d'entre eux reçoit le résultat qui vient d'être calculé,l'autre est en attente de réception d'informations de l'environnement.A la premiere étape,un seul exemplaire est activé;à partir de la seconde,on peut considérer que l'on a deux"files" de processus, decalée d'un rang l'une part rapport à l'autre et qui sont actives en parallèle.La première file contient un processus qui est toujours susceptible de recevoir de l'environnement les valeurs y,x et w(idem pour la seconde file pour ce qui est de x et w),ce qui exprime le schéma data driven.La seconde file contient un processus toujours susceptible de recevoir une valeur y transmise par la première file:les valeurs y circulent entre les processus.

Le processus t est écrit sous forme d'horloge,pour exprimer le schéma data driven utilisé dans le calcul.Le fait que le numérotage des ports ne soit pas supprimé(contrairement au cas des ports " α " et " γ "),permet d'exprimer le caractère systolique de l'échange des valeurs y et de différencier les différents exemplaires de processus activés.

Les processus Ex,Ey et Ew permettent d'importer,depuis l'environnement, les valeurs x,y et w;ils fonctionnent de la façon suivante: à chaque top d'horloge(émission d'un signal δ),ils se mettent en attente de valeurs (x,y,w), qui sont aussitôt après réception,transmises au processus t.Ces processus fonctionnent en data-driven.

Nous donnons maintenant pour fixer les idées,les deux premières étapes de calcul du processus t:

$t = (\beta?y \to \alpha x:\gamma?w:\beta!x*w+y:nil)[\psi]$

$\underline{\beta^0?y_0} \to ((t\&t^{y_0}_{\alpha\gamma})[\psi^1])[\varphi]$ où $t^{y_0}_{\alpha\gamma} = \alpha?x:\gamma?x:\beta!x*x+y_0:nil$

$\underline{\alpha^0?x_0} \to ((t\&t^{y_0,x_0}_{\gamma})[\psi^1])[\varphi]$ où $t^{y_0,x_0}_{\gamma} = \gamma?w:\beta!x_0*w+y_0:nil$

$\underline{\gamma^0?w_0} \to ((t\&\beta!x_0*w_0+y_0:nil)[\psi^1])[\varphi]$

$\underline{\beta^1!y'_0} \to (((t\&t^{y'_0}_{\alpha\gamma})[\psi^1])[\varphi]$

$\underline{\beta^2?y_1} \to (((t\&t^{y'_0}_{\alpha\gamma})[\psi^1]\ \&t^{y'_0}_{\alpha\gamma})[\psi^1])[\psi^1])[\varphi]$ où $y'_0 = x_0*w_0+y_0$

$\underline{\alpha^0?x_1} \to ((((t\&t^{y_1,x_1}_{\gamma})[\psi^1]\ \&\ t^{y'_0,x_1}_{\gamma})[\psi^1])[\psi^1])[\varphi]$

$\underline{\gamma^0?w_1} \to ((((t\&\beta!y_1+x_1*w_1:nil)[\psi^1]\ \&\beta!y'_0+x_1w_1:nil)[\psi^1])[\psi^1])[\varphi]$

$\underline{\beta^3!y'_1} \to (((((t\&t^{y'_1}_{\alpha\gamma})[\psi^1])[\psi^1])\ \&\ \beta!y'_0+x_1w_1:nil)[\psi^1])[\psi^1])[\varphi]$ où $y'_1=y_1+x_1*w_1$

$\underline{\beta^4?y_2} \to \ldots$

Remarquer que le processus importe les valeurs y sur des ports numérotés avec des chiffres pairs et qu'il exporte,au fur et à mesure du calcul,les différentes valeurs y' évaluées à l'étape précédente.

On notera d'autre part que ce système permet de calculer le produit de convolution de vecteurs de dimensions quelconque,sans que soit precisé le nombre de processus.Le calcul est exprimé indépendamment de la notion de processeur physique.

A la fin du calcul (determinée par la fin des séquences X et W), le résultat est disponible en exportation sur les ports β^2 , β^3 , β^4 , ...

Dans le programme précédant,l'horloge du système n'est pas décrite;pour combler cette lacune,il suffit de rajouter une horloge(au sens C.F.A) au système et de modifier légèrement le terme t; on obtient:

$(t' \;\&\; Ew \;\&\; Ex \;\&\; Ey \;\&\; H)$

où $H = ((\mu?\rightarrow\delta!:nil)\&(\mu!:nil))[\Psi_\mu]$

$t' = (\beta?y\rightarrow\alpha?x:\gamma?w:(\beta!x*w+y:nil \;\&\; \mu!:nil))[\varphi]$

avec

$\Psi(<\mu,i>) = \Psi_\mu(<\mu,i>) = <\mu,i>.$

L'horloge H est initialisée par le terme ($\mu!:nil$) et ensuite délivre un top sur réception d'un signal μ émit par le terme t',ce qui correspond à la fin d'une étape de calcul.Le système est,de cette façon,rendu synchrone.

5) REDUCTION

Les machines à réduction utilisent un transfert de contrôle de type demand driven et les deux types de transfert de données,suivant qu'il s'agisse de la réduction de chaine ou de graphe.

Le contrôle demand driven est utilisé pour éviter d'effectuer des calculs inutiles(voir le cas de la fonction de Fibonacci);il suppose deux(au moins) "postes"de calcul:un "demandeur" qui active un "serveur" quand il a besoin de la valeur élaborée par ce dernier(et uniquement dans ce cas); le serveur lui même ne pouvant être exécuté que sur activation d'un "demandeur".

Pour exprimer ce schéma en termes d'agents C.F.A ,il suffit de préfixer, dans le "demandeur" ,l'importation de la valeur demandée par l'émission d'un signal(disons sur le port δ) et de garder,dans le serveur,le terme qui réalise l'élaboration de la valeur demandée,par une réception de signal via le port .Le caractère symétrique de la communication assure que le serveur ne peut s'exécuter sans avoir été activé par un demandeur,ce dernier ne pouvant recevoir de valeur sans avoir activé un serveur.

Par exemple les deux processus suivants:

$$(\delta!:x?x:t) \;\&\; (\delta?:\ldots\ \alpha!v:t') \equiv t1 \;\&\; t2$$

fonctionnent suivant le schéma demand driven.Le demandeur est t1;le serveur t2. Ils se synchronisent par usage du port δ .La valeur élaborée par t2 est exportée via le port α .On voit bien que le serveur ne peut s'exécuter que sur demande du terme t1 c-a-d après réception du signal (émit par t1) sur le port δ.

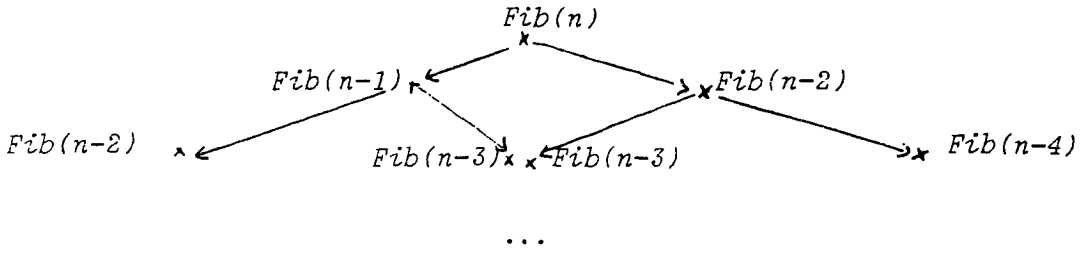
Un serveur peut être activé par plusieurs demandeurs;il doit,pour chacune des demandes,fournir un résultat;en outre,ces différents résultats doivent être identifiés de telle sorte que le résultat destiné au processus p,ne soit pas dirigé vers un autre processus p'. Pour résoudre ces problèmes,nous utiliserons l'opérateur horloge dans le terme représentant le serveur,pour permettre que soit à tout moment,disponible un exemplaire de serveur et qu'il puisse être identifié.Bien entendue cette identification est "masquée" à l'environnement du processus(en utilisant l'opération de renommage).

Un exemple caractéristique d'utilisation de ce schéma est fourni par l'évaluation (optimisée) de la fonction de Fibonacci.

Cette fonction est définie par:

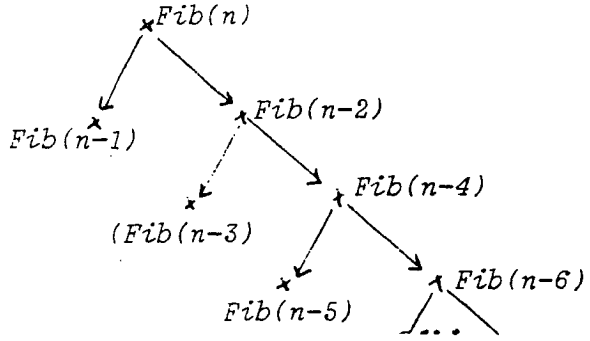
Fib(0)=Fib(1)=1,

Fib(n+2)=Fib(n)+Fib(n+1) L'optimisation ici,consiste à réduire le nombre de calculs redondants c-a-d passer d'un graphe d'exécution ayant la forme:



...

à un graphe où n'apparaisse qu'une seule fois chaque noeud.
L'idée pour ce faire ,consiste à n'activer de nouveaux processus qu'à partir de noeuds de même parité(celle de n dans le cas de Fib(n)). On obtient un graphe d'activation qui a l'allure suivante:



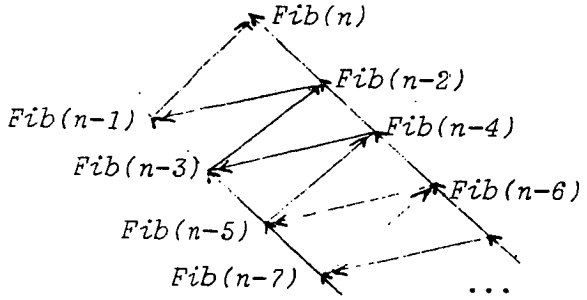
Etant donné ce graphe,le transfert de données s'effectue dans plusieurs directions:

1- des processus d'un niveau p vers le processus de niveau p-1 et de même parité que n (toujours pour Fib(n))

2- des processus de même parité que n,vers les processus de même niveau et de parité inverse.

3- entre les processus de parité inverse à celle de n,d'un niveau p vers le niveau p-1.

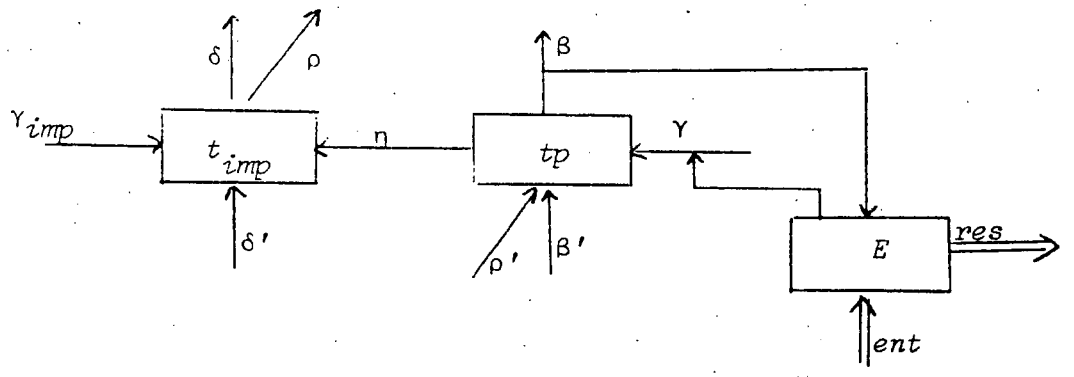
On obtient le graphe de transfert de données suivant:



Quand on supperpose les deux graphes,on voit immediatement qu'un noeud i s'exécute avec les noeuds i-1 et i-2 (pour i de même parité que n),en utilisant un schéma demand driven:le noeud i active ces deux noeuds et attend qu'ils lui communiquent le résultat de leurs exécutions;ces noeuds ne pouvant être activés que par le noeud i;ils le sont à la demande.En fait quand on examine le graphe de transfert de données,on se rend compte que l'on a, en plus du schéma demand driven,une partie des processus qui s'exécutent en data-driven,pour ce qui est de la "collecte"du résultat: les processus de parité inverse de celle de n n'activent pas de processus, mais doivent être prêts à recevoir les valeurs venant soit du niveau inférieur,soit du même niveau (directions 2 et 3 de communication).
En utilisant les agents C.F.A ,on obtient le programme suivant qui réalise la fonction de Fibonacci (optimisée):

$(tp \;\&\; timp \;\&\; E)$
où $\Psi(<ent,i>) = ent$, $\Psi(<res,i>) = res$ $\forall i$ et $\Psi(\lambda) = \varepsilon$ $\forall \lambda \in \Lambda$
avec
$tp = (\gamma p?x \rightarrow x<1 \rightarrow ((b?1:nil)\&(\beta!1:nil)\,[\psi^{-1}]);(\gamma p!x-2:nil)\&(\gamma imp!x-1:nil)\&$
$(\beta?Z:\rho?y:u+Z:nil)\&(\beta y+Z:nil)\,[\psi^{-1}]$
$)$
$timp = (\gamma imp?x \rightarrow x<1 \rightarrow ((\delta!:nil)\&(\rho!1:nil)[\psi^{-1}];(\delta?Z:\eta?y:((\delta!y+Z:nil)\&(\rho!y+Z:nil)))[\psi^{-1}]$
$E = (ent?x \rightarrow (\gamma p!x:nil)\&(\beta?y:res!y:nil)$

Cette solution propose un réseau de processus qui peut être schématisé comme suit:



Il s'agit de deux processus représentant les processus de même parité et de parité inverse de celle de n (processus pair et impair respectivement). On peut considérer que les transitions, dans le temps, de ces processus définissent une séquence de couples de processus qui fonctionnent comme suit: Les processus pair et impair d'un niveau i sont activés par le processus pair du niveau i-1, en utilisant les ports γ_p et γ_{imp}, suivant le schéma demand driven; ils lui transmettent les résultats de leurs exécutions via les ports β et ρ respectivement. Une fois activés, les processus impairs se communiquent leurs résultats (suivant un schéma data driven) via le port δ. A chaque niveau, le port η permet d'établir des communications entre processus de même niveau. Le renommage implicite introduit par les horloges (termes tp et timp) permet de différencier les divers niveaux de processus et de diriger les résultats vers les processus qui les attendent. Schématiquement, le programme précédent fonctionne comme suit: dans un premier temps (dépliage), autant de couples de processus que nécessaire sont activés; dans un deuxième temps (pliage) les résultats des exécutions sont acheminés depuis les processus du niveau le plus bas, vers le niveau le plus élevé. Le résultat final est accessible sur le port β du niveau le plus haut; le terme E du programme permet de le transférer à l'environnement via le port res.

Cet exemple illustre bien la possibilité que l'on a d'exprimer les deux schémas data driven et demand driven, ce qui est une de nos exigences dans le choix des structures de langages pour exprimer l'interprétation parallèle de langages fonctionnels. On notera aussi, comme dans l'exemple du produit de convolution systolique, que le programme n'indique pas explicitement le nombre de processus nécéssaires pour le calcul de fib(n), pour un n particulier. Les différents niveaux de processus sont engendrés au fur et à mesure des besoins. Il s'agit ici encore, de l'expression du calcul indépendamment de la machine physique d'implantation.

6) TRANSFERT DE DONNEES

Nous considérons ici,pour terminer cette présentation de l'expression des schémas d'exécution parallèle,le problème du transfert de données.
Le schéma de communication utilisé dans C.F.A considère des valeurs (du domaine V).Le mode de transfert "appel par valeurs" est donc directement exprimé dans les programmes C.F.A. La question qui mérite une attention particulière est la façon d'exprimer les autres modes de tranfert de données(évaluation paresseuse,appel par nécéssité,appel par nom ...) en utilisant les termes C.F.A .
La possibilité qui existe dans C.F.A,d'exprimer les constantes ou les (variables) séquences de valeurs par des termes sera très utile dans ce cadre.En fait,de la même façon que l'appel par valeur s'exprime par une opération d'importation de valeur via un port particulier,l'appel par nom (et ses différentes formes)s'exprime par activation de processus,avec des schémas de synchronisation particuliers.

i) **Appel par nom** : Il s'agit dans ce type de transfert de données,d'exécuter l'expression qui représente la valeur à importer,chaque fois que l'on a besoin de cette valeur.En termes C.F.A ,ce transfert de données s'exprime en utilisant un contrôle demand driven:on suppose l'existance d'un processus tv qui réalise l'expression représentant la valeur demandée.L'appel par nom s'exprime par l'activation du processus tv (émission d'un signal sur un port constituant la garde du processus),suivie d'une opération d'importation de la valeur v.Le processus tv s'exécute,après réception du signal,élabore la valeur v,l'exporte sur son port de sortie et se remet en attente d'un nouveau signal,pour une nouvelle communication de la valeur v.
ii) **Appel par nécéssité** : Il a été introduit pour éviter de recalculer la valeur à communiquer à chaque demande.Il est très similaire à l'appel par nom et s'exprime en modifiant légèrement l'expression de ce dernier.Le processus tv qui élabore la valeur à communiquer n'est plus écrit sous forme cyclique;après la première évaluation de la valeur v,il active un processus qui lui est susceptible d'exporter cette valeur à tout instant(il s'agit d'un processus réalisant une constante).De cette façon la valeur est disponible sans calculs supplémentaires,pour toute exécution postérieure à la première.
iii) **Evaluation paresseuse:** Il s'agit dans ce cas,non seulement de n'évaluer qu'une seule fois la valeur à communiquer,mais en plus de ne l'évaluer que partiellement.Cela suppose que cette valeur puisse être décomposée en valeurs élémentaires(par exemple séquence ou ensembles en les éléments qui les constituent).L'expression de ce type de transfert de données s'effectue(aussi) en modifiant celle de l'appel par nom.Le processus qui évalue la valeur à exporter (tv) est là encore cyclique,mais il attend,entre chaque cycle,un signal d'activation.Un cycle correspondant à l'elaboration et à l'expédition d'une valeur élémentaire constituant la valeur v.
Par exemple le processus:

$((\delta? \rightarrow \alpha?x:(\alpha!x+1:nil)[\varphi^1]) \;\&\; (\delta!:(\alpha!:nil)[\varphi^1]))[\varphi]$
avec $\forall i,\ \varphi(<\delta,i>) = \delta$ et $\varphi(<\alpha,i>) = \alpha$
évalue la séquence (infinie)des nombres entiers positifs.Les nombres sont évalués et expédiés via le port ,après chaque réception du signal .

V) CONCLUSION

L'interprétation de langages fonctionnels pose des problèmes qui sont inhérents au très haut niveau de programmation que permet ce genre de langage. Par exemple le fait que le seul opérateur proposé par le langage soit l'application d'une fonction à son argument,dispense l'utilisateur d'expliciter le transfert de contrôle .

Cependant on peut tirer profit de cette approche très haut niveau,puisqu'elle permet une très grande marge de manoeuvre à l'implanteur.C'est dans ce cadre et du fait de la non existence d'effets de bords lors de l'application d'une fonction à son argument,qu'une interprétation parallèle de ce type de langage peut être envisagée.

Cette interprétation parallèle repose essentiellement sur deux schémas: data-flow et réduction.Cependant si le schéma data-flow favorise le parallélisme, il ne permet pas toujours de prendre en compte les potentialités de calcul réelles de la machine de mise en oeuvre.Le schéma par réduction quant à lui,permet d'éviter le "gaspillage" des ressources,mais il brime le parallélisme dans certains cas.

Ce rapport exhibe une algèbre de processus(C.F.A) qui permet de concilier dans une même interprétation,les deux schémas.

Cette algèbre propose l'utilisation d'agents asynchrones,parallèles et qui communiquent de façon explicite, afin de réduire le nombre de calculs nécéssaires pour l'évaluation d'expressions fonctionnelles.Ils permettent d'exprimer explicitement le parallélisme et utilisent un schéma de communication et de synchronisation symétrique,ce qui facilite l'expression du contrôle.

Ces agents sont utilisés pour donner une définition formelle de l'implantation parallèle d'une fonction;ils sont,de plus,munis de relations (d'équivalence et d'ordre) qui permettent d'une part de décider de l'équivalence de deux implantations, et d'autre part, de juger de l'efficacitéde l'une par rapport à l'autre.

Avec de nombreux exemples,nous montrons l'usage de ces structures dans l'expression de l'évaluation d'expressions fonctionnelles.Par exemple l'application d'une fonction à son argument se résume à l'envoi,sur les ports d'entrée du processus réalisant la fonction,de la valeur de l'argument(dans les cas récursif et non récursif).

De la même façon,nous exprimons les deux schémas d'interprétation parallèle généralement proposés ainsi que les modifications qui leur sont souvent adjointes, ce qui permet d'exprimer par exemple très simplement les programmes systoliques qui allient "pipeline" et synchronisme.

Les agents C.F.A permettent,comme nous l'avons montré sur les exemples du produit de convolution systolique et sur la fonction de fibonacci,d'exprimer la solution d'un problème en termes de processus parrallèles, indépendamment de toute notion de processeur physique.

BIBLIOGRAPHIE

[1] **Arnold A.,Nivat M.** :Comportement de processus colloque AFCET:les mathématiques de l'informatique(mars 1982).

[2] **Austry D.,Boudol G.**:Algèbre de processus et synchronisation Sophia-Antipolis (Valbonne).

[3] **Burstall R.M.,Mc Queen D.B.,Sannella D.T.**:HOPE:an experimental applicative language.Internal Report CSR-62-80 (may 1980) Univ.Edinburgh .

[4] **Burstall R.M.,Darlington J.**:A Tranformation System for Developing Recursive Programs;JACM vol 24,N.1 (jan 1977).

[5] **Brookes S.D.**:A model for communicating sequential processes Doctoral thesis University College- Oxford University (janv.1983).

[6] **Darlington J.,Reeves M.**:Alice a multiprocessor reduction machine for the parallel evaluation of applicative languages. Proc. 1981 conf. on functional prog. lang. and comp. architecture,(oct 1981).

[7] **Davis A.L.**:The architecture and system method of DDM1:a recursively structured data-driven machine. Proc. 5th int. symp. on comp. architecture (april 1978) .

[8] **De Nicola R.,Hennessy M.C.B.**:Testing equivalences for processes Automata,languages and programming 10th colloquium. Barcelona(july 1983) and LNCS 154.

[9] **Dennis J.B.**:Packet communication architecture, proc.1975 comp. conf. on paralel processing.

[10] **Haynes L.S.,Lau R.L.,Siewiorek D.P.,Mizell D.W.**:A survey of highly paralel computing. IEEE jan.1982 .

[11] **Henderson P.,Morris J.H.**: A lazy Evaluator Proc. of POPL conf.Sigplan-Sigact Atlanta (1976).

[12] **Hoare C.A.R.,Brookes S.D.,Roscoe A.W.**:A theory of communicating sequential processes. U.of Oxford Comp. Lab. (may 1981).

[13] **Kahn G.,Mc Queen D.B.**:Coroutines and networks of parallel processes. Gilchrist B.(ed),Information processing .Amsterdam (1977) .

[14] **Keller R.M.,Lindstorm G.,Patil S.**:A loosely coupled applicative multiprocessing system. Proc. AFIPS NCC vol.48,(1979).

[15] **Kung H.T.**: Why systolics architectures ? IEEE jan.1982 .

[16] **Mago G.A.**:A cellular computer architecture for functional programing. Proc. IEEE COMPCOM 80 (feb.1980) .

[17] **Milner R.**:On relating synchrony and asynchrony. Univ. Edinburgh,Dep. of Comp.Sce. (dec 1980) .

[18] **Milner R.**:Synthesis of communicating behaviour. LNCS 64 (april 1978).

[19] **Milner R.**:An algebraic theory for synchronisation . LNCS 67 .

[20] **Pettorossi A.**:Organizing parallelism and communications for efficient distributed computations . CSSCCA-CNR Roma and Univ.Edingburgh Comp. Lab.

[21] **Pettorossi A.**:Toward a theory of parallelism and communications for increasing efficiency in applicative langages. LNCS 148 .

PI 217 **L'IRISA vu à travers les stages effectués par ses étudiants de DEA (1ère année de thèse) Edition 1983-1984**
Daniel Herman, 28 pages ; Janvier 1984.

PI 218 **Contribution de la classification automatique pour l'organisation et l'interrogation d'un corpus de « petites annonces »**
Philippe Peter, 32 pages ; Janvier 1984.

PI 219 **A micro-computer implementation of an interactive functional programming system**
Wei Zi Chu, 22 pages ; Janvier 1984.

PI 220 **Commande en boucle fermée de robots munis de capteurs extéroceptifs terminaux**
Bernard Espiau, 81 pages ; Janvier 1984.

PI 221 **Justification et validité statistique d'une échelle 0,1 de fréquence mathématique pour une structure de proximité sur un ensemble de variables observées**
I.C. Lerman, 48 pages ; Janvier 1984.

PI 222 **Spécification d'une machine de gestion mémoire pour les interpréteurs des langages logiques - version 1 (provisoire)**
Yves Bekkers, Bernard Canet, Ollivier Ridoux, Lucien Ungaro, 82 pages ; Février 1984.

PI 223 **Une approche à la validation des protocoles d'Enchère par la méthode des tests de spécification**
Christine Ecault, 36 pages ; Février 1984.

PI 224 **Présentation simplifiée d'une machine de gestion de mémoire pour les interpréteurs Prolog**
Yves Bekkers, Bernard Canet, Ollivier Ridoux, Lucien Ungaro, 20 pages ; Février 1984.

PI 225 **Algorithmes adaptatifs pour l'estimation du décalage entre deux signaux numériques**
Michèle Basseville, Danielle Pelé, 43 pages ; Avril 1984.

PI 226 **Modélisation avec pivot pour une loi générale**
Jean Pellaumail, 11 pages ; Mai 1984.

PI 227 **Systèmes de processus communicants et interprétation parallèle de langages fonctionnels**
Boubakar Gamatié, 30 pages ; Juin 1984.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique