ORSAY
n° d'ordre :

UNIVERSITE DE PARIS-SUD
CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'UNIVERSITE PARIS XI ORSAY

PAR

Jean-Louis Giavitto

Sujet :

$8_{1/2}$: un modèle MSIMD pour la simulation massivement parallèle

Soutenue le 10 Décembre 1991 devant la Comission d'examen :

MM.	Joffroy Beauquier	*Président*
	Guy Mazaré	*Rapporteur*
	Patrick Sallé	*Rapporteur*
	Gérard Berry	*Examinateur*
	Giovanny Coray	*Examinateur*
	Jean-Paul Sansonnet	*Directeur de la Thèse*

Remerciements

Je remercie Monsieur Joffroy Beauquier, Professeur au LRI, qui a bien voulu présider le jury de cette thèse. Monsieur Guy Mazaré, Professeur à l'INPG, et Monsieur Patrick Sallé, Professeur à l'IRIT ont commenté et évalué ce travail en acceptant la lourde tâche d'être rapporteurs; qu'ils acceptent mes sincères remerciements. Monsieur Gérard Berry, Professeur à l'École des Mines, et Monsieur Giovanni Coray, Professeur à l'EPF de Lausanne, ont eut la gentillesse de participer à ce jury et je les en remercie.

Monsieur Philippe Clermont m'a permis d'accéder à la Connection-Machine du Centre d'Études en Hyperparallélisme de l'ETCA, ce qui a beaucoup influé sur la direction prise par cette thèse. Les remarques de Monsieur Guy Vidal-Naquet ont toujours été éclairantes et les commentaires de Monsieur Paul Caspi me montrent combien l'expérience des langages temps-réel synchrones peut être profitable à la programmation parallèle.

C'est avec beaucoup de plaisir que je remercie Jean-Paul Sansonnet et Daniel Etiemble qui m'ont accueilli dans leur équipe et qui m'ont toujours soutenu. Je leur dois d'autant plus de gratitude que c'est une équipe formidable, dans lequel il fait bon travailler. Jean-Luc Béchennec, Franck Cappello, Franck Delaplace et Cécile Germain ont toujours eu l'oreille attentive et la remarque mordante. Je les remercie tous très vivement.

Fabrice Legrand a participé au développement de la première version de l'interprète 81/2. Pour quelqu'un qui supporte aussi difficilement que moi les lignes de codes écrites par les autres, son art de la programmation fut un véritable soulagement. S. Dumesnil, I. Biermann, F. Charlot, C. Mihail, X. Panet, E. Bonabeau, L. Leboucher ont, à des titres divers, développé les premiers exemples en 81/2.

Françoise Kieger, Pierre Esposito et Christophe Godin ont découvert, au risque d'éprouver notre solide amitié, mon orthographe, et m'ont supporté durant la rédaction de ce mémoire. Enfin, *A.*, *A.* et *A.*, ces démons familiers, peuvent être rassurés : il m'ont bien hanté durant ce travail.

À mes parents,
avec mes tendres excuses pour ce qui suit…

Marco Polo décrit un pont, pierre par pierre.

— Mais laquelle est la pierre qui soutient le pont ? demande Kublai Khan.

— Le pont n'est pas soutenu par telle ou telle pierre, répond Marco, mais par la ligne de l'arc qu'à elles toutes elles forment.

Kublai Khan reste silencieux, il réfléchit. Puis il ajoute :

— Pourquoi me parles-tu des pierres ? C'est l'arc seul qui m'intéresse.

Polo répond :

— Sans pierres il n'y a pas d'arc.

Italo Calvino, Les villes invisibles.

Tables des Matières

Sommaire

Ce mémoire propose un environnement permettant de **très grandes simulations digitales**. La réalisation de grandes simulations s'appuie sur l'utilisation efficace des grands calculateurs parallèles dont nous disposerons dans un proche avenir. Par ailleurs, énormément d'applications de simulation correspondent à des programmes dont on peut prévoir, à la compilation, le déroulement des calculs. C'est par exemple le cas de la simulation des systèmes dynamiques discrets (systèmes de contrôle/commande, mécanismes économiques, réseaux d'automates, circuits VLSI synchrones, systèmes continus discrétisés) qui constitueront un exemple privilégié d'applications. En conséquence, nous étudierons plus précisément **un modèle pour la programmation des applications à comportement prévisible sur les ordinateurs massivement parallèles**.

Notre objectif est double : *1*) offrir l'expressivité nécessaire aux applications de simulation, et en particuliers à la simulation des systèmes dynamiques discrets; *2*) préserver la simplicité et l'efficacité des modèles d'exécution SIMD tout en gagnant le rendement et la flexibilité des structures de contrôle offerts par le MIMD. Afin de pouvoir étudier plus précisément notre modèle de programmation, les ressources spécifiques de la *simulation massivement parallèle* vont constituer les entités explicites d'un langage expérimental appelé **Otto e Mezzo**, ou encore **8½**.

Le développement de Otto e Mezzo a été influencé par les premiers enseignements du projet **MEGA**, par la programmation de la **Connection-Machine** et les langages **LUCID**, **LUSTRE** et **SIGNAL**. La Connection-Machine nous a apporté le concept de collection et LUCID la notion de suite temporelle. MEGA explore les implications du parallélisme massif : quels sont les architectures et les modèles d'exécutions nécessaires à la réalisation et à l'utilisation d'ordinateurs comportant plusieurs milliers de processeurs. Les deux principales conséquences ont été le développement d'un langage *statique* permettant l'expression à la fois du *parallélisme de données* et du *parallélisme de contrôle*. Le langage étant statique, les principaux paramètres de l'exécution sont fixés à la compilation.

D'un point de vue technique, les apports de ce travail consistent en l'intégration du concept de *collection* et du concept de *suite temporelle* dans un langage *data-flow* et sa **compilation** pour une architecture MIMD synchronisable *massive*.

La collection est le support privilégié du parallélisme de données, tandis que les suites temporelles permettent un traitement statique du parallélisme de contrôle. Du point de vue des applications de simulation, les suites temporelles correspondent à la notion de trajectoire et les collections à celle de variable multi-dimensionelle.

La fusion des concepts de collection et de suite temporelle permet de répondre aux objections initialement soulevées à l'encontre du traitement inefficace des tableaux dans le modèle data-flow classique. De plus, cette extension permet de prendre en compte le problème de la répartition des données dans une architecture à mémoire distribuée, problème traditionnellement absent des préoccupations du data-flow. Le résultat est un langage MSIMD dans lequel le parallélisme est implicite, statique et à grain très fin.

La forme data-flow du langage simplifie les techniques d'analyse et d'optimisation des programmes. Nous en donnons plusieurs exemples à travers l'inférence du type des collections, l'analyse du graphe des dépendances, l'analyse temporelle des programmes, la propagation des constantes et l'optimisation des chemins critiques. La porte reste ouverte à de nombreuses analyses, optimisations et transformations de programme.

La compilation est rendue indispensable par les performances recherchées. Elle repose sur la sémantique du langage et se traduit par la résolution distribuée d'un système d'équations. La compilation donne une « forme triangulaire » à ce système, ce qui se traduit par une résolution efficace. La gestion de l'activité des tâches correspondantes n'est pas réalisée par un exécutif mais réglée dès la compilation par un ordonnancement statique de toutes les activités. Cette phase de la compilation permet de répartir les données en profitant d'une connaissance très fine de la structure des calculs et en tenant compte des coûts de communication. Une attention toute particulière a été accordée à la mise en œuvre de l'ordonnancement, afin que celui-ci ait une complexité compatible avec les temps de réponse attendus pour une compilation.

Ce travail a conduit au développement, à partir des techniques présentées dans cette thèse, d'un interprète/compilateur d'un sous-ensemble du langage 81/2. Cet interprète existe pour l'instant sur station de travail UNIX. La génération de code séquentielle (station de travail et ordinateur SIMD) doit aboutir à très court terme (début 92). L'implémentation de la génération de code pour une machine MIMD synchronisable (comme par exemple la machine PTAH développée dans le projet MEGA) est en cours et permettra l'exploitation effective du parallélisme de données et du parallélisme de contrôle.

∴

Organisation du rapport

Le premier chapitre pose le cadre de cette thèse : un modèle de calcul à grain très fin MSIMD data-flow pour les applications à comportement prévisible sur machine massivement parallèle. Un exemple important d'application à comportement prévisible et requérant une énorme puissance de calcul est détaillé : il s'agit de la simulation des systèmes dynamiques discrets. Le chapitre se termine par l'examen de deux structures de données fondamentales qui vont servir de support au parallélisme de données et de contrôle dans notre modèle : les collections et les streams.

Le langage **81/2**, qui correspond au modèle introduit dans le premier chapitre, est ensuite décrit. Le deuxième chapitre se poursuit par des exemples de programmes. À la fin du chapitre, une comparaison est faite avec des langages existants dans le domaine de la programmation data-flow, de la programmation temps-réel, de la programmation parallèle et de la programmation systolique.

La sémantique formelle du langage est présentée dans le style dénotationnel dans le chapitre *III*. Les chapitres *IV* et *V* sont dédiés à la sémantique statique. L'objectif est d'alléger l'écriture des programmes et d'interdire certaines expressions qui ont un comportement incompatible avec le caractère statique que l'on veut garder au langage. La principale difficulté vient du fait que les techniques mises en œuvre doivent être capables d'analyser des programmes impliquant des millions d'entités.

Le chapitre *VI* fournit les techniques nécessaires à la détection des expressions générant à coup sûr des famines ou des étreintes fatales. L'approche adoptée est basée sur le calcul d'un type représentant la structure temporelle du calcul d'une expression. La dépendance de cette structure vis-à-vis des données est modélisée à travers la notion de type existentiel.

Le chapitre *VII* présente le modèle d'exécution data-flow dynamique qui correspond au schéma naturel de l'évaluation par nécessité d'une expression 81/2. Ce modèle d'exécution n'a pas été implanté : nous l'examinons afin de mieux l'écarter pour un modèle d'exécution statique.

Le modèle d'exécution statique et le principe de la compilation sont détaillés dans le chapitre VIII et dans le chapitre *IX*. Le premier s'intéresse à la génération des tâches correspondant à l'évaluation des expressions 81/2. Le chapitre *IX* se concentre sur le problème du placement et de l'ordonnancement de ces tâches.

Le chapitre *X* présente l'environnement 81/2. Cet environnement est basé sur un interprète du langage et incorpore les diverses analyses nécessaires à la compilation.

Le chapitre *XI* évoque deux optimisations possibles du code généré, l'une basée sur la détection des expressions constantes et utilisant les techniques du chapitre VI, l'autre basée sur une transformation de programme permettant de réduire la durée d'exécution du chemin critique par déplacement des opérations de délai.

La conclusion place ce travail dans une perspective plus large et discute des apports mutuels de l'analyse des systèmes dynamiques et de la programmation des calculateurs parallèles.

Le lecteur trouvera à la fin de ce rapport une bibliographie commentée et une table des matières analytique.

I. En quête d'un langage de programmation massivement parallèle

L'objectif de cette thèse est de déterminer un modèle d'exécution adapté aux grandes simulations. Ces simulations sont très coûteuses en temps : de nombreuses simulations sont nécessaires à l'étude d'un système et les modèles deviennent de plus en plus complexes, intégrant de nouveaux paramètres et incorporant de plus en plus d'éléments. Elles nécessitent des puissances de calcul qui seules peuvent être atteintes à l'aide d'un *calculateur massivement parallèle*. Ces ordinateurs visent à obtenir une puissance de l'ordre du tera-ops (10^{12} opérations par seconde) à l'aide d'un très grand nombre de processeurs élémentaires qui communiquent entre eux. Cela implique l'expression et l'exploitation du parallélisme à un niveau très fin dans les programmes. Notre cadre de travail est le suivant :

- Les applications envisagées correspondent à des calculs dont le comportement est prévisible.
- La programmation des machines massivement parallèles requiert des modèles de calcul spécifiques qui permettent l'exploitation du parallélisme de données et du parallélisme de contrôle.

Nous allons détailler ces hypothèses et leurs conséquences au cours des cinq parties que comporte ce chapitre :

i) Nous allons tout d'abord caractériser les applications qui nécessitent l'usage des calculateurs parallèles. Parmi celles-ci, la plus grande part correspond aux applications *statiques*, c'est-à-dire à des applications dont le comportement est prévisible. Cette propriété permet d'utiliser des programmes à parallélisme statique qui utilisent efficacement les ressources matérielles du calculateur parallèle. Au contraire, les programmes dont le comportement est imprévisible génèrent de nombreuse difficulté qui rendent difficile une implémentation effective.

ii) Une très importante classe d'applications à comportement prévisible est celle qui correspond à la simulation des systèmes dynamiques discrets (SDD). Le formalisme des SDD intervient dans la modélisation de nombreux systèmes produits par l'homme et dans la discrétisation de nombreux systèmes « naturels ». La simulation des SDD constitue nos applications privilégiées. Elle vérifie et justifie nos hypothèses de travail.

iii) Le terme de « massivement parallèle » a été employé pour des ordinateurs très différents; aussi nous commençons par fixer notre propre définition : un calculateur massivement parallèle doit comporter beaucoup de processeurs et exploiter un parallélisme à grain très fin.

Les deux grandes familles d'architecture parallèle ont généré des langages spécifiques correspondant à l'expression du parallélisme de données et du parallélisme de contrôle. La critique de ces langages conduit à vouloir combiner leurs caractéristiques, d'autant plus que des architectures hybrides, capables d'exploiter les deux sources de parallélisme, deviennent disponibles.

iv) La combinaison des deux modèles de programmation conduit à la définition du *MSIMD data-flow*. Nous présentons plusieurs des propriétés caractéristiques de ce modèle qui en font le modèle naturel de la programmation massivement parallèle des applications à comportement prévisible.

v) Dans la dernière partie, nous détaillons les deux structures de données qui permettent l'expression *statique* du parallélisme de contrôle *et* du parallélisme de données dans un modèle « MSIMD data-flow ». Ces entités, qui interviennent naturellement dans les applications de simulation, constituent les ressources d'un langage expérimental appelé **Otto e Mezzo** (ou **8½** en abrégé) qui nous permettra d'étudier plus précisément notre modèle de programmation. Ce travail nous occupera pendant le reste de cette thèse.

I. Les applications à comportement prévisible

Deux grands types d'applications nécessitent l'usage d'un ordinateur parallèle : les applications qui doivent manipuler de grand volume de données et/ou de grand volume de calcul, et les applications qui sont par nature réparties et concurrentes (comme par exemple le « mail » sur un réseau d'ordinateurs). Nous utiliserons cependant un autre critère pour classer les applications du parallélisme : le critère statique/dynamique (Cf. tableau 1 ci-dessous).

<table>
<tr><th colspan="3">Applications Statiques</th><th>Applications dynamiques</th></tr>
<tr><td rowspan="2">. réaction chimique
. dynamique moléculaire

. Mécanique quantique
. balistique
. imagerie
. …</td><td colspan="2">simulation des systèmes dynamiques discrets</td><td rowspan="2">concurrence

(applications réparties)</td></tr>
<tr><td>systèmes dynamiques continus discrétisés

. éléments et différences finis
. mécaniques des fluides
. …</td><td>. croissance des cristaux
. trafic aérien
. modélisation socio-économique
. vie artificielle
. …</td><td>autres
(démonstration automatique, système expert)</td></tr>
</table>

Table 1 : Une classification des applications impliquant de gros volumes de données et/ou de calculs suivant le caractère prévisible ou dynamique des programmes. La simulation des systèmes dynamiques discrets n'implique pas nécessairement l'usage intensif de la virgule flottante.

Examinons d'un peu plus près le classement de la table 1. Les applications concurrentes ne demandent pas a priori de grandes puissances de calcul; elles utilisent des calculateurs parallèles (« distribués » serait un qualificatif plus adéquat) car c'est dans leur nature. À l'opposé, les applications qui nécessitent de grand volume de calcul et de données utilisent le parallélisme comme un moyen et non comme une fin. Les applications de calcul intensif qui ne sont pas statiques, comme par exemple la recherche dans un arbre d'états, correspondent en fait à des applications de complexité non-polynomiale (sinon on aurait pu les classer dans les applications statiques, en envisageant « le cas pire »). On peut alors douter de l'utilité d'un ordinateur, même parallèle, pour traiter ces dernières applications.

Un programme à *parallélisme statique* est un programme dont le comportement est prévisible, c'est-à-dire dont on peut prévoir avant l'exécution, les principales caractéristiques du déroulement des calculs :

- nombre de tâches et leur répartition;
- occupation et utilisation de la mémoire;
- temps de calcul des différentes tâches;
- communications effectuées entre tâches;
- ordonnancement des tâches;
- etc.

Il y a plusieurs degrés de prévisibilité. Dans un algorithme systolique, tout est prévisible. Un programme *LISP sur la Connection-Machine est déjà moins prévisible puisque les communications peuvent être quelconques (mais on peut noter qu'on essaie de rendre ces communications prévisibles quand c'est possible, Cf. [Dahl 90]). Un programme acteur typique est complètement imprévisible : on ne peut pas prévoir le nombre d'acteurs qui sera créé ni qui communique avec qui.

Par définition, les applications statiques correspondent à des programmes qui utilise un parallélisme statique. Mais on peut bien sur utiliser un langage permettant le parallélisme dynamique pour l'implémentation d'une application statique (« qui peut le plus peu le moins »).

Les programmes statiques permettent une gestion la plus efficace possible des ressources matérielles en prévoyant leur utilisation à la compilation. Par exemple, supposons qu'une tâche *A* calcule une valeur utilisée par une tâche *B* sur un autre processeur. Si les calculs effectués par *A* et par *B* sont prévisibles, alors il est possible d'émettre la valeur produite par *A* et de la conserver localement sur le processeur alloué à *B*, afin d'être utilisée aussi tôt que possible et sans délai de communication.

I.1. Le parallélisme dynamique

Au contraire, dans un environnement imprévisible, ou dynamique, il faut être constamment préparé à répondre à des demandes arbitraires de nouvelles ressources (par exemple la tâche *B* demande à la tâche *A* sa valeur quand il en a besoin). Cette disponibilité se paye en termes de messages, de buffers, en attente active, en délais de transmission, en coût de gestion, etc.

Nous soutenons ici la position suivante : *le parallélisme dynamique à grain fin n'est pas utilisable efficacement*. En conséquence, nous tournerons notre attention vers les modèles de programmation statique. Mais auparavant, nous allons motiver notre jugement.

Les applications parallèles dynamiques se traduisent entre autres par des créations de tâches au cours du calcul. Si on prend l'exemple de l'exploration parallèle d'un arbre de recherche, les tâches qui sont créées correspondent à l'exploration d'une nouvelle branche. La gestion dynamique des tâches doit alors faire faces à trois sortes de problèmes : *saturation de la machine, récupération des tâches mortes* et *distribution dynamique de charge*.

La saturation de la machine intervient rapidement dans le cas du parallélisme à grain fin : il y a davantage de tâches ou d'opérations que de PE. Il faut donc gérer la prolifération des tâches. Dans le cas du parallélisme implicite, on est par exemple obliger d'introduire des annotations dans les programmes afin d'interdire le parallélisme (Cf. par [MaRS 89]). Mais l'usage statique de ces annotations est complètement inadapté : le fait qu'une évaluation doit se faire en parallèle ou non dépend de l'état du PE où se fait l'évaluation et non de sa localisation statique dans le code source du programme. On en arrive alors à des mécanismes dynamiques de gestion du parallélisme au cours de l'exécution. Ces mécanismes relèvent de la répartition dynamique de la charge de calcul.

La répartition dynamique de la charge a pour but de répartir les calculs dans la machine. Mais plus le parallélisme est à grain fin, plus la migration d'une tâche (ou d'une opération) devient coûteux. En fait il n'est pas concevable de faire migrer les tâches quand celles-ci ont moins d'une dizaine d'instructions. Donnons l'exemple d'un

langage acteur : la tâche correspond au traitement d'un message par un acteur. C'est une entité trop petite pour pouvoir migrer. L'entité que le mécanisme de répartition de charge peut faire migrer c'est l'acteur. Mais un acteur ne représente pas une quantité de calcul connue (certains acteurs traiteront un ou deux messages, et d'autres des centaines, [GGF 91]). La migration des acteurs n'est donc pas adaptée à la répartition de charge. L'argument se généralise pour tous les langages à grain fin : la tâche est trop petite pour migrer et donc la répartition de charge doit se faire par la migration d'entités qui ne représentent pas une charge de calcul connue. La répartition de la charge est donc un problème difficile. Et ce problème est d'autant plus difficile que le nombre de processeurs est grand. Dans ce cas il n'est plus possible de maintenir une connaissance parfaite et instantanée de l'activité des PE. Il faut alors utiliser des algorithmes locaux de répartitions. Nous ferons une critique de ces algorithmes dans le chapitre VII.

Par ailleurs, de grandes quantités de tâches peuvent être créées et détruites dynamiquement [GGF 91]. Il faut donc disposer d'un ramasse-miette distribué : dans le cas d'un langage d'acteur, il faut récupérer les ressources allouées à un acteur qui ne recevra plus de messages; dans le cas d'un langage fonctionnel comme CRISTAL-SCHEME [Queinnec 90], il faut récupérer les continuations et les listes inatteignables. On trouvera dans [CMP 89] un état de l'art. On peut demander les propriétés suivantes à un bon algorithme distribué de récupération [CMP 89] [Lieberman, Hewitt 83] : localité et minimisation des communications, minimisation des synchronisations nécessaires, traitement simple des messages en cours d'acheminement (*flying references*), libération au plus tôt des ressources occupées par les tâches mortes, indépendance vis-à-vis des paramètres physiques (topologie du réseau, nombre de PE) et bien sur, faible coût de calcul. Les algorithmes basés sur le marquage [Hudak, Keller 82] [Hughes 85] des tâches vivantes ont plusieurs inconvénients dont le plus important est sans doute leur coût (il faut suspendre l'activité de toute la machine, faire la passe de marquage puis la passe de récupération). Les méthodes basées sur les compteurs de références [Beckerle, Ekanadham 86] [Bevan 87] [Watson, Watson 87] ne nécessitent pas de synchronisations globales et résolvent le problème des messages en chemin. Par contre, les tâches prises dans une chaîne de références circulaires ne sont pas libérées. De plus, car le parallélisme est fin, le coût d'un compteur de référence devient élevé. Dans l'état actuel, avec les contraintes que nous avons, le problème de la récupération est un problème important et non résolu.

I.2. Un modèle d'exécution statique pour des applications statiques

Évidemment, il est toujours possible d'utiliser le parallélisme dynamique d'une manière essentiellement statique. On peut par exemple créer un acteur par élément d'un tableau pour le manipuler ensuite dans un style SIMD. Mais dans cette utilisation d'un style de programmation pour un autre, on a perdu de l'information (par exemple on ne peut indiquer à l'exécutif combien il y aura d'acteurs qui représenteront les éléments du tableau). L'implémentation restera toujours moins efficace que celle qui aurait été faite dans un langage statique.

En conclusion, un modèle d'exécution pour une machine massivement parallèle doit obligatoirement être un modèle d'exécution statique car :

- les machines massivement parallèles ont pour objet de fournir de grandes puissances de calculs grâce au parallélisme à grain fin;
- les applications statiques sont, parmi les applications parallèles, celles qui nécessitent de grandes puissances de calculs;
- Un modèle d'exécution dynamique, même s'il permet une expressivité plus grande, ne permettra pas une gestion efficace des ressources de la machine, même s'il est utilisé de manière statique.

Le programmeur doit donc faire l'effort de formuler son problème de telle manière que le programme résultant soit à comportement prévisible, sous peine de ne pas pouvoir utiliser efficacement la puissance de calcul de son ordinateur massivement parallèle.

II. La simulation des systèmes dynamiques discrets : un exemple d'application statique

Avant de tourner notre attention vers les modèles de programmation à comportement prévisible, nous voulons décrire dans cette section une classe très importante des applications statiques. Cette classe correspond aux applications de simulation des systèmes dynamiques discrets. Nous allons montrer que ces applications sont des applications à comportement prévisible. Nous développerons donc dans la section suivante un modèle de programmation statique puisque la section précédente nous a montré leur intérêt par rapport aux modèles dynamiques.

II.1. Les systèmes dynamiques discrets

Notre acception d'un *système* est celle d'un ensemble d'entités telles qu'on ne peut définir la fonction ou les variations de l'une, indépendamment de celles des autres. Les entités constituant le système sont caractérisées par une valeur appelée paramètre ou variable. Les variations dans le temps de ces valeurs décrivent le système. Certaines variables voient leurs valeurs arbitrairement imposées : ce sont les *entrées* du système (ou *inputs*, *stimuli*, *afférences*, *admissions* [Walliser 77]); d'autres variables sont dites observables : ce sont ses *sorties* (ou encore *outputs*, *réponses*, *efférences*, *émissions*); le reste des variables définissent l'*état interne* du système.

Un système dynamique discret (**SDD**) est un système où les variations des variables dépendent d'événements discrets et où les valeurs des variables sont d'un type discret. Ainsi l'état du système ne change qu'à des instants donnés plutôt que continûment et la valeur d'une variable est décrite par un nombre fini[1] de valeurs d'un type scalaire (booléen, entier, réel).

Les SDD interviennent souvent dans la modélisation des systèmes artificiels et dans la discrétisation des systèmes naturels. Parmi les systèmes produits par l'homme, on peut citer :

. *systèmes de productions* (chaîne d'assemblage, processus de fabrication),

. *modélisation des mécanisme économique*,

. *réseaux d'ordinateurs*,

. *modélisations de trafics* (automobile, aérien, etc),

. *systèmes de files d'attentes*,

. *systèmes de contrôle et de commande* (pilotage d'une centrale nucléaire),

. *réseaux d'automates* (commande d'un ascenseur),

. *modèles connexionistes* (réseaux de neurones formels),

. *circuits VLSI synchrones*,

. *modélisations event-driven* (modèles de la physique qualitative),

. *modélisations des systèmes à états logiques ou symboliques* (comme par exemple l'exécution d'un programme parallèle),

. …

La modélisation des systèmes « naturels » donne souvent lieu à des modèles continus, mais les modèles continus se réduisent souvent à des SDD. En effet, le traitement mathématique des systèmes continus passe par le formalisme

[1] Par exemple une image de cinéma ne constitue pas un SDD car c'est une variable dont la valeur change 24 fois par seconde mais ce n'est pas une variable à valeur discrète : cette valeur est décrite par une fonction $\varphi(x, y)$ où (x, y) repèrent continûment un point de l'écran.

des équations différentielles. Une équation différentielle décrit une relation qui doit être vérifiée par les variables du système. La solution analytique d'une équation différentielle permet d'expliciter les valeurs des variables du système en fonction d'un petit nombre de variables (le temps et les entrées qui constituent les conditions aux limites). Seul un nombre limité de types d'équations différentielles a été résolu analytiquement et encore le caractère opératoire des fonctions obtenues dépend-il de la simplicité des conditions aux limites (Cf. [Smith 85, *préface*]). C'est pourquoi le seul moyen de décrire le comportement du système passe souvent par les méthodes d'approximations. Les méthodes d'approximations numériques, comme les différences finies ou les éléments finis, discrétisent le temps et les variables d'entrées. La solution est approchée en remplaçant l'équation différentielle par une équation algébrique portant sur la valeur des variables aux points de discrétisation. Ainsi, une classe très importante de systèmes dynamiques discrets provient de la discrétisation des systèmes dynamiques continus. L'approximation numérique de ces systèmes constitue une grande part du traitement numérique intensif.

Il est indubitable que les SDD prennent une importance toujours croissante, en particulier en Recherche Opérationnelle [IEEE 89], en Théorie des Systèmes [Pagels 88] et en Intelligence Artificielle. Pour ce dernier domaine, les systèmes dynamiques discrets permettent de modéliser des systèmes dont les états sont des états logiques ou symboliques (préoccupation de la physique qualitative par exemple). Ils sont aussi au centre des nouvelles préoccupations des sciences cognitives : qu'on pense aux *modèles connexionistes* proposés dans [PDP 86] ou aux *systèmes adaptatifs* présentés dans [SAB 90] ou [PPSN 90].

II.2. Quel langage pour la description des SDD ?

La *trajectoire* d'un système est la suite temporelle des valeurs des variables du système. La trajectoire d'un SDD est une courbe constante par morceaux. Chaque segment représente un état et la durée du segment représente le temps passé dans cet état (Cf. figure 1 ci-dessous). La suite de segments représente la suite des états du système.

Si les équations différentielles se sont imposées comme le paradigme de description des systèmes dynamiques continus, aucun formalisme n'a atteint un tel consensus pour les SDD [Ho 89]. Parmi les langages de description des SDD, on trouve :

> les automates à états finis (StateCharts), les réseaux de petri (colorés, temporisés, …), la logique temporelle, les chaînes de Markov (semi-chaînes de Markov, chaînes de Markov hiérarchisées, …), les réseaux de files d'attentes, les algèbres min-max, les algèbres de processus (CSP, CCS, …), les langages de simulation ad-hoc (Qnap2, REDUCE, ECO, …), …

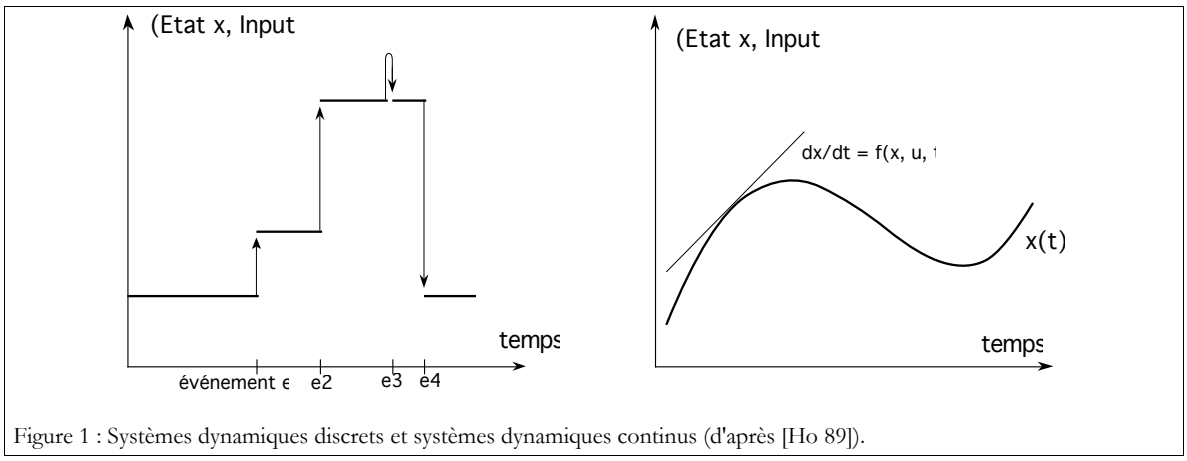


Figure 1 : Systèmes dynamiques discrets et systèmes dynamiques continus (d'après [Ho 89]).

Ces formalismes ne concernent pas uniquement les SDD et trouvent leurs applications dans d'autres domaines. Leur foisonnement provient de la diversité des questions à résoudre. En première approche on peut les classer par

deux critères : stochastiques/déterministes et temporisés/non-temporisés :

- Les modèles stochastiques/déterministes :
  Dans un modèle stochastique, les lois qui déterminent la trajectoire du système ne sont pas connues exactement mais on a une information sur la fréquence des décisions qui seront prises à chaque pas d'évolution du système. Les questions auxquelles on peut répondre à l'aide de ce genre de modèles sont des questions concernant non pas une trajectoire précise, mais un ensemble de trajectoires.
  Dans un modèle déterministe la suite des états est connue précisément, ou alors on connaît une propriété vérifiée par toutes les trajectoires possibles du système. En effet, des formalismes comme les algèbres de processus permettent de spécifier des systèmes indéterministes. Ce ne sont pas pour autant des modèles stochastiques, en ce qu'on ne s'intéresse pas à des propriétés vérifiées par un ensemble de trajectoires, mais à des propriétés vérifiées par toutes les trajectoires possibles.
- Les modèles temporisés/non-temporisés :
  Les modèles temporisés sont capables de répondre à des questions comme « Combien d'événements d'un certain type auront lieu dans un certain intervalle de temps ? », « Combien de temps faut-il attendre pour voir apparaître un événement d'un certain type ? ».
  Dans les modèles non-temporisés le temps ne fait pas partie intégrante du modèle : on ne s'intéresse pas au temps passé dans un état, mais uniquement à la succession des états.

Dans ce mémoire, nous sommes essentiellement intéressés par les modèles déterministes non-temporisés, c'est-à-dire par *le calcul de la suite des états d'un système*. En effet, il suffit d'introduire une horloge, i.e. une variable dont les événements mesurent l'écoulement du temps, pour pouvoir traiter certaines questions temporelles dans le cadre d'un système non-temporisé.

II.3. Le calcul de la suite des états d'un système

On peut aborder le problème du calcul de la trajectoire d'un système par au moins deux voies différentes : en donnant le principe d'évolution des valeurs des variables en fonction du temps, ou bien en donnant les relations qu'elles doivent respecter et les valeurs initiales. Cette dernière forme correspond à une présentation structurelle du système tandis que la première correspond à une description de sa dynamique.

Pour expliquer la différence entre les deux approches, prenons l'exemple d'un point mobile astreint à se déplacer sur un cercle. On peut définir un cercle comme le lieu géométrique des points qui vérifie une équation (E1) ou bien comme la trajectoire du point (x, y) quand t varie entre 0 et 2π (E2) :

$$\text{(E1) :} \quad 1 = x^2 + y^2 \qquad\qquad \text{(E2) :} \quad \begin{cases} x = \sin(t) \\ y = \cos(t) \end{cases}$$

l'équation (E1) est une description structurelle : la définition de la trajectoire du point se fait implicitement par une propriété caractéristique vérifiée par tous les éléments de cette trajectoire. La définition (E2) est une définition explicite de la trajectoire en fonction du temps (qui définit implicitement la structure de la trajectoire). Elle comporte plus d'informations que la définition (E1) car elle donne une relation précise entre la position du point et le temps.

La présentation structurelle d'un système présente un aspect plus intentionnel : c'est une approche qui est plus *explicative* que *descriptive*. Ainsi, nous préférons expliquer la chute d'une pomme comme un résultat de la relation liant deux corps, la terre et la pomme, plutôt que de simplement décrire son mouvement accéléré vers le bas. C'est à partir de descriptions structurelles qu'on construit de nouveaux systèmes. Le problème qui se pose est de pouvoir calculer les trajectoires à partir des relations qui décrivent le système, c'est-à-dire de passer d'un système du type (E1) à un système de type (E2). En effet, la validation d'un système passe par l'observation de son comportement et donc par le calcul explicite de ses trajectoires.

Il faut bien remarquer qu'il existe plusieurs paramétrages possibles du cercle (c'est-à-dire plusieurs systèmes du

type (E2) décrivant la même trajectoire). Ils diffèrent par la relation entre *t* et la position du point. Mais comme nous l'avons précisé plus haut, nous ne nous intéressons pas à la quantité de temps écoulé (mais uniquement à son écoulement) : nous ne cherchons pas à obtenir un paramétrage particulier, n'importe lequel nous convient.

II.4. Les relations fonctionnelles

L'expression de la relation liant les paramètres d'un système peut prendre des formes très diverses. Dans l'exemple précédent, la relation entre les deux variables décrivant la position du point mobile est du type « $\phi(x, y) = 0$ ». Parmi toutes les formes possibles de relations, il y en a une qui nous intéresse plus particulièrement, c'est la forme d'une relation fonctionnelle

$$y = f(x) \qquad (1)$$

x et ***y*** étant deux variables d'un système et f une fonction quelconque. L'équation précédente lie leurs valeurs. Toute variation de ***x*** entraîne une variation *simultanée* de ***y*** afin de maintenir l'équilibre décrit par l'équation (1). Il faut bien comprendre que la relation (1) est une relation vraie à tout moment : c'est une relation vérifiée par les *trajectoires* de x et de y. On pourrait écrire plus précisément

$$\forall t \in \mathbb{T}, y(t) = f(x(t))$$

avec $\mathbb{T}$ un ensemble dénombrable d'instants. Le système est discret car $\mathbb{T}$ est un ensemble dénombrable et car x et y sont à valeur dans un ensemble scalaire (i.e., ce ne sont pas des fonctions).

Une relation fonctionnelle peut mettre en jeu plusieurs variables, mais une seule apparaît au membre gauche de l'égalité. Par exemple,

$$x = y + z \qquad (2)$$

où ***x***, ***y*** et ***z*** sont des variables dont les valeurs évoluent dans le temps. L'équation (2) exprime une égalité qui doit être vérifiée a tout instant. La figure 2 illustre la relation (2) à l'aide du principe des vases communicants.

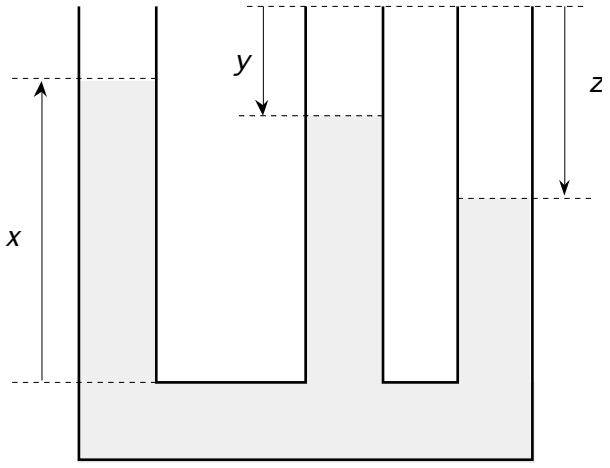


Figure 2 : Illustration de la relation d'équilibre $x = y + z$. On peut imaginer que les quantités y et z sont commandées à travers la position de pistons. Toute variation de y ou de z entraîne à travers le fluide incompressible, une variation correspondante de x. Mais inversement, une variation de x (par l'intermédiaire d'un piston) doit modifier les quantités y *ou* z. De quelle manière ? pour un x donné, il y a plusieurs façons de choisir un couple (y, z) qui réponde à la contrainte $x = y + z$. Quand la relation est fonctionnelle, on indique implicitement qu'on ne se posera pas ce problème : x est fonction de y et de z

La relation (2) est à distinguer soigneusement de la relation

$$0 = x - y - z \qquad (3)$$

car si (3) exprime la même contrainte que (2), il y a une information implicite dans l'équation (2) qui n'est pas

présente dans l'équation (3) : l'équation (2) indique que la quantité x peut se calculer en fonction des valeurs de y et de z. On devrait écrire comme on a l'habitude en physique :

$x := y + z$ ou bien encore $y + z =: x$

pour exprimer que x est *défini* à partir de y et de z.

L'intérêt des relations fonctionnelles est qu'il est possible de calculer explicitement la suite des valeurs des variables au cours du temps. Notre travail fait la supposition suivante : *beaucoup de SDD peuvent être décrits par un ensemble de relations fonctionnelles*. En effet, les modèles sont souvent de type *causal*, ce qui revient à dire que la variation d'une certaine quantité est définie comme conséquence de la variation d'autres quantités; cela s'exprime tout naturellement à l'aide des relations fonctionnelles.

II.5. La simulation

Le recours à la simulation est nécessaire quand elle reste la solution la moins coûteuse (en temps, en effort, …) pour calculer explicitement l'évolution d'un système à partir des relations liant ses paramètres. En effet, ces relations expriment souvent une loi *locale* (le passage d'un état à un autre) et il faut obtenir un résultat *global* (toute la trajectoire du système). Mais il n'est pas toujours possible de trouver une formule simple permettant de donner immédiatement les valeurs des variables du système à un instant `t` sans devoir calculer ces valeurs à tous les instants précédents.

La simulation intervient aussi en analyse des systèmes quand on veut obtenir un modèle d'un système de type « boîte noire ». De ce genre de systèmes on ne connaît que les entrées et les sorties. On adopte alors une démarche expérimentale en construisant des systèmes artificiels et en observant leurs évolutions par simulation. La simulation permet de tester différentes hypothèses sur la nature du mécanisme caché dans la boite noire, en comparant les résultats de la simulation au système réel [Zeigler 89].

II.6. La discrétisation

Calculer explicitement les valeurs des variables d'un système à partir d'un ensemble d'équations du type (1) peut se faire sur un calculateur digital à la condition de disposer d'un modèle discret du système. C'est-à-dire : *a*) le nombre de variables décrivant le système est fini, ainsi que le nombre d'équations exprimant les relations entre ces variables ; *b*) les valeurs des variables peuvent se représenter de manière discrète; *c*) les variations des valeurs des variables se produisent à des instants qui sont en nombre dénombrable. Ces trois hypothèses étant satisfaites par les SDD, il suffit pour chaque instant où une variation d'un des paramètres du système se produit, de propager le changement pour calculer les variations conséquentes.

II.7. Un modèle de calcul pour les applications de simulation

Une constatation préliminaire est que la simulation d'un SDD peut nécessiter d'énormes ressources de calcul. En effet,

- La simulation utilise beaucoup de variables, met en jeu des relations complexes et doit se dérouler pour une longue période de temps.
- Lorsque le SDD simulé correspond à la discrétisation d'un système dynamique continu, le pas de discrétisation peut être très fin ce qui implique un très grand nombre de variables. Qu'on pense par exemple à la discrétisation d'une image par une grille 128×128 : la simulation comportera une variable dont la valeur est un vecteur de dimension 16384.
- Les résultats de la simulation sont recalculés de nombreuses fois avec des entrées différentes. C'est par

exemple le cas du cycle conception/simulation/optimisation que l'on peut rencontrer lors de la création de la coque d'un avion. C'est encore le cas quand on désire obtenir ou vérifier des propriétés statistiques sur le comportement d'un système.

Deux illustrations permettent d'insister sur les très grandes puissances de calcul qui sont requises :

- [Rhigter, Walrand 89] donne l'exemple de la simulation d'un réseau de Jackson (M/M/1 files d'attentes, les entrées suivent une loi de Poisson et le routage est aléatoire) avec un trafic d'intensité 0,5. La recherche d'un cycle d'attente du réseau demandera de simuler le réseau pendant une durée de l'ordre de une à deux semaines sur une machine de 10 MIPS. Et la recherche de plusieurs cycles d'attente est nécessaire quand on veut estimer la valeur des intervalles de confiances du réseau.
- L'autre exemple est tiré de [Collins, Jefferson 90] et concerne la simulation de l'évolution d'une colonie de fourmis. Chaque organisme individuel possède une représentation propre, ainsi que chaque gène de cet organisme. Chaque événement ayant une influence biologique est simulé séparément. Une simulation à un tel niveau de précision se dénomme simulation micro-analytique. La simulation d'un environnement comprenant une dizaine de milliers d'organismes, comprenant un génome de plusieurs milliers de bits, et se poursuivant sur une centaine de générations, nécessite l'utilisation d'un ordinateur massivement parallèle (en l'occurrence une Connection-Machine 2) pendant plusieurs jours.

La simulation des SDD est donc une cible de choix pour les machines parallèles. La discrétisation des variables se traduit par du parallélisme de données : on pourra manipuler en parallèle de très grand ensemble de données homogènes. Les relations liant les différentes variables, ou la simulation en parallèle de plusieurs systèmes, se traduit par des calculs dont le parallélisme porte sur le contrôle. Un modèle de programmation permettant de combiner les deux types de parallélisme serait donc idéal.

La description d'un SDD est statique : elle est donnée a priori et les variables sont en nombre fixe. Les lois d'évolution du système sont données une fois pour toutes. Les calculs à effectuer sont donc connus avant leur exécution : la simulation d'un SDD est une application dont le comportement est prévisible.

La simulation des SDD constitue un des champs d'applications privilégiés qui *vérifient* et *justifient* nos hypothèses de travail : il existe des applications dont le comportement est prévisible, qui requièrent l'utilisation à la fois du parallélisme de données et du parallélisme de contrôle, avec un grain très fin, et qui ont une très grande importance aussi bien dans la pratique que dans la théorie.

III. Les architectures parallèles

Nous allons à présent nous concentrer sur le parallélisme. La demande toujours plus grande en puissance de calcul n'a pu être complètement satisfaite par l'accroissement de la vitesse des circuits électroniques du calculateur séquentiel de Von Neumann. Une solution a alors été d'introduire du parallélisme entre les différentes fonctions réalisées par le calculateur de Von Neumann; deux méthodes ont été employées :

– *modifier* les principales unités fonctionnelles (mémoire, unité de contrôle, unité arithmétique et logique, …) afin que leurs opérations puissent se recouvrir;

– *dupliquer* certaines unités afin qu'elles puissent fonctionner en parallèle.

La première approche a conduit au concept de *pipe-line* et aux calculateurs vectoriels, la seconde au concept de *array-processor* et aux calculateurs parallèles. En réalité la frontière est moins tranchée qu'il n'y parait, puisqu'un calculateur vectoriel possède généralement plusieurs unités arithmétiques et qu'un array-processor possède une seule unité de contrôle.

unité de traitement	unité de contrôle	mémoire partagée (1 mémoire)	mémoire distribuée (n mémoires)
1	1	*modèle de von Neuman*	—
1	m	—	—
n	1	*supercalculateur vectoriel*, *VLIW*	*cellulaire*, *SIMD* : Illiac IV, MPP, Connection-Machine
n	m	*data/demand driven* : MaRS *multiprocesseurs à mémoire partagée* : NYU, RP3, Alliant	*MIMD à passage de messages* : Hypercube Intel, Transputer, Touchstone, PTAH

Table 2 : un classement des architectures des ordinateurs suivant la duplication de la mémoire, du contrôle et du traitement. Le lecteur trouvera une excellente présentation des architectures parallèles qui sont citées dans [Almasi, Gottlieb 89] et dans [Sansonnet 91].

Les calculateurs parallèles peuvent se décrire suivant les fonctions qui sont dupliquées : la duplication de la mémoire conduit à la distinction mémoire-partagée/mémoire-distribuée, la duplication des unités de contrôle et de traitement au critère SIMD/MIMD (acronymes pour Single Instruction Multiple Datastreams et Multiple Instruction Multiple Datastreams). La configuration « *1* unité de calcul/*n* unités de contrôle » ne présente pas d'intérêt évident. Les configurations du type « *m* unités de contrôle/*n* unités de calcul » où $m < n$ correspond au cas des machines partitionables (par exemple [PASM 81]). La table 2 ci-dessus résume la situation.

III.1. Les architectures massivement parallèles et leur programmation

Le terme d'architecture massivement parallèle est utilisé par beaucoup d'auteurs dans des sens assez différents. Nous voulons éviter que les grands réseaux d'ordinateurs faiblement couplés (par exemple ARPA) soient considérées comme des architectures massivement parallèles. De même, nous ne voulons pas faire entrer dans cette catégorie les petits réseaux, même fortement couplés (comme par exemple les réseaux de terrain pour le contrôle-commande).

Nous définissons une architecture massivement parallèle comme une architecture où le facteur de duplication est supérieur au millier et qui exploite un parallélisme à grain fin. Cela impose des contraintes architecturales très fortes. En particulier, la mémoire doit être distribuée (afin d'éviter la contention des accès; voir cependant [LHMRSP 91] pour un autre point de vue). La communication se fera donc par passage de messages [Athas, Seitz 88]. Les contraintes architecturales sur le réseau de communication sont décrites dans [GBES 89] [GBES 90] [CBE 91].

Des contraintes spécifiques pèsent aussi sur l'exploitation du parallélisme : de tels ordinateurs requièrent à l'évidence de nouveaux modes de programmation. Il faut tout d'abord répondre à des questions qui ne se posent pas dans le cas d'une machine séquentielle : exhiber, caractériser et quantifier le parallélisme présent dans un programme, l'exploiter à travers l'ordonnancement des tâches et la gestion de la distribution des données, réaliser les synchronisations nécessaires, etc. Mais la programmation des ordinateurs massivement parallèles se distingue en plus de la programmation parallèle en ce qu'il n'est pas possible de considérer l'activité de chaque processeur individuellement. À la gestion du temps et de l'espace, s'ajoute la difficulté du nombre : on ne doit pas gérer une dizaine de processus représentant des millions d'opérations, mais des millions de petites tâches d'une dizaine d'opérations. Autrement dit, les machines massivement parallèles veulent exploiter un parallélisme dont la granularité est *très* fine [Athas 87].

La *granularité* du parallélisme correspond à la « grandeur » des « morceaux de programmes » qui peuvent ou ne peuvent pas se dérouler en parallèle. Elle correspond à l'exécution d'une quantité très variable de code. Par exemple en *Bourne shell* (le langage de commande sous UNIX), les opérations sont des programmes entiers, ce qui met en jeu des centaines de milliers d'instructions. Dans des langages comme MultiLISP [Halstead 85] ou QLISP [Gabriel, McCarthy 84], l'unité de parallélisme correspond à l'application d'une fonction, ce qui représente de l'ordre du millier d'instructions. Dans les langages acteurs, du type SAL, ACT3 [Agha 85] ou OAL [GGF 91], la tâche correspond au

traitement d'un message, c'est-à-dire plutôt à une centaine ou à une dizaine d'instructions; dans le même ordre de grandeur, l'unité de parallélisme en Concurrent-Prolog est l'unification de la clause [Shapiro, Taekuchi 83]. Pour descendre à une unité de parallélisme de l'ordre d'une instruction, il faut se tourner vers les langages de type data-flow comme VAL [McGraw 82], Id [NPA 86] ou SISAL [MAS 85] ou bien vers les langages de manipulation en parallèle de données comme *LISP [*LISP 86], Actus [Perrot, Zarea-Aliabadi 86] ou PARALATION LISP [Sabot 88].

Cependant une granularité très fine n'est pas la seule caractéristique requise pour un langage de programmation des machines massivement parallèles : une de nos hypothèses de travail est qu'il faut pouvoir combiner l'expression du parallélisme de données et du parallélisme de contrôle. En effet, un des problèmes des architectures SIMD massives (comme la *Connection-Machine*) est celui du rendement : les processeurs ont un taux d'utilisation qui peut être très faible. Nous reviendrons ci-dessous sur ce problème, mais on peut retenir que l'exploitation du contrôle permet de gagner en flexibilité et augmente l'*efficacité* du calculateur (l'efficacité se définit comme le rapport A/N où A est le facteur d'accélération et N le nombre de processeurs).

Un des buts de cette thèse est de proposer pour les machines massivement parallèles, un modèle de programmation à grain très fin qui combine l'expression *statique* du parallélisme de données et du parallélisme de contrôle.

III.2. Exploitation du parallélisme de donnée et de contrôle

Le premier objectif que l'on se doit d'assigner à un langage parallèle est de pouvoir exprimer le parallélisme. Ce n'est qu'ensuite qu'on peut lui demander de refléter les capacités de l'architecture cible. En effet, l'expression du parallélisme est une condition nécessaire (quoique non suffisante) à son exploitation.

Dans ce paragraphe, nous allons rappeler brièvement les propriétés essentielles des deux modèles de programmations parallèles classiques, tels qu'ils ont été induits par la classification des architectures parallèles que M. J. Flynn a proposé en 1969.

III.2.1. Le modèle SIMD

Les modèles de programmation SIMD correspondent à l'exploitation du *parallélisme de données* : la même opération est répétée sur les éléments d'un ensemble homogène de données.

La programmation SIMD doit donc organiser les données en des ensembles homogènes dont les éléments seront traités en parallèle. En contrepartie, le contrôle du programme est simple : chaque processeur exécute la même instruction, ou bien suivant l'état d'un drapeau, ne fait rien. Il existe donc une seule suite d'instructions qui commande le calculateur : l'exécution est séquentielle, même si les instructions opèrent en parallèle sur des ensembles de données.

On fait généralement deux critiques aux modèles de programmation SIMD :

- ils ne sont pas capables d'exprimer tout le parallélisme possible présent dans un algorithme (par exemple le parallélisme de contrôle ou de flux);
- l'utilisation effective des ressources matérielles de calcul peut être très faible.

Ce dernier inconvénient est illustré par l'implémentation des structures de contrôle conditionnel. Chaque processeur doit exécuter la même instruction sur chaque processeur. Ainsi, le *si… alors… sinon…* est implémenté en séquentialisant les deux branches : les processeurs ayant évalués à vrai la condition sont inactifs pendant l'exécution de la branche *sinon* et vice-versa. En cas de conditionelles imbriquées, on voit que cette implémentation peut réduire exponentiellement l'utilisation des processeurs.

III.2.2. Le modèle MIMD

Le modèle de programmation MIMD est capable de répondre à ces deux critiques. Chaque processeur exécute en effet son propre programme : c'est le modèle le plus général et qui offre le plus de liberté. La programmation MIMD doit donc organiser les interactions entre processeurs (plus exactement, entre processus). Cela se fait à l'aide de structures de contrôle comme le passage de messages, les sémaphores, l'appel de fonction distante, etc.

Les modèles MIMD permettent de maximiser l'utilisation des processeurs mais on leur fait principalement les deux critiques suivantes :

– le modèle MIMD ne permet pas l'exploitation efficace du parallélisme de données;

– le développement et la mise au point des programmes MIMD peut se révéler extrêmement difficile.

C'est ce que nous allons détailler.

Le modèle MIMD est très général : il est donc capable d'exploiter le parallélisme de données. Cette exploitation est cependant la plupart du temps inefficace : il faut associer à chaque donnée une tâche, ce qui induit un niveau d'interprétation qui est coûteux. Dans les travaux que nous avons mené dans [GGF 91], nous avons étudié l'adéquation d'un langage Acteur, nommé OAL, dans l'expression du parallélisme de données. Le modèle acteur propose un noyau minimal permettant la gestion du parallélisme de contrôle. La représentation et la gestion des tableaux est pour le moins inefficace : par exemple il faut associer un acteur à chaque élément d'un tableau et recourir à un arbre binaire pour communiquer avec cet acteur. Chaque opération parallèle sur le tableau demande une opération de diffusion, sans parler des synchronisations nécessaires (la communication acteur n'est pas déterministe).

Dans une exécution MIMD, chaque processus déroule indépendamment son propre code. En conséquence, le programmeur doit maîtriser l'indéterminisme qui résulte de l'exécution autonome de chaque programme. Cela requiert la gestion correcte de tous les cas de figure d'interactions entre processus, et ceux-ci sont potentiellement en nombre exponentiel. Cela complique la compréhension des programmes et leur construction correcte. De plus, cette gestion passe par la synchronisation explicite des différents processus. Cette synchronisation représente un coût d'exécution non négligeable.

IV. Le modèle MSIMD

Les considérations précédentes motivent le développement de nouveaux paradigmes de programmation [Steele 90] : « Briefly put, SIMD isn't enough. (…) Briefly put, MIMD is too much. ». La question qui se pose est donc de déterminer comment étendre le modèle SIMD ou comment restreindre le modèle MIMD afin de *combiner les avantages des deux modèles à la fois*.

L'absence de support adéquat pour le parallélisme de données dans un modèle d'exécution MIMD peut se résoudre par l'introduction de primitives spécifiques. On obtient un modèle que nous appellerons **MSIMD**. MSIMD est l'acronyme de *Multiple SIMD*. Cependant, il faut décider d'une gestion simple et sûre des interactions entre les différentes parties SIMD d'un programme MSIMD, afin d'éviter les ennuis causés par l'indéterminisme. La question peut se reformuler de la façon suivante : *peut-on exprimer le parallélisme de contrôle sans payer le prix de l'asynchronisme ?*

La réponse que nous apportons à cette question est celle du modèle *data-flow* : *les différentes parties SIMD d'un programme MSIMD doivent être gérées à la manière data-flow*. Afin de motiver cette réponse, nous allons nous pencher plus précisément sur l'ordonnancement des opérations d'un programme.

IV.1. Le séquencement des programmes

Un programme correspond à un ensemble ordonné d'opérations[1]. Il faut distinguer les opérations produites par l'exécution d'un programme des instructions du programme (par exemple une boucle permet de générer plus d'opérations que la boucle ne contient d'instructions). Le nombre des instructions d'un programme est en général réduit mais le nombre d'opérations peut être très grand. Par exemple l'algorithme de Gauss pour la résolution d'un système d'équations linéaires correspond à une dizaine d'instructions alors que le nombre d'opérations est de l'ordre de n^3 où n est la taille du système.

Les opérations d'un programme sont ordonnées entre elles : on ne peut intervertir deux opérations quelconques sans risquer de changer le résultat du calcul. Si une opération o_1 doit précéder une opération o_2 alors $o_1 < o_2$. Deux opérations incomparables sont des opérations qui peuvent se dérouler en parallèle : les variables partagées entre les deux opérations doivent alors être uniquement utilisées en lecture. Ainsi, si la relation d'ordre est une relation totale, le programme est purement séquentiel; « plus » l'ordre est partiel, « plus » le programme est parallèle.

Les primitives de synchronisation permettent de spécifier complètement ou partiellement la relation d'ordre entre opérations. C'est par exemple le cas de l'opérateur → en CSP (équivalent du point-virgule en PASCAL) qui sert à séquencer deux processus. Cette spécification de ce qui doit être séquentiel peut prendre des formes plus subtiles : par exemple on veut seulement éviter certains séquencements; interviennent alors des structures de contrôle du type sémaphore, région critique, moniteur, etc. Mais le programmeur peut aussi spécifier les opérations qui sont incomparables : ce sont les primitives de création de tâches (le *PAR* en OCCAM, le *pcall* en MultiLISP, la *qlambda* en QLISP, le *fork* UNIX, le *cobegin* en Parallel-Pascal, etc.).

IV.2. Les modèles de programmation à séquencement implicite

Pourtant, il faut remarquer que la spécification des opérations d'un programme induit naturellement un ordre entre celles-ci. Par exemple quand on écrit dans le langage C l'expression

```
a = b + c;
```

on peut de bonne foi supposer que les valeurs de b et de c seront calculées avant celle de a : l'évaluation de l'expression précédente induit un ordre naturel entre les opérations. Cependant dans un langage comme C, la spécification du séquencement des opérations ne tient pas compte de cet ordre naturel : il doit être explicitement spécifié par le programmeur à l'aide de structure de contrôle (par exemple le point-virgule). Il est ainsi parfaitement possible que b ne corresponde pas à une valeur calculée au moment de l'évaluation de a (si le compilateur est suffisamment sophistiqué, le programmeur verra peut être un message du genre : « b utilisé avant d'être initialisé »).

Dans un modèle d'exécution data-flow, le programmeur *ne peut pas* spécifier de relation d'ordre entre les opérations du langage. Le séquencement des opérations est induit par *les dépendances entre données*.

La dépendance la plus classique est sans doute celle attachée à l'application de fonction. Quand on écrit dans un langage data-flow l'instruction :

```
a = b + c
```

il faut comprendre que les opérations attachées à l'évaluation de la valeur de b seront obligatoirement exécutées avant l'opération d'addition car l'évaluation de la fonction + demande à ce que ses arguments soient évalués avant d'être appliquée.

Le parallélisme apparaît du fait que les applications de fonctions peuvent se faire, moyennant quelques hypothèses, dans un ordre quelconque. En effet, l'application d'une fonction peut se voir comme un processus de

[1] Le point de vue adopté ici est habituel dans le domaine de l'extraction automatique du parallélisme, voir par exemple [Feautrier 91b].

réécriture, chaque règle de réécriture s'appliquant en parallèle à un sous-terme du terme à réduire. Un théorème de confluence permet d'appliquer les réécritures dans n'importe quel ordre. Ce modèle d'exécution correspond à la réduction de graphe en parallèle; il est utilisé par exemple pour la machine MaRS [MaRS 86] ou ALICE [DCF 87]. Un exemple similaire est donné par le langage **Γ** [BCLM 88] [Berry, Boudol 89], ou encore par LINDA [Carriero, Gelerntner 89]. Dans **Γ**, les données correspondent à des « molécules » qui se transforment quand elles rencontrent d'autres molécules (les règles de transformation constituent des règles de réécritures). Implicitement, les transformations ont le droit de se faire dans n'importe quel ordre (i.e. toutes les opérations sont incomparables, du point de vue de la relation d'ordre). Dans LINDA, la réécriture se produit sur des tuples.

La propriété importante est que *l'ordre induit par la dépendance des données suffit à déterminer le résultat d'un calcul.* En particulier, il n'y a plus de problème d'indéterminisme. Pour assurer cette propriété, il faut invoquer un théorème de confluence, comme plus haut, ou bien introduire des contraintes au niveau du langage : *l'assignation unique* [Tesler, Enea 68] permet, par exemple, d'assurer cette propriété pour les langages impératifs.

Les langages data-flow sont donc les langages dans lesquels le séquencement des opérations est implicitement assuré par la dépendance entre les données. Un ordonnancement minimal des opérations est automatiquement déduit du programme ce qui permet par suite un *parallélisme maximal.* Cet ordonnancement peut être inféré statiquement (i.e. par un compilateur), ou dynamiquement (par l'évaluateur, comme c'est le cas dans les architectures data-flow). Le séquencement implicite des programmes par les données possède de nombreux avantages :

- Le programmeur n'a pas de synchronisations à gérer explicitement par : celles-ci sont implicites et correspondent aux contraintes justes suffisantes pour assurer l'évaluation correcte.
- Il n'y a pas de redondance entre le séquencement imposé par les dépendances entre données et le séquencement imposé par le programmeur. Cela diminue les erreurs.
- Les langages qui ne permettent pas le séquencement implicite des programmes doivent offrir des primitives permettant de spécifier explicitement soit « ce qui est parallèle », soit « ce qui est séquentiel ». Dans le premier cas le programmeur doit rajouter des synchronisations en considérant les interactions en nombre exponentiel entre processus parallèles, dans le second cas le parallélisme possible doit être extrait automatiquement. L'extraction automatique du parallélisme consiste à ne pas tenir compte du séquencement superflu spécifié par le programmeur; cependant l'analyse des dépendances entre données est rendue très difficile car le compilateur doit tenir compte de cette synchronisation explicite quand c'est nécessaire (dans le cas de la réutilisation d'une variable par exemple) et ne pas en tenir compte quand elle n'intervient pas dans la sémantique du langage (afin d'extraire le parallélisme).
- Dans le cas des langages data-flow, les synchronisations nécessaires sont déduites automatiquement et correspondent à un ordonnancement minimal. Par suite, le parallélisme exploitable est maximal. On voit par exemple difficilement un programmeur OCCAM écrire

```
PAR
   PAR
        d = …;
        e := …;
        f := …;
        g := …;
   b := d + e;
   c := f + g;
a := b + c;
```

plutôt que d'écrire :

```
d = …;
e := …;
f := …;
g := …;
a := d + e + f + g;
```

puis indiquer un placement des tâches a à g, afin d'exprimer et exploiter tous le parallélisme possible. Cet exemple montre bien que l'expression du parallélisme doit être implicite.

Nous reviendrons dans la dernière partie de ce chapitre sur d'autres avantages du modèle data-flow.

IV.3. Conclusion provisoire

Les considérations précédentes plaident en faveur d'un modèle d'exécution « *MSIMD data-flow* » (ou encore *MFMD* : Multiple Flow Multiple Datastream). En effet, un tel modèle est capable de capturer le parallélisme de données (à travers l'aspect SIMD) et apporte les avantages du MIMD (le parallélisme de contrôle) sans en avoir les inconvénients.

Cependant, on peut se poser la question de savoir s'il existe des architectures qui sont capable de tirer partie de l'expression à la fois du parallélisme de données et du parallélisme de contrôle. Aujourd'hui, de nouveaux ordinateurs parallèles apparaissent qui offrent les supports matériels nécessaires à l'implémentation des modèles MSIMD. Ces architectures correspondent aux calculateurs *MSIMD* (comme par exemple la **Connection-Machine 5** ou **SPHYNX** [Clermont 90]) aux *ordinateurs parallèles reconfigurables* (comme la machine **PASM**, Cf. [PASM 81]) et aux machines *MIMD synchronisables* (comme par exemple la machine **PTAH** [Cappello, Béchenec 91], [CNB 91]).

Par ailleurs, les modèles d'exécution SIMD et MIMD constituent les cas extrêmes du modèle MSIMD (ce qui peut se figurer par les équations : MIMD = Multiple SISD et SIMD = Single SIMD). Un modèle de programmation MSIMD peut donc s'adapter aussi bien à une architecture MIMD ou SIMD pure, ce qui se rapproche d'une notion de *portabilité* (voir aussi à ce sujet [Skillicorn 90] pour une approche de la portabilité basée sur les structures de données qui supportent le parallélisme).

La discussion précédente a été motivée par les différentes sources de parallélisme exploitées par les calculateurs parallèles. Nous devons à présent introduire les propriétés des applications visées : à savoir le caractère prévisible des calculs.

V. Les ressources d'un modèle de calcul MSIMD à comportement prévisible

Les considérations précédentes nous ont conduit à choisir un modèle « MSIMD data-flow » pour la programmation parallèle à grain très fin des calculateurs massivement parallèles. Dans cette dernière partie, nous allons étudier plus précisément les implications du modèle « MSIMD data-flow *à parallélisme statique* ». Pour cela nous allons entrer plus en détails dans le modèle data-flow. Notre modèle data-flow est dans la lignée du langage *LUCID* et utilise les **streams** afin de représenter de manière prévisible (ou encore *statique*) le parallélisme de contrôle. Le parallélisme de données s'introduit dans ce cadre data-flow à travers la notion de **collection**.

V.1. Le modèle data-flow

Les principes du data-flow ont été proposés dans le début des années 1970 comme une alternative au modèle d'exécution *control-driven* [Dennis 74] (voir [Thakkar 87] pour un tutoriel et [Dennis 91] pour une perspective historique du data-flow statique). Un principe général est que toute variable d'un programme data-flow ne reçoit qu'une seule valeur au cours du temps. Cela permet de compiler les programmes en un graphe, le graphe data-flow, où les nœuds correspondent à des instructions et les arcs à des dépendances de données. L'évaluation d'un graphe data-flow est cadencée par la disponibilité des données plutôt que par un ou plusieurs compteurs de programmes : une instruction peut s'exécuter dès que toutes ses données sont présentes.

V.1.1. La contrainte du parallélisme statique

L'exécution parallèle d'un graphe data-flow s'imagine facilement sur une machine abstraite qui aurait autant de

processeurs que de sommets dans le graphe. Chaque processeur attend ses données et émet le résultat vers les PE auquel il est relié.

La notion de fonction se traduit de la manière suivante : chaque invocation de fonction se traduit par la création d'un nouveau graphe, correspondant au corps de la fonction, et qui sera détruit après obtention du résultat. La notion d'itération existe aussi et correspond à un graphe cyclique : une donnée produite en sortie du graphe est réinjectée à son entrée. Tant qu'on ne considère pas les fonctions récursives, il est facile d'implémenter cette machine abstraite sur un réseau de processeurs réels : le nombre de sommets étant fixé, il suffit de les répartir sur les PE. Cependant, si on admet les appels récursifs, alors il n'est plus possible de connaître a priori le nombre de sommets qui vont être activés : le parallélisme est dynamique.

Puisque nous nous sommes fixés la contrainte d'un modèle dont les calculs doivent avoir un comportement prévisible, il nous faut interdire les fonctions récursives. C'est le cas dans un langage comme FORTRAN 77 par exemple. Cependant, si on ne dispose pas des fonctions récursives, il faut disposer d'un concept d'itérations sous peine de réduire l'expressivité du langage à une pauvreté rédhibitoire. Or ce concept pose un problème dans le modèle data-flow.

V.2. Itération et data-flow : la notion de stream

Si l'itération est une structure de contrôle naturelle pour le programmeur, elle pose beaucoup de problème pour sa modélisation mathématique. Elle empêche en effet que les affectations de variables soit prises pour des assertions sur la valeur de cette variable. Les langages à assignation unique ont cette propriété mais pour cela il faut supposer que les variables référencées dans un pas d'itération sont différentes des variables référencées le pas d'itération suivant. Rappelons que cette propriété permet de cadencer le programme uniquement à l'aide des dépendances entre données.

Nous avons indiqué que les boucles de programmes correspondaient à des boucles dans le graphe data-flow. En fait, une autre manière de représenter les itérations dans les langages de programmation a existé très tôt : le *stream*. Afin de respecter l'assignation unique, on considère que chaque variable dénote une suite de valeur. Ce point de vue rend compte très simplement du fait suivant : un sommet du graphe data-flow (pris dans une boucle) voit en fait passer une suite de valeurs dans le temps.

Plutôt que de restreindre les seules variables d'une boucle à dénoter une séquence, on peut généraliser le procédé. C'est ce qui est fait dans un langage comme LUCID [Ashcroft, Wadge 76] [Wadge, Ashcroft 85] ou la structure de donnée de base est la suite de valeurs. Les opérateurs du langage agissent tous sur des suites. Par exemple, si A et B correspondent aux suites :

$A = 1,\ 2,\ 3,\ 4,\ 5, \ldots$
$B = 2,\ 7,\ 4,\ 8,\ 33, \ldots$

alors, l'expression $A + B$ a pour valeur

$A + B \Rightarrow 3,\ 9,\ 7,\ 12,\ 38, \ldots$

V.2.1. L'algèbre des streams

Nous préférons utiliser le terme de « stream » à celui de « suite de valeurs » car un stream représente une suite potentiellement infinie de valeurs. Par ailleurs, la notion de stream que nous considérons, modèle ses propriétés sur celles de la *trajectoire* d'un paramètre d'un système dynamique. Par suite c'est une notion essentiellement *temporelle* : il y a une flèche du temps et la valeur d'un élément dans un stream ne dépend pas de la valeur d'éléments survenus ultérieurement.

Ainsi, contrairement à LUCID, nous contraignons les opérateurs à vérifier une propriété *markovienne*, c'est-à-dire

que la valeur courante d'un stream ne peut dépendre que des *n* précédentes valeurs. Le nombre *n* doit être une constante calculable à la compilation. Cette contrainte existe aussi dans des langages pour la programmation temps-réel comme LUSTRE [CPHP 87] et SIGNAL [LGBBG 86] qui sont dérivés de LUCID. Cette hypothèse est compatible avec les applications que nous envisageons. Du point de vue de l'implémentation, elle permet de forcer un ordre d'évaluation séquentielle des valeurs d'un stream et rend possible la réservation statique de l'espace mémoire nécessaire aux calculs.

Seul un nombre limité d'opérations est donc disponible sur les streams :

– la spécification de la valeur d'un stream à un instant donné;

– l'observation d'un stream à un instant donné;

– la référence à la valeur du stream à l'instant précédent l'instant courant;

– l'algèbre sur les valeurs des types de base étendue naturellement pour opérer sur les streams (comme dans l'exemple de l'addition donné plus haut);

– le sous-échantillonnage.

Ces points seront décrits dans le chapitre suivant avec la présentation du langage 81/2. En particulier, nous verrons que les streams « n'avancent pas tous à la même vitesse ».

V.2.2. Pourquoi la notion de stream est fondamentale

La notion de stream est essentielle pour plusieurs raisons :

– Le stream permet une représentation data-flow de l'itération et par suite rend possible un parallélisme MIMD statique avec ses avantages (comme par exemple l'estimation à la compilation du coût des communications et des charges des PE, Cf. chap. X).

– Une suite dans le temps de valeurs correspond à la notion de *trajectoire* dont nous avons besoin pour modéliser les systèmes discrets dynamiques.

– Le stream est une suite de valeurs. Les suites de valeurs constituent une structure de données importante et utile qu'on retrouve dans beaucoup de langages de programmation, par exemple sous le nom de *séquence* en Common-Lisp et en FP [Backus 78], sous le nom de *vecteur* en APL, de *stream* en LUCID, LUSTRE [CPHP 87], SIGNAL [Le Guernic, Benveniste 87] ou d'*ensemble* en SETL. Mais on retrouve aussi cette structure de donnée dans la programmation des PLD (*programmable logic device* [Bolton 90]) et la conception des circuits VLSI (voir par exemple [Johnson 83], [Delgado 86]).

– Le stream est un paradigme de communication : qu'on pense au pipe UNIX (et sa généralisation au *socket* et au *stream* unix) qui était la seule opération de communication entre processus qui a existé pendant longtemps. Le mode de communication impliqué par un stream est *statique* : on connaît l'émetteur et le receveur.

– Dans un langage à assignation unique le stream est, pour citer [Wadge, Ashcroft 85], une structure « mathématiquement respectable » et même [Waters 91] : « (…) series expressions are to loops as structured control constructs are to gotos ». Cela permet :

  - *la transparence référentielle* : cela veut dire que toute référence a une variable peut être remplacée par sa définition. Un ensemble de définitions peut être vu comme un ensemble d'équations.

  - Il est donc plus facile de *vérifier et raisonner formellement sur les programmes*.

  - Ceci permet *la transformation automatique des programmes,* en vue de leur optimisation (Cf. par exemple [Leiserson, Saxe 83] et [Waters 91]).

  - Enfin, pouvoir considérer un programme comme un système d'équations a pour avantage d'assurer que le programme calculera un résultat, si ce résultat est formellement dérivable du système

d'équations. C'est la propriété de *declarative completness* [HOS 85].

V.3. Répétition et parallélisme de données : la notion de collection

La notion de stream ne rend pas compte de la notion de séquence finie. Une séquence finie correspond à une répétition (boucle *pour*) dont on connaît les bornes. Ces répétitions interviennent quand on veut manipuler des tableaux ou plus généralement des ensembles homogènes de données. Si on pense aux systèmes dynamiques discrets, ces répétitions interviennent pour la modélisation d'une population, la représentation des paramètres multi-dimensionnels ou encore la discrétisation de paramètres continus (par exemple la discrétisation d'une image, ou le maillage d'un objet dans le cas des éléments finis). Il est donc vital de traiter correctement ce concept, tant du point de vue de l'expressivité que du point de vue de l'efficacité, car il est le support du parallélisme de données.

Le traitement des tableaux pose un problème dans le modèle data-flow : des problèmes d'efficacité [Gajski, Padua, Kuck 82] (l'assignation unique oblige conceptuellement à copier tout le tableau chaque fois qu'on modifie la valeur d'un de ses éléments) et des problèmes de traitement mathématique (en particulier comment interagit la notion de stream et de tableau[1]). C'est pourquoi nous introduisons un nouveau concept, celui de *collection*, chargé de représenter spécifiquement les ensembles homogènes et finis de données.

V.3.1. La notion de collection

Le terme de *collection* est introduit dans [Sabot 88] [Blelloc, Sabot 90] et [Sipelstein, Blelloch 91] pour désigner une structure de données permettant de manipuler *comme un tout* un ensemble d'éléments. Le terme de *séquence* est parfois aussi utilisé[2]. Le vocabulaire ne doit pas cacher que c'est une structure de données utilisée depuis longtemps dans beaucoup de langages. Cependant l'intérêt provient ici de ce que la collection est vue comme le support privilégié du parallélisme exploité par les architectures SIMD [Hillis, Steele 86] [Blelloch 89] : le concept de collection permet d'appliquer la même opération en parallèle sur tous les éléments d'un ensemble.

Plusieurs critères permettent de dresser une taxinomie des collections :

- Le premier est la manière dont les éléments sont agrégés. Par exemple l'ensemble PASCAL est une collection qui se distingue d'un tableau (en FORTRAN 90 [FORTRAN 90] ou en APL [Legrand 84] on peut manipuler des tableaux comme un tout, et par exemple écrire « A + B » pour désigner l'addition des deux tableaux A et B). Les ensembles et les tableaux n'ont pas les mêmes propriétés (par exemple les éléments d'un tableau sont naturellement ordonnés par leur index alors que cela n'est pas vrai dans un ensemble).
- Le deuxième critère de classification est le type des éléments d'une collection, et en particulier, si les éléments d'une collection peuvent être d'autres collections.
- Enfin les collections se distinguent par les opérations qu'elles autorisent. Plus particulièrement, on distinguera trois classes d'opérations qui correspondent à l'*alpha-notation*, la *réduction* et les opérations de permutations.

Nous allons détailler chacun de ces trois points.

V.3.2. Les différents types de collections et d'éléments manipulés

En APL, *Lisp, C* ou Paralation Lisp [Sabot 88], la collection est un vecteur, c'est-à-dire une fonction de {1, …, n} dans un ensemble de valeurs. C'est une structure de données qui est différente d'un multi-ensemble car les

[1] [Wadge, Ashcroft 85] : « We spent a great deal of effort trying to find a simple algebra of arrays (…) with little success »

[2] voir aussi [Waters 91] pour une grande bibliographie sur le sujet et [More 79] pour une axiomatisation de la notion de « tableau rectangulaire multi-dimensionnel » afin d'en faire une notion primitive.

éléments d'un multi-ensemble ne sont pas ordonnés. Nous préférons le terme de vecteur à celui de liste, car ce dernier implique souvent un accès séquentiel aux éléments et une dynamicité supplémentaire (la capacité de rajouter des éléments dans la liste). Les valeurs peuvent être des entiers, des flottants, des caractères : ces types de données, qui ne sont pas des collections, seront regroupés sous le terme de scalaires.

Les éléments d'une collection peuvent être d'autres collections. C'est le cas d'APL2 ou de Paralation-Lisp, mais pas celui de *LISP qui ne permet pas la manipulation aisée de vecteur de vecteurs (quoique qu'on puisse déclarer une pvar de pvar en *LISP) : par exemple, il n'est pas possible de faire l'addition de deux vecteurs de vecteurs, il faut passer par une boucle d'itération explicite.

L'imbrication des collections permet l'expression de multiples niveaux de parallélisme. L'exemple classique est le suivant [Sabot 88] : on désire tracer en parallèle un ensemble de droite. Le tracé d'une droite peut se réaliser par le tracé en parallèle des points de la droite. La représentation naturelle est donc une collection de collections de points. L'imbrication des collections permet de représenter un ensemble de droites; sans imbrication de collections, il n'est pas possible de représenter le parallélisme provenant du tracé en parallèle de plusieurs droites, chacune de ces droites étant tracée elle aussi en parallèle.

Quand un langage permet de manipuler des collections imbriquées, il offre aussi des opérateurs permettant de restructurer une collection. C'est l'opération *unnest* dans Algrès (une extension de SQL), et dans APL on peut citer la *caténation*, le *laminage*, la *rotation*, la *transposition simple* et *dyadique*… (entre autres) .

Les langages précédents ne sont pas les seuls à permettre de manipuler des collections. SETL et SQL permettent de manipuler des ensembles plutôt que des vecteurs. SETL permet aussi de manipuler des fonctions énumérées : ce sont des fonctions où l'on a explicitement associé une image à un point (une fonction énumérée peut donc se représenter par un ensemble de couples). Les éléments manipulés en SQL sont des tuples. Les langages relationnels classiques, comme SQL, ne permettent pas de hiérarchie, mais de nombreuses extensions du modèle entité-relation le permettent. Les éléments manipulés en SETL sont des scalaires, des paires, des vecteurs, des ensembles. FORTRAN 90 et Artus sont deux exemples de langages qui manipulent des tableaux. Enfin, on peut aussi citer les automates cellulaires : un automate cellulaire représente une collection d'état. Généralement l'état est une valeur prise dans un ensemble fini.

V.3.3. Accès à un élément

Un critère important permettant de caractériser le type d'une collection est la manière dont on peut accéder à un élément à travers la collection :

– On ne peut pas y accéder :

  il est impossible d'accéder à un élément donné à travers la collection. C'est le cas d'un ensemble. Cela ne veut pas dire qu'on ne peut plus accéder à la valeur de cet élément (Cf. paragraphe suivant) mais on ne peut y accéder individuellement.

– On peut y accéder à travers un (ou plusieurs) index :

  cela correspond au cas des vecteurs, des listes, et plus généralement des tableaux. Cet index peut être un nombre, ou bien une *clef* (comme pour les tuples SQL). Dans ce cas là on est proche de la notion de tableau associatif (fonction énumérée en SETL ou *xapping* en CM-LISP [Hillis 85]).

Dans ce mémoire, on appellera *géométrie* l'ensemble d'index qui est associé à une collection.

V.3.4. Alpha-notation

L'alpha-notation est la capacité d'appliquer une fonction à chaque élément d'une collection. Par exemple, si *A*

est une collection de 3 booléens

$A = \langle$ *true*, *false*, *true* $\rangle$

on peut appliquer la fonction *not* à chaque élément :

not $A \Rightarrow \langle$ *false*, *true*, *false*$\rangle$

Le mécanisme se généralise pour les fonctions à plusieurs arguments :

$B = \langle 1, 2, 3 \rangle$
$C = \langle 4, 5, 6 \rangle$
$B + C \Rightarrow \langle 5, 7, 9 \rangle$

Implicitement dans cet exemple, nous avons supposé que l'on pouvait apparier les arguments de l'addition. Cela ne peut se faire que s'il est possible d'accéder à un élément à travers un index, et que l'ensemble d'index est le même pour les deux collections : les collections doivent avoir la même géométrie. On ne pourrait pas faire de même dans le cas où la notion de collection se réduit à celle d'ensemble.

L'alpha-notation permet de transformer une fonction qui s'applique à des scalaires en une fonction qui s'applique à des collections ordonnées. Le résultat est la collection constituée de l'application élément par élément de la fonction scalaire. Cette application peut être implicite ou explicite : opérateur *mapc* en LISP, *for each* en SETL, α en CM-Lisp.

La notation explicite est requise si on veut éviter les ambiguïtés possibles dans le cas où les collections sont imbriquées. Un exemple est celui de la fonction *reverse* qui peut s'appliquer à une liste dans son entier ou bien à chaque élément de cette liste :

$A = \langle \langle 1, 2, 3\rangle, \langle 4, 5, 6\rangle, \langle 7, 8, 9\rangle \rangle$

reverse $(A) = \langle \langle 7, 8, 9\rangle, \langle 4, 5, 6\rangle, \langle 1, 2, 3\rangle \rangle$ $\qquad \alpha$ *reverse* $(A) = \langle \langle 3, 2, 1\rangle, \langle 6, 5, 4\rangle, \langle 9, 8, 7\rangle \rangle$

V.3.5. La réduction

La réduction permet d'itérer l'application d'une fonction sur les éléments d'une collection. Par exemple en APL (ce qui porte le nom de *beta-réduction*) :

$+ / 1\ 2\ 3 \Rightarrow (1 + (2 + 3)) = 6$ (syntaxe APL)

L'ordre de la réduction est fixé en APL, ce n'est pas le cas en FORTRAN 90 ou en *LISP. Dans ce cas, l'utilisation d'une fonction qui n'est pas associative produit un résultat indéfini. La fonction qui sert à la réduction peut appartenir à un ensemble d'opérateur réduit (*and, or, +, *, min* et *max* en FORTRAN 90) ou bien être n'importe quelle fonction définie par le programmeur (en CM-LISP).

Une autre forme de réduction est le *scan* : le résultat est la collection des résultats partiels des réductions successives :

$+ // 1\ 2\ 3 \Rightarrow 1\ 3\ 6$ (syntaxe APL)

L'intérêt des réductions est de s'implémenter en parallèle en utilisant n PE et $\log_2(n)$ étapes, avec n le nombre d'éléments de la collection.

Dans le cas d'une collection imbriquée, il faut indiquer à quel niveau on applique la réduction. C'est l'opérateur d'axe en APL. Mais cela peut se faire grâce à l'alpha-notation si celle-ci est explicite :

$\beta+ \langle \langle 1, 2, 3\rangle, \langle 4, 5, 6\rangle, \langle 7, 8, 9\rangle \rangle \Rightarrow \langle 12, 15, 18 \rangle$ ou bien $\langle \langle 12, 15, 18 \rangle \rangle$
$\alpha\ \beta+ \langle \langle 1, 2, 3\rangle, \langle 4, 5, 6\rangle, \langle 7, 8, 9\rangle \rangle \Rightarrow \langle 6, 15, 24 \rangle$ ou bien $\langle \langle 6\rangle, \langle 15\rangle, \langle 24\rangle \rangle$

(les alternatives dépendent de la manière de considérer le résultat de la réduction : est-ce une collection d'un seul élément ou bien un scalaire ?)

V.3.6. Opérations de sélection

Il est souvent intéressant de réaliser une réduction qui ne tienne compte que de certains éléments. Les éléments sont sélectionnés par un masque en FORTRAN 90 :

SUM (A, MASK = A.GT.0.0)

réalisera la somme des éléments positifs du tableau A.

En *LISP (et en POMP-C [Bougé 90]), le mécanisme de sélection est un mécanisme plus général. La sélection des éléments *actifs* est opérée par une structure de contrôle : le **when* ou le *where* (en *LISP et en POMP-C respectivement). Cette structure de contrôle est de la forme :

where cond *do* body

Toute opération (les réductions mais aussi les alphas) qui se trouve dans le corps du where n'est exécutée que pour les éléments dont l'index a été sélectionné par l'expression « cond ». Cette expression est une collection dont chaque élément est un booléen. En effet, dans ces langages, chaque collection a le même ensemble d'index[1]. Par exemple

$A \Rightarrow \langle 1, 2, 3 \rangle$
$B \Rightarrow \langle 4, 5, 6 \rangle$
$C \Rightarrow \langle$ *false*, *true*, *false* $\rangle$
$D \Rightarrow \langle 0, 0, 0 \rangle$

après évaluation de

where C do $D = B + C$

on peut vérifier que

$D \Rightarrow \langle 0, 7, 0 \rangle$

(la syntaxe simplifiée ne correspond ni a *LISP ni à POMP-C).

La création d'une nouvelle collection représentant une sous-collection d'une collection donnée se fait en APL en utilisant l'opérateur de compression et un masque booléen. En Paralation-Lisp, on travaille sur une sous-collection d'une collection donnée, en l'isolant dans un segment (on peut diviser un vecteur en segments contigus grâce à l'opérateur *collect*, les segments étant déterminées par un vecteur d'indices : les éléments de même indice se retrouvent dans le même segment). En SQL la sélection est une opération fondamentale, c'est le *select* des algèbres relationnelles.

V.3.7. Opérations de réindexation

Quand les collections sont indexées, il est possible de permuter l'index des éléments. Si, dans l'optique d'une exécution parallèle, on considère qu'un processeur est attaché à un élément d'une collection, alors la renumérotation est une opération qui implique une communication.

Cette communication peut être statique : dans le cas d'un automate cellulaire, le schéma de communication est fixe et indépendant d'un programme donné (communication avec les voisins, la géométrie de la collection définissant ce qu'est la relation de voisinage). La communication est dynamique dans le cas où on peut réaliser une permutation obtenue comme le résultat d'un calcul.

La réindexation n'est d'ailleurs pas restreinte aux permutations. Elle correspond à l'opération de *gather* sur un tableau :

$A = B(C) \quad \Rightarrow \quad A(i) = B(C(i))$ pour tous les i appartenant aux index de C

[1] Plus exactement la notion de géométrie est une notion qui n'est pas attachée à la collection mais à la configuration courante du calculateur (le *shape* en *LISP). Cette configuration peut être changée en cours de calcul. Une configuration décrit comment les éléments d'une collection seront affectés aux PE.

V.3.8. Structuration

Les opérations de réindexation ne sont qu'une partie des opérations de structuration sur une collection. On peut imaginer diverses opérations de *projection*, de *troncature*, de *duplication*, de *concaténation*, d'*aplatissement* et de *redressement* (dans le cas de collections imbriquées). APL [Legrand 84] et Artus en offrent quelques exemples.

V.3.9. Récapitulatif

Le tableau qui suit résume les notions de collection qui existent dans différents langages :

. La colonne *alpha* indique si l'itération est implicite ou explicite.

. La colonne *beta* indique si la réduction peut se faire à partir d'une fonction utilisateur (user) ou non (statique).

. La colonne *communication* rend compte des permutations possibles : *statique* correspond à des patterns de communication fixe, *éther* à des gathers quelconques.

. La colonne *sélection* indique s'il est possible de sélectionner et d'agir sur une partie seulement des éléments d'une collection.

	type de la collection	**type des éléments**	**alpha**	**beta**	**commu-nication**	**sélection**	**style de langage**
LISP	liste	scalaire ou liste	map	statique [10]	statique [5]	rien	applicatif
***LISP**	pvar [1]	scalaire	alpha	user	éther	*when	applicatif
CM-LISP	xector [1], xapping [2]	scalaire, xector, xapping	α, • [12]	user	éther	?	applicatif
Paralation Lisp	field [1]	scalaire ou vecteur	elwise	user	éther	masque, pack [13]	applicatif
FORTRAN 90	tableau [3]	scalaire	doall	statique	éther	masquage [8] pack [13]	impératif
APL	vecteur (tableau)	scalaire ou vecteur	alpha	statique [11]	éther	masquage	applicatif
SETL	ensemble, fct. énumérée	scalaire, tuple, ensemble, map	forall, alpha	user	— [6]	compré-hension [9]	ensembliste
SQL	relation	tuple [4]	alpha	statique	— [7]	select.	relationnel
automate cellulaire	tableau	état (∈ ensemble fini)	alpha	—	statique	—	automate a états finis
81/2	tissu	stream, tissu	alpha, ^	user	statique ou ether	if, gather	déclaratif

Table 3 : Une comparaison des différents types de collection qui existent dans les langages de programmation

Notes :

1 un *xector* ou un *vecteur* ou une *pvar* ou un *field* est une collection dont les éléments sont indexés par un intervalle [1, n].

2 un *xapping* est une collection dont les éléments sont indexés par une clef dont le type est quelconque. C'est une structure très semblable à une liste d'associations.

3 un tableau est une collection dont les éléments sont indexés par un uplet $(i_1, i_2, \ldots)$ où $i_1 \in [1, n_1]$, etc.

4 En première approximation, un tuple correspond à une structure (mais l'égalité des tuples se fait par valeur alors que l'égalité sur une structure correspond à une égalité par nom)

5 On suppose qu'il n'y a qu'un ensemble restreint d'opérations primitives de sélection sur les listes, comme par exemple *car*, *cdr* ou bien encore *reverse*.

6 Un ensemble n'a pas d'index. Par contre il y a une notion de gather avec les fonctions énumérées (ce qui correspond à la composition des fonctions).

7 On ne peut changer la clef d'un tuple (c'est elle qui définit l'existence du tuple). La permutation n'a donc pas de sens.

8 On peut spécifier un masque pour les opérations de réduction.

9 On peut en SETL définir un ensemble en compréhension : { x ∈ A | P(x) } est l'ensemble des éléments de A qui vérifient P. Cela correspond à une sélection.

10 La famille des map en LISP permet de concaténer les résultats successifs ou bien de les lister. Il y a donc une forme primitive de réduction dont les opérateurs sont *cons* ou *list*.

11 Il se peut qu'en APL2 on puisse réduire avec une fonction utilisateur.

12 Les opérateurs α et • ont des effets opposés et se comportent exactement comme les opérateurs *eval* et *quote* : • inhibe l'effet de α.

13 L'opération de packing est semblable à la compression en SETL : on peut créer une nouvelle collection à partir d'une collection et d'un masque.

VI. Un modèle pour la programmation massivement parallèle des applications statiques

Rappelons l'articulation de la discussion menée dans ce chapitre :

- Les applications statiques, i.e. dont le parallélisme des calculs est exprimé de manière statique, existent et sont fondamentales. Elles comprennent en particulier les applications de simulation des systèmes dynamiques discrets (nous nous intéresserons plus particulièrement à la classe des systèmes déterministes non-temporisés décrits par des relations fonctionnelles). Ces applications ont d'énormes besoins en puissance de calcul.
- Nous voulons établir un modèle de programmation à grain très fin des machines massivement parallèle pour ces applications.
- L'examen des modèles MIMD et SIMD nous a conduit à fusionner les deux modèles pour obtenir un modèle MSIMD data-flow statique.
- L'examen des modèles data-flow nous a montré que le parallélisme de contrôle statique avait pour support naturel la notion de stream, à la condition de restreindre les opérateurs sur les streams à vérifier une hypothèse markovienne. Cette restriction est compatible avec les applications de simulations.
- Le concept de collection permet l'expression du parallélisme de données.
- La notion de stream correspond à une séquence temporelle et constitue le support du concept de trajectoire.
- Le concept de collection correspond aux séquences spatiales qui permettent la représentation des variables multi-dimensionnelles et la discrétisation des paramètres continus des « systèmes dynamiques continus discrétisés ».

Le modèle de programmation que nous proposons consiste en la combinaison des notions de stream et de collection dans un environnement data-flow. Ce modèle est construit autour d'une structure de données : les *streams de collections* (ou les *collections de streams*, le point de vue étant dual). On appellera une telle structure de données un **tissu**.Un tissu est capable de capturer le parallélisme de données et le parallélisme de contrôle.

Le concept de tissu a pour ambition de capturer l'aspect spatial et temporel des entités qui interviennent dans la simulation parallèle. Un tissu est constitué de points (les éléments de la collection). Chaque point a une trajectoire, i.e. la valeur de chaque point d'un tissu correspond à une suite de valeurs dans le temps. Afin de pouvoir étudier plus précisément notre modèle de programmation, les tissus constitueront les ressources d'un langage expérimental appelé **Otto e Mezzo**, ou en abrégé, **81/2**.

Il n'y a pas de construction explicitement parallèle en 81/2 : le séquencement des opérations est implicite (en accord avec l'analyse que nous avons développé précédemment sur les langages à séquencement implicite). L'extraction automatique du parallélisme présent dans un programme 81/2 et son exploitation sur une machine est simple car :

– La forme data-flow du langage simplifie l'analyse des dépendances entre expressions et facilite leur transformation en tâches. Cela rend donc possible l'exploitation du *parallélisme de contrôle* offert par des ordinateurs comme le BBN, le Cosmic-Cube, les réseaux de transputer, l'iPSC/2, etc.

– Les tissus supportent naturellement l'expression du *parallélisme de données* et les opérations entre collections donnent lieu tout naturellement à l'exploitation de celui-ci sur des machines comme l'ILLIAC IV, le MPP, la Connection-Machine ou le GF11.

– La notion de suite permet au compilateur d'utiliser l'effet *pipe-line* (logiciel ou matériel) : les valeurs d'un tissu à un tic `t` peuvent se calculer pendant le calcul des valeurs d'un tissu à un tic `t+1`. Il est donc possible d'utiliser le parallélisme de flux qui est exploité sur des machines comme le Cray1, le Cyber 205 ou l'IBM 3090.

– Le concept de tissu corrige certains des défauts des modèles data-flow initiaux [Gajski, Padua, Kuck 82], principalement en introduisant un traitement spécifique des tableaux avec un concept adéquat du temps. Par ailleurs, le tissu est une structure de données qui est naturellement *répartie*, préoccupation habituellement absente du data-flow.

– La transparence référentielle rend possible un traitement formel des programmes ce qui permet d'envisager l'optimisation des programmes par transformation.

– L'extension du modèle data-flow par le concept de collection permet de combiner les avantages des modes de programmation parallèle SIMD et MIMD.

81/2 doit être perçu comme un outil permettant d'expérimenter les modèles de programmation statique permettant l'exploitation des nouvelles architectures hybrides[1].

81/2 ne supporte pas tous les styles de programmation parallèle. Par exemple, il sera difficile de représenter en 81/2 des calculs au comportement dynamique, tel que l'exploration parallèle d'un arbre d'états (où l'arbre est créé au fur et à mesure de l'exploration). Cependant nous pensons que c'est un bon compromis pour une très large classe d'applications, qui préserve « la simplicité et l'efficacité du SIMD » tout en gagnant « le rendement et la flexibilité des structures de contrôle offerts par le MIMD » ([Steele 90]).

[1] Il semble bien que ce soit ce type d'architectures qui servira à remporter la course au tera-ops. Les premières informations publiques concernant la **Connection-Machine 5**, de Thinking Machine Corporation, ont été annoncées en novembre 1991 et font états de performances attendues de l'ordre de *2 tera-flops* pour une machine comprenant 16000 PE. Une machine à 1000 PE a déjà été construite. « The CM5 has some characteristics of both a SIMD and a MIMD machine. It is primarily designed to run single-threated, data-parallel software, but takes advantage of multiple threads of execution to do this more efficiently. Like a MIMD machine, each processor is capable of fetching its own instructions locally, but like a SIMD machine there is a special network of processors closely synchronized. This "control network", in conjunction with the compiler and operating system, provide the user with the illusion of a single threaded machine. » (message de D. Hillis retransmit dans comp.parallel sur le réseau electronique "news").

II. *Otto e Mezzo*

Le chapitre précédent a mis en place les grandes lignes du modèle. Afin de pouvoir étudier plus précisément notre modèle de programmation, les ressources spécifiques de la programmation massivement parallèles, le stream et la collection, seront combinées dans le langage 81/2 pour former un nouvel objet : le **tissu**. Les principales constructions du langage sont exposées et la présentation se poursuit par quelques exemples. Ce chapitre se termine par une revue des langages qui présentent des concepts comparables dans différents domaines d'application.

I. Une présentation de 8$_{1/2}$

I.1. Le concept de tissu

81/2 est un langage déclaratif permettant de définir des *suites de vecteurs* (ou des *vecteurs de suites*, les deux points de vue sont possibles). On appellera une telle structure de données un **tissu**.

Un tissu est constitué de *points* qui sont les éléments du vecteur. Chaque point est repéré par un indice entier et positif. Chaque point possède une valeur. Cette valeur est soit un tissu, soit une suite de valeurs *scalaires*. Une valeur scalaire est une valeur d'un des types de base : *entier*, *booléen*, *flottant*… La valeur d'un point peut donc se décrire par un arbre dont les nœuds sont des points. Les feuilles de cet arbre sont les points dont la valeur est un scalaire. L'arbre des points d'un tissu se nomme sa *géométrie*. La géométrie décrit la « structure spatiale » d'un tissu et correspond à l'imbrication des vecteurs. Les vecteurs vont permettre de manipuler un ensemble fini d'éléments *comme un tout*. Ils correspondent aussi à une structure de données réparties permettant d'effectuer des traitements en parallèle [Hillis, Steele 86]. C'est pourquoi on préfère les appeler des **collections** [Sipelstein, Blelloch 91].

Les suites utilisées possèdent un nombre a priori infini d'éléments. De plus elles ont une interprétation temporelle : les éléments de la suite sont disponibles *séquentiellement*. Chaque élément de la suite correspond à un instant dans le temps. Quand la suite « progresse », le temps progresse aussi. Aussi on leurs donne un nom particulier, celui de **stream**.

Un tissu est donc un **stream de collections**, ou bien encore, une **collection de streams** (Cf. figure 1). Les

aspects spatiaux et temporels d'un tissu sont orthogonaux : on peut généralement parler de l'un en oubliant l'autre point de vue. Un tissu correspond donc à un objet à deux dimensions : quand on donne un point et un instant, on obtient une valeur. On peut représenter cet objet bi-dimensionnel comme un volume dans un repère temps/géométrie/valeur (Cf. figure 1). La valeur d'un point *p* d'un tissu à un instant *t* a pour coordonnées dans ce repère, un couple d'entiers positifs (*p*, *t*). Un tissu n'est pas une matrice car un *t* n'est pas borné.

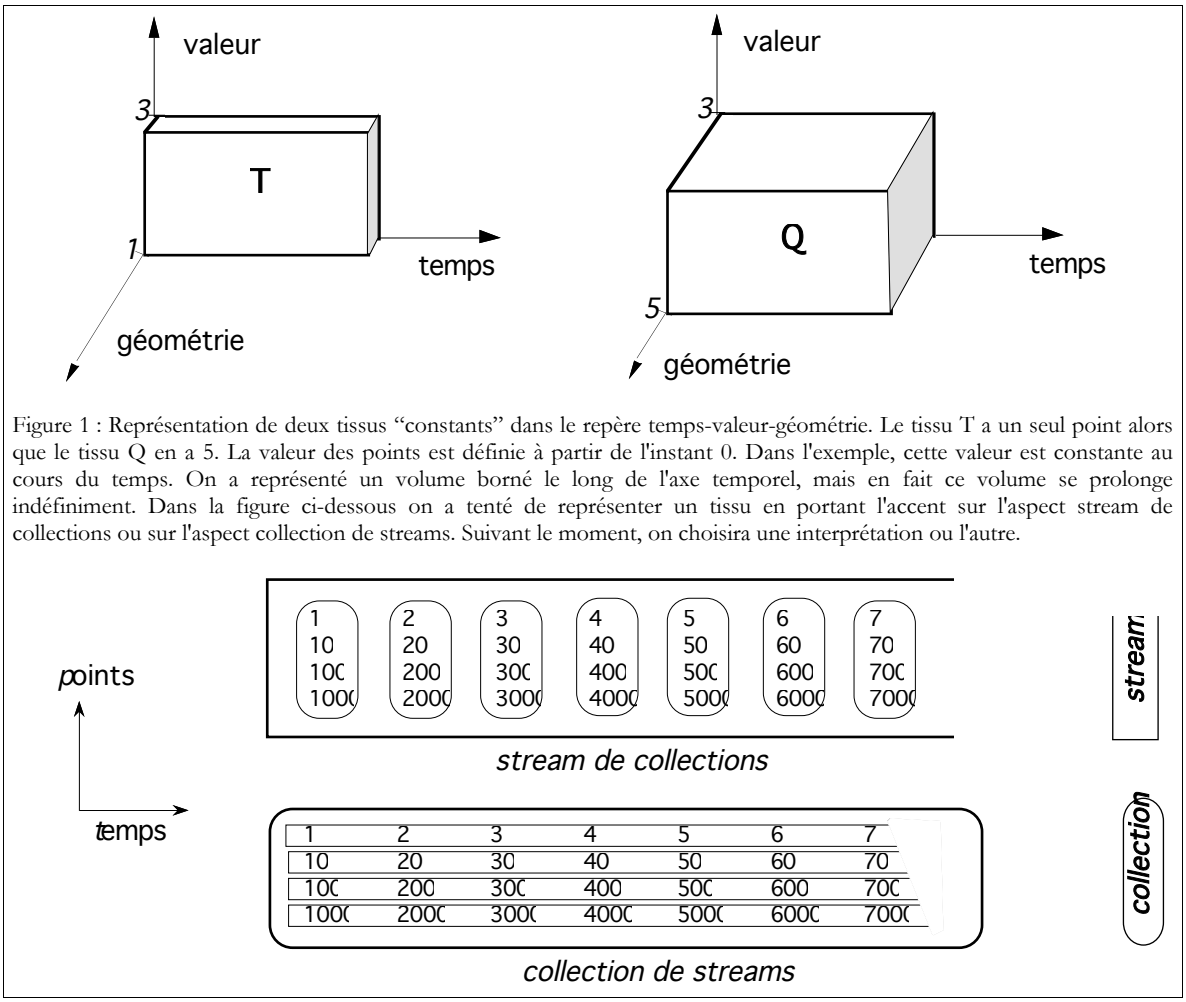


Figure 1 : Représentation de deux tissus "constants" dans le repère temps-valeur-géométrie. Le tissu T a un seul point alors que le tissu Q en a 5. La valeur des points est définie à partir de l'instant 0. Dans l'exemple, cette valeur est constante au cours du temps. On a représenté un volume borné le long de l'axe temporel, mais en fait ce volume se prolonge indéfiniment. Dans la figure ci-dessous on a tenté de représenter un tissu en portant l'accent sur l'aspect stream de collections ou sur l'aspect collection de streams. Suivant le moment, on choisira une interprétation ou l'autre.

I.1.1. Un langage déclaratif

81/2 est un langage déclaratif : un programme consiste uniquement en des définitions de tissus. Contrairement à un langage impératif où il faut indiquer la suite des actions à effectuer pour donner une valeur à une variable, on spécifie en 81/2 des relations qui sont toujours vraies entre les valeurs des variables. Une définition prend la forme d'une équation qui se termine par un point-virgule :

A = B + C ;

est une relation qui est vérifiée à chaque instant *t* et pour chaque point *p* de A. Ces relations permettent de définir des tissus à partir d'autres tissus.

Un programme 81/2 consiste donc en une série d'équations qui permettent de lier une variable à une expression. Comme le langage est déclaratif, il est toujours possible de remplacer toute occurrence d'une variable par sa définition. La définition d'une variable est l'expression qui apparaît à droite d'un signe « = ». Par exemple la définition de A est l'expression « B + C ». Il n'y a qu'une seule définition pour chaque variable, mais on verra plus loin que des règles de visibilité permettent d'avoir des variables qui ont le même identificateur apparent mais qui sont en réalité différentes.

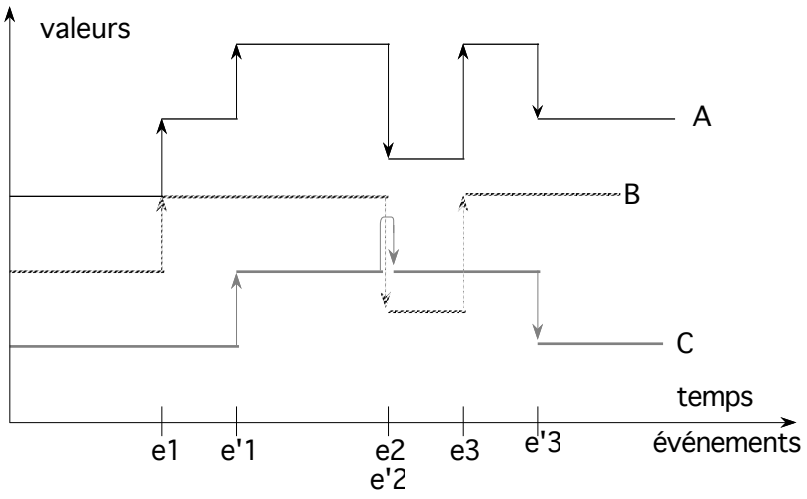
I.1.2. La comparaison temporelle de deux suites

Un tissu correspond à une suite de collections (on dira plus simplement une « suite de valeurs », le terme de valeurs désignant indifféremment un scalaire, une collection de scalaire ou un tissu). Si on raisonne en termes de système dynamique discret, un tissu correspond à une variable du système. Chaque élément de la suite de valeurs du tissu correspond donc à l'occurrence d'un événement intervenant dans la trajectoire de la variable.

Que se passe-t'il si on compare deux trajectoires ? Les événements caractéristiques de chacune des trajectoires sont distincts. Pour pouvoir comparer les deux trajectoires, il faut disposer d'une horloge universelle qui permet de comparer les événements de chaque trajectoire (Cf. figure 2). Quand on écrit une relation

A = B + C ;

B et C ne partagent pas forcément les mêmes événements. Les événements caractéristiques de A sont les événements qui modifient la trajectoire de A. C'est donc la réunion des événements qui concernent B et de ceux qui concernent C.



B	2	3	1,5	3		
C	1	2	2	1		
A	3	4	5	3,5	4	3
B ⊕ C	3	5	3,5	4		

Figure 2 : Exemple de deux trajectoires B et C qui n'ont pas les mêmes événements. Cependant les deux trajectoires partagent l'événement e2 (ou e'2). La trajectoire A correspond à l'addition des deux trajectoires B et C. Pour simplifier, on suppose que les tissus B et C ont un seul point qui a une valeur réelle. En-dessous on a représenté la suite des valeurs de B, C et A, ainsi que l'addition terme à terme des suites B et C, ce qui est noté par B ⊕ C. On ne tient donc pas compte du fait que les éléments des deux suites surviennent à des dates différentes. On remarque bien que la suite ainsi obtenue est différente de la suite des valeurs de A.

Si on retraduit ces considérations en termes de suites, on voit qu'il y a un problème : les suites B et C « ne progressent pas suivant le même rythme ». Faire l'addition terme à terme n'a pas de sens ainsi que le montre la table

de la figure 2. On voit donc qu'il nous faut un peu raffiner la notion de suite de valeurs afin qu'elle corresponde à celle de trajectoire.

Afin de simplifier la manipulation répartie des streams sur un calculateur parallèle, on va tout simplement éviter ces « trous » en les « remplissant » : il suffit d'intercaler entre chaque élément de la suite initiale, *des éléments de remplissage*. Ces éléments de remplissage s'appellent des **tics**. Le but d'un tic est de rendre possible la datation de chaque élément d'une suite : le $n^{ème}$ élément de la suite se produit au $n^{ème}$ instant de l'horloge universelle. Les éléments de la suite initiale, qui correspondent à l'occurrence d'un événement qui a une influence sur la trajectoire, s'appellent des **tops**. *La valeur d'un tic est la valeur du top qui précède.*

On suppose que l'horloge universelle est suffisamment précise pour distinguer des événements différents. De la précision de cette horloge dépend le nombre de tics qui seront utilisés pour synchroniser les différentes suites. Il n'est pas important que cette horloge soit « trop précise » car les tics sont des entités virtuelles qui n'ont pas obligatoirement d'incidence au niveau de l'implémentation du langage (c'est le cas de l'implémentation dynamique proposée au chapitre VII mais ce n'est pas le cas de l'implémentation statique proposée au chapitre VIII & IX). L'horloge mesure le passage du temps dans le programme. Plus exactement, l'horloge universel compte les événements qui correspondent à une modification du système. Pour simplifier, on peut appeler l'occurrence d'un événement un *instant* ou un tic. Les tics sont repérés par un nombre entier positif, le premier instant étant repéré par 0.

Pour résumer, un top correspond à un changement de la valeur courante du tissu (éventuellement, le tissu peut changer de valeur pour reprendre la même valeur). Un tic correspond à l'observation de la valeur d'un tissu. Les streams en 81/2 sont tels que les opérations entre tissus peuvent se faire élément par élément car les différents tissus, grâce aux tics, sont *synchrones* entre eux. Une définition « A = *f* (B, C); » spécifie un tissu A dont les changements seront justes ceux nécessaires à maintenir la relation. Autrement dit, les tops de A dépendent directement des tops de B et de C et de la nature de la fonction *f*. Enfin, l'évaluation d'une expression 81/2 consiste à calculer les valeurs des points de l'expression pour chaque top de son horloge.

I.1.3. Valeur indéfinie

Dans l'exemple précédent, on a pu additionner les valeurs de B et C à chaque instant car celles-ci étaient *définies*. Or il se peut qu'une valeur soit indéfinie. Par exemple on verra qu'il est possible de définir un tissu T qui n'a de valeur qu'à partir de l'instant 3. Cet instant serait alors aussi un top. Avant l'instant 3, la valeur de T est indéfinie. On note cette valeur *NIL*. Les expressions en 81/2 sont généralement strictes, ce qui veut dire que quand NIL intervient dans un calcul, le résultat est NIL (les exceptions sont les formes conditionelles).

I.1.4. La géométrie d'un tissu

La géométrie d'un tissu est spécifiée à l'aide d'une qualification qui suit le nom de la variable au moment de sa définition ou de sa déclaration. Par exemple dans la définition

T [5] = A + B; (2)

« [5] » est la spécification d'une géométrie. En l'occurrence il s'agit de la géométrie d'un *tissu plat* : un tissu dont la valeur des points est un stream de scalaire. La définition (2) spécifie un tissu de nom T et qui doit comporter 5 points. Le nombre de points d'un tissu est appelé son **cardinal**.

La géométrie des tissus complexes (dont la valeur des points est un tissu) se définit en composant les géométries simples :

U [[10] [10] [10] [10]]

est un tissu comportant 4 points, chaque point ayant pour valeur un tissu de 10 points. Comme cette notation peut rapidement devenir pénible à écrire, on dispose de la contraction suivante :

U [4, 10]

De manière générale, la spécification d'une géométrie de la forme [a, G] doit s'interpréter comme une abréviation de **[**[G] … [G]**]** *où* [G] *apparaît* a *fois.*

La spécification de la géométrie d'un tissu est optionnelle dans le cas d'une définition : la plupart du temps le compilateur est capable d'inférer la géométrie du tissu à partir de l'expression de définition. La géométrie d'un tissu est *statique*, c'est-à-dire que l'ensemble des points d'un tissu est un ensemble déterminable au moment de la compilation et qui ne dépend pas du temps. Le chapitre IV explique les techniques à mettre en œuvre pour vérifier ces propriétés.

I.2. Les constantes

Une expression 81/2 construit un tissu à partir d'autres tissus. Les tissus élémentaires sont les *tissus constants* et les *tissus externes*. Les tissus constants sont présentés dans ce paragraphe et les tissus externes dans le suivant. Les sections qui suivent sont dédiées à la présentation des opérateurs permettant de construire des expressions.

I.2.1. Les tissus constants

Toute constante scalaire peut se transformer implicitement en un tissu constant. Par exemple

5

est une expression correspondant à un tissu constant dont la valeur est 5. L'horloge d'un tissu constant ne comporte qu'*un seul top* en 0.

La géométrie d'un tissu constant est la plupart du temps inférée par le compilateur. Une constante scalaire permet donc de noter de manière ambiguë plusieurs tissus constants de géométrie différente. Mais une constante dénote toujours un tissu plat. Par exemple

T [1] = 5 ; (3)
U [100] = 5; (4)

dans l'équation (3) la constante 5 dénote un tissu constant de 1 point. Dans l'équation (4) la constante 5 dénote un tissu constant de 100 points. Chaque point a pour valeur le scalaire 5. L'instant 0 est le seul top de U.

I.2.2. Les tissus externes

Les tissus externes sont des tissus qui n'ont pas de définition. Leur valeur est fournie au moment de l'évaluation par un mécanisme étranger à l'évaluateur 81/2. Les tissus externes peuvent par exemple correspondre à une entrée (en termes de SDD, ce sont des *stimuli*). Dans le cas de l'interprète 81/2 présenté dans le chapitre X, la valeur de ces tissus est demandée interactivement à l'opérateur. Dans le cas d'une exécution compilée, ces valeurs peuvent être cherchées dans un fichier. Ces tissus doivent être *déclarés* à l'aide du mot-clef *outside.*

outside T [3] ;

La spécification de la géométrie est *obligatoire.*

Il y a un tissu externe qui est prédéfini : il s'agit du tissu identifié par **CLOCK**. CLOCK est un tissu plat, qui comporte un seul point et dont la valeur booléenne est toujours *vraie.* Les tops de CLOCK ne sont pas déterminés mais toute évaluation d'un programme 81/2 contenant CLOCK vérifie les propriétés suivantes :

– l'instant 0 est un top de CLOCK;

– pour tout instant t, il y a un top de CLOCK qui se produit après l'instant t

– le nombre de tics entre deux tops de CLOCK est borné par une constante finie.

On verra plus tard que ces propriétés nous sont nécessaires afin de faire progresser certains calculs.

I.3. Les opérateurs spatiaux

Dans cette section ainsi que dans les suivantes, nous allons présenter les opérateurs permettant de construire des expressions. Cette section est réservée aux opérateurs qui agissent sur la dimension spatiale d'un tissu : ce sont des opérateurs qui n'ont pas d'influence sur l'aspect stream. On peut donc dans ce qui suit faire comme si un tissu était juste une collection. L'action d'un opérateur est répétée aux éléments successifs du stream.

Il faut néanmoins préciser quels sont les tops de « *f* (A, B) » quand *f* est un opérateur géométrique. Les tops d'une expression géométrique sont obtenus simplement en faisant l'union des tops des arguments. En effet, l'expression « A + B » change de valeur chaque fois que A **ou** que B changent de valeur. Il est donc naturel que les tops de cette expression soient l'union des tops de A et de B. La table 1 donne un exemple pour l'addition (les tops sont en **gras**, les tics en maigres).

événements	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	…
A	NIL	**0**	0	**1**	1	1	**2**	2	2	**3**	3	…
B	**10**	10	**11**	11	**12**	**12**	12	12	**13**	**13**	13	…
A+B	NIL	**10**	**11**	**12**	**13**	**13**	**14**	14	**15**	**16**	16	…

Table 1 : Exemple d'opérations géométriques. Dans cet exemple on considère deux tissus A et B qui ont un seul point ayant une valeur entière. On a figuré dans le tableau le début de la trajectoire de ces deux tissus ainsi que la trajectoire du tissu obtenu en additionnant A et B. L'addition se fait élément par élément de la trajectoire. Les tops du résultat correspondent aux événements de A **et** à ceux de B.

I.3.1. L'arithmétique

L'arithmétique sur les scalaires s'étend naturellement sur les collections grâce à l'*alpha-notation* : à partir d'une fonction scalaire *f* on obtient une fonction entre collections en appliquant *f* élément par élément. Par exemple

$$\begin{pmatrix}1\\2\\3\\4\end{pmatrix} + \begin{pmatrix}10\\11\\12\\13\end{pmatrix} \Rightarrow \begin{pmatrix}11\\13\\15\\17\end{pmatrix}$$

(le signe ⇒ doit se lire : *a pour valeur*, les grandes parenthèses sont une notation qui ne fait pas parties du langage : elles sont utilisées dans les exemples pour donner les valeurs d'une collection). L'alpha-notation est implicite pour tous les opérateurs du langage. Les expressions arithmétiques demandent que les arguments aient la même géométrie. Comme on peut le remarquer dans la table 1, les opérateurs arithmétiques sont stricts : le résultat vaut NIL si un des arguments a pour valeur NIL.

Nous allons examiner plus en détail l'opérateur conditionnel. Celui-ci n'adopte pas la syntaxe C, le point d'interrogation et le deux-points étant réservés à un autre usage. Une expression conditionnelle 81/2 prend la forme suivante

```
if A then B else C fi
```

Cette conditionnelle s'applique à chaque point de la collection (rappelons que l'on oublie dans cette section l'aspect temporel d'un tissu). Par exemple

$$\text{if}\begin{pmatrix}\text{true}\\ \text{false}\\ \text{false}\\ \text{true}\end{pmatrix}\ \text{then}\begin{pmatrix}1\\ 33\\ \text{NIL}\\ 4\end{pmatrix}\ \text{else}\begin{pmatrix}14\\ 2\\ 3\\ 28\end{pmatrix}\ \text{fi}\quad\Rightarrow\quad\begin{pmatrix}1\\ 2\\ 3\\ 4\end{pmatrix}$$

Comme on le voit dans cet exemple, l'opérateur conditionnel n'est pas strict (mais il est strict en son premier argument).

La beta-réduction est définie grâce à l'opérateur « \ » :

$$+\setminus\begin{pmatrix}1\\ 2\\ 3\\ 4\end{pmatrix}\quad\Rightarrow\quad(10)\qquad\qquad\&\&\setminus\begin{pmatrix}\text{true}\\ \text{false}\\ \text{false}\\ \text{true}\end{pmatrix}\quad\Rightarrow\quad(\text{false})$$

(cette notation ne correspond pas à la convention APL où on utilise « / », mais ce dernier est utilisé pour la division dans le langage C). La beta-réduction permet de réduire une collection en utilisant une fonction dyadique. La réduction d'une collection en utilisant l'addition donne par exemple la somme des éléments de la collection. La réduction d'une collection en utilisant la fonction *min* a pour valeur l'élément minimal de la collection.

Le *scan* est obtenu en doublant l'opérateur de réduction :

$$+\setminus\setminus\begin{pmatrix}1\\ 2\\ 3\\ 4\end{pmatrix}\quad\Rightarrow\quad\begin{pmatrix}1\\ 3\\ 6\\ 10\end{pmatrix}$$

le résultat d'un scan correspond à la collection des résultats partiels de la réduction.

I.3.2. L'imbrication et concaténation des tissus

Les grandes parenthèses utilisées pour exprimer une collection en extension ont un équivalent dans le langage : c'est l'opérateur « { · } »

A = { 1, 2, 3 } ;

A est un tissu de cardinal 3. Chaque élément est un élément scalaire qui a un seul top en 0. Les virgules sont optionnelles. Les accolades permettent de créer des imbrications :

B = { A {4, 5, 6} } ;
B ⇒ { { 1, 2, 3 }, {4, 5, 6} }

Si on compare la collection d'un tissu à une liste, les accolades correspondent à la fonction *list* en LISP. Par convention, les deux expressions 1 et {1} sont équivalentes : elles dénotent toutes deux un tissu plat constant de cardinal 1. Cette convention n'est pas contradictoire car les constantes scalaires dénotent un tissu de manière ambiguë; par contre elle simplifie l'écriture des programmes. Les accolades vont nous permettre à présent d'écrire simplement nos exemples.

Un analogue à la fonction *append* en LISP existe, c'est l'opérateur « # » qui permet la concaténation de deux tissus :

C = A # { 4, 5, 6 } ;
C ⇒ { 1, 2, 3, 4, 5, 6 }

I.3.3. Cardinal et index d'un tissu

Le cardinal d'un tissu s'obtient par l'opérateur « | . | » :

| { 1, 2, 3, 4 } | ⇒ 4
| { {24, 36} {44, 2} {63, 7} } | ⇒ 3

Un autre opérateur utile est la quote qui permet de générer un tissu plat constant dont le premier point a pour valeur 0 et le dernier point, le cardinal du tissu moins 1. Cet opérateur est le pendant du *iota* APL. Par exemple :

' { {24, 36} {44, 2} {63, 7} } ⇒ { 0 1 2 }

Cet opérateur n'est pas primitif : on peut le réécrire en fonction d'autres opérateurs, comme on le verra dans la section consacrée aux exemples.

I.3.4. Projection, troncature et duplication

Il est possible d'accéder à la valeur d'un point grâce à l'opérateur point :

A = { { 1, 2, 3 }, {4, 5, 6} } ;

A.0 ⇒ { 1, 2, 3 } A.1 ⇒ {4, 5, 6} A.2 ⇒ *erreur*
A.0.1 ⇒ 2

Cet opérateur est un opérateur de projection. Seule une constante peut apparaître à droite du point. Des expressions comme : A.B où A et B sont des tissus (B serait un tissu de cardinal 1) sont interdites.

Il existe un opérateur permettant de tronquer et de dupliquer un tissu. C'est l'opérateur « · : [·] ». L'argument entre crochets doit être une géométrie. Par exemple :

A = { { 1, 2, 3 }, {4, 5, 6} } ;
A : [1] ⇒ { { 1, 2, 3 } }
A : [5] ⇒ { { 1, 2, 3 }, {4, 5, 6}, { 1, 2, 3 }, {4, 5, 6}, { 1, 2, 3 } }

Si la géométrie indiquée a un cardinal plus petit que l'argument, l'argument est tronqué. Dans le cas contraire, on duplique les éléments dans l'ordre jusqu'à obtenir le cardinal désiré. À la place de la spécification d'une géométrie, il est possible de mettre un tissu : la géométrie qui sera utilisée sera la sienne :

B = { 1, 2, 3 } ;
A : [B] ⇒ A : [3] ⇒ { { 1, 2, 3 }, {4, 5, 6}, { 1, 2, 3 } }

L'effet d'une géométrie complexe correspond à l'application répétée de troncatures/duplications. On commence par dupliquer ou tronquer le tissu afin que son cardinal correspondent au cardinal de la géométrie. Ensuite on applique chaque sous-géométrie aux points correspondants :

A : [[a] G] ⇒ ((A:[b]).0 : [A]) # (cdr(A:[b]) : [G])

avec b le cardinal de [[a] G] et la fonction cdr renvoyant un tissu privé de son premier point. Un exemple :

A : [3 4]
⇒ A : [[4] [4] [4]] *par définition d'une géométrie*
⇒ { { 1, 2, 3 }, {4, 5, 6}, { 1, 2, 3 } } : [[4] [4] [4]] *on ajuste les cardinaux*
⇒ { { 1, 2, 3 }:[4], {4, 5, 6}:[4], { 1, 2, 3 }:[4] } *on applique les sous-géométries*
⇒ { { 1, 2, 3, 1 }, {4, 5, 6, 4}, { 1, 2, 3, 1 } } *on obtient le résultat final*

I.3.5. Réindexation

Projection, troncature et duplication ne sont que des particularisations de l'opérateur général de *réindexation* (appelé aussi *sélection* car il permet de « choisir » des points dans un tissu). La syntaxe est la suivante :

Source **(**Adresse**)**

Le tissu Source est le tissu qui va fournir les valeurs. Le tissu Adresse est un tissu dont les valeurs sont des entiers compris entre 0 et (|Source| - 1). Le résultat est un tissu qui a la géométrie de Adresse si le tissu Source est un tissu plat. Les valeurs du résultat sont obtenues de la manière suivante :

(Source(Adresse)).i ≡ Source.(Adresse.i) $0 \leq i <$ |Adresse|

l'expression précédente est incorrecte en 81/2 car on ne peut avoir un deuxième argument variable pour l'opérateur point. Mais elle exprime intuitivement la sémantique de la réindexation. Donnons un exemple :

C = { 1, 2, 3, 4, } ; D = { 2, 1, 0 } ;

C (D) ⇒ { 3, 2, 1 }
C (0) ⇒ 1
C (10) ⇒ *erreur*

Si la géométrie du tissu Source est complexe, le résultat a pour géométrie une géométrie obtenue en substituant aux points-feuilles de Adresse, les valeurs des points de Source :

A = { { 1, 2, 3 }, {4, 5, 6} } ;
E = { { 1, 0}, { 0, 1} } ;

A ({1,0}) ⇒ { {4, 5, 6}, {1, 2, 3 } }
A (E) ⇒ { {{4, 5, 6}, { 1, 2, 3 }}, {{ 1, 2, 3 }, {4, 5, 6}} }

Deux options sont envisagées actuellement : dans une version complètement statique du langage, le tissu Adresse doit être un tissu constant. On peut alors calculer au moment de la compilation les communications nécessaires à l'exécution du programme. C'est un schéma qui est adapté à l'approche complètement statique d'une machine reconfigurable comme PTAH [Cappello, Béchennec 91] où la topologie des communications doit être connue avant l'exécution du programme. Une deuxième version permet un tissu quelconque en place du tissu Adresse. Dans ce cas, les tops du résultat correspondent à l'union des tops des deux arguments.

I.4. Les opérateurs temporels

Dans cette section, nous allons examiner les opérateurs temporels. Ces opérateurs ont un effet sur l'aspect stream d'un tissu et n'agissent pas sur la dimension spatiale. Pour simplifier la présentation, on va donc supposer que les tissus sont des tissus plats réduits à un seul point.

Afin de faciliter la lecture des exemples, on va utiliser la notation suivante :

< **1**, **2**, 2, 2, **3**, 3, **4**, … >

pour indiquer un stream dont le premier élément est 1, le deuxième 2, etc. *Cette notation ne fait pas partie du langage* 81/2. Les tics seront représentés par des valeurs en caractères maigres, les tops par des valeurs en gras. Ainsi dans l'exemple précédent, Les instants 0, 1, 4 et 6 sont des tops. Quand on parle de l'horloge d'un tissu, on veut parler de l'ensemble de ses tops.

I.4.1. Définition markovienne *des streams*

Contrairement à LUCID, nous contraignons les opérations sur les streams à vérifier une propriété *markovienne* : la valeur courante d'un stream ne peut dépendre que des n précédentes valeurs. Le nombre n doit être une constante calculable à la compilation. Cette contrainte existe aussi dans des langages pour la programmation temps-réels comme LUSTRE et SIGNAL qui sont dérivés de LUCID. Cette hypothèse est compatible avec les systèmes que nous voulons simuler. Du point de vue de l'implémentation elle permet de forcer un ordre d'évaluation séquentielle des valeurs d'un stream et de réserver à la compilation, l'espace mémoire nécessaire aux calculs.

I.4.2. L'observateur temporel

De la même façon qu'il existe un opérateur permettant d'extraire la valeur d'un point, il existe un opérateur

permettant d'observer la valeur d'un tissu à un top donné :

T ⇒ < **1**, **2**, 2, 2, **3**, 3, **4**, … >
T@2 ⇒ < NIL, NIL, NIL, NIL, **3**, 3, 3, … >

est un tissu qui a un seul top correspondant au troisième top de T. Sa valeur est indéfinie avant l'occurrence de ce top. Ensuite la valeur est une valeur constante qui est celle de T au troisième top. Si le tissu en argument n'a pas « suffisamment » de top, le résultat est un tissu dont la valeur est toujours indéfinie :

U = 1;
U ⇒ < **1**, 1, 1, … >
U@1 = < NIL, NIL, NIL, … >

I.4.3. Le retard

L'opérateur de retard permet de référencer la valeur du top précédent. Il se note à l'aide de « $ »

A ⇒ < **1**, **2**, 2, 2, **3**, 3, **4**, … >
$A ⇒ < **NIL**, **1**, 1, 1, **2**, 2, **3**, 3, … >

Il est plus simple de comprendre l'effet du délai si on fait se correspondre les valeurs des streams :

événements	*0*	*1*	*2*	*3*	*4*	*5*	…
A	**1**	**2**	2	**3**	3	**4**	…
$A	**NIL**	**1**	1	**2**	2	**3**	…
0	**0**	0	0	0	0	0	…
$0	**NIL**	NIL	NIL	NIL	NIL	NIL	…

Table 2 : Exemple de tissu retardé. On utilise la convention que les tops sont en fonte grasse et les tics en maigre.

L'horloge d'un tissu A retardé est exactement la même que l'horloge de A. La première valeur de $A qui soit définie est celle qui se produit sur le second top de A. La valeur en un top de $A est la valeur du top de A qui précède. Il faut remarquer qu'un délai appliqué à un tissu qui a un seul top a une valeur toujours indéfinie (c'est le cas quand on retarde une constante, Cf. table 2).

I.4.4. Quantification des équations

La première valeur d'un tissu retardé n'est pas définie. Éventuellement, on a besoin d'imposer une valeur à ce premier top. Pour cela, on utilise la « quantification » des équations (on parlera aussi de « définition multiple »). La définition :

T@0 = A ;

permet d'indiquer que la valeur du premier top de T est donné par la valeur du premier top de A. Le qualificatif « @0 » ne doit pas être confondu avec l'opérateur « @ » qui apparaît dans une expression (la partie gauche d'une définition n'est pas une expression). Il limite la portée d'une équation à un top donné. L'argument derrière l'arobasque est obligatoirement une constante entière.

L'expression précédente ne correspond pas à une définition complète : il manque la définition du tissu pour les tops autres que 0. Le programme suivant est complet :

A ⇒ < **3**, 3, 3, 3, **4**, **5**, 5, 5 …>

T@0 = 0 ;
T = A ;

T ⇒ < **0**, 0, 0, 0, **4**, **5**, 5, 5, … >

L'ordre des équations n'a pas d'importance. On aurait pu tout aussi bien écrire :

T = A ;
…
T@0 = 0 ;

La quantification des équations permet de donner une valeur initiale à un tissu défini à l'aide d'un retard. Par exemple :

A ⇒ < **3**, 3, 3, 3, **4**, **5**, 5, 5 …>

U@0 = 0 ; U = $A ;

U ⇒ < **0**, 0, 0, 0, **3**, **4**, 4, 4, … >

Un autre exemple est donné par la figure 3 et montre que la quantification correspond à un *aiguillage* : l'équation qui donne la valeur d'un tissu est sélectionnée en fonction du nombre de tops déjà passés.

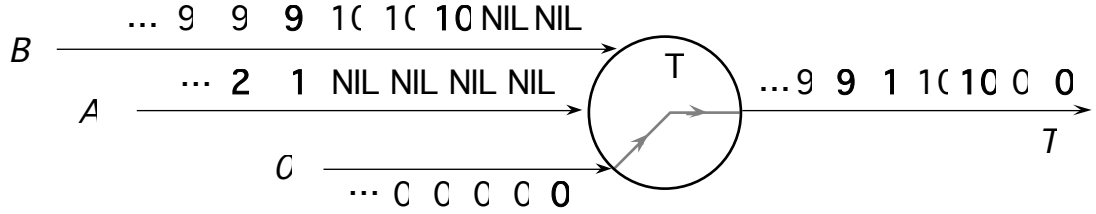


Figure 3 : Un exemple de quantification. Le premier top de T correspond au premier top de 0. Le second top de T est spécifié par la définition sans quantification (car il n'y a pas de définition de la forme « T@1 = … ». Le second top de T est le premier top de B qui se produit après l'occurrence du premier top de T. Le premier top de T ayant eu lieu à l'instant 0, on doit attendre le premier top de B qui se produit *après* l'instant 0 : c'est un top qui a lieu à l'instant 2 et la valeur de B est 10.

La valeur du troisième top de T est spécifiée par la définition T@2 = A. On doit donc attendre le premier top de A qui survient *après* l'instant 2. Il s'agit de l'instant 4, et la valeur de A est 1.

Les autres tops de T sont spécifiés par l'équation sans quantification. Le quatrième top de T est le top de B qui se produit après l'instant 4. Après la survenue de ce top, la suite des valeurs de T est semblable à la suite des valeurs de B. On remarquera que la valeur de T en un tic est toujours la valeur du top *de* T qui précède.

I.4.5. L'échantillonneur

Nous avons indiqué au début du chapitre que la « vitesse » de chaque suite pouvait être différente. Cela nous a amené à introduire des tics afin de pouvoir comparer les suites élément par élément. Jusqu'à présent nous n'avons pas rencontré d'opérateur permettant d'agir sur une horloge. L'opérateur « when » permet de le faire. Dans l'équation

T = A when B;

le tissu B doit être un tissu de cardinal 1 et de valeur booléenne. L'horloge du résultat est constituée des tops de B sur lesquels B est vrai. La valeur du tissu résultat est la valeur du tissu A sur les tops de la nouvelle horloge.

Cette opération est une opération de contrôle : elle peut se voir comme une vanne permettant de faire passer ou non le flot des valeurs de A. C'est pourquoi cet opérateur est appelé un *échantillonneur* ou encore un *trigger*. La table 3

donne un exemple et la figure 4 illustre les différents opérateurs d'échantillonnage.

événements	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
A	**1**	**2**	2	**3**	**4**	4	**5**	**6**	6
B	**NIL**	NIL	**false**	**true**	true	**true**	true	**false**	false
A when B	NIL	NIL	NIL	**3**	3	**4**	4	4	4

Table 3 : Exemple d'opérateur when.

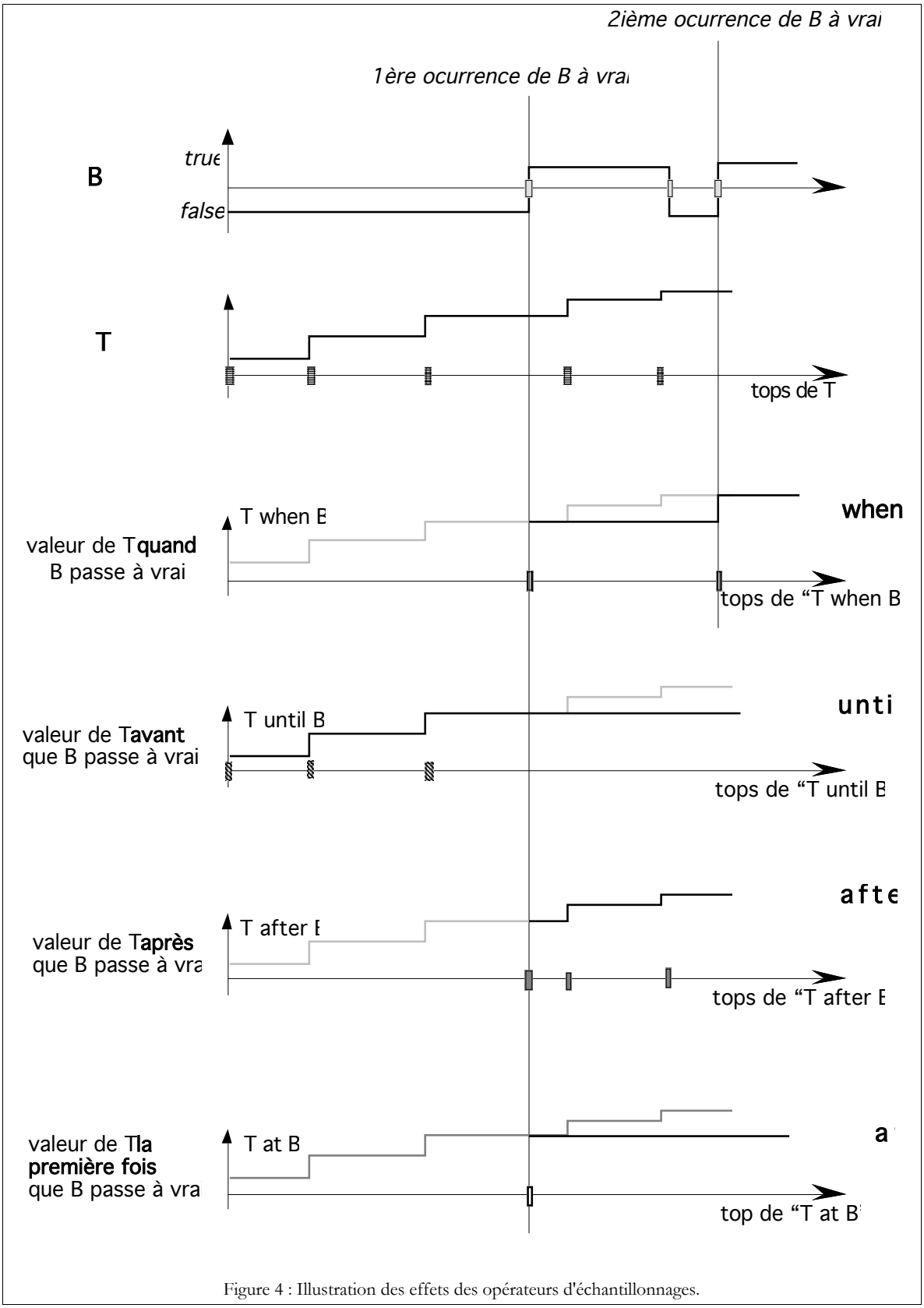


Figure 4 : Illustration des effets des opérateurs d'échantillonnages.

I.4.6. Les filtres

Les filtres sont des opérateurs du même type que l'opérateur when. Ils sont de la forme

tissu until / at / after *commande*

Ils permettent de découper une suite suivant la première occurrence à vrai de la commande (Cf. figure 4 et table 4). L'opérateur « until » permet de geler la progression d'une suite. L'opérateur « after » correspond à un filtre passe-haut. L'opérateur « at » correspond à la version dynamique de l'opérateur « @ ».

événements	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
A	**1**	**2**	2	**3**	**4**	4	**5**	**6**	6
B	**NIL**	NIL	**false**	**true**	true	**true**	true	**false**	false
A until B	**1**	**2**	2	2	**2**	2	2	2	2
A after B	NIL	NIL	NIL	**3**	**4**	4	**5**	**6**	6
A at B	NIL	NIL	NIL	**3**	3	3	3	3	3

Table 4 : Exemple d'échantillonnages (le premier top à vrai est indiqué par une colonne surlignée)

I.4.7. Le traitement de la valeur NIL

NIL est une valeur stricte pour les opérateurs arithmétiques 81/2. Ce n'est pas le cas pour la conditionnelle (sauf si c'est la valeur de l'expression booléenne). Ce n'est pas le cas non plus en tant que valeur de la commande d'un échantillonneur ou d'un trigger. Dans ce cas, la valeur NIL agit de manière similaire à la valeur *false*.

I.5. La structuration du calcul

Trois concepts permettent de structurer les programmes 81/2. Le premier est le *bloc*. Il consiste à enrichir le concept de tissu complexe afin d'en faire le support de la notion de module. Le deuxième outil de structuration est la *fonction*. En 81/2 il y a quatre manières différentes d'appliquer une fonction. Enfin, un opérateur particulier, *every*, permet de créer des sous-suites, ce qui permet de structurer le flot des calculs.

I.5.1. Le bloc

Il y a quelques différences entre un tableau et un *record* PASCAL :

– Les éléments d'un record peuvent être de type différent alors que les tableaux contiennent généralement des éléments de même type.

– Les éléments sont accédés à travers un label plutôt que par un indice. On peut obtenir un indice comme résultat d'un calcul, mais pas un label

Mais une collection 81/2 a un aspect statique qui la rapproche plus du record que d'un tableau : par exemple on ne peut pas accéder à un point par un index calculé. D'où l'idée d'enrichir la notion de tissu complexe afin de supporter les mécanismes de label d'un record. Ces mécanismes sont surtout intéressants car ils permettent de limiter la visibilité d'un identificateur. Un bloc 81/2 est un tissu énuméré où les éléments de l'énumération sont des définitions et pas des expressions. Par exemple :

```
Auto = {
    vitesse = V1;
    essence@0 = 50; essence = $essence - 0.6*v1;
}
```

Auto est un bloc constitué de deux tissus : vitesse et essence. On peut accéder à ces tissus par l'opérateur point, exactement comme les champs d'une structure C :

```
Auto.vitesse
```

Les identificateurs vitesse et essence ne sont pas visibles en dehors de Auto (comme les labels des champs d'une structure C) : le bloc est une unité d'encapsulation.

I.5.2. Les fonctions : application, alpha-notation, beta-réduction et scan explicites

La définition d'une fonction correspond simplement à une expression paramétrée :

```
f (x, y) = x + y ;
```

L'expression de définition d'une fonction peut faire appel à d'autres fonctions mais il ne peut y avoir de définition récursive. Les arguments d'une fonction ne peuvent être que des tissus. Par contre les fonctions ne sont pas typées : on ne précise pas le type des arguments ni le type du tissu qui est défini. En conséquence, les fonctions ainsi définies sont *polymorphes* (i.e. elles peuvent s'appliquer à des tissus qui ont des géométries différentes).

81/2 fournit 4 mécanismes d'application de fonction sur les tissus :

nom	**signature**		**syntaxe**
apply:	$(\text{collection}^p \rightarrow X) \times \text{collection}^p \rightarrow X$		f(T, …)
alpha:	$(\text{scalar}^p \rightarrow \text{scalar}) \times \text{collection}^p \rightarrow \text{collection}$		f^ (T, …)
beta:	$(\text{scalar}^2 \rightarrow \text{scalar}) \times \text{collection} \rightarrow \text{scalar}$	$f \setminus T$	
scan:	$(\text{scalar}^2 \rightarrow \text{scalar}) \times \text{collection} \rightarrow \text{collection}$	$f \setminus\setminus T$	

(X est un scalaire ou une collection, p est l'arité de la fonction *f*, T est une collection). Le premier mécanisme correspond à l'application habituelle. Par exemple :

```
horloge (x) = (x == x);
B = horloge(A);
```

A ⇒ < {**3 NIL**}, {3, NIL}, {3 **10**}, {3 10}, {**4** 10}, {**5 11**} …>
B ⇒ < {**true NIL**}, {true, NIL}, {true **true**}, {true true} {**true** true}, {**true true**} …>

La fonction horloge est une fonction qui a pour but de créer un tissu booléen qui vaut toujours vrai ou NIL et qui a les mêmes tops que son argument (on peut utiliser ce tissu pour commander un when à partir d'un tissu plat de cardinal 1 par exemple). Les noms de fonctions doivent être distincts des noms de tissus. Ainsi, l'expression

```
F (A)
```

n'est pas ambiguë : si F est un tissu, cette expression correspond à une réindexation. Si c'est un nom de fonction, c'est une application.

La deuxième manière d'appliquer une fonction correspond à l'appliquer à chaque point d'un tissu : c'est une alpha-notation. Les opérateurs de l'arithmétique scalaire 81/2 sont tous implicitement alpha-dénotés. Mais une notation explicite peut être nécessaire (Cf. plus loin dans ce chapitre les exemples). Dans ce cas le symbole « ^ » vient s'intercaler entre la fonction et la liste des paramètres :

```
C = horloge ^ (A) ;
D = f ^ (A, A).
```

La troisième manière d'appliquer une fonction correspond à la réduction. L'ordre des applications de la réduction n'est pas spécifié. Une réduction se note de la même manière que pour les opérateurs arithmétiques : le symbole « \ » pour la beta-réduction, ou le symbole « \\ » pour le scan est inséré entre la fonction et l'argument : « C = f \ A ; ». Les fonctions utilisées dans une réduction sont toujours des fonctions à deux arguments.

II.5.3. L'opérateur every

L'opérateur « every » permet de créer des *sous-calculs* : il insère l'évaluation d'un programme 81/2 tout entier sur un tic de l'horloge globale. La forme d'une expression *every* est la suivante :

résultat every T where { *body* }

où *body* est un programme 81/2 et *résultat* est une variable définie dans *body*. La variable T est une variable qui est définie en dehors de *body*. L'évaluation de cette expression se fait de la manière suivante :

- Les variables qui ne sont pas définies dans *body* doivent être définies à l'extérieur. Les variables définies dans *body* sont locales à l'expression et invisibles de l'extérieur.
- Les tops de l'expression *every* sont les tops de T.
- À chaque top de T, le programme *body* est évalué. Tout se passe comme si le programme principal était « gelé » pendant l'évaluation de *body*. En particulier pendant l'évaluation de *body*, les variables externes sont des tissus constants qui ont pour valeur, la valeur que ces tissus externes ont à l'instant du top de T. Comme ce sont des tissus constants, à l'intérieur de *body* ces tissus ont un seul top, en 0.
- La valeur de l'expression est la valeur du premier top du tissu défini dans *body* et identifié par *résultat*. Si l'expression qui définit *résultat* dans *body* n'a pas de top, alors la valeur de l'expression est indéfinie et l'évaluation du programme tout entier ne se terminera pas.

Donnons un exemple. Supposons que l'on dispose de deux suites de scalaires et que l'on veuille élever les éléments de la première suite à la puissance indiquée par les éléments de la deuxième suite, par exemple :

A ⇒ < **1**, **2**, **3**, **4**, **5**, … >
B ⇒ < **2**, **4**, **2**, **2**, **1**, … >

et on veut obtenir < **1**, **16**, **9**, **16**, **25**, … >. Une définition possible de ce tissu est la suivante :

```
C =    r every A where {
                    i@0 = 1; i = $i + 1 when CLOCK;
                    t@0 = A; t = $t * A when CLOCK;
                    r = t at (i == B);
       } ;
```

L'usage de CLOCK sera éclairci un peu plus loin dans les exemples. Il nous suffit pour l'instant de savoir que la définition de i réalise un compteur et que les valeurs de t constituent la suite des puissances de la valeur initiale. Le tableau 5 ci-dessous représente ce qui se passe pour les deux premiers tops de A.

événements	*0*			*1*					…
A	**1**			**2**					…
B	**2**			**4**					…
	sub	*0*	*1*	*sub*	*0*	*1*	*2*	*3*	
	i	**1**	**2**	i	**1**	**2**	**3**	**4**	
	t	**1**	**1**	t	**2**	**4**	**8**	**16**	
	i == B	**false**	**true**	i == B	**false**	**false**	**false**	**true**	
	r	NIL	**1**	r	NIL	NIL	NIL	**16**	
C	**1**			**16**					…

Table 5 : Exemple d'utilisation de l'opérateur every. L'évaluation du corps de l'every (à chaque top de A) correspond à l'exécution d'un *sous*-programme dans lequel chaque tissu défini à l'extérieur du every est devenu une constante. Le label *sub* indique les événements locaux aux calculs du corps de l'every. De manière générale, l'arrêt du sous-calcul lié à l'every dépend du calcul qui est effectué et ne peut pas être prévu à la compilation, ni d'ailleurs au début de l'exécution de l'every.

I.6. Les définitions récursives et le calcul d'une expression

Il est tout à fait possible que l'expression de définition d'un tissu implique ce tissu : c'est une définition récursive. La récursion peut être directe (l'identificateur en partie gauche apparaît en partie droite d'une définition) ou indirecte (dans ce cas il y a une ou plusieurs variables intermédiaires). Utilisée sans précaution, la récursion risque fort de ne pas définir grand chose. Par exemple le programme

```
A = B;                (E1)
B = A;                (E2)
```

définit A (et B) par une récursion indirecte. Dans cet exemple, les valeurs des tissus A et B vont toujours être NIL et A et B auront un ensemble de tops réduit à l'ensemble vide.

Pour comprendre pourquoi, il faut considérer (E1)+(E2) comme un système d'équations entre des suites de collections. Ces équations doivent être vérifiées pour chaque top. Le système d'équations (E1)+(E2) admet une infinité de solutions : tous les couples de tissus de la forme (T, T) où T est un tissu quelconque.

De manière générale, quand il y a plusieurs solutions possibles, un programme 81/2 calcule « la plus petite solution » au sens d'une certaine relation d'ordre. Cette relation d'ordre est définie en détail dans le chapitre suivant. Mais on peut retenir que le plus petit tissu est celui qui a le moins de tops possibles. Dans notre exemple, le plus petit tissu possible qui vérifie les équations est un tissu qui n'a pas de tops.

II. Exemples de programmes 81/2

Nous donnons ici quelques exemples permettant de comprendre le langage 81/2. Le lecteur intéressé trouvera des exemples plus importants dans [Giavitto 91c], [Dumesnil 91] et [Legrand 91]. Ce dernier montre comment utiliser 81/2 pour programmer des réseaux comportementaux. Les réseaux comportementaux sont des systèmes adaptatifs utilisés pour modéliser les comportements animaux [Maes 90]. Dans [Dumesnil 91] est décrit un algorithme d'apprentissage en 81/2 pour un réseau à couches étudié dans le cadre de la machine PTAH.

La syntaxe des opérateurs arithmétiques est similaire à la syntaxe du langage C.

II.1. Exemples de définition récursive

Dans le paragraphe précédent, on a montré que la récursion utilisée sans précaution ne permettait pas de définir grand chose. En fait, il y a deux manières d'utiliser des équations récursives sans que le résultat se réduise à NIL : la récursion temporelle et la récursion spatiale.

II.1.1. Récursion temporelle

Nous allons introduire la récursion temporelle à partir de l'exemple du compteur. Nous voulons définir un tissu dont la valeur s'incrémente de 1 au cours du temps. La première expression qui nous vient est d'écrire :

```
T = $T + 1 ;
```

Cette définition est récursive : la valeur de T au top courant est la valeur de T au top précédent incrémentée de un. Cette définition admet une infinité de solutions : c'est tous les tissus dont la valeur est augmentée de 1 à chaque top. La figure 5 montre quelques solutions possibles.

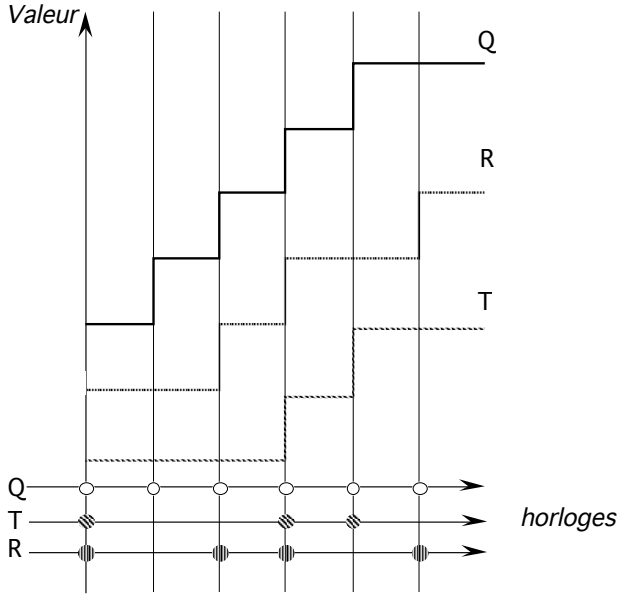


Figure 5 : Q, R et T sont différentes solutions possibles à l'équation X = $X +1.

Le tissu NIL (dont l'horloge est l'ensemble vide) est aussi solution. En fait, notre problème vient du fait que l'horloge du compteur est insuffisamment contrainte : à quel rythme doit compter le compteur ? L'opérateur *when* permet d'imposer un rythme quelconque. Utilisons par exemple celui de CLOCK :

T = ($T + 1) when CLOCK

Cette fois-ci on a imposé une horloge au tissu T mais ses valeurs sont indéfinies (la première valeur de $T est NIL et donc le tissu vaut constamment NIL). Qu'à cela ne tienne, on utilise une équation quantifiée pour imposer une valeur initiale :

T@0 = 0 when CLOCK; (E3)
T = ($T + 1) when CLOCK;

On peut vérifier que la seule solution à ce système d'équations est un tissu T qui a même horloge que CLOCK et dont la valeur s'incrémente de 1 à chaque top. L'expression « when CLOCK » dans l'équation (E3) a pour but d'imposer à T un premier top au même instant que le premier top de CLOCK. Il se trouve que le premier top de CLOCK se produit à la date 0, au même moment que le premier top de 0. On aurait donc pu écrire « T@0 = 0 » mais cela n'aurait pas été vrai pour un tissu quelconque autre que CLOCK.

La position de l'opérateur when a peu d'importance : les trois programmes suivant calculent tous la même chose :

T@0 = 0 when CLOCK;	T@0 = 0 when CLOCK;	T@0 = 0 when CLOCK;
T = ($T + 1) when CLOCK;	T = $T + (1 when CLOCK) ;	T = ($T when CLOCK) + 1;

En effet, ce qui est important c'est d'imposer une horloge à T. On peut le faire directement comme ci-dessus. Mais comme l'horloge d'une opération arithmétique correspond à l'union des horloges des arguments, on peut aussi appliquer le *when* à un quelconque des arguments.

On aperçoit à présent l'utilité de la constante CLOCK : elle donne accès à une horloge qui a un nombre infini

de tops. Sans cette constante, les programmes 81/2 qui ne font pas appels à des tissus externes ont obligatoirement un nombre de tops réduit à 1.

La forme de récursion utilisée ici est dénommée *récursion temporelle* car la valeur courante du tissu est définie en fonction de sa valeur passée.

I.1.2. Récursion spatiale

Il est possible de définir la valeur d'un point en fonction de la valeur d'un point « voisin ». Cette forme de récursion est la *récursion spatiale*. Donnons un exemple :

```
I [10] = (0 # (I + 1)) : [I] ;
```

I est un tissu de 10 éléments scalaires. Le premier élément est la constante 0. Le deuxième élément se calcule ainsi :

$$\mathbf{I.1} \Rightarrow ((0 \# (I + 1)) : [I]).1 \Rightarrow (0 \# (I + 1)).1 \Rightarrow (I + 1).0 \Rightarrow I.0 + 1 \Rightarrow 1$$

et de manière plus générale,

$$\begin{cases} I.0 & 0 \\ I.i & I.(i - 1) + 1 \quad \text{pour } 0 < i < 10 \end{cases}$$

Autrement dit, I ⇒ < **c**, c, c, c …> avec c = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }.

Le mécanisme qui est utilisé ici est tout à fait semblable à la récursion temporelle. Dans la récursion temporelle, l'opérateur « $ » permet de référencer une valeur déjà calculée. Ici c'est l'opérateur « # » qui joue ce rôle.

La récursion spatiale est importante car elle montre que contrairement à notre intuition, le calcul de la valeur d'un point d'un tissu ne peut pas toujours se dérouler en parallèle avec le calcul de la valeur des autres points. Ce phénomène se produit quand un tissu dépend *instantanément* de lui-même.

III.1.3. Trois façons d'écrire factorielle

Voici trois façons d'écrire factorielle en 81/2. La première manière utilise la récursion temporelle :

```
fact@0 = 1; i@0 = 0;
i = $i + 1 when CLOCK;
fact [1] = $fact * i ;
```

La valeur du i^{ème} top de fact est la factorielle de i. On peut aussi utiliser la récursion spatiale :

```
i [10] = 'i + 1;
fact [10] = (1 # (fact * i)) : [fact] ;
```

le tissu i est le tissu constant {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Les deux expressions de la factorielle que nous avons données utilisent en fait un index (soit énuméré dans le temps, soit dans l'espace). La troisième manière de définir factorielle est dans le style d'APL en utilisant un scan :

```
fact [10] = * \\ (+ \\ 1:[10])
```

Grâce à l'opérateur d'index on aurait aussi put écrire :

```
fact [10] = * \\ ('fact + 1)
```

L'expression « + \\ 1:[10] »permet de générer le tissu constant {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Ce tissu est ensuite réduit par multiplications successives. Du point de vue du parallélisme, cette dernière manière est la plus efficace : une beta-réduction peut s'exécuter en le logarithme du cardinal du tissu alors que la récursion temporelle ou spatiale sérialise le calcul de la factorielle.

II.2. Utilisation des collections

II.2.1. Représentation de tableaux

Monsieur Eden a proposé en 1958 un modèle de croissance des tumeurs. Pour simplifier, on supposera que le processus de croissance se déroule sur une grille. Au départ, on donne les cellules atteintes. On va utiliser un tissu de booléen, la valeur vraie indiquant une cellule atteinte, la valeur fausse une cellule saine. À chaque cycle, des sites sont choisis aléatoirement parmi les sites *frontières*. Un site est un site frontière s'il n'est pas occupé et s'il possède au moins un voisin occupé.

Il faut d'abord définir comment on va représenter la grille. Plusieurs solutions sont possibles. Nous allons en examiner une : on utilise ici l'imbrication des tissus pour représenter une grille 2D comme un vecteur de colonnes. On commence par définir alors deux fonctions auxiliaires, left() et right() qui permettent de décaler un vecteur vers la gauche et vers la droite :

```
left (x, c) = (x # c) ('x + 1) ;
right (x, c) = (c # x) : [x] ;
```

L'argument c correspond à la valeur qu'il faut donner sur un bord.

```
nord (x, c) = right ^ (x, c:[x]) ;
sud (x, c) = left  ^ (x, c:[x]) ;
est (x, c) = left (x, c:[x]) ;
ouest = right (x, c:[x]) ;
```

Ces diverses définitions permettent de décaler le vecteur (de colonnes) ou bien chaque colonne (du vecteur) afin de réaliser la translation correspondante vers le nord, le sud, l'est ou l'ouest.

Une fois qu'a été choisie une représentation pour la grille, il est simple de décrire la propagation d'une tumeur :

```
outside start [10000];        /* la tumeur croit sur une grille 100x100 */

T = start@0;
dT@0 = false; dT = $T;
Voisinage = nord (dT, false) || sud(dT, false) || est (dT, false) || ouest (dT, false) ;
ok = if (Voisinage && !dT) then random() else false fi ;
T =  if ok then true else dT fi
       when CLOCK ;
```

random() est une fonction qui renvoie vrai ou faux de manière aléatoire.

II.2.2. Étiquetage de composantes connexes

On se donne une image binaire I représentée par un tissu 2D (i.e. du type de ceux utilisés ci-dessus). On désire étiqueter les composantes connexes de cette image, c'est-à-dire calculer un tissu C de même géométrie que I et dont chaque point appartenant à la même composante connexe partage la même valeur. Cette valeur, l'étiquette, doit être différente pour chaque composante connexe et vaudra -1 en tout point où I == 0.

L'algorithme va consister à calculer une valeur différente pour chaque point et ensuite à propager cette valeur aux voisins qui appartiennent à la même composante connexe. Entre deux étiquettes, un point choisit l'étiquette de valeur maximale. On itère tant qu'on a pas atteint un état global stable.

```
outside I [256, 256] ;        /* une image binaire 256×256; la valeur d'un point est 0 ou 1 */
```

```
findex (i) = 'i + i * |i| ;
int index [I] = findex ^ ('I) ;
```

index est un tissu constant de valeur : { {0 1 2 … 255} {256 257 … } … }.

```
c@0 = if I then index else -1 fi;
c =   if I then max (c1, c2, c3, c4) else $c fi
      until stop ;

dc@0 = -1; dc = $c when CLOCK;
stop = && \ ($dc == dc);

c1 = max (dc, nord(dc, -1));
c1 = max (dc, sud(dc, -1));
c1 = max (dc, est(dc, -1));
c1 = max (dc, ouest(dc, -1));
```

Si on désire compter les composantes connexes, il suffit de rajouter les équations :

```
composantes [I] = if  (index == c after stop) then 1 else 0 ;
nombre_composante = + \ composantes;
```

Le tissu « composantes » permet de donner la valeur 1 à un seul point par composante connexe, les autres ayant pour valeur 0. La réduction de ce tissu permet de compter les composantes connexes.

Remarquons que cet algorithme permet de trouver un chemin entre l'entrée et la sortie d'un labyrinthe. Le labyrinthe est représenté par une carte. Un chemin est une composante connexe dans cette carte. Une composante connexe qui contient le point d'entrée et le point de sortie est un chemin solution. Cet algorithme montre comment par un changement de représentation judicieux, on peut transformer un algorithme qui est a priori dynamique en un algorithme statique (la recherche d'un chemin dans un labyrinthe est un exemple paradigmatique de l'exploration d'un arbre d'état dynamique).

II.2.3. La fonction reverse

Donnons ici une définition possible pour la fonction reverse : cette fonction montre que la notation explicite de l'alpha-notation est nécessaire si on veut éviter des ambiguïtés.

```
reverse (x)  =  x (|x| - 'x - 1) ;
```

par exemple :

```
reverse ({ 4 5 6 })
⇒ { 4 5 6 } ( {3 3 3} - { 0 1 2 } - { 1 1 1 } )
⇒ { 4 5 6 } ({ 2 1 0 })
⇒ { 6 5 4 }
```

On remarquera les géométries des constantes (i.e. |x| et 1) sont implicites. Cela permet de définir une fonction reverse qui est *polymorphe* en ce sens qu'elle peut s'appliquer à des tissus qui ont des géométries différentes. Si on applique la fonction reverse à une collection imbriquée :

```
reverse ({ {1 2} {3 4} {5 6} }) ⇒ { {5 6} {1 2} {3 4} }
reverse ^ ({ {1 2} {3 4} {5 6} }) ⇒ { {2 1} {4 3} {6 5} }
```

on remarque bien que l'application de cette fonction ne s'évalue pas en le même résultat que l'alpha-notation.

II.3. Utilisation de l'opérateur *every*

Supposons que l'on veuille calculer la suite des racines carrées des valeurs successives d'un tissu de cardinal 1. Les itérations successives de f (y) = (y + x^2/y) /2 en partant de x^2 convergent vers x. À partir de là il est facile d'écrire un programme qui calcule la racine carrée :

```
sqrt (x) =    r every x where {
      s@0 = x + 1;
      s = ($s + x/$s) / 2 when CLOCK;
      delta = (x - (s * s)) / x ;
      r = s at (delta < 0.01) ;
} ;
```

II.4. Transformation d'espace en temps et vice-versa

Il serait agréable de pouvoir enregistrer *n* valeurs consécutives d'un tissu S dans un tissu T de cardinal *n*. Les tops de T se produiraient alors tous les *n* tops de S. Ce comportement peut parfaitement s'écrire à l'aide des opérateurs existants. C'est une transformation du temps dans l'espace.

```
record (S) =
{
      t@0 = S@0 # (0:[n-1]);
      cpt@1 = S@0 ; cpt = if (cpt == n) then 1 else $cpt + 1 fi when (S == S) ;
      t [n] = t:[n-1] # S ;
      T = t when (cpt == n)
} . T ;
```

Ce programme s'explique facilement : la fonction record utilise un bloc pour cacher les tissus auxiliaires t et cpt. Le tissu cpt est un compteur qui compte de 1 à n de manière cyclique, au rythme des tops de S. Le tissu t enregistre les valeurs successives de S (au début, n-1 valeurs sont initialisées à 0). Le tissu T correspond au tissu t à l'instant où il y a n nouvelles valeurs de S.

Puisqu'on a une fonction record, on aimerait une fonction play qui fasse l'inverse : à partir d'un tissu T de *n* points, on dérive un tissu S de cardinal 1 prenant successivement la valeur des points de T. Il faut pour cela insérer des tops : la seule façon de faire est d'utiliser l'opérateur every. On ne peut donc pas exprimer l'opérateur play par une fonction : le tissu S doit être utilisé dans le corps d'un every. Voici comment on peut faire :

```
dummy  = stop every T where
{
      cpt@0 = 1; cpt = $cpt + 1 when CLOCK;
      t@0 = T; t = (0 # $t) : [n] when CLOCK ;
      S = t.n ;
      … utilisation de S…
      stop = true at (i == n);
}
```

(il est possible de se passer de l'opérateur every si on peut modifier la définition de T afin de la « geler » à l'aide d'un when, le temps de jouer les *n* valeurs)

II.5. Expressions équivalentes

Les opérateurs que nous avons présentés ne correspondent pas à un noyau minimal. Nous donnons ici quelques équivalences.

II.5.1. Expression de la quote

La quote n'est pas un opérateur primitif. On peut en effet définir que :

'X ≡ (+ \\ 1:[|X|]) - 1

i.e. on crée un tissu constant valant 1 et de même cardinal que X. Une opération de scan permet de générer un tissu de même géométrie dont la valeur des points va de 1 à |X|. On soustrait 1 pour avoir le résultat désiré.

II.5.2. Expression des filtres en termes de when

Il est possible d'exprimer les filtres *until*, *after* et *at* uniquement en terme de *when*. La définition se fait en utilisant un signal auxiliaire de valeur fausse après le passage à vraie de la condition de déclenchement.

```
outside T, B ;

cpt@0 = 0; cpt = $cpt + 1 when B ;

clock (x) = (x == x) ;
T_until_B = T   when   ( (clock(T) && (cpt < 1)) when clock(T)) ) ;
T_after_B = T   when   ( (clock(T) && (cpt >= 1)) when clock(T) ) ;
T_at_B = T when (cpt == 1) ;
```

Le tissu cpt permet de compter le nombre d'occurrences où B est passé à vrai. La fonction clock() permet de générer un tissu booléen dont les tops sont exactement ceux de son argument et dont la valeur est toujours vraie. À partir de là, il est facile d'écrire les expressions voulues. On remarquera que l'on n'introduit pas de tops d'horloge superflus pour until et after puisque la condition du when le plus externe a pour horloge celle de T. En effet,

- cpt a une valeur initiale à 0
- clock(T) n'est valué à vrai qu'après le démarrage de T,
- (cpt < 1) est valué à faux après le premier passage à vrai de B.
- (cpt >= 1) est valué à vrai avec le premier passage à vrai de B.

II.6. La traduction des itérations

Deux types de boucles existent dans les langages impératifs : la boucle *for* qui consiste à itérer sur un ensemble connu à l'avance, et la boucle *while* qui consiste à répéter une action en fonction d'un résultat calculé.

La boucle while correspond naturellement au concept de stream. Et l'imbrication des boucles while peut se traduire par l'utilisation de l'opérateur every. Notons cependant que l'on montre en théorie du calcul automatique, que tout programme contenant des *tant-que* imbriqués peut se réécrire en un programme ne contenant qu'un seul tant-que englobant tout le programme.

Pour la traduction des boucles *for*, on peut utiliser l'aspect stream d'un tissu. Mais cela ne permet pas de les paralléliser. Il vaut mieux les représenter par un tissu dont le cardinal correspond au nombre d'itérations de la boucle. S'il y a des dépendances entre les itérations, la définition de ce tissu sera récursive. Sinon, l'expression sous forme d'un tissu permettra une exécution parallèle. Donnons un exemple :

```
float T [10] ; for (i = 0; i <10; i++) T[i] = 2*i;     /* boucles en C */
float T [10] = 2 * 'T ;                                 /* tissu 81/2 correspondant */
```

cet exemple n'implique pas de dépendances entre itérations. L'exemple suivant

```
float U [10] ;
for (j = 0; j <10; j++) if (j == 0) U[i] = 1; else U[i] = U[i-1] + 2.7*i;
```

implique une récursion spatiale (pour simplifier on utilise un tissu auxiliaire u) :

```
float U [10] = (1 # u) : [U] ;          /* programme 81/2 correspondant */
u = U + 2,7 * 'U
```

La sélection offre le moyen de traiter des indiçages plus compliqués qu'un simple décalage :

```
float V [10] ; for (k = 0; k <10; k++) if (k%3) V[i] = k; else V[i] = U[(2k-1)%10];

int K [10] = ('k*2 - 1) % 10 ;                  /* programme 81/2 correspondant */
float V [10] = if ('K % 3) then 'K else U(K) fi ;
```

II.7. Les extensions

Nous envisageons dans cette section un certain nombre d'extensions qu'il serait possible d'ajouter avec plus ou moins d'effort au langage et qui peuvent avoir leur utilité.

II.7.1. Arithmétique

On peut bien sûr ajouter toutes les fonctions arithmétiques qui seraient nécessaires. L'alpha-notation est implicite, ce qui veut dire qu'un opérateur arithmétique définit de manière ambiguë toute une série d'opérations. Dans la version actuelle de l'interprète, il est par exemple possible de faire appel aux fonctions trigonométriques.

II.7.2. Une algèbre de blocs

Nous n'avons pas considéré d'autres manipulation que la création et l'accès à un élément d'un bloc. Cependant cette voie doit être explorée, par exemple dans la ligne de [Cardelli, Wegner 85]. Par exemple la possibilité d'utiliser l'opérateur de concaténation « # » entre blocs correspond à une forme d'héritage rudimentaire.

Il est aussi nécessaire de pouvoir moduler la visibilité des tissus définis dans un bloc, à la manière des clauses *public* et *private* d'une classe C++.

II.7.3. Extension des opérateurs temporels

Il arrive souvent que l'on ait besoin de l'horloge d'un tissu dans la commande d'un *when*. Il faut écrire à chaque fois « … when (X == X) ». Plutôt que de répéter cela, on a introduit dans l'interprète une nouvelle construction: *synchro*. Cet opérateur est défini par :

```
Y synchro X   ≡   Y when (X == X)
```

Nous avons donné plus haut la définition des opérateurs *play* et *record*. Si ces opérateurs sont très utilisés, il peut être intéressant de les incorporer au le langage.

L'opérateur every permet d'insérer des sous-calculs dont on ne connaît pas la durée (i.e. on ne connaît pas le nombre de tics qui seront évalués à l'intérieur de l'every avant qu'une valeur de retour ne soit élaborée). Il serait éventuellement agréable de posséder une forme de every correspondant à une exécution déterminée. C'est le cas par

exemple dans la définition de l'opérateur play. Cela revient à introduire une notion de boucle *for*. Mais a priori on peut s'en passer comme il a été montré plus haut.

II.7.4. Opérations de sélection

La sélection peut paraître lourde à manipuler quand on veut uniquement extraire un sous-ensemble contigu des points d'un tissu. On peut proposer des opérateurs basés sur FORTRAN 90 pour l'extraction de sous-tableaux. Une syntaxe est encore à proposer.

II.7.5. Structures localement dynamiques

La possibilité de prévoir à la compilation les ressources nécessaires à l'exécution est un des objectifs du langage. Par exemple il n'est pas possible de définir des tissus dont la géométrie varie au cours du temps car cela obligerait à faire de l'allocation dynamique répartie (Cf. chapitre IV).

Cependant, il existe des structures de données qui utilisent dynamiquement des ressources et qu'on sait gérer efficacement sur un seul processeur (par exemple une pile). Il est donc possible d'étendre le langage en enrichissant les *types scalaires* afin d'inclure ces types dynamiques.

Une donnée de type scalaire correspond en fait a une donnée dont la représentation sera localisée sur un seul processeur. La déclaration :

int stack S [100] ;

correspondra alors à une collection de 100 piles d'entiers. Chaque pile sera localisée sur un seul processeur. Le programme

int T [100] = … ;
boolean Q [100] = …;
S = if Q then *push* ($S, T) else *pop* ($S, T) fi;

créera un stream correspondant aux *push*s et aux *pop*s de ces piles sur chacun des processeurs, suivant les valeurs de Q.

II.7.6. Collections ad-hoc

La notion de collection utilisée dans 81/2 est proche de celle de vecteur : en particulier, l'index d'un point correspond à un entier. Il est possible de compliquer cette indexation grâce à l'imbrication des tissus. Mais on ne peut représenter de géométrie qui ne soit pas des vecteurs de vecteurs…

Cela implique en particulier que pour manipuler des tableaux, il faut passer par une *représentation* des tableaux en termes de vecteurs. Cette représentation est assez naturelle mais on peut par exemple choisir entre une représentation sous forme de « ligne de colonnes » ou bien sous forme de « colonne de lignes », ou même pourquoi pas, de « vecteur de diagonales ». Cette dernière représentation est par exemple judicieuse pour la gestion des matrices *n*-diagonales. Ce problème de représentation, qui n'est pas bien gênant pour les tableaux, devient plus important avec d'autres structures de données plus complexes. Par exemple, les simulations en mécanique des fluides sont parfois basées sur des automates cellulaires utilisant un pavage hexagonal de l'espace. Comment représenter ce pavage à l'aide d'un vecteur ?

Ces problèmes de représentation incitent à introduire d'autres notions de collections. En fait on peut considérer le langage 81/2 tel qu'il a été présenté, comme l'algèbre

streams(*vecteurs_complexes*(*scalaire*))

où *streams*(·) est un type paramétré (à la manière des packages génériques en ADA qui à partir de la définition d'une pile générique, permettent par instanciation de créer aussi bien des piles d'entiers que des piles d'ensembles). Rien n'empêche d'enrichir le langage afin de manipuler des collections de nature différente.

Dans ce cas, la spécification de la géométrie « · **[**·**]** » doit être vue comme un constructeur de type qui prend deux paramètres, un type *t* et un entier *n*, et qui renvoie un type « vecteur de *n* éléments de type *t* ». La transformation d'un type *t* en un type *streams*(*t*) est implicite. Par exemple,

int X **[** *5* **]**

est un stream (le constructeur est implicite) dont les éléments sont des vecteurs de 5 éléments de type entier. La prise en compte de collections de nature différente se fait alors simplement en introduisant de nouveaux constructeurs de collections. Par exemple,

int X **(***p*, *q***)**

pourrait définir un stream de tableaux (*p*, *q*) d'entiers et

int X **<***r*, *s***>**

pourrait définir un pavage hexagonal. Ces constructeurs peuvent bien sûr se combiner :

(*int* **<**2, 3**>**) X **[**10**]**

définit un objet X qui est un vecteur de 10 éléments qui sont eux-mêmes des pavages hexagonaux de 2 par 3[1]. Tout le reste du langage reste identique. Ainsi, on pourrait écrire des définitions comme

```
int A <2, 3> = 5 when CLOCK ;
(int <2, 3>) X [10] = A : [10] ;      /* ou plus simplement :  X [10] = A : [10] ; */
```

Une partie des opérations sur les collections restent valides (par exemples les 4 types d'applications de fonctions). Mais les opérations d'accès à un élément et les opérations proprement géométriques (par exemple les restructurations) sont propres à chaque type de collections.

II.7.7. Collections à géométrie variable

Nous avons insisté sur le caractère statique du langage mais puisque nous évoquons les extensions possibles, nous voudrions examiner la possibilité d'avoir des tissus dont la géométrie est dynamique. Donnons d'abord un exemple d'application où les géométries variables interviendraient naturellement.

Les systèmes de Lindenmayer [Prusinkiewicz 89] sont des systèmes de réécriture utilisés en biologie pour la modélisation des phénomènes de croissance. Par exemple, on peut modéliser le développement de filaments multi-cellulaires trouvés chez certaines bactéries et algues. On a deux symboles o et n qui décrivent l'état cytologique de la cellule, « être sur le point de se diviser ou non », et de deux autres symboles d et g indiquant la polarité de la cellule (orientation à droite ou à gauche). Une cellule peut se retrouver dans un des quatre états suivants

od, og, nd, ng

Chaque division cellulaire respecte les règles suivantes :

od → og nd
og → ng od
nd → od
ng → og

Des tissus à géométrie variable permettent de coder immédiatement ces règles de réécritures [Bonabeau, Leboucher 91] :

[1] On a utilisé une syntaxe à la C++ ou l'expression d'un type s'obtient en supprimant le nom de la variable dans sa déclaration.

```
od = 0; og = 1; nd = 2; ng = 3;

c@0 = cellule-axiome ;

c =   if croître then
            if gauche then { ng, od } else { og, nd } fi
      else
            if gauche then og else od fi
      fi when CLOCK ;
croître = ($c == od:[$c]) || ($c == og:[$c]) ;
gauche = ($c == ng:[$c]) || ($c == og:[$c]) ;
```

Tout repose sur l'opérateur :[] qui transforme une constante en un tissu de géométrie adéquate. Par exemple, en partant de od on obtient successivement :

od, {ng od}, {og {og nd}}, {{ng od} {{ng od} od}}, {{og {og nd}} {{og {og nd}} {og nd}}} …

et ainsi de suite.

D'autres exemples d'application viennent naturellement à l'esprit : par exemple les méthodes multi-grilles dans lesquelles le pas de discrétisation d'une équation aux différences finies est ajusté en fonction des résultats obtenus.

La collection est la structure qui permet de répartir les données sur un calculateur parallèle. Implicitement, les éléments d'une collection sont sur des processeurs différents. La possibilité d'avoir une géométrie dynamique implique donc la capacité de répartir dynamiquement de nouvelles données. Cela nécessite des techniques de *load-balancing* dynamiques qui sont coûteuses et difficiles à mettre en œuvre. Par ailleurs, il n'est plus possible de compiler le langage. En effet, une expression comme

A + B

lorsque la géométrie de A et de B est variable, implique une interprétation dynamique de l'opération « + » qui selon le cas sera une addition entre collections ou une addition entre collections de collections, etc. Pour les mêmes raisons, il n'est plus possible de typer statiquement le langage.

III. Une comparaison avec les langages existants

Nous allons comparer 81/2 avec des langages existant dans le domaine de la programmation data-flow, de la programmation temps-réel, de la programmation parallèle et de la programmation systolique.

III.1. La programmation data-flow : LUCID

Si on se restreint aux scalaires, 81/2 correspond à un sous-ensemble de LUCID [Ashcroft, Wadge 76] [Wadge, Ashcroft 85]. Le sous-ensemble choisi est celui qui reste compilable. En particulier :

- Il est possible en LUCID de faire référence à un élément ultérieur (opérateur *next*). Cette possibilité de LUCID pose des problèmes quand il s'agit d'ordonnancer les évaluations. En particulier, on ne peut plus ordonner les évaluations *a priori* suivant les tics croissants. Le seul mécanisme d'évaluation est celui par nécessité, les éléments d'un stream étant représentés par des données indexées par leur tic. L'algèbre des streams LUCID ne vérifie donc pas la propriété markovienne et ne modélise pas l'écoulement du temps. En terme de sémantique dénotationelle, [Caspi 91], cela se traduit par le fait que l'ordre considéré entre les suites LUSTRE ou les tissus 81/2 est l'ordre préfixe alors que l'ordre sur les suites LUCID est l'ordre de Scott (i.e. l'ordre usuel sur les fonctions de $\mathbb{N} \rightarrow D$ par exemple).

- La récursion est permise en LUCID. Il y a donc création dynamique de streams.
- L'opérateur *every* qui existe en 81/2 peut se simuler en LUCID à partir des opérateurs *whenever* et *upon* qui permettent de « geler » un stream et « extraire » des valeurs. Ces opérateurs, plus généraux, permettent des interactions complexes entre la progression des différents tissus, ce qui requiert encore une gestion dynamique (on doit par exemple conserver la valeur d'un tic donné pendant un temps inconnu). La structure de contrôle *every* étant une structure fondamentale, les concepteurs de LUCID ont proposé un nouvel opérateur : *is current* qui permet la construction d'*every* de manière plus immédiate. Mais cet opérateur souffre du même défaut : sa gestion doit être dynamique.

Plusieurs tentatives ont été faites pour introduire les tableaux en LUCID. Une notion qui a été proposée est celle de *ferd*. Un *ferd* est une suite (potentiellement infinie) de streams. Cela correspond donc à une notion de tableau dynamique (c.-à-d. dont le cardinal n'est pas fixé). Mais ce tableau n'est pas manipulé comme un tout, il est manipulé exactement comme un stream avec des primitives équivalentes. Ce n'est donc pas un support pertinent pour le parallélisme SIMD. Une autre extension qui a été proposée est celle d'un temps multi-dimensionnel, i.e. les éléments d'un stream ne sont plus indexés par un tic, mais par un uplet de tics (le nombre de composants étant éventuellement infini, l'indexation se faisant alors par un stream de streams d'entiers). Le problème qui se pose est toujours celui de la gestion statique de cette construction. Ces deux extensions ne sont, à notre connaissance, que des propositions.

Le traitement des tableaux pose un problème dans le modèle data-flow : des problèmes d'efficacité [Gajski, Padua, Kuck 82] (l'assignation unique oblige conceptuellement à copier tout le tableau chaque fois qu'on modifie la valeur d'un de ses éléments) et des problèmes de traitement mathématique (en particulier comment interagit la notion de stream et de tableau [Wadge, Ashcroft 85] : « We spent a great deal of effort trying to find a simple algebra of arrays (…) with little success »). C'est pourquoi nous avons introduit dans le modèle data-flow le concept de *collection*, chargé de représenter spécifiquement les ensembles homogènes et finis de données.

III.2. La programmation temps-réel : LUSTRE et SIGNAL

Le langage LUCID a inspiré deux langages dans le domaine de la programmation temps-réel : LUSTRE [Bergerand 86] [CPHP 87] et SIGNAL [LGBBG 86] [Le Guernic, Benveniste 87]. Ces langages ont été développés pour réaliser des programmes *réactifs* c'est-à-dire des programmes en interaction avec leur environnement, à un rythme imposé par cet environnement [Halbwachs 90]. Ce sont des programmes permettant d'exprimer et de vérifier des contraintes temporelles.

Les domaines d'application visés (la programmation parallèle) ont conduit 81/2 à introduire la notion de collection et à adopter une conception du temps différente de celle des ces langages. Nous allons détailler ces deux points.

III.2.1. Le concept de temps est différent

Voulant répondre aux problèmes de la programmation temps-réel, les langages synchrones ont une conception instantanée du temps : les événements n'ont pas de durée et peuvent se produire simultanément. Ainsi les processus réagissent instantanément aux événements et la valeur d'une variable est disponible uniquement sur les tops de son horloge. Par exemple, l'expression $a + b$ a un sens en SIGNAL seulement si a et b ont la même horloge.

Au contraire $a + b$ est toujours défini en 81/2. En effet, la valeur des paramètres est toujours connue. La seule différence entre un tic et un top c'est que sur un top, la valeur de la variable est susceptible de changer. Par suite, un stream 81/2 n'est pas le *current* d'une suite LUSTRE ou SIGNAL (*current* est un opérateur permettant d'observer une suite sur tous les tics de l'horloge de base d'un programme synchrone). Par exemple l'équivalent de l'expression 81/2 « $a + b$ » n'est ni « current($a + b$) » ni « current(a) + current(b) » mais « (current(a) + current(b)) when (horloge(a) ∨

horloge(*b*)) » ou *horloge* est un opérateur qui produit une suite booléenne sur l'horloge de base du programme, qui vaut vraie sur les top de son argument et faux sur les tics.

La conception du temps en 81/2 est proche de la conception initiale de LUCID. Une conséquence importante est que *le calcul d'une expression est complètement indépendant de son utilisation*. Donnons un exemple; le programme SIGNAL suivant [Le Guernic, Gautier 90] :

$c := a > 0 \mid x := a$ when c

est un programme SIGNAL correct. Mais si on ajoute l'équation

$y := x + a$

le programme devient un programme incorrect [Benveniste, LeGuernic 87]. En effet l'équation ajoutée vient perturber les définitions existantes en stipulant que x et a doivent avoir la même horloge. Cela n'est pas possible car l'horloge de x est des sous-horloge de celle de a. Remarquons que pour le programme équivalent en LUSTRE, c'est explicitement l'équation de y qui est déclarée eronnée. Par comparaison, le programme « équivalent » 81/2

$c = a > 0$; $x = a$ when c ; $y := x + a$;

est valide : en 81/2, les changements de valeur de y correspondent aux changements de valeur de x et de a. Comme les changements de la valeur de x représentent un sous ensemble des tops de a, les tops de y sont les mêmes que ceux de a.

Qu'est ce qui se cache derrière cette différence ? [Le Guernic, Benveniste 87] note que le choix de donner un sens à l'expression $x := y + z$ uniquement si y et z sont simultanément disponibles, est un choix arbitraire mais qui est naturel pour le temps-réel. Il correspond à l'hypothèse de *fort synchronisme* et permet de vérifier des contraintes temporelles de simultanéité. Nous faisons le choix de donner un sens à l'expression précédente dès que y et z ont une valeur, que celle-ci soit en train de changer ou pas. Ce choix est justifié par les facilités qu'il apporte dans la programmation parallèle. En particulier, la définition d'un tissu est toujours correcte (mais éventuellement son horloge est vide). C'est une propriété de *forte modularité* car elle permet d'utiliser arbitraitrement n'importe quel tissu dans une définition.

81/2 a repris la caractéristique des streams LUSTRE et SIGNAL qui consiste à interdire les références vers le futur. De même, il n'est pas non plus possible d'atteindre une valeur quelconque dans le passé car cela exigerait une mémoire de taille non-bornée. Les valeurs accessibles sont celles qui sont dans une fenêtre temporelle fixe ou qui ont une date donnée. Par suite les opérations à effectuer sont naturellement ordonnées par la succession des tics, ce qui n'est pas le cas en LUCID.

Remarquons enfin qu'un programme SIGNAL peut comporter de l'indéterminisme, ce qui n'est jamais le cas en 81/2. Il a existé un opérateur *every* en LUSTRE, qui n'existe plus et n'avait rien à voir avec l'opérateur baptisé *every* en 81/2. Une version *statique* de ce dernier (le nombre d'itérations effectuées par l'every est fixé avant son exécution) correspond à l'opération de multiplexage qui existe en SIGNAL.

III.2.2. Les tableaux en LUSTRE

Les tableaux ont été considérés dès la conception de LUSTRE mais leur implémentation n'a été étudiée que très récemment [Rocheteau, Halbwachs 91]. Un tableau est un ensemble indexé de variables synchrones. Il est possible de définir un par un les éléments d'un tableau grâce à l'opérateur d'accès à un élément « $T[0] = a + b$ ». Il est aussi possible de les définir plus globalement, par segment, à l'aide de la construction : « T [K1..K2] = R [L1..L2] + S [K1..K2] » où K1, K2, L1, L2… doivent être des constantes. Les tableaux doivent être homogènes (i.e. chaque élément doit avoir le même type). L'alpha-notation est implicite pour les opérateurs du langage. Il n'y a pas de réduction ni de sélection.

Les domaines d'utilisation des tableaux sont très différents. Les tableaux ont été introduits en LUSTRE afin de

programmer des circuits logiques reprogrammables, les Programmable Active Memories; la notion 81/2 de collection est plus riche pour répondre aux besoins de la programmation parallèle. Les architectures cibles et les contraintes imposées par les applications sont donc différentes mais il serait cependant très intéressant de comparer les techniques de compilation mise en œuvre et en particulier, les stratégies de placement.

Eric Payan [Mazare, Payan 90] [Payan 91] a effectué un travail similaire pour la programmation d'un réseau de cellules asynchrones [CEMO 87]. L'implémentation a consisté à transformer le tableau en autant de variables que d'éléments. Cette approche n'est plus envisageable quand le nombre d'éléments est très grand (cas des applications 81/2).

III.3. La programmation parallèle : OCCAM et *LISP

Nous allons comparer 81/2 à deux modèles d'exécution, celui de *LISP (qui correspond à un modèle SIMD) et celui de OCCAM (qui correspond à un modèle MIMD).

III.3.1. La programmation MIMD

OCCAM est un langage pour la programmation des architectures MIMD (et plus particulièrement du TRANSPUTER). La communication en OCCAM est bloquante : l'émetteur est en attente de l'acceptation par le récepteur. Le programmeur est naturellement conduit à partir de là, à adopter un style de programmation où le cadencement des calculs est imposé par la disponibilité des données. OCCAM est pourtant loin d'un modèle data-flow car la spécification des processus est explicite et à la charge du programmeur. Une autre différence importante est l'existence d'un support explicite pour l'exploitation d'un parallélisme de type SIMD. Les tableaux OCCAM ne sont pas des collections : il n'est pas possible de les manipuler comme un tout. Il faut remarquer que les tissus 81/2 sont des *entités réparties* (Cf. le chapitre IX), alors qu'un processus OCCAM est une entité localisée explicitement sur un PE.

III.3.2. La programmation SIMD

La forme data-flow d'un programme 81/2 rend possible l'exploitation, si la machine cible le permet, de plus d'une source de parallélisme SIMD à la fois. Par exemple le calcul de *A* et de *B* dans

$A = T * T$; $B = T + T$;

peut se dérouler en parallèle, car *A* et *B* sont indépendants. On répond ainsi à la demande exprimée dans [Steele 90] qui plaide pour un style de programmation combinant les avantages des modèles synchrones (SIMD) et asynchrones (MIMD).

Un des problèmes importants du SIMD est mis en évidence par le traitement des conditionnelles : l'obligation d'organiser les calculs de manière à ce que chaque processeur exécute (ou n'exécute pas) la même instruction, conduit à sérialiser l'exécution des branches d'une conditionnelle (les PE ayant évalué la condition à vrai étant inhibée pendant le déroulement de la branche *else*, et vice-versa). Le modèle de programmation MIMD permet d'éviter cela grâce à de multiples *threads* de contrôle. En contrepartie, il se pose des problèmes d'indéterminisme et de synchronisation. 81/2 est un modèle de programmation déterministe et synchrone, permettant l'exécution de multiples *threads* de contrôle (comme pour *A* et *B* dans l'exemple plus haut). En particulier, nous verrons que le temps d'exécution d'une conditionnelle correspond au maximum du temps d'exécution des deux branches, et non pas à la somme comme dans *LISP par exemple.

On se référera au tableau 2 du chapitre I (§ III.3.9.) pour une comparaison entre la notion de collection en 81/2 et celle qui existe dans d'autres langages.

III.4. La programmation systolique : CRYSTAL et ALPHA

CRYSTAL [Chen 86] et ALPHA [Mauras 89a] [Mauras 89b] sont des langages de programmation basés sur la notion de variables multi-dimensionnelles et d'équations récurrentes. Une variable ALPHA est un ensemble de points indexés dans ZZ^n, n quelconque. Le domaine d'indexation doit être un polyèdre convexe (il doit être décrit par un ensemble fini d'inéquations linéaires). Le temps correspond à un index qui prend sa valeur dans ZZ (il peut a priori y avoir plusieurs tels index). De plus, les opérations de manipulation sont homogènes et ne dépendent pas de l'axe auquel elles s'appliquent. Par exemple, il n'existe pas d'opérateur de délai, celui-ci étant réalisé par un *right* qui s'applique sur l'axe temps.

Cette approche est très élégante car complètement homogène : la dimension temporelle n'est pas distinguée des dimensions spatiales. C'est volontairement que cette distinction est soigneusement faite en 81/2 car cela permet, à travers la notion de collection, l'exploitation d'un parallélisme de type SIMD.

L'expression simple de ce type de parallélisme conduit [Mauras 89b] à suggérer l'introduction d'opérateurs de *généralisation* correspondant en fait à des opérations de réduction. Il est par ailleurs possible d'analyser les équations récurrentes linéaires pour en extraire le parallélisme (une équation est linéaire si les index au membre droit sont des combinaisons linéaires de l'index de la variable définie en partie gauche). C'est le cas en ALPHA (CRYSTAL adopte des équations d'une forme plus générale). Mais cette analyse est difficile et demande des techniques toutes nouvelles (voir par exemple [Feautrier 91]).

Les opérateurs *when*, *until*, *after*, *at* et *every* n'ont pas d'équivalents dans ce type de langage : ils permettraient en effet de générer des domaines non polyédriques d'index. Cette restriction correspond au domaine de la programmation systolique où on veut « câbler » l'algorithme obtenu : il faut donc un langage complètement statique et des données de structures très régulières. Mais ces restrictions font aussi toute la force de ces langages, car elles permettent des techniques optimales d'ordonnancement des calculs.

III. Sémantique dynamique

Le langage 81/2 a été présenté informellement dans le chapitre précédent. Il nous faut maintenant fonder rigoureusement notre modèle pour pouvoir donner un sens à des expressions telles que $T = \$T + 1$. Pour cela, nous allons associer à chaque programme 81/2 un objet mathématique qui représentera « le calcul » effectué. L'approche adoptée utilise les techniques usuelles de la sémantique dénotationnelle [Scott 91]. La section III donne la motivation de certains choix qui ont été fait dans le *design* du langage. Les buts poursuivis ont été de préserver autant que faire se peut la *localité* et le caractère *statique* des calculs à effectuer.

Nous supposerons dans ce chapitre que les programmes 81/2 sont corrects : c'est-à-dire que tout identificateur est défini une seule fois, que les expressions sont correctement typées, etc. Définir quels sont les programmes corrects est le but de la sémantique statique qui est traitée dans les trois chapitres suivants.

I. La modélisation d'un tissu

Deux points de vue sont possibles pour représenter un tissu :

– *un tissu est une suite de collections*, ou bien

– *un tissu est une collection de suites*,

suivant que l'on privilégie l'aspect temporel ou spatial d'un tissu. Une collection ne correspond pas à une suite : les suites représentent un historique des valeurs et sont donc des suites infinies; les collections peuvent être hiérarchisées, ce qui n'est pas le cas pour les suites. De plus, on n'utilise pas les mêmes opérations sur ces deux types d'objets (par exemple, on n'utilise pas d'opérateur de beta-réduction sur une suite temporelle).

Nous choisissons ici de privilégier l'aspect « suite de collections » correspondant à une vue macroscopique du langage : 81/2 apparaît comme un langage séquentiel permettant de manipuler des collections comme un tout. Nous aurons cependant besoin des opérateurs ↑ et ↓ qui font passer d'un stream de collections à une collection de streams et vice-versa. Le point de vue microscopique est plus proche de l'implémenteur : il correspond au traitement en parallèle de différents flots de données (qui sont corrélées). Luc Bougé étudie dans [Bougé 90] l'équivalence des deux points de vue, dans le cadre de la sémantique opérationnelle d'un langage impératif SIMD (correspondant à POMP-

C).

I.1. La représentation de l'horloge d'un tissu

Le premier problème que nous rencontrons quand il s'agit de représenter un tissu, est celui qui tient à la modélisation de l'aspect temporel. Il faut décider comment représenter le fait que les tissus « n'évoluent pas à la même vitesse ». En effet, la présence de l'opérateur *when* permet de définir des tissus qui évoluent moins vite que d'autres. La suite des valeurs correspondant à une définition ne suffit donc pas à caractériser celle-ci : pour pouvoir comparer des tissus entre eux, il faut disposer d'une datation de chaque élément d'une suite. Or il est important de pouvoir comparer des tissus entre eux car, par exemple, les valeurs des tissus *T* et *Q* peuvent être produites sur des processeurs différents avant d'être consommées sur le processeur chargé de calculer *T+Q*.

Pour pouvoir comparer des tissus entre eux, nous avons introduit dans le chapitre précédent une distinction entre tic et top et nous avons supposé qu'il existe une horloge de base permettant de dater tout événement. Cette horloge mesure le passage du temps dans le programme. La datation la plus simple est de considérer que le $n^{ième}$ élément d'une suite se produit au $n^{ième}$ instant. Cela implique d'enregistrer une valeur à chaque instant, que celui-ci soit un tic ou un top. La modélisation mathématique d'un tissu va donc impliquer deux suites : la suite des valeurs du tissu (y compris les tics) et une suite de booléens. En effet, pour ne pas perdre d'information, il faut pouvoir distinguer les tics et les tops. C'est à quoi servira cette suite de booléens, la valeur vraie indiquant un top, la valeur fausse un tic (Cf. figure 1). Cette dernière suite sera appelée l'horloge du tissu.

horloge de base	0	1	2	3	4	5	6	7	8
T = <0, 1, 2, 3, 4>	0		1		2		3		4
horloge (T)	true	false	true	false	true	false	true	false	true
valeur (T)	0	0	1	1	2	2	3	3	4
Q = <0, 1, 2, 3, 4>	0	1	2	3	4				
horloge (Q)	true	true	true	true	false	false	false	false	
valeur (Q)	0	1	2	3	4	4	4	4	4

Table 1 : Les deux tissus *T* et *Q* correspondent aux mêmes suites de valeurs mais n'ont pas la même horloge. Un tissu est donc caractérisé par deux suites, la suite des valeurs et son horloge. Une horloge est une suite booléenne qui indique à quel moment d'une horloge universelle, un tissu change de valeur. La suite des valeurs est la suite des observations, sur l'horloge universelle, des valeurs d'un tissu.

I.2. La représentation de la géométrie d'un tissu

Il est possible en 81/2 d'écrire des équations qui définissent un tissu dont la géométrie varie dans le temps, par exemple :

T@0 = 0;
T = {$*T*} when *Clock*;

Intuitivement ce programme pourrait définir un tissu dont le premier élément est 0, le second {0}, le troisième {{0}}, etc. De telles expressions sont rejetées par la sémantique statique du langage (Cf. le prochain chapitre) car elles ne répondent pas à l'objectif d'être statiques. Cependant la possibilité de définir des tissus à géométrie variable présente un intérêt certain. Ces tissus pourraient par exemple offrir une solution simple à l'expression des méthodes multi-grilles, ou permettre une représentation directe des systèmes de Lindenmayer (Cf. les exemples donnés dans le chapitre précédent).

Cette extension demanderait à pouvoir associer, à chaque top d'un tissu, une représentation de sa structure. Cette représentation peut par exemple être une surjection σ de *COLLECTION* dans *D*, où *D* est un domaine d'une des deux formes : $SCALAIRE^n$ ou D^m, avec n et m des entiers quelconques. Nous n'irons cependant pas plus loin dans cette direction.

I.3. Les tissus

Un tissu sera donc représenté par deux suites infinies, celle de ses valeurs et celle de son horloge.

I.3.1. Collections, streams et tissus de D

À partir d'un ensemble quelconque *D*, on note par D^c les collections des éléments de *D*. Les suites infinies des éléments d'un ensemble *D* se notent D^∞ . Les opérateurs *C* et ∞ se définissent par

$$D^c \cong \{\perp_{D^c}\} + D + (D^c \times D^c)$$
$$D^\infty \cong D \times D^\infty$$

Un tissu est une suite infinie de collections de scalaires. On a donc :

$$COLLECTION = SCALAIRE^c$$
$$TISSU = COLLECTION^\infty$$

SCALAIRE représente l'ensemble des valeurs des types de base. Pour fixer les idées, on peut prendre les entiers relatifs avec la convention que *0* est interprété comme la valeur booléenne fausse et tous les autres entiers comme la valeur vraie dans les opérateurs relationnels. On ajoute un élément distinct de tous les autres, $\perp$, pour obtenir un domaine. $\perp$ modélise l'élément indéfini. Les équations précédentes sont des équations entre domaines qui ont effectivement une solution (on pourra par exemple se reporter à [Scott 91]).

Les ensembles *COLLECTION* et *TISSU* sont des domaines; en particulier ils sont munis d'une relation d'ordre partiel et ils ont un plus petit élément. On note $\perp_D$ le plus petit élément du domaine *D* (on omettra l'indice quand il n'y a pas d'ambiguïté possible). Les éléments de *SCALAIRE* sont tous incomparables entre eux sauf avec $\perp$ qui est inférieur à tous : *SCALAIRE* est un « domaine plat » (*flat domain*). L'ordre sur *COLLECTION* est celui attaché à $\times$ et $+$: dans $A+B$ les éléments de *A* ne sont pas comparables à ceux de *B* mais $\perp_{A+B}$ est inférieur à tout élément. Dans $A \times B$ $\langle a, b\rangle \leq \langle c, d\rangle$ si $a \leq c$ et $b \leq d$. Par suite, $\perp_{COLLECTION}$ est différent par exemple de $\langle \perp_{SCALAIRE} \rangle$ qui est la collection à un élément scalaire, de valeur indéterminée. Le plus petit élément de $TISSU^\infty$ est une suite infinie de $\perp_{COLLECTION}$.

I.3.2. Les constructeurs et les observateurs sur les suites et les collections

On va définir un certain nombre d'opérations sur les suites et les collections :

$< c_1\ c_2\ c_3 >$ = la suite infinie (un stream) formée des éléments $c_1\ c_2\ c_3 \perp \perp \perp \ldots$

$\langle a, b \rangle$ = la collection formée des éléments *a b*

$c ; s$ = la suite de premier élément *c* se poursuivant par la suite *s*

$c_1 ++ c_2$ = la collection formée par la concaténation de c_1 et de c_2

$s . i$ ou $c . i$ = le *i*ème élément de la suite *s*, ou le i*ème* élément de la collection *c*, le premier élément étant indexé par 0. Cet opérateur est surchargé et se note de la même manière pour les suites ou pour les collections.

$|c|$ = le nombre d'élément de la collection *c*

On notera que les opérateurs « ++ » et « ; » ne sont pas des opérateurs stricts, par exemple $c;\perp \neq \perp$. Nous allons aussi utiliser la fonction continue suivante qui rend une collection ou une suite privée de son premier élément :

$$rest : X \rightarrow X, \qquad rest(\perp) = \perp, \text{ et } rest(c\,;s) = s \quad \text{si } X \text{ est une suite}$$
$$rest(\perp) = \perp, \text{ et } rest(\langle\langle a, b \ldots \rangle\rangle) = \langle b \ldots \rangle \quad \text{si } X \text{ est une collection.}$$

I.3.3. Alpha-notation

Quel que soit le domaine D, il est possible d'associer à chaque fonction f de $D \times D$ dans D, une fonction $\alpha_D f$ de $D^n \times D^n$ dans D^n définie par induction sur n :

$$\alpha_D f(c, d) = \begin{cases} f(c, d) \text{si } c, d \quad D \\ \bullet\ f(c.0, d.0)® \ ++\ \alpha_D f(rest(c), rest(d)) \text{sinon} \end{cases}$$

I.3.4. Beta-réduction

De manière analogue, nous définissons la beta-réduction comme l'association à une fonction f de $D \times D$ dans D de la fonction $\beta_{D,n} f$ de D^n dans D définie par :

$$\beta_{D,n} f(\langle\langle a_1 \ldots a_n \rangle\rangle) = \langle f(a_1, \ldots f(a_{n-1}, a_n)) \rangle$$

I.3.5. Sélection

Enfin nous définissons la sélection s d'une collection c de D par une autre collection d de D comme :

$$s = c(d) \quad \Leftrightarrow \quad s.i = c.(d.i) \text{ pour } 0 \leq i < |d|$$

s est défini même si un élément de d est un index invalide (i.e. s'il existe un i tel que $d.i < 0$ ou $d.i \geq |c|$). Dans ce cas, la valeur de l'élément est la valeur indéfinie $\perp_{COLLECTION}$.

II. Sémantique dénotationnelle d'un programme 8½

Il est possible de considérer une expression 8½ de manière « globale » ou de manière « locale ». Le point de vue global regarde les équations 8½ comme des équations entre suites considérées comme un tout. Le point de vue local correspond à une vision « data-flow » et fait correspondre un réseau à une expression 8½. Ce réseau reçoit des valeurs en entrées et produit éventuellement des valeurs sur ses sorties. Le point de vue est local car il n'y a pas de notion de suite, même si c'est bien une suite de valeurs qui est présentée en entrée et si l'observation des sorties résulte en des suites de valeurs observées.

Le point de vue local est certainement plus « opérationnel » car la manipulation de suites infinies doit certainement passer par une telle implémentation. [Kahn 73], [Kahn, MacQueen 77] associent une *fonction* à chaque nœud du réseau (un nœud correspond à un opérateur 8½). Cette fonction transforme une valeur en entrée en une valeur en sortie. Une autre approche possible est celle proposée par [Milner 73] [Milne, Milner 79], qui associent à un nœud un *comportement*. Un comportement est une fonction qui à une valeur en entrée associe une valeur en sortie *et* un comportement. Ce nouveau comportement sera utilisé pour traiter la prochaine valeur en entrée.

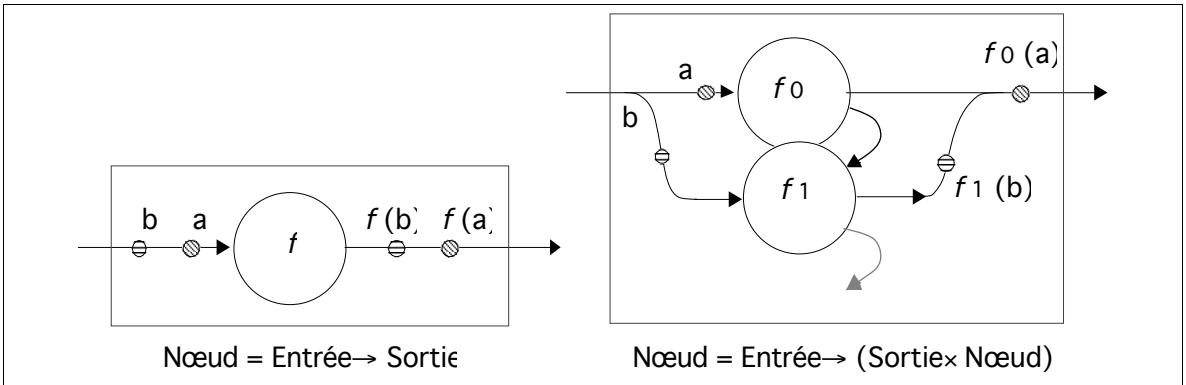


Figure 1 : Une équation 81/2 peut être vue comme un réseau data-flow. À chaque nœud on peut associer une fonction qui transforme une valeur en entrée : c'est le style de [Kahn 73]. Afin de prendre en compte, de manière purement fonctionelle, les nœuds dont le comportement varie en fonction des entrées reçues, on associe à un nœud un comportement : c'est le style adopté dans [Milner 73].

Nous adoptons ici le point de vue global : on va associer à chaque expression 81/2 une suite et on manipulera des suites dans leur totalité. On remarquera cependant que les fonctions que nous utiliserons sur ces suites sont définies par induction à partir du constructeur « ; », i.e. elles prennent la forme suivante : $F(a ; s) = g(a) ; G(a, s)$, où pour tout a, $\lambda s.G(a, s)$ est aussi une fonction inductive. Cela veut dire que la sémantique obtenue est très comparable à celles basées sur la dénotation d'un comportement, le nouveau comportement étant représenté par $\lambda s.G(a, s)$.

II.1. Sémantique des expressions

On va s'intéresser à un sous-ensemble du langage 81/2, le langage L1. Certaines facilités syntaxiques de 81/2 sont absentes de L1. Par exemple, tout tissu constant doit se noter sous la forme `v:[n]` ; la géométrie ne peut être omise (dans le langage 81/2 la plupart des qualifications de géométrie peuvent être omises car la sémantique statique permet de desambiguïser la notation). Une autre facilité absente de L1 est la nomination des tissus d'un bloc, ce qui permet d'y accéder à travers un nom symbolique plutôt qu'à travers un nombre. Il est trivial de transformer un programme 81/2 qui fait appel à cette facilité pour n'utiliser que l'indexation numérique. Enfin les définitions quantifiées, qui utilisent une syntaxe impliquant plusieurs équations syntaxiquement indépendantes, sont remplacé par l'opérateur fby. Par exemple « T@0 = 0; T@2 = 2; T = $T +2; » se transforme en « T = 0 fby (($T + 2) fby (2 fby ($T + 2))) »

Afin de réduire les constructions étudiées au noyau minimal, L1 ne contient pas les opérateurs « synchro, clock, at, until, after », dont des définitions en termes de « when » ont été données précédemment. De plus, nous ne définirons qu'une version unaire de l'opérateur d'agrégation. En effet, l'agrégation n-aire peut se définir par : {a, b, c, … } = {a} # {b} # {c} # …

Par ailleurs, les expressions ne font intervenir que des identificateurs (et non d'autres sous-expressions) et on suppose résolu le problème de leur portée : chaque identificateur est unique et distinct des autres. Enfin, afin de simplifier l'exposé, nous ne traiterons que les fonctions d'une variable (la technique utilisée se généralisant aisément) et nous ne donnerons pas la sémantique du scan (qui en essence est la même que celle de la réduction).

II.1.1. Domaine des expressions

Nous définissons les constructions syntaxiques abstraites du langage L1. Il nous faut d'abord un certain nombre d'ensembles de base:

ID	domaine des identificateurs de tissu (i, j, k ∈ ID)
FID	domaine des identificateurs de fonction (f ∈ FID, ID∩FID = Ø)
BINOP	domaine des opérateurs scalaires binaires du langage (binop ∈ BINOP)
EXP	domaine syntaxique des expressions de L1 (e ∈ EXP) :

EXP ::=	n:[m]	les constantes entières
|	i	les identificateurs
|	i = e	définition de tissu
|	f (i, j, k…) = e	définition de fonction
|	i . n	les observateurs
|	i @ n	
|	i binop j	l'arithmétique
|	if i then j else k	conditionnelle
|	f (i, j, k…)	application
|	f ^ (i, j, k…)	alpha-notation
|	f \ i	réduction
|	i (j)	sélection
|	i fby j	suivi par
|	$ i	délai
|	i when j	trigger
|	{i}	imbrication spatiale
|	i # j	agrégation
|	i every j where e	imbrication temporelle

II.1.2. Sémantique des expressions

La sémantique d'une expression est donnée par deux fonctions : *Val* **[[** *e* **]]** et *Hor* **[[** *e* **]]** qui associent à une expression *e* respectivement la suite de ses valeurs et la suite d'horloge.

La valeur et l'horloge d'un tissu dépendent de la valeur et de l'horloge des tissus référencés par les identificateurs apparaissant dans la définition. Nous introduisons donc un environnement qui associe à un identificateur la valeur et l'horloge du tissu dénoté. Plus précisément, afin de ne pas multiplier les domaines, nous associons à un identificateur (de tissu ou de fonction), un triplet correspondant à ⟨ *valeur*, *horloge*, ⊥⟩ si l'identificateur dénote un tissu et ⟨ ⊥, ⊥, *fonction*⟩ si l'identificateur dénote une fonction. La notion de fonction 81/2 est rendue par une fonction qui prend une valeur, une horloge (l'argument de la fonction 81/2) et un environnement pour rendre une valeur et une horloge (la valeur calculée). L'association variable ↔ ⟨ *valeur*, *horloge*, *fonction*⟩ est rendue par une fonction de ENV = ID → VAL × HOR × FCT. La fonction *Env* **[[** *e* **]]** transforme un environnement et l'enrichit par les liaisons variable-valeur qui apparaissent dans l'expression *e*. En résumé,

VAL =	*TISSU*
HOR =	*TISSU* (et plus précisément $BOOLEEN^{c^\infty}$)
FCT =	VAL × HOR → ENV → VAL × HOR
ENV =	ID → VAL × HOR × FCT
Val :	EXP → ENV → *TISSU*
Hor :	EXP → ENV → *TISSU*
Env :	EXP → ENV → ENV

Les fonctions *Val, Hor* et *Env* sont définies par cas sur la structure d'une expression. Dans ce qui suit, afin d'alléger les notations, nous adoptons les conventions suivantes :

- Les **[[** … **]]** permettent de « quoter » les éléments de EXP.
- *env* est un élément de ENV et par convention, *env*(id) = ⟨val(id), hor(id), fct(id)⟩.
- Les variables s, t désignent des tissus et h,l des tissus booléens qui représenteront des horloges.
- La variable c représente une collection.
- La variable v désigne un élément de VAL × HOR × FCT
- i, j, id sont des identificateurs de variables (des éléments de ID) et f un identificateur de fonction.
- La lettre ρ dénote un environnement.
- Nous omettons les injections nécessaires entre certains domaines syntaxiques (par exemple BINOP) et les valeurs sémantiques associées (par exemple l'opérateur scalaire correspondant).

Référence à une variable

Env **[[** id **]]** $env = env$

Val **[[** id **]]** env = val (id)

Hor **[[** id **]]** env = hor (id)

Rappelons que $env = \langle val, hor, fct \rangle$.

Liaison variable-valeur

Env **[[** id = e **]]** $env = env$**[** $\langle Val$ **[[** id=e **]]** env, Hor **[[** id=e **]]** $env, \perp_{FCT}\rangle$ **/** id **]**

Val **[[** id = e **]]** $env = Val$**[[** e **]]** FIX_{ENV} (λρ:ENV. env**[** $\langle Val$ **[[** e **]]** ρ, Hor**[[** e **]]** ρ, $\perp_{FCT}\rangle$ **/** id **]**)

Hor **[[** id = e **]]** $env = Hor$**[[** e **]]** FIX_{ENV} (λρ:ENV. env**[** $\langle Val$ **[[** e **]]** ρ, Hor **[[** e **]]** ρ, $\perp_{FCT}\rangle$ **/** id **]**)

La fonction f **[**v **/** id**]** est la fonction qui appliquée à x vaut v si x vaut id et $f(x)$ sinon. La fonction $\mathrm{FIX}_{ENV}(f)$ retourne le plus petit point fixe de la fonction f de ENV → ENV, i.e. le plus petit environnement v tel que $v = f(v)$.

Pour voir pourquoi cette définition est correcte, prenons d'abord l'exemple d'une expression **[[** id = e **]]** dans laquelle l'identificateur id n'apparaît ni dans e, ni dans *env*. Alors Val**[[** e **]]** env[v/id] a même valeur que Val**[[** e **]]** env pour n'importe qu'elle valeur de v. (En effet, id n'apparaissant pas dans e *et* pas dans *env*, la valeur dénotée par id ne joue pas de rôle dans l'élaboration de la valeur de e; pour s'en convaincre, il suffit de considérer les équations de définition de *Val*.) Et donc, FIX_{ENV} (λρ:ENV. env**[** $\langle Val$ **[[** e **]]** ρ, Hor **[[** e **]]** ρ, $\perp_{FCT}\rangle$ **/** id **]**) a pour valeur env**[** $\langle Val$ **[[** e **]]** env, Hor **[[** e **]]** $env, \perp_{FCT}\rangle$ **/** id **]**. En effet,

(λρ:ENV. env**[** $\langle Val$ **[[** e **]]** ρ, Hor **[[** e **]]** ρ, $\perp_{FCT}\rangle$ **/** id **]**)
(env**[** $\langle Val$ **[[** e **]]** env, Hor **[[** e **]]** $env, \perp_{FCT}\rangle$ **/** id **]**)

= env**[** $\langle Val$ **[[** e **]]** env**[** $\langle Val$ **[[** e **]]** env, Hor **[[** e **]]** $env, \perp_{FCT}\rangle$ **/** id **]**,
Hor **[[** e **]]** env**[** $\langle Val$ **[[** e **]]** env, Hor **[[** e **]]** $env, \perp_{FCT}\rangle$ **/** id **]**,
$\perp_{FCT}\rangle$ **/** id **]**

= env**[** $\langle Val$ **[[** e **]]** env, Hor **[[** e **]]** $env, \perp_{FCT}\rangle$ **/** id **]**

ce qui montre que cette dernière valeur est le point fixe que nous cherchons. Par suite,
Val **[[** id = e **]]** $env = Val$ **[[** e **]]** env[Val **[[** e **]]** env / id] et en réappliquant la même remarque,

$$Val\ [\ [\ \text{id} = \text{e}\ ]\ ]\ env\ =\ Val\ [\ [\ \text{e}\ ]\ ]\ env$$

ce qui est bien ce à quoi on s'attendait : quand on introduit la définition d'un tissu qui n'interfère avec aucun autre, on augmente l'environnement et la valeur du tissu est celle de son expression dans l'ancien environnement.

La présence du point fixe est rendue nécessaire par le fait que l'identificateur en partie gauche peut apparaître dans l'expression en partie droite. Prenons comme exemple le calcul de la valeur de l'expression [[T = T]] dans l'environnement ε qui associe $\langle \perp, \perp, \perp \rangle$ à tout identificateur (l'environnement « vide ») :

$$Val\ [\ [\ \text{T} = \text{T}\ ]\ ]\ \varepsilon = Val\ [\ [\ \text{T}\ ]\ ]\ \text{FIX}_{\text{ENV}}\ (\lambda\rho{:}\text{ENV}.\ \varepsilon[\ \langle\ Val\ [\ [\ \text{T}\ ]\ ]\ \rho,\ Hor\ [\ [\ \text{T}\ ]\ ]\ \rho, \perp_{\text{FCT}} \rangle\ /\ \text{T}])$$

$$= (\ \text{FIX}_{\text{ENV}}\ (\lambda\rho{:}\text{ENV}.\ \varepsilon[\ \langle\ Val\ [\ [\ \text{T}\ ]\ ]\ \rho,\ Hor\ [\ [\ \text{T}\ ]\ ]\ \rho, \perp_{\text{FCT}} \rangle\ /\ \text{T}])\ (\text{T})\ ).0$$

en utilisant $Val\ [\ [\ \text{T}\ ]\ ]\ env = \text{val}(\text{T})$. Or, ε est le point fixe cherché car :

$$((\lambda\rho{:}\text{ENV}.\ \varepsilon[\ \langle\ Val\ [\ [\ \text{T}\ ]\ ]\ \rho,\ Hor\ [\ [\ \text{T}\ ]\ ]\ \rho, \perp_{\text{FCT}} \rangle\ /\ \text{T}])\ \varepsilon)\ (\text{Q}) = \varepsilon[\langle \perp, \perp, \perp \rangle / \text{T}])\ (\text{Q}) = \varepsilon(\text{Q})$$

et cela pour tout Q. ε est donc l'environnement cherché et par suite, la valeur de T dans l'expression T = T est la suite $<\perp_{\text{COLLECTION}}, \perp_{\text{COLLECTION}}, \perp_{\text{COLLECTION}}, \ldots >$.

Définition et appel d'une fonction

$$Env\ [\ [\ \text{f}\ (\text{i}) = \text{e}\ ]\ ]\ env = env\ [\ \langle\ \perp, \perp, f \rangle\ /\ \text{f}\ ]$$

avec $f = \lambda\text{v}{:}\text{VAL}\times\text{HOR}\times\text{FCT}, \rho{:}\text{ENV}.\ \langle\ Val[\ [\ \text{e}\ ]\ ]\ \rho[\text{v}/\text{i}], Hor[\ [\ \text{e}\ ]\ ]\ \rho[\text{v}/\text{i}] \rangle$

$$Val\ [\ [\ \text{f}\ (\text{i}) = \text{e}\ ]\ ]\ env = \perp$$

$$Hor\ [\ [\ \text{f}\ (\text{i}) = \text{e}\ ]\ ]\ env = \perp$$

Nous ne décrivons que la sémantique des fonctions à une variable afin de ne pas obscurcir l'exposé. Rappelons que les identificateurs de fonctions sont distincts des variables de tissus. Nous supposons que la définition d'une fonction est correcte. En particulier, les appels récursifs ne sont pas permis, pas plus que les définitions imbriquées (cela veut dire entre autre qu'on ne peut avoir de liaison variable-valeur dans le corps d'une fonction). Une définition de fonction est une expression qui a pour valeur ⊥ (il aurait été plus logique de séparer les expressions et les déclarations de fonctions en deux domaines syntaxiques disjoints, mais cela aurait introduit des fonctions et des domaines supplémentaires).

L'application d'une fonction consiste juste à évaluer l'expression qui a été enregistrée à travers *f* dans l'environnement courant, avec la liaison requise pour le paramètre. Remarquons que ce paramètre ne peut être une fonction. Par ailleur, les définitions imbriquées étant incorrecte, une application de fonction ne peut pas modifier l'environnement courant. Donc :

$$Env\ [\ [\ \text{f}\ (\text{i})\ ]\ ]\ env = env$$

$$Val\ [\ [\ \text{f}\ (\text{i})\ ]\ ]\ env = (\text{fct}\ (\text{f})\ \langle \text{val}(\text{i}), \text{hor}(\text{i}), \perp \rangle\ env).0$$

$$Hor\ [\ [\ \text{f}\ (\text{i})\ ]\ ]\ env = (\text{fct}\ (\text{f})\ \langle \text{val}(\text{i}), \text{hor}(\text{i}), \perp \rangle\ env).1$$

Nous utilisons ici une définition à portée dynamique. En effet, l'environnement de la définition est celui de l'application de la fonction. Un mécanisme d'appel de fonction avec portée statique aurait été réalisé en capturant $\lambda\text{v}{:}\text{VAL}\times\text{HOR}\times\text{FCT}.\ \langle\ Val[\ [\ \text{e}\ ]\ ]\ env[\text{v}/\text{i}], Hor[\ [\ \text{e}\ ]\ ]\ env[\text{v}/\text{i}] \rangle$ dans l'environnement avec la définition suivante de FCT : $\text{FCT} = \text{VAL}\times\text{HOR} \rightarrow \text{VAL}\times\text{HOR}$.

Remarque :

Aucune autre expression de L1 ne modifie les liaisons variable-valeur, et on a donc :

$$Env\ [\ [\ e\ ]\ ]\ env = env$$ pour toute expression *e* qui n'est pas une liaison variable-valeur

Nous omettrons ces équations dans la suite.

Constantes

Val [[v:[n]]] *env* = cte(v^n)
Hor [[v:[n]]] *env* = true^n ; cte(false^n)

avec les définitions auxiliaires suivantes :

cte(v) = *v*;cte(*v*) $v^n = \begin{cases} \text{si } n = 1 \text{ alors } v \\ \text{sinon } \bullet v ® \mathbin{+\!\!+} v^{n-1} \end{cases}$

Une constante n'a qu'un seul top, au début.

Observateurs

Val [[id . n]] *env* = dot (n, val(id))
Hor [[id . n]] *env* = dot (n, hor(id))

avec la fonction dot définie par : dot (n, s) = s.0.n ; dot (n, *rest*(s))

Val [[id @ n]] *env* = dat (n, hor(id), val(id))
Hor [[id @ n]] *env* = hdat (n, hor(id))

avec les fonctions dat et hdat définie par :

dat (n, h, s) = si h.0 alors si n=0 alors s.0 ; cte(s.0) sinon ⊥ ; dat (n-1, *rest*(h), *rest*(s))
sinon ⊥ ; dat (n, *rest*(h), *rest*(s))
hdat (n, h) = si h.0 alors si n=0 alors *true* ; cte(*false*) sinon false ; hdat (n-1, *rest*(h))
sinon *false* ; hdat (n, *rest*(h))

ici *true* et *false* dénotent des constantes booléennes « polymorphes » qui ont la même géométrie que celle du tissu référencé par id. Cette géométrie est connue, ainsi qu'il a été dit plus haut.

Construction fby

Val [[i fby j]] *env* = fby (hor(i), val(i), hor(j), val(j))

Hor [[i fby j]] *env* = hfby (hor(i), val(i), hor(j))

La fonction `fby` est définie par :

fby (h, s, l, t) = si (h.0 ∧ (s.0 ≠ ⊥)) alors s.0 ; fby2 (s.0, *rest*(l), *rest*(t))
sinon s.0 ; fby (*rest*(h), *rest*(s), *rest*(l), *rest*(t))
fby2 (c, h, s) = si (h.0 ∧ (s.0 ≠ ⊥)) alors s sinon c ; fby2 (c, *rest*(h), *rest*(s))
hfb (h, s, l, t) = si (h.0 ∧ (s.0 ≠ ⊥)) alors *true* ; hfby2 (*rest*(l), *rest*(t))
sinon *false* ; hfb (*rest*(h), *rest*(s), *rest*(l), *rest*(t))
hfb2 (h, s) = si (h.0 ∧ (s.0 ≠ ⊥)) alors *true* ; *rest*(h) sinon *false* ; hfb2 (*rest*(h), *rest*(s))

Autrement dit, i fby j débute comme i jusqu'au premier top ayant une valeur différente de ⊥, et ensuite se poursuit comme j à partir du moment ou j a un top dont la valeur est définie.

On teste explicitement que la valeur de i et de j est non-nil sur un top car il est faux de penser que la valeur d'un tissu est différente de ⊥ sur un top (Cf. l'opérateur de délai). Le pourquoi de la chose est discuté dans la prochaine section. (Notons que si la fonction λx.x≠⊥ n'est pas stricte, elle est néanmoins continue).

Arithmétiques

Val [[i binop j]] *env* = bop (binop, val(i), val(j))

Hor **[[** i binop j **]]** *env* = hbop (hor(i), hor(j))

Les définitions de **bop** et hbop sont les suivantes :

bop (s, t) = binop (s.0, t.0) ; bop (binop, *rest*(s), *rest*(t))
hbop(h, l) = (h.0 $\underline{\vee}$ l.0) ; hbop (*rest*(h), *rest*(l))

On supposera que les fonctions « binop » vérifient la propriété suivante :

binop (x, y) = $\perp$ $\Rightarrow$ x = $\perp$ $\vee$ y = $\perp$.

L'opérateur $\underline{\vee}$ est un opérateur non strict :

$\perp \underline{\vee} \perp$ = *false*
$\perp \underline{\vee}$ x = x $\underline{\vee} \perp$ = x *si* x $\neq \perp$
x $\underline{\vee}$ y = x $\vee$ y *si* x $\neq \perp$ *et* y $\neq \perp$

Cet opérateur est continu et permet explicitement de considérer une horloge indéfinie comme un tic. En particulier si l'un des deux arguments correspond à un top, alors c'est aussi un top pour l'expression toute entière.

Alpha-notation

Val **[[** f ^ i **]]** *env* = alpha (λj.(fct (f) j *env*).0, val(i), hor(i))

Hor **[[** f ^ i **]]** *env* = alpha (λj.(fct (f) j *env*).1, val(i), hor(i))

avec : alpha (f, s, h) = (α f $\langle$ s$\uparrow$, h$\uparrow$, $\perp\rangle$) $\downarrow$

L'opération s$\uparrow$ permet de transformer une suite de collections en une collection de suites et l'opération c$\downarrow$ réalise l'inverse : elle transforme une collection de suites en une suite de collections. Ces deux fonctions ont pour définition :

c$\downarrow$ = c$\downarrow$0 ; c$\downarrow$1 ; c$\downarrow$2 …
c$\downarrow$i = $\langle$ c.0.i c.1.i … c.*n*.i $\rangle$ avec *n* = |c|

s$\uparrow$ = $\langle$ s$\uparrow$0, s$\uparrow$1, …, s$\uparrow$*n* $\rangle$ avec *n* = |s.0|
s$\uparrow$i = s.0.i ; *rest* (s$\uparrow$i)

les « … » dans les définitions se remplacent facilement par une récurrence. Donnons un exemple avec le calcul de la valeur de Q :

f (x) = x+1;
Q = f ^ T;

on suppose que T à pour valeur v = < $\langle$ 0, 1, 2 $\rangle$ > et pour horloge h = $\langle$ true, true, true$\rangle$; cte(false[3]). On est amené à calculer :

v$\uparrow$ = $\langle$ <0>, <1>, <2> $\rangle$
h$\uparrow$ = $\langle$ <true>;cte(false), <true>;cte(false), <true>;cte(false) $\rangle$

Val **[[** f ^ T **]]** ε
= α λj.(λp:VAL×HOR×FCT, ρ:ENV. $\langle$ *Val* **[[** x+1 **]]** ρ[p/x], *Hor* **[[** x+1 **]]** ρ[p/x]$\rangle$ j ε).0 (v$\uparrow$, h$\uparrow$, $\perp$)
$\downarrow$
= $\langle$ *Val* **[[** x+1 **]]** ε[$\langle$ <0>, <true> ; cte(false), $\perp\rangle$ /x] ,
Val **[[** x+1 **]]** ε[$\langle$ <1>, <true> ; cte(false), $\perp\rangle$ /x],
Val **[[** x+1 **]]** ε[$\langle$ <2>, <true> ; cte(false), $\perp\rangle$ /x] $\rangle$ $\downarrow$

= …
= $\langle$ <1>, <2>, <3> $\rangle$ $\downarrow$
= < $\langle$ 1, 2, 3 $\rangle$ >

Beta-réduction

Pour montrer le mécanisme de la beta-réduction, il faut des fonctions à deux arguments. Le lecteur aura compris qu'elles se généralisent sans peine à partir du mécanisme que nous avons mis en place, en introduisant des paramètres supplémentaires à la fonction capturée dans l'environnement. Nous supposons donc ici que la fonction f prend deux arguments :

Val [[f \ i]] *env* = beta (λj, k.(fct (f) j k *env*).0, val(i), hor(i))

Hor [[f \ i]] *env* = beta (λj, k.(fct (f) j k *env*).1, val(i), hor(i))

avec beta (f, s, h) = (β f ⟨ s↑, h↑, ⊥⟩) ↓

Sélection

Val [[i(j)]] *env* = select (val(i), val(j))

Hor [[i(j)]] *env* = *Hor* [[i]] *env*

select (s, t) = select-c (s.0, t.0) ; select(*rest* (s), <t.0>)
select-c (c, d) = si |d| > 0 alors ⟨ c.(d.0) ⟩ ++ select-c(*rest* (c), *rest* (d)) sinon ⊥

On remarque que le tissu j servant d'adresse n'est utilisé que pour son premier élément : c'est ici qu'intervient le caractère « statique » de la communication; les adresses sont des constantes calculables à la compilation. Une version non statique de la communication serait :

Val [[i(j)]] *env* = select_dynamique (val(i), val(j))
Hor [[i(j)]] *env* = hbop (hor(i), hor(j))
select_dynamique = select-c (s.0, t.0) ; select(*rest* (s), *rest* (t))

où une sélection est effectuée chaque fois que l'un des deux arguments bouge.

Conditionnelle

Val [[if i then j else k]] *env* = cond (val(i), val(j), val(k))

Hor [[if i then j else k]] *env* = hbop (hor(i), hbop (hor(j), hor(k))

cond (s, t, u) = cond-c(s.0, t.0, u.o) ; cond (*rest* (s), *rest* (t), *rest* (u))
cond-c (a, b, c) = α *si_alors_sinon*$_{SCALAIRE}$ (a, b, c)

Délai

Val [[$ i]] *env* = délai (val(i), hor(i), ⊥, ⊥)

Hor [[$ i]] *env* = hor(i)

délai (s, h, current, next) = si (h.0 ∧ (s.0≠⊥)) alors next ; délai (*rest* (s), *rest* (h), next, s.0)
sinon current ; délai (*rest* (s), *rest* (h), current, next)

Cette définition fait du retard un élément « passif » qui n'a pas d'horloge propre : un retard a la même horloge que le tissu qui est retardé. Un comportement « actif » aurait consisté par exemple à supprimer le premier top d'horloge d'un tissu. Cette approche participe de l'attitude générale qui consiste à confondre l'horloge d'un tissu et son domaine de définition (i.e. les tops sur lesquels sa valeur n'est pas ⊥). Bien que séduisante, cette approche introduit des « anomalies » qui rendent l'interprétation des équations récursives peu naturelle (nous en discuterons dans le prochain paragraphe).

Trigger

Val [[i when j]] *env* = trigger (val(i), val(j), hor(j), ⊥)

Hor **[[** i when j **]]** *env* = htrigger (val(j), hor(j))

Les fonctions auxiliaires sont définies ci-dessous :

trigger (i, j, h, v) = si ((j.0 ≠ ⊥) ∧ h.0 ∧ j.0) alors i.0 ; trigger (*rest* (i), *rest* (j), *rest* (h), i.0)
sinon v ; trigger (*rest* (i), *rest* (j), *rest* (h), v)

htrigger (j, h) = si (j.0 ≠ ⊥) alors (h.0 ∧ j.0) ; htrigger (*rest* (j), *rest* (h))
sinon *false* ; htrigger (*rest* (j), *rest* (h))

Il faut remarquer qu'on teste explicitement que la valeur de j est définie : en un tic t où la valeur de j est indéfinie, « a when b » a pour valeur, la valeur du top précédent. L'opérateur ∧ est un opérateur non strict qui traite ⊥ comme la valeur *false* :

⊥ ∧ ⊥ = *false*
⊥ ∧ x = x ∧ ⊥ = *false*
x ∧ y = x ∧ y *si* x ≠ ⊥ *et* y ≠ ⊥

ce qui permet explicitement de considérer qu'une horloge indéfinie correspond à un tic. L'opérateur ∧ est continu.

Héritage

Val **[[** i # j **]]** *env* = enrich (val(i), val(j))
Hor **[[** i # j **]]** *env* = enrich (hor(i), hor(j))

enrich (s, t) = (s.0 ++ t.0) ; enrich (*rest* (s), *rest* (t))

Imbrication spatiale

Val **[[** {i} **]]** *env* = bloc (val(i))
Hor **[[** {i} **]]** *env* = bloc (hor(i))

bloc (s, t) = ⟨ s.0 ⟩ ; bloc (*rest* (s))

Imbrication temporelle

Val **[[** i every j where e **]]** *env* = chaque (⟨ *Val* **[[** e **]]**, *Hor* **[[** e **]]**, ⊥⟩, *env*, i, j, hor(j), ⊥)
Hor **[[** i every j where e **]]** *env* = hor(j)

chaque = λ v:VAL×HOR×FCT, ρ:ENV, i:ID, j:ID, h:*TISSU*, previous:*COLLECTION*.
si h.0 alors run (v, ρ, i) ; chaque (v, next(ρ), i, j, *rest* (h), run (v, ρ, i))
sinon previous ; chaque (v, next(ρ), i, j, *rest* (h), previous)

run = λ v:VAL×HOR×FCT, ρ:ENV, i:ID. first ((v freeze(ρ) i).0, (v freeze(ρ) i).1)

freeze = λ ρ:ENV. (λi:ID. ⟨cte((ρ i).0), cte((ρ i).1), (ρ i).2⟩)

next = λ ρ:ENV. (λi:ID. ⟨*rest*((ρ i).0), *rest*((ρ i).1), (ρ i).2⟩)

first (s, h) = if h.0 alors s.0 sinon first (*rest* (s), *rest* (h))

La valeur de l'expression « i every j where e » s'obtient de la manière suivante. Sur chaque top de j, on évalue l'expression e dans un environnement où les identificateurs extérieurs à la clause « where » sont liés à une suite constante dont la valeur est la valeur courante de l'identificateur. La valeur de l'expression est la première valeur (i.e. la valeur sur le premier top) du tissu dénoté par i dans l'expression e. L'horloge de l'expression est celle de j.

La fonction run permet de d'évaluer l'expression e dans un environnement modifié par freeze. Freeze est une fonction qui modifie un environnement afin que tout identificateur soit lié à une suite constante dont la valeur est celle du premier élément de la suite initiale liée à l'identificateur. La fonction next est une fonction qui modifie un

environnement pour fournir un environnement dans lequel les identificateurs sont liés au reste de leur suite initiale. Enfin la fonction first permet de récupérer la valeur du premier top d'un tissu.

II.2. Sémantique d'un programme

Un programme 81/2 est une composition d'expressions (prog ∈ PROG) :

prog ::= e | e ; prog

(attention à ne pas confondre le « ; » opérateur sur les suites et **[[...** ; … **]]** la composition des expressions 81/2). La sémantique d'un programme est un environnement où les identificateurs sont liés aux tissus qu'ils réfèrent. On fournit aussi un environnement de base :

Prog : PROG → ENV → ENV

Prog **[[** e **]]** *env* = *Env* **[[** e **]]** *env*
Prog **[[** e ; prog **]]** *env* = (*Prog* **[[** prog **]]** ∘ *Env* **[[** e **]]**) *env*

env_{base} = ε[⟨cte(true), cte(true), ⊥⟩ /Clock]

∘ dénote la composition de fonction. L'environnement de base que nous nous donnons contient le tissu Clock, mais il peut bien sûr contenir tout tissu prédéfini qui serait utile (en particulier tous les tissus déclarés *outside* et non définis).

II.3. Un exemple de calcul de la sémantique d'une expression

Nous allons calculer la valeur de T dans le programme :

T@0 = 0 ;
T = $T + 1 when Clock ;

ce qui correspond en L1 au programme :

P0 = 0 ;
P1 = 1 ;
Q = P1 when Clock ;
R = $T ;
S = R + Q ;
T = P -> S ;

On note p0, p2, q, r, s, t (resp. hp0, hp1, hq, hr, hs, ht) les valeurs (resp. les horloges) des tissus P0, P1, Q, R, S et T. En se reportant aux équations sémantiques, on peut vérifier que les équations suivantes doivent être vérifiées :

p0 = cte(0)
hp0 = true ; cte(false)

p1 = cte(1)
hp1 = true ; cte(false)

q = trigger (cte(1), cte(true), cte(true), ⊥) = cte(1)
hq = htrigger (cte(true), cte(true)) = cte(true)

r = délai(t, ht, ⊥, ⊥)
hr = ht

s = bop(+, r, q)
hs = hbop(hr, hs)

t = fby (true ; cte(false), cte(0), s)
ht = fby (true ; cte(false), true ; cte(false), hs)

soit encore, en développant la définition de t et de ht :

t = 0 ; *rest* (bop (+, r, q))
= 0 ; bop (+, *rest* (r), cte(1))
= 0; *1+ rest* (r)

ht = true ; *rest* (hbop(hr, cte(true)))
= true ; *rest* (hr)

où *1+* est la fonction *1+* = λu. (u.0 + 1 ; *1+ rest* (u)). Or ht = hr, et donc on a l'équation :

hr = true ; *rest* (hr)

d'où l'on déduit que hr = ht = cte(true). En injectant ces résultats dans la définition de r, il vient :

r = délai (0; *1+ rest* (r), *cte* (true), ⊥, ⊥)
= délai1 (0 ; *1+ rest* (r), ⊥)

avec délai1 l'application partielle de délai à (·,·,cte(true),·), i.e. délai1(u, v) = v ; délai1(*1+ rest* (u), u.0). Et donc :

r = ⊥ ; délai1(*1+ rest* (*1+ rest* (r)), 0)
= ⊥ ; 0 ; délai1(*1+ rest* (*1+ rest* (*1+ rest* (r))), 1)

et on montrerait facilement par récurrence que r = ⊥ ; 0 ; 1; 2 ; 3 ;… et par suite que t = 0 ; 1 ; 2 ; …

III. Remarques sur la sémantique adoptée

Nous donnons ici la motivation de certains choix qui ont été fait dans le *design* du langage. Les buts poursuivis ont été de préserver autant que faire se peut la *localité* et le caractère *statique* des calculs à effectuer.

III.1. Horloge simple et horloge complexe

L'horloge qui est associée à un tissu 81/2 a la même structure que les valeurs de ce tissu. Ainsi, une suite de booléens est attachée à chaque suite de scalaires d'un tissu, et si le tissu est complexe, l'horloge du tissu sera aussi complexe.

Une option envisageable aurait été d'associer une seule horloge a un tissu : l'horloge d'un tissu pourrait être définie comme « l'union » des horloges de ses composants. Par exemple, si *R* est un tissu dont l'horloge est 2I**N** (i.e. *vraie* sur les tops qui ont une date paire et *faux* sur les tics qui sont impairs) et si *S* est un tissu d'horloge 3I**N**, alors *T* = { *R*, *S* } est un tissu d'horloge 2I**N**+3I N.

Cette approche est séduisante car elle associe une seule horloge à une entité qui a une cohérence en soi. Malheureusement avec cette façon de faire, l'observation de la composante d'un tissu est différente de cette composante isolée. Pour reprendre l'exemple précédent, avec

T = { *R*, *S* }
r = *T*.0

l'horloge de *r* est différente de l'horloge de *R*. Il est vrai que sur cet exemple, les valeurs de *r* et les valeurs de *R* restent identiques. Mais il est facile, par exemple en comptant les tops de *r* et de *R* d'obtenir des tissus dont

l'observation des valeurs diffère. Ainsi, l'option « une horloge simple par tissu » ne permet pas d'assurer que les opérateurs spatiaux sont sans effet sur les propriétés temporelles des tissus : les dimensions spatiales et temporelles ne sont plus orthogonales.

Le choix qui a été fait est de minimiser autant que faire se peut les relations entre les opérations spatiales et temporelles. En effet, leur mélange ne peut amener lors de l'implémentation que des calculs supplémentaire obligeant de plus à propager des données. Par exemple, dans l'exemple précédent, il aurait fallu amener les valeurs de l'horloge de *S* aux processeurs dédiés au calcul de *R* et inversement, afin de pouvoir générer l'horloge correcte de *T*.

III.2. Le traitement des valeurs indéfinies

Nous avons choisi de ne pas identifier une collection ayant un élément indéfini avec la collection indéfinie. Ce choix est encore justifié par le besoin de localité : supposons que le calcul de la valeur d'un tissu implique une division, et que pour un point du tissu, le dénominateur soit nul. La valeur du tissu en ce point est alors indéfinie. Si on veut donner au tissu tout entier une valeur indéfinie, il faut propager à chaque tic et à chaque processeur impliqué dans le calcul d'un tissu, le fait que les valeurs calculées sont correctes.

Cette attitude est tout à fait possible dans un environnement SIMD : sur la Connnection-Machine, une exception levée par un des 64536 processeurs engendre l'arrêt de tous. Cette attitude nous semble moins raisonnable (et matériellement trop coûteuse) dans un environnement MIMD où des activités dépendant de séquenceurs différents s'exécutent concurremment. Aussi avons-nous adopté le point de vue exposé : la valeur indéfinie va se propager suivant les calculs, rendant indéfinies les valeurs qui dépendent d'elle de manière stricte.

III.3. L'horloge d'un tissu n'est pas la fonction caractéristique du domaine de définition de ses valeurs

L'usage des valeurs indéfinies qui est évoqué au paragraphe précédent est celui concernant les situations exceptionnelles (une division par 0 par exemple). Il est possible d'éviter ces cas exceptionnels, à l'aide de tests appropriés. Un autre usage est fait des valeurs exceptionnelles en 81/2, qui est relié au domaine de définition des valeurs d'un tissu.

Les valeurs d'un tissu sont observables a priori à tout instant : avant son premier top, la valeur d'un tissu est la valeur indéfinie, ensuite, la valeur en un tic est la valeur du dernier top. Ce procédé peut sembler lourd puisque l'on dispose de toute façon de l'horloge du tissu, qui indique en quels tops le tissu a une valeur (hors les cas exceptionnels). On est alors tenté de faire débuter une horloge, à partir du moment où les tissus ont une valeur qui n'est pas indéfinie. Par exemple :

clock(*a*)	·	·	·	•	•	·	·	•	·	•
a	·	·	.	1	2	·	·	·	3	3
clock(*b*)	·	·	·	·	·	•	·	·	•	•
b	·	·	·	·	·	10	·	·	20	30
clock(*c* = *a*+*b*)	·	·	·	·	·	•	·	•	•	•
c* = *a*+*b	·	·	·	·	·	12	·	13	23	33

(• représente la valeur vraie, les valeurs observées ne sont pas répétées quand elles sont semblables à celle qui précède). Autrement dit, l'horloge de *a*+*b* correspond au « ou logique point-à-point » de l'horloge de *a* et de l'horloge de *b à partir du moment où* les tissus *a* et *b* ont déjà eu leur premier top.

Cette manière de voir est cependant incompatible avec une interprétation naïve des équations récursives. Cela s'illustre simplement sur l'exemple d'un compteur : avec l'interprétation précédente, le programme suivant :

$T@0 = 0$; (1)
$T = \$T + (1 \text{ when Clock})$; (2)

définit le tissu T comme identique à la constante 0. En effet, l'équation (1) force le tissu T à avoir la valeur 0 au premier top de 0 (i.e. à l'instant 0). L'horloge de $\$T$ se définit comme l'horloge de T moins son premier top et l'horloge de T se définit comme étant le top 0 suivi de l'horloge de $\$T$ puisque l'horloge de 1 when Clock est cte(true). En reportant dans (1) et (2), il vient que l'horloge ht de T doit vérifier :

ht.0 = true → *équation 1* : 0 est un top
rest(ht) = *rest* (ht) → *équation 2* : *rest* (ht) est l'horloge de $\$T$

autrement dit, la seule condition que doit vérifier l'horloge d'une solution à (1)+(2) est :

ht.0 = true

La constante 0 est donc bien une solution aux équations (1)+(2) et on montre facilement que c'est la plus petite.

On peut aussi interpréter cela de la façon suivante : les équations de définition doivent être vérifiées sur les tops pour lesquelles elles sont quantifiées (i.e. (1) doit être satisfait pour le premier top et (2) pour les autres). L'équation (2) est donc satisfaite de-facto par un tissu qui n'a pas de top en dehors de 0.

Ce n'est évidemment pas le résultat escompté. Il est possible de forcer l'interprétation voulue en forçant un top pour l'équation (2). Cela est possible si la somme est définie et cela est à son tour possible si $\$T$ a un top. Pour cela il suffit que T possède deux valeurs initiales (auquel cas l'horloge de T possède au moins deux tops et l'horloge de $\$T$ au moins un). Le programme du compteur devient :

$T@0 = 0$;
$T@1 = 1$ when Clock ; (3)
$T = \$T + (1 \text{ when Clock})$;

On peut vérifier que ce programme réalise bien un compteur. Mais il faut avouer que l'ajout de l'équation (3) n'est pas de prime abord très naturel. Que deviendrait alors un programme plus complexe ?

En conséquence de quoi, nous avons dissocié le domaine de définition d'un tissu de son horloge. Il faut remarquer que cela n'a pas de grande influence sur l'observation des valeurs d'un tissu. Ainsi, l'exemple précédent devient :

clock(*a*)	·	·	·	•	•	·	·	•	·	•
a	⊥	⊥	⊥	1	2	·	·	·	3	3
clock(*b*)	·	·	·	·	·	•	·	·	•	•
b	⊥	⊥	⊥	⊥	⊥	10	·	·	20	30
clock(*c* = *a*+*b*)	·	·	·	•	•	•	·	•	•	•
c* = *a*+*b	⊥	⊥	⊥	⊥	⊥	12	·	13	23	33

on voit que la suite des valeurs non-⊥ est la même que dans l'exemple précédent, par contre « il y a plus de tops ». Notons aussi que la valeur ⊥ agit comme la valeur *false* quand cette valeur sert de commande à un trigger ou à un filtre.

III.4. Un langage statique

Plusieurs choix rendent le langage statique :

– Les schémas de communication sont des constantes (bien qu'il soit possible d'introduire sans problème des communications dynamiques).

– Les fonctions récursives sont interdites, de même que les fonctions d'ordre supérieur (i.e. acceptant une

fonction en argument ou retournant une fonction). En effet, ces dernières permettent la récursion de manière « cachée » comme le montre l'exemple suivant :

F (f, x) = si (x == 0) alors 1 sinon x * f(f, x - 1) fi

La fonction F n'intervient pas elle-même dans sa définition mais on peut appliquer F à elle-même, ce qui la transforme en fonction récursive (et permet de calculer la fonction factorielle).

– On ne peut définir de fonctions imbriquées et la portée des variables dans une fonction est dynamique. Cela rend la notion d'application de fonction très voisine de celle de macro-expansion. Puisque les fonctions récursives sont interdites, cette macro-expansion peut prendre place à la compilation. Cependant la notion de fonction 81/2 n'est pas semblable à une notion de macro du niveau lexical comme celle introduite par le `#define` du préprocesseur C. En effet, une fonction peut être argument d'une alpha, d'une beta ou d'un scan. La notion de fonction 81/2 est donc très proche des macros Common-Lisp.

Par contre plusieurs aspects du langage restent « dynamiques » :

– Les tops d'un tissu ne sont en général pas calculables statiquement (à cause de l'opérateur when);

– les arguments d'une sous-expression ont une horloge quelconque;

– Le temps passé dans un every n'est en général pas calculable statiquement.

III.5. Implémentation directe des équations sémantiques

Un petit évaluateur, directement basé sur une première version des équations sémantiques, a été écrit en C++ et permettait d'évaluer une expression 81/2. L'implémentation était « brutale » : le calcul de *Val* et *Hor* à un instant *t*, impliquait le recalcul de *Val* et de *Hor* à l'instant *t*-1, ainsi qu'il apparaît dans les équations sémantiques. Cet interprète était donc inutilisable (le temps de calcul impliqué par l'évaluation des expressions à un tic donné croissant avec le rang du tic) mais offrait l'avantage d'une implémentation directe des équations sémantiques.

IV. Le typage des géométries

Une sémantique formelle du langage 81/2 a été proposée dans le chapitre précédent. Elle s'applique à des programmes corrects. Le but de ce chapitre, et des deux suivants, est de définir précisément ce qu'est un programme correct.

Un programme correct est un programme correctement typé. Le typage permet de restreindre les expressions valides du langage. Nous voulons plus particulièrement interdire 3 sortes d'expressions :

1) les expressions impliquant une géométrie non constante comme par exemple dans :

$Q@0 = 0$; (E1)
$Q = \{ \$Q \}$;

2) les expressions dont l'ordonnancement des calculs ne peut être fait de manière statique comme par exemple dans :

R = si A alors S sinon B fi; (E2)
S = si A alors B sinon R fi;

3) les expressions dont la valeur se réduit de manière certaine à ⊥ comme par exemple :

$T = \$1$; (E3)

La détermination statique de la géométrie d'un tissu a la conséquence importante de permettre la réservation statique de la mémoire nécessaire à l'exécution du programme. L'expression (E1) qui ne vérifie pas cette propriété provoque par exemple un effet « larsen spatial » qui saturera la mémoire. Une gestion statique de la mémoire est une propriété impérativement requise pour éviter les problèmes et le coût de l'allocation dynamique dans un environnement distribué.

On comprend l'intérêt pour un langage qui se veut statique de pouvoir fixer un ordre (partiel) d'exécution à la compilation. Cependant, on peut se demander si l'interdiction d'expressions du type de (E2) n'est pas une mesure trop draconienne. Autrement dit, les programmes dont l'ordonnancement est statique constituent-ils une classe de programme suffisamment générale pour être utile ? Une réponse que l'on peut avancer est celle basée sur l'expérience des langages synchrones comme LUSTRE ou SIGNAL qui appliquent cette restriction [CPHP 87]. À ma connaissance, cette contrainte n'est pas jugée rédhibitoire.

On peut regarder un programme 81/2 comme un réseau de processus : un processus est attaché à chaque définition de tissu, consomme les valeurs envoyées par les processus qui apparaissent en partie droite de sa définition et calcule une valeur émise vers les autres processus. De ce point de vue, un tissu dont la valeur est réduite à ⊥ s'interprète comme un processus souffrant de *famine*. Il est donc important de pouvoir détecter des expressions du type (E3).

I. Le typage d'un programme $8_{1/2}$

L'outil qui va nous permettre de détecter les situations précédentes est le typage d'un programme. Le concept de type auquel nous adhérons ici est celui défendu par L. Cardelli dans [Cardelli 89]. Un type est vu comme une spécification partielle et la vérification de type (*type checking*) correspond à vérifier qu'un programme répond à cette spécification. Cependant le typage diffère de la vérification générale de programmes en ce qu'il doit exister un moyen *automatique* de vérifier les contraintes du typage (alors que la vérification de programme relève de la preuve de théorème qui est un problème indécidable).

Un système de typage (Cf. par exemple [Reynolds 85]), est basé sur la définition d'un ensemble de types et de règles d'affectation d'un type à une expression. Les règles de typage seront données suivant l'approche de la « sémantique naturelle » décrite dans [Kahn 87].

Le typage en 81/2 est complètement statique. Cela veut dire que le typage est vérifié le plus tôt possible, à la compilation, et n'entraîne pas de tests effectués dynamiquement à l'exécution du programme. Nous distinguerons *trois systèmes de typage indépendants*.

Le type géométrique

Le premier système de typage, le typage géométrique, permet de vérifier des propriétés portant sur la géométrie d'un tissu. C'est un typage explicite. Cependant, le type géométrique d'une expression est fonction du type géométrique de ses sous-expressions. Il est donc souvent possible à l'utilisateur d'omettre les indications de typage, celles-ci étant inférées par le compilateur. Le type géométrique d'un tissu correspond à son cardinal (dans le cas d'un tissu simple) et les règles de typage permettent de restreindre les combinaisons possibles de tissus : une expression bien-typée est une expression dont la géométrie est connue à la compilation. Une expression qui ne peut pas être typée à l'aide des règles est rejetée comme incorrecte.

Ordonnancement des tissus

Le deuxième système de typage est celui permettant de fixer statiquement un ordre (partiel) de calcul des expressions. Ce typage est transparent à l'utilisateur : le calcul du type de séquencement d'une expression est effectué par le compilateur et n'est jamais explicité par le programmeur. Il correspond à construire un ordre partiel entre sous-expressions. « Plus » cet ordre est « partiel », plus il y a de parallélisme présent dans le calcul. Il faut remarquer que la construction de cet ordre partiel **ne** se réduit **pas** à vérifier qu'il n'y a pas de cycle dans le graphe des dépendances entre définitions. Par exemple

$$T\,[763] = (\{1\} \,\#\, (f \,\hat{}\, T)) : [T] \qquad \text{(E4)}$$

est une collection de 763 éléments dont la valeur est $\langle 1, f(1), f(f(1)), f^3(1), \ldots f^{762}(1) \rangle$. Il existe un ordre de calcul simple : le calcul du point i précède le calcul du point j si $i<j$.

La question de savoir si un ordonnancement existe est traitée dans le chapitre qui suit.

Le type temporel

Le troisième système de typage, le typage temporel, a pour but de caractériser statiquement l'horloge d'un tissu afin de pouvoir prédire si un tissu est équivalent à ⊥. Ce système de typage fait l'objet du chapitre VI.

II. Le typage des géométries

Nous associons à **certaines** expressions 81/2 un type à l'aide d'un système de règles. Nous soulignons « certaines expressions » car ce sont évidemment les expressions bien typées : les expressions dont il est impossible de dériver un type sont incorrectes.

Le typage géométrique en 81/2 répond à deux objectifs :

– interdire les expressions dont la géométrie ne peut être déterminée à la compilation;

– interdire les expressions dont la géométrie n'est pas *homogène*.

Un tissu a une géométrie homogène si tous les points de ce tissu ont la même géométrie. Par exemple « { 1 2 3 4 } » a une géométrie homogène (tous les points sont scalaires). { { 1 } { 2 } { 3 } } a aussi une géométrie homogène, mais pas { { 1 } { 2 3 } { 4 } } (le deuxième point n'a pas la même géométrie que les autres). Une géométrie homogène est nécessaire afin de garantir que la géométrie d'une sélection est indépendante de la valeur du sélecteur.

On suppose dans tout ce chapitre que la structure de bloc a été *aplatie* par une première passe qui a rendu tous les identificateurs différents. En conséquence, un programme se présente comme une suite d'équations à un seul niveau. Les expressions du type *identificateur.label* ont été transformées en un nouvel identificateur par la passe d'aplatissement. Par exemple le programme suivant :

```
Q = … ;
T = {
     i@0 = 0 ; i = if (i < 10) then $i+1 when Q else 0 fi;
     a = if (i=0) then Q else $a + Q fi;
};
R = 2 * T.a ;
```

est transformé en (T_i et T_a étant des identificateurs qui n'apparaissent pas ailleurs dans le programme) :

```
Q = … ;
T_i@0 = 0 ; T_i = if (T_i < 10) then $T_i+1 when Q else 0 fi;
T_a = if (T_i=0) then Q else $T_a + Q fi;
R = 2 * T_a ;
```

Nous ne nous étendrons pas plus sur les règles de visibilité des identificateurs (elles sont semblables par exemple à celles qui régissent le nom des attributs d'un objet C++).

II.1. Les règles du typage géométrique

II.1.1. La sémantique naturelle

Les règles de typage sont données sous la forme de formules « EG ≦ e : G » qui se lisent« avec les hypothèses EG l'expression e a pour type géométrique G ». EG dénote un environnement : le type d'une expression dépend du type des variables qui interviennent dans l'expression. L'environnement EG permet de donner un type à chaque

variable d'une expression. Il faut remarquer que dans la règle $EG \trianglelefteqslant e : G$ il est implicitement supposé que EG définit un type pour au moins chaque variable libre de e.

Chaque règle est composée de prémisses séparées de la conclusion par une ligne horizontale. Chaque règle contient des variables; une instance de la règle est obtenue en remplaçant les variables par un objet du type approprié. Nous suivrons les conventions suivantes :

variable	*type d'objet*
EG	environnement des types géométriques
id	identificateurs
F, G, H	types géométriques
e, f, g	expressions

Chaque instance d'une règle donne en conclusion un typage valide si les prémisses de l'instance sont des typages valides. Ce style de présentation, introduit par Plotkin dans [Plotkin 81], est à la base de Centaure, un système interactif de spécification sémantique [Kahn 87].

II.1.2. Le typage géométrique

Un type géométrique est soit un entier soit une liste dont les éléments sont des types géométriques :

$GEOMETRIE \cong \mathbb{N} \cup GEOMETRIE^*$

L'interprétation en est simple : si le type géométrique d'une expression est un entier *n*, alors l'expression définit un tissu simple de cardinal *n*; sinon le tissu est un tissu complexe, dont le type géométrique du point *i* est décrit par le *i* ème élément de la liste. Nous utiliserons les notations du chapitre précédent : $\langle , \rangle$ est le constructeur de liste; si *l* est une liste, *l.i* est le *i*ème élément de *l*; *n* ++ *l* représente une liste dont le premier élément est *n* et qui se poursuit comme *l*.

Une définition de fonction n'a pas de type géométrique mais l'application d'une fonction, oui : la géométrie de l'application est la géométrie de l'expression substituée. Pour simplifier l'exposé nous ne considérons que les fonctions à deux arguments.

Nous décrivons le typage sur un sur-ensemble du langage L1 (la différence essentielle est que les sous-expressions d'une expression donnée ne sont pas réduites à être des identificateurs).

Type d'un identificateur :

$$EG, id : G \trianglelefteqslant id : G \qquad \text{(GT0)}$$

Liaison variable - valeur :

$$\frac{EG, id : G \trianglelefteqslant e : G}{EG \trianglelefteqslant id=e : G} \qquad \text{(GT1)}$$

Typage des constantes :

$$EG \trianglelefteqslant Clock : 1 \qquad \text{(GT2)}$$

$$EG \trianglelefteqslant n : 1 \qquad EG \trianglelefteqslant n :[m] : m \qquad EG , id : G \trianglelefteqslant n :[id] : G \qquad \text{(GT3)}$$

Un scalaire « n » dénote un tissu plat de cardinal 1. Les constructions suivantes (GT3) permettent de créer des tissus constants plus complexes :

n : [m]

crée un tissu plat constant de cardinal m où chaque point a pour valeur n (attention à ne pas confondre le « : » signe de la relation de typage et le caractère « *:* » faisant partie de l'opérateur dyadique infixe « ·*:*[·] »). Une forme plus riche de l'opérateur de broadcast existe en 81/2 et permet de remplacer m par l'identificateur d'un tissu : n*:*[id]. Dans ce cas la géométrie de la constante est celle de ce dernier.

Arithmétique :

$$\frac{EG \Subset e : G \quad EG \Subset f : G}{EG \Subset \text{ e binop f} : G} \qquad \frac{EG \Subset e : G \quad EG \Subset f : G \quad EG \Subset g : G}{EG \Subset \text{ si e alors f sinon g fi} : G} \qquad \text{(GT4)}$$

Observateurs :

$$\frac{EG \Subset e : G}{EG \Subset e\,.\,i : G.i} \qquad \frac{EG \Subset e : G}{EG \Subset e\,.@i : G} \qquad \text{(GT5, 6)}$$

Application de fonction :

$$\frac{f(a, b) = e \quad EG, a : F, b : H \Subset e : G \quad EG \Subset g_1 : F, \quad EG \Subset g_2 : H}{EG \Subset f(g_1, g_2) : G} \qquad \text{(GT7)}$$

La notation « EG, a : F, b : H » indique un environnement où la géométrie de l'identificateur a est F et celle de b est H; la géométrie des autres identificateurs est décrite par EG. La première des clauses, « f(a, b) = e », n'est pas une prémisse mais est présente pour indiquer les conditions d'application de la règle (i.e. f est une fonction dont les paramètres formels sont a et b, et le corps, e). Cette règle indique que si G est la géométrie de e, avec pour hypothèse que les géométries de a et de b sont F et H respectivement, alors la géométrie de « $f(g_1, g_2)$ » est G, si les géométries de g_1 et de g_2 sont F et H respectivement.

Alpha-notation

Dans ce qui suit, N et M désignent des géométries qui correspondent à des entiers (donc à des tissus plats) et K et L des géométries qui correspondent à des listes (donc à des tissus complexes) :

$$\frac{f(a, b) = e \quad EG, a : 1, b : 1 \Subset e : G \quad EG \Subset g_1 : N \quad EG \Subset g_2 : N}{EG \Subset f \,\hat{}\, (g_1, g_2) : \textit{unfold}\,(G, N)} \qquad \text{(GT8.1)}$$

avec la fonction *unfold* (G, N) définie par :

unfold (G, N) = si (N = 1) alors G sinon G ++ *unfold* (G, N-1)

autrement dit, une alpha-notation sur des tissus simples produit un tissu dont le cardinal est celui des arguments de départ et dont les éléments ont une géométrie qui correspond à l'application de f à chaque paire d'éléments des arguments.

$$\frac{EG \Subset g_1 : \bullet \circledR, EG \Subset g_2 : \bullet \circledR}{EG \Subset f \,\hat{}\, (g_1, g_2) : \bullet \circledR} \qquad \text{(GT8.2)}$$

Une alpha appliquée à des tissus complexes sans point donne en résultat un tissu sans point. Cette règle sert à arrêter les dérivations induites par la règle suivante :

$$\frac{\begin{array}{c} f(a, b) = e \\ EG, a : F, b : H \Subset e : G \\ EG \Subset g_1 : F \mathbin{++} K \\ EG \Subset g_2 : H \mathbin{++} L \\ EG, g_1' : K, g_2' : L \Subset f \mathbin{\hat{}} (g_1', g_2') : I \end{array}}{EG \Subset f \mathbin{\hat{}} (g_1, g_2) : G \mathbin{++} I} \qquad \text{(GT8.3)}$$

Cette règle exprime que la géométrie d'une alpha correspond à la concaténation des géométries de la fonction f appliquée à chaque paire d'éléments des arguments.

Beta-réduction

Erreur ! (GT9)

Sélection

$$\frac{\begin{array}{c} EG \Subset e : G = \mathit{unfold}\,(H, n) \\ EG \Subset f : F \end{array}}{EG \Subset e\,(f) : \mathit{leaf}\,(F, H)} \qquad \text{(GT10)}$$

avec la fonction *leaf* définie par

leaf (F, H) = si H ∈ I N
alors F
sinon si F∈I N alors *unfold* (H, F) sinon *leaf* (F.0) ++ *leaf* (*rest* (F), H)

Remarquons que puisque *G* a une géométrie homogène, cela dispense de connaître la valeur de chaque point de f. Ainsi f peut être une constante (cas de la communication statique) ou bien être un tissu à part entière (cas de la communication dynamique). La fonction *leaf* à pour but de produire une géométrie semblable à celle de son premier argument mais dont les feuilles ont été remplacées par des éléments de G (donc de géométrie égale à H).

Retard

$$\frac{EG \Subset e : G}{EG \Subset \$e : G} \qquad \text{(GT11)}$$

Opérations temporelles dyadiques

$$\frac{\begin{array}{c} EG \Subset e : G \\ EG \Subset f : 1 \end{array}}{EG \Subset e \mathbin{\mathit{bitop}} f : G} \qquad \frac{EG \Subset e : G}{EG \Subset e \text{ synchro } f : G} \qquad \text{(GT11)}$$

bitop désigne l'une des quelconques opérations temporelles : fby, when, after et until.

Bloc

$$\frac{EG \Subset e : G}{EG \Subset \{ e \} : \bullet G \circledR}$$

(GT12)

Agrégation

$$\frac{\begin{array}{c} EG \Subset e : N \\ EG \Subset f : M \end{array}}{EG \Subset e \# f : N+M} \qquad \frac{\begin{array}{c} EG \Subset e : K \\ EG \Subset f : L \end{array}}{EG \Subset e \# f : \mathit{append}\,(K, L)} \qquad \text{(GT13)}$$

avec *append* la fonction de concaténation de liste (K et L sont des géométries de tissus complexes).

Imbrication temporelle

$$\frac{EG \Subset i : G}{EG \Subset i \text{ every } j \text{ where } e : G} \qquad \text{(GT14)}$$

Cette règle indique que la géométrie de « i every j where e » est celle de l'identificateur i. Nous ne nous sommes pas préoccupés des portées des variables : on a supposé que toutes les variables d'un programme sont distinctes. En réalité l'expression e introduit une nouvelle portée dans laquelle doit être défini l'identificateur i. La géométrie de l'expression est celle de l'expression attachée à i dans e.

II.2. Constante polymorphe et inférence de la géométrie

Dans un langage typé classique (PASCAL, C…) le compilateur associe un type à chaque expression et à chaque sous-expression. Cependant le programmeur n'a pas besoin de déclarer explicitement le type de toutes les sous-expressions et de toutes les expressions. Les annotations concernant le typage s'introduisent seulement en certains points du programme, par exemple à l'introduction d'une nouvelle variable. Le type des autres expressions est déduit à partir du contexte.

De par la forme des règles (GT1) à (GT14), le type d'une expression dans un certain environnement EG est directement fonction du type de ses sous-expressions immédiates dans un environnement EG′ (qui est souvent le même que EG). Le processus de typage se déroule donc de manière montante dans l'arbre d'une expression : à partir du type des feuilles (identificateurs et constantes) et des règles permettant la combinaison du type des sous-expressions en le type d'une expression plus grosse, il est possible d'inférer le type de toute expression.

Il faut donc connaître le type de chaque identificateur : pour cela chaque équation 81/2 est de la forme :

T **`spécification—de—la—géométrie`** = …

où **`spécification—de—la—géométrie`** est un élément de *GEOMETRIE* (le constructeur ⟨, ⟩ est remplacé par une paire de crochets et la géométrie d'un tissu simple est indiquée par une liste de 1 élément plutôt que par un scalaire). Par exemple,

R [5] = … *R* est un tissu plat de 5 éléments scalaires

S [[3] [3]] = … *S* est un tissu complexe de cardinal 2. Le premier point est un tissu plat de trois éléments, le deuxième point est un tissu plat de 3 éléments.

On introduit aussi la notation suivante, qui a déjà été vu au chapitre II, et qui correspond à la pratique de la déclaration des tableaux :

T [2, 5] = … *T* est un tissu complexe de 2 éléments, chaque élément étant un tissu simple de cardinal 5

Cette notation ne présente pas d'ambiguïté avec la notation des listes. Son interprétation générale est la suivante :

[a, b …] ≡ [$[b \ldots]_1 \ldots [b \ldots]_a$] (il y a « a » crochets de type [b …]).

Ces annotations explicites deviennent rapidement pénibles dès qu'il y a plus d'une dizaine d'équations, principalement pour trois raisons :

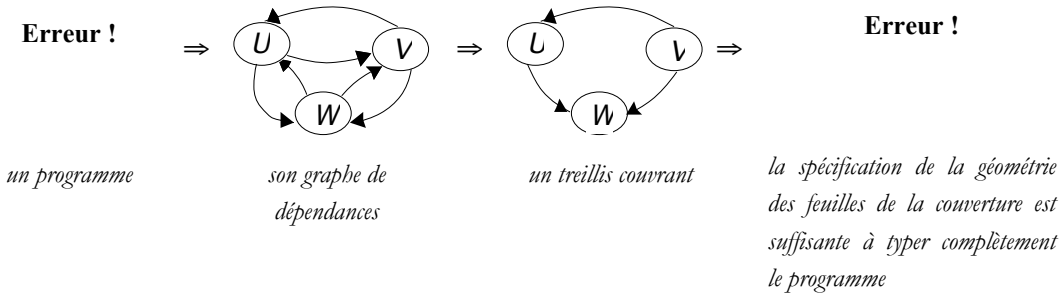
1) La répétition de la spécification de la géométrie est un processus fatigant et non exempt d'erreur. On aimerait pouvoir dire qu'un tissu a la même géométrie qu'un autre, sans autre précision.

2) L'annotation du type d'une constante est redondante. Par exemple dans :

   U [10] = $*U* + 1 *:*[10]

   la spécification de la géométrie de la constante « 1 » est redondante puisque (GT4) requiert que *U* et 1 aient la même géométrie.

3) L'annotation du type d'une variable est souvent redondante. Il n'y a pas besoin de typer toutes les définitions, mais uniquement les définitions correspondant aux éléments minimaux d'un treillis couvrant le graphe des dépendances univoques de géométries entre définitions. La figure suivante donne un exemple :



Le type de *U* et le type de *V* sont impliqués par le type de *W* (on aurait pu choisir une autre couverture avec par exemple *U* en feuille d'où on aurait déduit le type de *V* et *W*). Le graphe des dépendances de géométries est obtenu en examinant les règles de typage. Par exemple pour les opérateurs temporels, les types géométriques de l'expression et de la première sous-expression dépendent l'un de l'autre de manière univoque (ils sont égaux).

Pour résoudre le point 1), nous avons introduit la possibilité de spécifier une géométrie symboliquement : « *T* [*S*] = … » indique que *T* à la même géométrie que *S*. Pour résoudre le point 2), les constantes scalaires ont été étendues afin de noter (de manière ambiguë) les constantes *plates* de cardinal quelconque : ce sont des constantes *polymorphes*. Plus exactement, il s'agit de *polymorphisme ad-hoc* suivant la classification proposée par L. Cardelli et P. Wegner dans [Cardelli, Wegner 85]. On rencontre une situation analogue pour les constantes en C : la constante « 0 » dénote aussi bien un pointeur (sur un objet de n'importe quel type), un caractère, un entier, un flottant, un « double », etc. C'est le contexte d'utilisation qui permet de désambiguïser. Enfin, pour répondre au point 3), la déclaration de la géométrie d'un tissu devient optionnelle.

Les facilités introduites rendent plus difficile l'inférence du type de chaque expression. De plus les programmes peuvent devenir ambigus, c'est-à-dire admettre plusieurs géométries.

II.2.1. Un premier algorithme

Le problème qui se pose est donc de trouver un algorithme performant permettant d'inférer les types géométriques des variables d'un programme 81/2, de déceler les erreurs de typage et de détecter les programmes ambigus.

Le langage ML utilise un algorithme de typage [Hindley 69] [Milner 78] qui est susceptible de répondre à nos besoins. L'algorithme est encore un algorithme montant sur l'arbre d'une expression. Il assigne une *variable de type* à chaque identificateur et à chaque constante. La déclaration explicite du type de la définition attachée à l'identificateur force la valeur de la variable de type correspondante. Un algorithme d'unification [Robinson 65] est utilisé pour propager la valeur à toutes les occurrences de la variable et vérifier les contraintes.

Cet algorithme est très général. S'il aboutit, c'est que le programme admet (au moins un) typage. S'il reste des variables de type à la fin de la vérification, alors le programme est ambigu : il admet autant de typage que d'instanciations des variables de type restantes. Donnons trois exemples, un programme correctement typé, un programme invalide et un programme ambigu. Le type associé à un identificateur du programme est cet identificateur noté en minuscule. Les variables de type des constantes sont indiquées en indice.

	Programme 81/2	Équations correspondantes	Solution du typage
i)	$T@0 = 0_a$;	t = a	a = 10
	$T = \$T + 1_b$;	t = b	b = 10
	$U[10] = 1_c + T$;	u = c = t = 10;	u = t = c = 10
ii)	$T@0 = 0_a$;	t = a	
	$T[1] = \$T + 1_b$;	t = 1, t = b	——
	$U[10] = \$ T$;	u = 10, u = t	
iii)	$T@0 = 0_a$;	t = a	a = t
	$T = \$T + 1_b$;	t = b	b = t
	$U[10] = 1_c \# T$;	u = 10;	u = 10
		u = c + t	c = 10 - t

Dans le cas i), chaque variable de type a une valeur positive non nulle : le programme est correctement typé. Dans le cas ii) il n'y a pas de solution : le programme est incorrect. Dans le cas iii) il reste une variable de type t : le programme est ambigu.

L'algorithme de typage présenté est très puissant mais est aussi lourd (il faut un algorithme d'unification) et mal-commode à mettre en œuvre. Par exemple un « occur-check » est nécessaire afin de détecter que « $T = \{ \$T \}$ » est incorrect et éviter de boucler. Par ailleurs, l'unification ne suffit pas quand le type doit être calculé (et non pas propagé). Par exemple :

$R = \ldots$	
$S = \ldots$	
$T[10] = R \# S$;	t = 10 = r + s
$U[6] = \$R \# R$;	u = 6 = 2r

En fait cet exemple nous suggère une solution simple et efficace : l'inférence des géométries des équations 81/2 consiste à résoudre un système d'équations linéaires entre variables à valeur entière.

II.2.2. L'algorithme d'inférence et de type-checking géométrique

L'algorithme que nous proposons a pour but de déterminer le *cardinal* de chaque expression d'un programme 81/2. Plus précisément, on commence par déterminer la dimension de chaque tissu. On note $|T|$ le cardinal de T, $|T|_2$ le cardinal d'un élément de T, $|T|_3$ le cardinal d'un élément d'un élément de T, etc. La géométrie d'un tissu peut s'écrire comme :

$[T] = [\,|T|, |T|_2, \ldots, |T|_d\,]$ avec d la dimension du tissu

L'idée est de déterminer les $|T|_i$ et par suite, la géométrie. L'intérêt de cette approche est que le calcul d'un cardinal ne fait appel qu'à la résolution d'un système linéaire. On ne peut pas résoudre tous les $|T|_1$ puis les $|T|_2$, etc, car $|T|_{i-1}$ peut dépendre de $|T|_{i+1}$ et inversement. Par exemple :

$A = \{ B \}$;
$B = 1 + \$(A.0)$;

1 étant une constante, la géométrie de B est de dimension 1. Cela permet de fixer la dimension de A à 2. Ensuite on a résolu le système d'équations suivant :

$|A| = 1$
$|A|_2 = |B|$
$|B| = |A|_2$

Le système admet une infinité de solutions : les annotations de typage (il n'y en a aucune) ne sont pas suffisamment

fortes pour induire une géométrie unique pour chaque tissu.

On doit donc calculer tous les $|T|_i$ en même temps. Cela implique de calculer d'abord la dimension de chaque tissu. L'algorithme de typage comporte donc 2 étapes :

Étape 0 :

On commence par éliminer toutes les équations qui ont une annotation de type. Le type des variables correspondantes est en effet connu.

1ère étape :

On commence par calculer la *dimension* de chaque expression. La dimension d'une expression est le nombre de dimensions de la valeur d'un tissu. Par exemple, une constante a une dimension de 1. Le calcul de la dimension se fait de la manière suivante.

On associe à chaque expression et sous-expression du programme 81/2 une variable. Cette variable représente la dimension de l'expression. On forme ensuite le système des équations que doivent vérifier ces variables. Une expression 81/2 peut générer plusieurs équations. On trouvera en annexe à ce chapitre les équations associées à chaque expression. Les équations peuvent prendre 3 formes :

t = 1 qui correspond aux expressions de la forme *T* [*n*] = …, *n*, *n*:[*m*]…

t = v + 1 qui correspond à l'équation de la forme *T* = { *V* } et elle est transformée en une équation linéaire « 1 = t - v » ;

t = v qui correspond aux expressions de la forme *T* = *V* binop …, *T* = … binop *V*, *T* = *V* bitop …, *T* = $*V*, *T* = *V*@…, *T* [*V*] = …, *T* = *V* # … ; … *T* # *V* … Ces contraintes sont transformées en une équation linéaire homogène du type « 0 = v - t »

On obtient finalement un système d'équations linéaires N = A .D où N et C sont des vecteurs et C une matrice. D est le vecteur des dimensions et N un vecteur de 1 ou de 0 correspondant aux dimensions connues (celles des constantes qui valent 1). A est la matrice des contraintes entre dimensions. Cette matrice est une matrice creuse qui ne contient que des 1 et des -1.
On résout le système. Un *algorithme efficace* (plus efficace en temps et en mémoire que la résolution générale d'un système linéaire) est donné en annexe. Cet algorithme permet en plus de prendre en compte les contraintes générées par les expressions du type « *T*.n » qui ne sont pas linéaires. Trois cas peuvent se produire :

a) Le système n'a pas de solution : on déclare une erreur de géométrie, il y a des tissus dont la géométrie est dynamique.

b) Le système admet une infinité de solutions : le système n'est pas assez contraint pour inférer une géométrie unique pour chaque équation. Le programmeur doit ajouter des annotations.

c) Il y a une solution unique : on doit vérifier que chaque dimension est entière et positive Si oui, on passe à l'étape suivante de l'algorithme, sinon on déclare une erreur de géométrie

2ème étape :

On associe à chaque identificateur du programme *d* variables, où *d* est la dimension du tissu. On forme les équations qui doivent être vérifiées par ces variables. On obtient un système d'équations linéaires qu'il suffit de résoudre. 3 cas peuvent à nouveau se produire :

d) Le système n'a pas de solution : on déclare une erreur

e) Il y a une infinité de solutions : le programme est ambigu car on peut donner une valeur arbitraire à *n*-*r* cardinaux et déterminer en fonction de ceux-là les *r* cardinaux restants (*r* est le rang de la matrice A et *n* le nombre de cardinaux). Mais il se peut qu'il y ait une seule solution admettant des valeurs

entières strictement positives.

f) Le système a une seule solution : on doit vérifier que chaque cardinal est entier et strictement positif. Si c'est le cas, on a trouvé les cardinaux cherchés. Sinon on déclare une erreur.

Les règles correspondant à la formation des équations que doivent vérifier le i*ème* cardinal sont données en annexe du chapitre. Nous donnons à présent un exemple qui implique les principales constructions. Les constructions intéressantes sont celles qui permettent la navigation dans l'imbrication des tissus : l'accès à un point, l'alpha-notation, la réduction et le bloc. On remarquera au passage le traitement des fonctions. Enfin nous donnons des exemples de systèmes générant une erreur de typage.

II.3. Exemples de type-checking géométrique

On veut inférer et vérifier le typage géométrique du programme suivant :

(1) A = { 1, 2, 3, 4 } ;
(2) B = + \ A ;
(3) f (x) = { x } ;
(4) C = f ^ A ;
(5) D = f ^ C ;

Remarquons que *toutes* les annotations de géométrie ont été omises. Les expressions sont numérotées afin d'indiquer dans la suite la provenance des équations.

La première chose à faire est de calculer la hauteur de la géométrie de chaque expression :

(1) *hauteur*(A) = 1 une constante est toujours plate
(2) *hauteur*(B) = *hauteur*(A) = 1 la hauteur d'une expression est aussi la hauteur de ses sous-expressions sauf pour « . » et « { } »
(3) *hauteur*(f) = *f* = λx. x+1 la hauteur d'une fonction est une `fonction de la` hauteur de ses arguments
(4) C = *f* (*hauteur*(A)) = 2
(5) *hauteur*(D) = *f* (*hauteur*(C)) = 3

la hauteur de la géométrie d'une expression se calcule par propagation à partir de la hauteur des constantes (si un programme ne contient aucune constante alors on peut prouver que la valeur calculée par chaque tissu est ⊥). En effet, les équations de définition sont de la forme : « $h_1 = h_2$ », « $h_1 = 1$ » et « $h_1 = h_2 + 1$ ». Il n'y a donc absolument aucune difficulté à résoudre le système.

On forme ensuite le système des équations que doit vérifier les différents cardinaux :

(1) |A| = 4 par simple énumération : $|\{ x_1, \ldots, x_n \}| = n$
(2) |B| = 1 par définition d'une beta
(3) |f|(x) = λx.1 le cardinal d'une fonction est fonction du cardinal de ses arguments. Cette fonction est calculée comme : |f|(x) = λx . |f (X)| où X est tel que |X| = x
(4) |C| = |A| = 4 par définition d'une alpha , |f ^(A, …)| = |A|
(5) |D| = |C| = 4
(3) $|f|_2$ = λx. x car $|\{ X \}|_{n\geq 2} = |X|_{n-1}$. De manière générale, on a : $|f|_n(x)$ = λx . |f (X)|$_n$ où X est tel que $|X|_n = x$
(4) $|C|_2 = |f|(|A|) = 1$ par définition d'une alpha :
(5) $|D|_2 = |f|(|C|) = 1$ $|f \hat{}\ (A, \ldots)|_n = |f|_{n-1}(|A|_{n-1}, \ldots)$

(5) $|D|_3 = [f|_2(|C|_2) = 1$

D'où la détermination des géométries (il suffit d'aligner les cardinaux successifs d'un tissu) :

(1) [A] = [4]
(2) [B] = [1]
(4) [C] = [4 1]
(5) [D] = [4 1 1]

II.3.1. Exemple d'expressions incorrectement typées

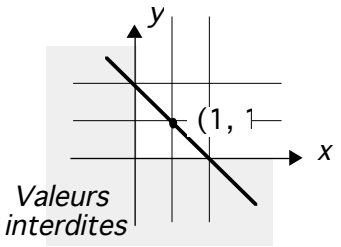
Voici à présent des exemples d'échec :

$T = \{ T \}$;	⇒	h = h + 1	⇒	pas de solution (cas a)
$T = \{ T.0 \}$;	⇒	h = h +1 - 1	⇒	une infinité de solutions (cas b)
$T = + \setminus T$;	⇒	h = h	⇒	une infinité de solutions (cas b)
$T = \{1\}.0.0$;	⇒	h = 1 - 1	⇒	une solution impossible (cas c)
$T = 0 \# T$;	⇒	t = 1 + t	⇒	pas de solution (cas d)
$T = 1 + \{ T.0 \}$;	⇒	t = t	⇒	une infinité de solutions (cas e)
$T = 0 \# U; V[1] = T;$	⇒	t = 1, u = 0	⇒	une solution impossible (cas f)

II.3.2. Un exemple de système non-réellement ambigu

Dans l'exemple suivant il y a une seule solution correcte parmi une infinité de solutions possibles (il y a une seule solution possible sur $\mathbb{N}^+$ mais une infinité sur ZZ) :

$$\begin{cases} X = 1 \\ Y = 1 \\ T[2] = X \# Y \end{cases} \qquad \begin{cases} 0 < x \quad \mathbb{N} \\ 0 < y \quad \mathbb{N} \\ 2 = x + y \end{cases}$$



Ce programme est déclaré ambigu par l'algorithme de typage proposé. Une analyse plus fine est possible mais semble trop coûteuse pour être mise en œuvre. On peut calculer une représentation de l'espace vectoriel des solutions (qui est de dimensions n-r, n étant le nombre de variables et r le rang du système). Il faut ensuite déterminer si cet espace vectoriel a une intersection avec le cône des entiers positifs et déterminer le cardinal de cette intersection. Des algorithmes existent [Feautrier 88] mais qui nous semblent trop coûteux pour le gain en confort réalisé par l'utilisateur et le thésard.

II.4. Évaluation de la faisabilité

Dans l'algorithme proposé, il faut résoudre un système linéaire. Il est donc important d'estimer la taille de la matrice A. Prenons un programme contenant des centaines d'équations (c'est un gros programme). L'imbrication des expressions va rarement dépasser 3 (Cf. par exemple les analyses quantitatives données dans [Henessy, Patterson 91]). On a donc 300 géométries à déterminer (c'est une hypothèse très pessimiste). De manière pessimiste, on peut supposer que chaque tissu est un tissu à quatre dimensions. En effet, les applications visées étant essentiellement la

simulation de systèmes dynamiques du monde physique, on trouvera les trois dimensions spatiales et le temps comme espace des paramètres d'un processus. On a donc $300.4 = 1200$ cardinaux à évaluer.

Le nombre moyen d'équations générées par une expression est approximativement de 1,5 (ce chiffre est obtenu en faisant la moyenne sur les expressions $81/2$ du nombre d'équations générées; c'est donc une mesure qui ne tient pas compte de la distribution réelle des opérateurs dans un programme). La matrice de calcul des cardinaux est donc une matrice 1800×1200. Cette matrice est **très** creuse. Si on regarde le nombre moyen d'arguments par opérateur $81/2$ (mesure statique) on obtient environ 1,5. Cela veut dire que sur une ligne de la matrice il y a en moyenne 1,5 coefficients non nuls. La résolution d'un système linéaire de cette taille peut sembler tout à fait hors de portée d'une station de travail. *Il n'en est rien.*

En effet, les équations à résoudre ont une forme particulière qui ne met en jeu que deux variables au plus à chaque fois (à une exception près). Il est donc possible de transformer la résolution de ce système d'équations en un problème équivalent consistant à calculer la longueur de tous les chemins d'un nœud donné vers tous les autres nœuds d'un graphe (des détails supplémentaires sont fournis dans l'annexe à ce chapitre). Il y a autant de nœuds que d'expressions et autant d'arcs que d'équations. Une estimation pessimiste de la taille du graphe est donc de 1200 nœuds et de 1800 arcs. L'algorithme de résolution peut se voir comme une descente en profondeur dans un graphe. La complexité de l'algorithme est donc proportionnelle au nombre d'arcs [AHU 83, *p 216*]. Celui-ci étant de l'ordre du millier, l'algorithme est non seulement praticable, mais a des temps de réponses compatibles avec une utilisation interactive.

III. Annexe au chapitre

III.1. Équations de définition de la dimension d'un tissu

Les dimensions des expressions d'un programme 81/2 sont calculées en résolvant un système d'équations linéaires. Chaque expression du programme génère zéro ou plusieurs équations. Nous donnons ci-dessous la liste des équations générées par chaque type d'expression. On note les expressions par des capitales, la dimension de l'expression par la minuscule correspondante.

E = n:[m]	1 = e	E = $F	0 = f - e
E = Id	0 = id - e	E = F fby G	$\begin{cases} 0 = f - e \\ 0 = g - e \end{cases}$
E = Id = F	0 = f - id	E = F synchro G	0 = f - e
E = F : [Id]	0 = e - id	E = F $\left\{\begin{matrix} \text{at} \\ \text{after} \\ \text{when} \\ \text{until} \end{matrix}\right\}$ G	$\begin{cases} 1 = g \\ 0 = f - e \end{cases}$
E = F.n	—		
E = F@n	0 = f - e		
E = F binop G	$\begin{cases} 0 = f - e \\ 0 = g - e \end{cases}$	E = { F, G, H, … }	$\begin{cases} 0 = f - e \\ 0 = f - g \\ 0 = f - h \\ \dots \end{cases}$
E = if F then G else H fi	$\begin{cases} 0 = f - e \\ 0 = g - e \\ 0 = h - e \end{cases}$	E = F # G	$\begin{cases} 0 = f - e \\ 0 = f - g \end{cases}$
E = F(G)	0 = g + f - e	E = Id every Jd where F	0 = id - e

III.1.1. Traitement des fonctions

Une fonction correspond à un système d'équations paramétré par les arguments de la fonction. Par exemple

Fct (X, Y) = { X } # Y ; → S(x, y) = $\begin{cases} fct = y \\ y = x + 1 \end{cases}$

On note *fct* la dimension de l'application de Fct à deux arguments X et Y de dimensions x et y respectivement. Ce système paramétré est obtenu uniquement à partir du corps de l'expression. Il y a quatre manières d'appliquer une fonction, ce qui conduit à quatre instanciations du système S(x, y)

E = Fct (x, y) → e = *fct*, S(x, y)

E = Fct ^ (X, Y) → e = *fct* +1, S(x-1, y-1)

E = Fct \ (X) → S (x-1, y-1)

E = Fct \\ (X) → e = *fct* + 1, S(x-1, y-1)

Les équations correspondent à l'application d'une fonction sont adjointes au système d'équations du programme, après instanciation des variables en jeu. Remarquons qu'une dimension de 0 correspond à un scalaire.

III.1.2. Les contraintes non-linéaires

Pour que le système d'équations obtenu soit un système d'équations linéaires, nous n'avons pas associé d'équation à une observation spatiale. En effet, si *T* est un tissu plat, alors *T*.0 est encore un tissu plat, alors que si *Q* est un tissu complexe de dimension n > 1, *Q*.0 est un tissu complexe de dimension n-1. La contrainte à vérifier est donc :

E = F . n → e = si f == 1 alors 1 sinon f -1 (C1)

qui n'est pas linéaire. Par ailleurs, nous aurions pu rajouter la contrainte suivante sur une beta :

E = Fct \ (X) → e = si *fct* == 0 alors 1 sinon *fct* (C2)

qui correspond à la conversion implicite d'un scalaire avec la collection de dimension 1.

Remarquons que le nombre d'équations est bien supérieur au nombre d'inconnues, donc, la plupart du temps, l'élimination des équations non-linéaires ne change rien à la détermination de la valeur des inconnues. Si le système linéaire n'a pas de solution, le système total n'a pas de solution non plus. Une fois le système linéaire résolu on peut calculer la dimension de e. Si E apparaît ailleurs, on obtient deux valeurs pour e : celle calculée lors de la résolution du système linéaire et celle calculée à partir de la valeur de f et de (C1) ou (C2). Il faut vérifier que ces deux valeurs sont compatibles. Si elles ne le sont pas, il y a une erreur de typage.

III.1.3. Résolution efficace du système complet d'équations

On peut remarquer que toutes les équations sont de la forme $x_i - x_j = a_{i,j}$ ou bien $x_i = a_i$, sauf pour la sélection. Ce n'est pas une grande contrainte que de forcer la sélection à prendre son deuxième argument parmi les tissus plats. Alors, toutes les équations ont la forme citée.

Dans ce cas il y a une représentation naturelle des contraintes sur les x_i. On associe à chaque x_i un nœud v_i et à chaque contrainte $x_i - x_j = a_{i,j}$ un arc $v_i \rightarrow v_j$ et de poids $a_{i,j}$. Ajoutons à ce graphe un nœud particulier, v_{-1}, et des arcs de poids a_i de v_{-1} vers x_i pour chaque contrainte de la forme $x_i = a_i$.

La valeur de x_i cherchée est alors la longueur d'un chemin de v_{-1} vers x_i. De plus, il faut que la longueur de chaque chemin de v_{-1} vers x_i et qui ne contient pas de cycle soit identique. Ce calcul et la vérification peuvent se faire rapidement grâce à une descente récursive dans le graphe. Par exemple :

```
type&check () :
   Pour i ← 0 à n faire x_i ← ⊥, u_i ← false
   descente (v_-1)
   Pour chaque v_j  ≠ v_-1 faire
        si x_i = ⊥ alors ERREUR Ambiguïté
```

Le drapeau u_i est associé au nœud v_i et permet de signaler la visite du nœud par la procédure type&check.

```
descente (v_i) :
   si u_i = false alors
        u_i = true
        Pour chaque arc v_i, j faire
             si x_j ≠ ⊥ alors
                  si x_j ≠ x_i + a_i, j alors ERREUR Type Checking
   (1)
             sinon
                  x_j ← x_i + a_i, j                                    (1)
                  si x_j ≤ 0 alors ERREUR Type Checking
                  descente (v_j )
   sinon
   /* on a déjà visité le nœud, il suffit de vérifier l'égalité des chemins
*/
        si x_j ≠ x_i + a_i, j alors ERREUR Type Checking
   (1′)
```

Cet algorithme présente de plus l'opportunité de réintégrer les contraintes non-linéaires que nous avions écartées. Il suffit de rajouter à l'arc représentant la contrainte non-linéaire un drapeau $nl_{i,j}$ et de remplacer les lignes (1) et (1′) par :

```
si nl_i, j alors
   si (((x_i =1) & (x_j ≠1)) OR ((x_i > 1) & (x_j ≠ x_i - 1)) ) alors ERREUR Type
Checking
sinon
   si x_j ≠ x_i + a_i, j alors ERREUR Type Checking
```

et la ligne (2) par :

```
si nl_i, j alors
   si x_i = 1 alors  x_j ← 1 sinon x_j ← x_i - 1
sinon
   x_j ← x_i + a_i, j
```

III.2. Équations de définition des cardinaux d'un tissu

Les cardinaux successifs des expressions d'un programme 81/2 sont calculés en résolvant un système d'équations linéaires. Chaque expression du programme génère une ou plusieurs équations. Nous donnons ci-dessous la liste des équations générées par chaque type d'expression. On note les expressions par des capitales, les cardinaux successifs par la minuscule correspondante indicée par le rang du cardinal. On calcule autant de cardinaux successifs que de dimensions de l'expression.

III.2.1. Traitement des fonctions

Une fonction correspond à un système d'équations paramétré par les (cardinaux successifs des) arguments de la fonction. Par exemple

$$\text{Fct (X, Y)} = \{ \text{X} \} \# \text{Y} ; \quad \rightarrow \quad S(x_1, y_1, x_2, y_2, \ldots) = \begin{cases} 0 = fct_1 - 1 - y_1 \\ 0 = fct_n - y_n , n>1 \\ 0 = y_n - x_{n-1} , n> 1 \end{cases}$$

On note fct_n le n*ième* cardinal de l'application de Fct à deux arguments X et Y de dimension x et y respectivement. Ce système paramétré est obtenu uniquement à partir du corps de l'expression. Il y a quatre manières d'appliquer une fonction, ce qui conduit à quatre instanciations du système S(…)

E = Fct (x, y) $\quad e_n = fct_n, S(x_1, y_1, \ldots)$

E = Fct ^ (X, Y) $\quad e_1 = x_1, x_1 = y_1, S(x_1, y_1, \ldots)$

E = Fct \ (X)	$e_1 = 1, S(x_1, x_1, x_2 , x_2 , \ldots)$
E = Fct \\ (X)	$e_1 = x_1, S(x_1, x_1, x_2 , x_2 , \ldots)$

III.2.2. Résolution efficace du système complet

Mis à part la sélection et la première équation associée à « E = F # G », toutes les équations ont la forme $x_i - x_j = a_{i,j}$ où bien $x_i = a_i$. On peut appliquer la contrainte que l'argument « destination » d'une sélection doit être plat et ignorer pour un temps la contrainte (C3) : « $0 = f_1 + g_1 - e_1$ » générée par l'expression « E = F # G ». On peut alors résoudre le système efficacement, en utilisant la méthode précédemment exposée. À la fin de la descente, trois cas peuvent se produire :

– Une erreur a été déclarée. Le système partiel (i.e. sans les contraintes de type C3) n'a pas de solution. Il en va de même pour le système total et donc la détection d'une erreur est correcte.

– Aucune erreur n'a été détectée et il ne reste pas de variable sans valeur. Il suffit de vérifier que les contraintes de type C3 sont vérifiées par les valeurs trouvées. Si c'est le cas, le typage a abouti, sinon on déclare une erreur.

– Aucune erreur n'a été détectée et il reste des variables sans valeur. On considère alors le système général dans lequel on remplace toutes les variables à présent connues par leur valeur. On obtient ainsi un système linéaire de taille réduite qu'on peut résoudre par une méthode générale. (Ou bien on peut déclarer le programme ambigu ce qui contraint le programmeur à ajouter des annotations.)

E = n:[m]	$\begin{cases} m = e_1 \\ 0 = e_n , n> 1 \end{cases}$	E = $F	$0 = f_n - e_n$
E = Id	$0 = id_n - e_n$	E = F fby G	$\begin{cases} 0 = f_n - e_n \\ 0 = g_n - e_n \end{cases}$
E = F : [Id]	$\begin{cases} 0 = id_1 - e_1 \\ 0 = f_n - e_n , n>1 \end{cases}$	E = F synchro G	$0 = f_n - e_n$
E = Id = F	$0 = f_n - id_n$	E = F {at, after, when, until} G	$\begin{cases} 1 = g_1 \\ 0 = g_n, n> 1 \\ 0 = f_n - e_n \end{cases}$
E = F.n	$0 = f_{n+1} - e_n,$		
E = F@n	$0 = f_n - e_n$	E = { $F^1, \ldots, F^p$ }	$\begin{cases} p = e_1 \\ 0 = f^1_n - e_{n+1}, n> 1 \\ 0 = f^1_n - f^i_n , 1<i\leq p \end{cases}$
E = F binop G	$\begin{cases} 0 = f_n - e_n \\ 0 = g_n - e_n \end{cases}$		
E = if F then G else H fi	$\begin{cases} 0 = f_n - e_n \\ 0 = g_n - e_n \\ 0 = h_n - e_n \end{cases}$	E = F # G	$\begin{cases} 0 = f_1 + g_1 - e_1 \\ 0 = f_n - g_n , n>1 \\ 0 = f_n - e_n, n>1 \end{cases}$
E = F(G)	$0 = g_n + f_n - e_n$	E = Id every Jd where F	$0 = id_n - e_n$

V. Le calcul des dépendances

L'exécution d'un programme consiste à effectuer des opérations élémentaires dans un certain ordre. Une opération élémentaire en 81/2 correspond au calcul de la valeur d'un point à un tic donné. Si l'ordre entre opérations est total, le programme est purement séquentiel; si l'ordre est partiel, il est possible de réaliser en parallèle des opérations qui ne sont pas comparables.

Le but du typage de l'ordonnancement est de déterminer s'il existe un ordre partiel $\Re$ entre les opérations d'un programme 81/2. Nous demandons de plus que $\Re$ vérifie deux propriétés :

- $\Re$ est *statique* : il est indépendant d'une exécution particulière du programme;
- $\Re$ est *sans recouvrement* : les opérations correspondant à un tic donné précèdent strictement les opérations concernant les tics ultérieurs.

Ces deux propriétés seront expliquées un peu plus loin. La recherche de cet ordre partiel est liée à l'analyse des dépendances entre opérations.

Le calcul des dépendances

Deux opérations sont dépendantes si l'une utilise le résultat de l'autre pour sa propre réalisation. Dans un langage séquentiel comme Pascal ou FORTRAN, l'ordre des opérations est fixé de manière indépendante de leurs dépendances. Par exemple en Pascal, l'ordre des instructions est spécifié grâce au « **;** » qui est l'opérateur de séquencement des instructions. On parle traditionnellement de « vraie dépendance » entre une opération O_1 *qui précède une opération* O_2, si O_2 utilise en lecture une variable référencée par O_1 en écriture; O_1 est « anti-dépendant » de O_2 dans le cas contraire; et enfin, si O_1 et O_2 utilisent la même variable en écriture, on parle de « dépendance de sortie ». Dans ces trois cas, on ne peut intervertir l'ordre d'exécution de O_1 et O_2 sans changer la sémantique du programme [Berstein 66]. Un des buts du compilateur est de déterminer quelles sont les opérations indépendantes, afin de ne pas tenir compte du séquencement explicite et de les paralléliser (c'est le problème de l'extraction du parallélisme).

Dans un langage data-flow l'ordre des opérations n'est pas explicité par le programmeur : *il n'y a pas d'opérateur de séquencement*. De plus, la valeur d'une variable est définie par une seule opération (c'est ce que recouvre le terme de langage à assignation unique ou encore de transparence référentielle). En conséquence il n'y a que des « vraies

dépendances » entre opérations et leurs analyses sont simples. Il faut noter que l'analyse à la compilation des dépendances d'un programme data-flow est utile uniquement si on désire l'ordonnancement statique des opérations du programme. Cela n'est pas nécessaire pour une exécution sur les machines data-flow qui fournissent un support matériel pour l'ordonnancement dynamique des instructions [Plas *& al*, 76].

Si l'on excepte les itérations, une opération dans un langage data-flow comme VAL ou SISAL correspond à une expression. Par suite, l'analyse des dépendances se ramène à la recherche des occurrences d'une variable dans une expression. Les itérations, ou le concept de stream, compliquent l'analyse car plusieurs opérations correspondent à la même expression. En 81/2 une difficulté supplémentaire est ajoutée par la dimension spatiale d'un tissu : une expression dénote plusieurs opérations correspondant à la suite des valeurs *et à la collection* des valeurs. Cependant l'itération temporelle et spatiale des opérations est réalisée « de manière mathématiquement respectable », [Wadge, Ashcroft 85], ce qui permet toujours une analyse beaucoup plus simple que dans le cas d'un langage comme FORTRAN[1].

Ordonnancement statique et cyclique

Le but du typage de l'ordonnancement est de déterminer s'il existe un ordre partiel entre les opérations d'un programme 81/2 qui soit indépendant des valeurs produites par une exécution particulière du programme. Par exemple,

R = si *A* alors *S* sinon *B* fi;
S = si *A* alors *B* sinon *R* fi;

n'est pas un programme qui admet un ordonnancement qui soit indépendant de l'exécution : si à un top donné, *A* a la valeur vraie, il faut d'abord calculer *B* puis *S* et enfin *R*. Si la valeur de *A* est fausse, il faut calculer dans l'ordre *B* puis *R* puis *S*. L'ordonnancement n'est pas *statique*.

Les opérations à exécuter correspondent à un tic donné. Un ordonnancement est sans recouvrement si les opérations du tic t précèdent strictement les opérations d'un tic $t' > t$.

Un ordonnancement des opérations est *complètement statique* si c'est un ordonnancement statique sans recouvrement. Il faut de plus que l'ordonnancement des opérations d'un tic ne dépende pas de ce tic (Cf. figure 1).

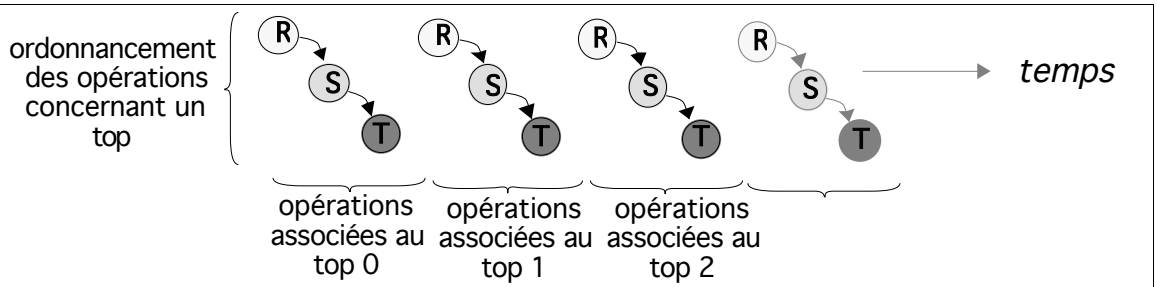


Figure 1 complètement statique : un ordonnancement est complètement statique si l'ordonnancement des opérations respecte l'ordre des tops.

Un ordonnancement complètement statique présente l'avantage suivant : il suffit d'établir un ordonnancement *O* des opérations associées au calcul d'un tic t. L'ordonnancement total est obtenu par répétition de l'ordonnancement *O* où les opérations concernant le tic t+1 viennent se substituer aux opérations concernant le tic t. Les opérations concernant le tic t sont en nombre fini, alors que le nombre total d'opérations est éventuellement

[1] On trouvera une introduction au problème de l'analyse des dépendances d'un programme FORTRAN dans [Zima 91, *chap 4*]. Les techniques en jeu font souvent appel à des méthodes d'analyse exacte ou approchée de l'ensemble des solutions entières d'un système d'équations (ou d'inéquations) linéaires, Cf. par exemple [Feautrier 91].

infini.

Un ordonnancement complètement statique présente le désavantage de ne pas être « aussi partiel que possible ». En effet, un ordonnancement complètement statique vérifie que les dernières opérations du tic t doivent précéder les premières opérations du tic t+1, alors que cette contrainte n'est pas forcément requise par le programme. Nous donnons un exemple d'ordonnancement statique qui n'est pas complètement statique dans la partie de ce chapitre réservée aux exemples.

Dans ce chapitre, nous nous intéressons au problème d'établir s'il existe un ordonnancement complètement statique d'un programme 81/2 donné. La présentation sera informelle (un exemple d'approche formelle du typage sera donné dans le chapitre suivant).

I. Le graphe des dépendances

Pour ordonner (de manière complètement statique) les opérations correspondant à l'exécution d'un programme, il suffit de savoir ordonner les opérations concernant un top donné. Une opération élémentaire est associée au calcul de la valeur d'un point d'un tissu. Le graphe des dépendances entre opérations correspond donc à un graphe des dépendances entre points.

Il n'est pas possible de représenter explicitement toutes les dépendances entre chaque point car le nombre de points d'un tissu peut être arbitrairement grand. Par contre on peut représenter explicitement des dépendances entre tissus car les tissus sont créés par un humain et donc leur nombre reste petit. On va donc essayer de déterminer un graphe *G* des dépendances entre tissus tel que le graphe *P* des dépendances entre points soit « inclus » dans *G*. Si ce dernier ne contient pas de cycle, alors *P* ne contient pas de cycle non plus. Le graphe des dépendances *G* (ou graphe d'ordonnancement) sera alors la représentation sagittale d'une relation d'ordre.

Il faut que *G* ne soit pas une approximation trop grossière de *P*. Par exemple il ne faut pas que

$$f(x) = x + 1; \qquad (E1)$$
$$T = (1 \# f(T)) : [1000]$$

soit rejeté comme incorrect. En effet, ce schéma de programme correspond à une itération bornée séquentielle et représente une structure de contrôle importante. Il admet un ordonnancement du calcul des points d'un tissu bien que le graphe des dépendances présente un cycle (*T* dépend directement de *T*).

On va donc affiner le graphe d'ordonnancement entre tissus en distinguant plusieurs sortes des dépendances entre tissus :

- *Dépendance point à point*

  Un tissu *T* dépend point à point d'un tissu *S* si la valeur de chaque point de *T* dépend uniquement de la valeur du point correspondant de *S*, comme par exemple dans « $T = S$ ». On notera $T \xrightarrow{p} S$.

- *Dépendance positive*

  Un tissu *T* dépend positivement d'un tissu *S* si la valeur de chaque point *i* de *T* dépend de la valeur des points $j < i$ de *S*. C'est le type de dépendance qui relie *T* et *S* dans « $T = Q \# S$ ». On notera $T \xrightarrow{+} S$.

- *Dépendance totale*

  Un tissu *T* dépend totalement d'un tissu *S* si la valeur d'un point *i* de *T* dépend de la valeur de chaque point *j* de *S*. Un exemple est donné par la somme de tous les éléments d'un tissu « $T = + \backslash S$ ». On notera $T \xrightarrow{t} S$.

Le graphe d'ordonnancement des tissus représente une approximation du graphe des dépendances entre points. En effet, le graphe des dépendances entre points est un sous-graphe du graphe obtenu par « dépliage » du graphe

d'ordonnancement. Ce dépliage s'obtient de la manière suivante :

- Chaque sommet représentant un tissu T est remplacé par $|T|$ sommets représentant les points de T.
- Les arcs $T \xrightarrow{p} S$ sont remplacés par des arcs entre le sommet $T.i$ et le sommet $S.i$.
- Les arcs $T \xrightarrow{+} S$ sont remplacés par des arcs entre le sommet $T.i$ et les sommets $S.j$ pour tous les $j < i$.
- Les arcs $T \xrightarrow{t} S$ sont remplacés par des arcs entre le sommet $T.i$ et tous les sommets $S.j$.

Ce dépliage est illustré par la figure 2 ci-dessous :

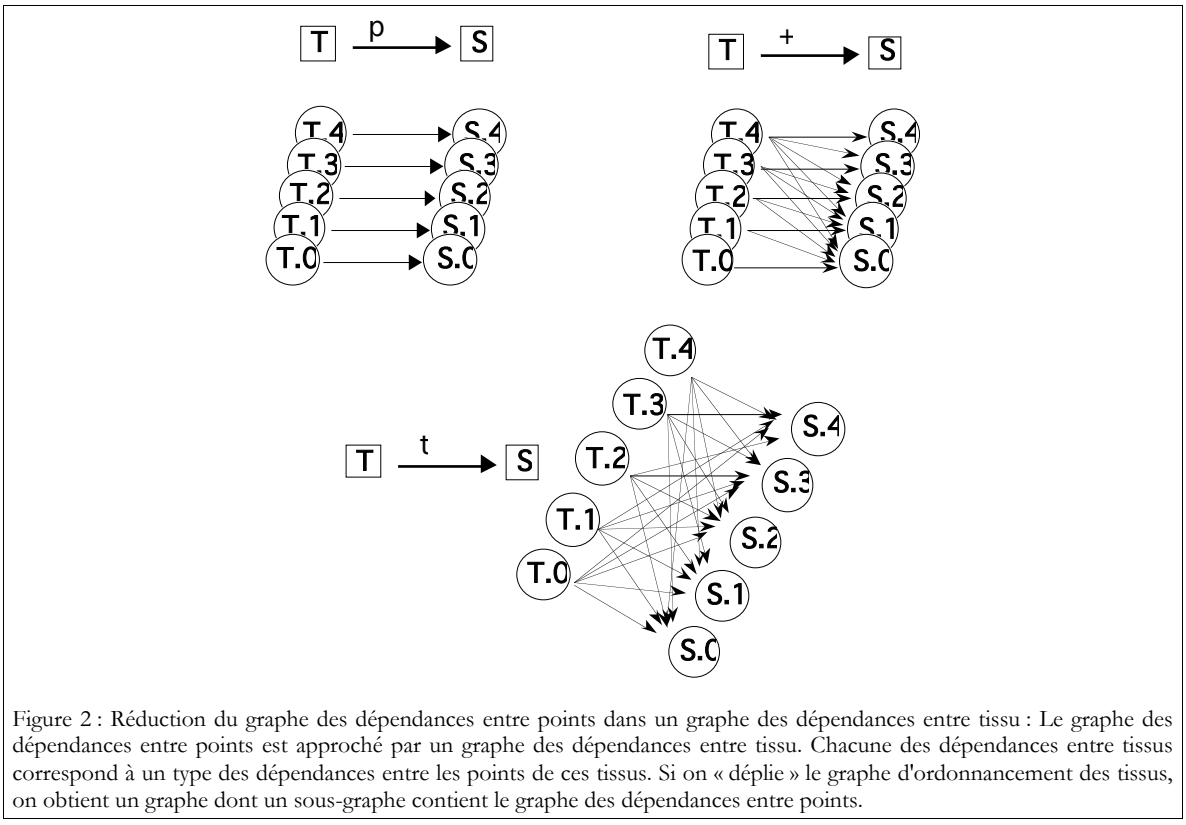


Figure 2 : Réduction du graphe des dépendances entre points dans un graphe des dépendances entre tissu : Le graphe des dépendances entre points est approché par un graphe des dépendances entre tissu. Chacune des dépendances entre tissus correspond à un type des dépendances entre les points de ces tissus. Si on « déplie » le graphe d'ordonnancement des tissus, on obtient un graphe dont un sous-graphe contient le graphe des dépendances entre points.

Si un tissu R dépend d'un tissu S et ce dernier d'un tissu T alors R dépend aussi de S. La composition des trois sortes de dépendance est définie par les règles suivantes :

$$
\begin{array}{llll}
R \xrightarrow{p} S & \text{et} & S \xrightarrow{p} T & \Rightarrow \quad R \xrightarrow{p} T \\
R \xrightarrow{p} S & \text{et} & S \xrightarrow{+} T & \Rightarrow \quad R \xrightarrow{+} T \\
R \xrightarrow{p} S & \text{et} & S \xrightarrow{t} T & \Rightarrow \quad R \xrightarrow{t} T \\
\\
R \xrightarrow{+} S & \text{et} & S \xrightarrow{p} T & \Rightarrow \quad R \xrightarrow{+} T \\
R \xrightarrow{+} S & \text{et} & S \xrightarrow{+} T & \Rightarrow \quad R \xrightarrow{+} T \\
R \xrightarrow{+} S & \text{et} & S \xrightarrow{t} T & \Rightarrow \quad R \xrightarrow{t} T \\
\\
R \xrightarrow{t} S & \text{et} & S \xrightarrow{p} T & \Rightarrow \quad R \xrightarrow{t} T \\
R \xrightarrow{t} S & \text{et} & S \xrightarrow{+} T & \Rightarrow \quad R \xrightarrow{t} T \\
R \xrightarrow{t} S & \text{et} & S \xrightarrow{t} T & \Rightarrow \quad R \xrightarrow{t} T
\end{array}
$$

Cependant on a pas besoin de faire apparaître toutes les relations obtenues par transitivité : le graphe des dépendances complet s'obtient par fermeture transitive en suivant les lois de composition précédente.

Si un tissu T dépend point à point d'un tissu S, alors le calcul de la valeur du point i de S doit précéder le calcul

de la valeur du point *i* de *T* (mais les calculs de *S* et de *T* peuvent s'entrelacer). Si un tissu *T* dépend totalement d'un tissu *S*, le premier calcul concernant une valeur de *T* doit prendre place après le dernier calcul concernant une valeur de *S*. Si la dépendance est positive, ce sont les *i* premiers calculs de *S* qui doivent avoir été achevés avant de commencer le *i*ème calcul de *T*.

Si un tissu ne dépend pas de lui-même (il n'y a pas de cycle impliquant le sommet associé au tissu) alors on peut calculer les valeurs des points du tissu en parallèle. Sous certaines conditions, un tissu peut dépendre de lui-même. Dans ce cas la dépendance est une dépendance positive : la valeur d'un point *i* dépend de la valeur des points $j < i$ de ce tissu. La valeur des points d'un tissu dépendant positivement de lui-même doit donc être calculée dans l'ordre des points d'index croissants.

I.1. La construction du graphe des dépendances

Les règles de construction du graphe des dépendances associé à une expression sont données sous la forme de formules « EO ⩦ e → G » qui se lisent : « avec les hypothèses EO, le graphe des dépendances de l'expression e est G ». Un tel graphe d'ordonnancement se décrit par un quadruplet $G = (G_v, G_p, G_+, G_t)$. G_v est l'ensemble des sommets du graphe, G_p correspond à l'ensemble des arcs point à point. C'est une partie de $G_v \times G_v$ telles que si $(v, w) \in G_p$ alors $v \xrightarrow{}_p w$. G_+ et G_t dénotent respectivement les arcs positifs et totaux du graphe G.

L'union de deux graphes F et G se définit par : $F \cup G = (F_v \cup G_v, F_p \cup G_p, F_+ \cup G_+, F_t \cup G_t)$. On notera par $\{ v \xrightarrow{}_p w \}$ le graphe ($G_v = \{v, w\}$, $G_p = \{(v, w)\}$, $G_+ = \varnothing$, $G_t = \varnothing$). Les graphes $\{ v \xrightarrow{}_+ w \}$ et $\{ v \xrightarrow{}_t w \}$ se construisent de manière analogue avec respectivement G_+, et G_t.

On va attacher un sommet à chaque expression et sous-expression. Le sommet attaché à une expression e se note v_e. On ne représente pas explicitement toutes les dépendances : si $v_1 \rightarrow v_2$ et $v_2 \rightarrow v_3$ alors on a bien sûr aussi une dépendance $v_1 \rightarrow v_3$ obtenue par la fermeture des trois relations partielles dans le graphe des dépendances.

EO dénote un environnement : le type d'une expression dépend du type des variables qui interviennent dans l'expression. L'environnement EO permet de donner un type à chaque variable d'une expression. Il faut remarquer que dans la règle EO ⩦ e → G il est implicitement supposé que EO définit un type pour au moins chaque variable libre de e. Nous décrivons le typage sur un sur-ensemble du langage L1. Nous observerons les conventions suivantes

variable	*type d'objet*
EO	environnement des graphes d'ordonnancement
id	identificateur
F, G, H	ordonnancements
e, f, g	expressions

Type d'un identificateur :

$$EO, id \rightarrow G \Subset id \rightarrow G \qquad (OT0)$$

Liaison variable - valeur :

$$\frac{EO, id \varnothing G \Subset e \varnothing G}{EO \Subset id=e \varnothing G \approx \{ v_{id} \xrightarrow{}_p \varnothing v_e \}} \qquad (OT1)$$

le graphe lié à « id = e » est le graphe de e augmenté du sommet id et de l'arc allant de v_{id} au sommet v_e (en examinant les règles on trouvera que le sommet associé à une expression est toujours présent dans le graphe associé à cette expression). La dépendance entre un identificateur et l'expression qui est dénotée, est une dépendance point à point (on n'a qu'à penser que la liaison entre une expression et un identificateur correspond à une copie des valeurs

de l'expression; la copie peut se faire point à point).

Typage des constantes :

$$EO \Subset \text{Clock} \rightarrow (\{v_{const}\}, \varnothing, \varnothing, \varnothing, \varnothing) \qquad EO \Subset n \rightarrow (\{v_{const}\}, \varnothing, \varnothing, \varnothing, \varnothing) \qquad \text{(OT2, 3)}$$

Une constante ne dépend de rien. Mais on introduit un sommet v_{const} afin de rester homogène avec le comportement des autres expressions.

Arithmétique :

$$\frac{EO \Subset e \varnothing G \quad EO \Subset f \varnothing F}{EO \Subset e \text{ binop } f \varnothing G \approx F \approx \{v_{e \text{ binop } f\ p} \varnothing v_e\} \approx \{v_{e \text{ binop } f\ p} \varnothing v_f\}}$$

$$\frac{EO \Subset e \varnothing H \quad EO \Subset f \varnothing F \quad EO \Subset g \varnothing G}{EO \Subset \text{si } e \text{ alors } f \text{ sinon } g \text{ fi} \varnothing H \approx F \approx G \approx \{v_{\text{si } e \ldots\ p} \varnothing v_e\} \approx \{v_{\text{si } e \ldots\ p} \varnothing v_g\} \approx \{v_{\text{si } e \ldots\ p} \varnothing v_f\}}$$

Le résultat d'une opération arithmétique dépend point à point du résultat de ses arguments.

Observateurs :

$$\frac{EO \Subset e \varnothing G}{EO \Subset e \,.\, i \varnothing G \approx \{v_{e.i\ t} \varnothing v_e\}} \qquad \frac{EO \Subset e \varnothing G}{EO \Subset e@i \varnothing G \approx \{v_{e@i\ p} \varnothing v_e\}}) \qquad \text{(OT5, 6)}$$

Le résultat d'une observation spatiale dépend du résultat de tous les points : en effet, le fait de connaître le point observé n'apporte pas d'information supplémentaire du point de vue des relations de dépendance que nous construisons. La valeur de l'observation ne pourra être disponible qu'à partir du moment où toutes les valeurs du tissu sont disponibles.

Application de fonction :

$$\frac{\begin{array}{c} f(a, b) = e \\ EO, a \varnothing F, b \varnothing H \Subset e \varnothing G \\ EO \Subset g_1 \varnothing F, \quad EO \Subset g_2 \varnothing H \end{array}}{EO \Subset f(g_1, g_2) \varnothing G \approx \{v_{f(g1, g2)\ p} \varnothing v_e\}} \qquad \text{(OT7)}$$

Le nœud v_e qui apparaît au dénominateur est le sommet de G correspondant à l'expression e. La notation « EO, a → F, b → H » indique un environnement où le graphe des dépendances de l'identificateur a est F et celui de b est H; le graphe des dépendances des autres identificateurs est décrit par EO. La première des clauses, « f (a, b) = e », n'est pas une prémisse mais est présente pour indiquer les conditions d'application de la règle (i.e. f est une fonction dont les paramètres formels sont a et b, et le corps, e). Cette règle indique que si G est le graphe des dépendances de e, avec pour hypothèse que les graphes des dépendances de a et de b sont F et H respectivement, alors le graphe des dépendances de « $f(g_1, g_2)$ » est G, si les graphes des dépendances de g_1 et de g_2 sont F et H respectivement.

Autrement dit, le graphe des dépendances d'une application de fonction correspond au graphe des dépendances de l'expression définissant la fonction, dans lequel on substitue les sommets correspondant aux paramètres formels par les sommets correspondant aux expressions qui sont les paramètres réels (Cf. figure 3).

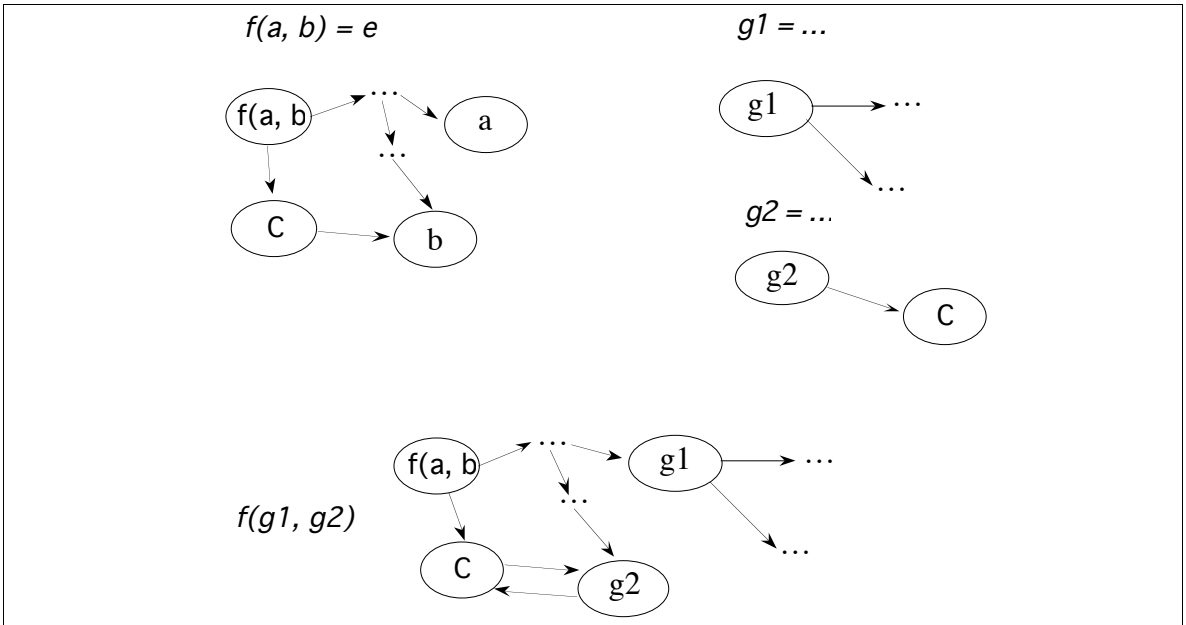


Figure 3 : instanciation d'une fonction. Le graphe des dépendances obtenu est composé du graphe de la fonction dans lequel les paramètres formels ont été remplacés par les sommets correspondant aux expressions qui sont les paramètres réels.

Alpha-notation

$$\frac{\begin{array}{c} f(a, b) = e \\ EO, a \varnothing F, b \varnothing H \trianglelefteqslant e \varnothing G \\ EO \trianglelefteqslant g_1 \varnothing F\ ,\quad EO \trianglelefteqslant g_2 \varnothing H \end{array}}{EO \trianglelefteqslant f\hat{}(g_1, g_2) \varnothing G \approx \{v_{f\hat{}(g1, g2)\ p} \varnothing v_e\}} \qquad \text{(OT8)}$$

Cette règle exprime que la valeur d'une alpha peut être calculée en un point donné quand la valeur de la fonction f appliquée aux points correspondants des arguments est disponible. Pour voir en quoi l'alpha est identique à l'application de fonction, supposons que f soit une dépendance point à point de ses arguments (par exemple f(x, y) = x + y) alors la valeur de l'alpha en un point donné est disponible quand la valeur des points correspondants des paramètres réels est disponible.

Beta-réduction

Erreur !) (OT9)

La valeur d'une beta n'est disponible que quand les valeurs de tous les points de l'argument sont disponibles.

Projection, broadcast et sélection

$$\frac{\begin{array}{c} n \leq |e| \\ EO \trianglelefteqslant e \varnothing G \end{array}}{EO \trianglelefteqslant e : [n] \varnothing G \approx \{v_{e:[n]\ p} \varnothing v_e\}} \qquad \frac{\begin{array}{c} n > |e| \\ EO \trianglelefteqslant e \varnothing G \end{array}}{EO \trianglelefteqslant e : [n] \varnothing G \approx \{v_{e:[n]\ t} \varnothing v_e\}} \qquad \text{(OT10)}$$

La valeur d'une projection dépend point à point de la valeur d'un argument. La valeur d'un broadcast dépend totalement de la valeur de l'argument.

L'accessibilité de la valeur d'une sélection « e(f) » dépend de la valeur des points de e. Une première modélisation des dépendances pourrait donc être :

$$\frac{EO \trianglelefteqslant e \varnothing G}{EO \trianglelefteqslant e\ (f) \varnothing G \approx \{v_{e\ (f)\ t} \varnothing v_e\}} \qquad \text{(OT10')}$$

mais cette modélisation présente l'inconvénient d'être trop grossière, en particulier quand la sélection est utilisée pour

réaliser un schéma itératif similaire à (E1) :

T [10] = f^ (0 # T (id), $\mathcal{Q}$) ;
id [9] = (+ \\ 1) -1 ; (id = { 0, 1, 2, 3, 4, 5, 6 7, 8 })

En fait, dans « e(f) » l'argument f est une constante. Il est donc toujours possible de connaître les valeurs de f et de raffiner la dépendance. En particulier, on dira que f est *lipschitzienne* si : f.i < i. Dans ce cas,

(e(f)).i = e.(f.i) = e.j avec j < i

et par suite e(f) dépend positivement de e : $\dfrac{\begin{array}{c}f \textit{ lipschitzienne} \\ EO \trianglelefteqslant e \varnothing G\end{array}}{EO \trianglelefteqslant e(f) \varnothing G \approx \{v_{e(f)} {}_{+}\varnothing v_e\}}$ (OT10″)

Retard

$$EO \trianglelefteqslant \$e \rightarrow (\{v_{\$e}\}, \varnothing, \varnothing, \varnothing, \varnothing,) \qquad \text{(OT11)}$$

La valeur d'un tissu retardé ne dépend de rien. En effet, nous sommes dans le cas des ordonnancements complètement statiques et par suite, les valeurs des tops précédents sont toutes disponibles.

Opérations temporelles dyadiques

$$\frac{\begin{array}{c}EO \trianglelefteqslant e \varnothing G \\ EO \trianglelefteqslant f \varnothing F\end{array}}{EO \trianglelefteqslant e \text{ binop } f \varnothing G \approx F \approx \{v_{e \text{ bitop } f} {}_{p}\varnothing v_e\} \approx \{v_{e \text{ bitop } f} {}_{p}\varnothing v_f\}} \quad \text{(OT11)}$$

bitop désigne l'une des quelconques opérations temporelles : fby, when, after, until *et* synchro.

Bloc

$$\frac{EO \trianglelefteqslant e \varnothing G}{EO \trianglelefteqslant \{ e \} \varnothing G \approx \{ v_{\{e\}} {}_{p}\varnothing v_e\}} \qquad \text{(OT12)}$$

Cette règle n'est pas évidente: a priori, la valeur d'un point d'un bloc (il y en a en l'occurrence un seul) n'est disponible que quand toutes les valeurs de e (qui constituent *la* valeur du point) sont disponibles. La dépendance entre « {e} » et « e » devrait donc être une dépendance totale. Cette condition est néanmoins trop forte. Si on raisonne en termes du graphe « déplié » des dépendances entre points, la *i*ème valeur du dépliage de « {e} » est aussi la *i*ème valeur du dépliage de e (le dépliage aplatit toute la hiérarchie). Il suffit donc d'une dépendance point à point.

Agrégation

$$\frac{\begin{array}{c}EO \trianglelefteqslant e \varnothing G \\ EO \trianglelefteqslant f \varnothing F\end{array}}{EO \trianglelefteqslant e \# f \varnothing G \approx F \approx \{v_{e \# f} {}_{p}\varnothing v_e\} \approx \{v_{e \# f} {}_{+}\varnothing v_f\}} \qquad \text{(OT13)}$$

Pour un i > |e|, (e # f).i = f.(i - |e|) et par suite les points de e # f dépendent positivement de ceux de f.Par contre pour i < |e|, (e # f).i = e.i et donc les points de e # f dépendent point à point de e. Remarquons qu'on peut utiliser cette règle pour calculer les dépendances de « { A, B, … } » à partir de « { A, B, … } ≡ {A} # {B} # … ». On obtient { A, B, … } $_{+}\rightarrow$ {X} avec X un élément du bloc qui n'est pas le premier.

Imbrication temporelle

$$\frac{\begin{array}{c}EO \trianglelefteqslant e \varnothing G \\ EO \trianglelefteqslant j \varnothing F\end{array}}{EO \trianglelefteqslant i \text{ every } j \text{ where } e \varnothing G \approx F \approx \{v_{i \text{ every}\ldots} {}_{t}\varnothing v_e\} \approx \{v_{i \text{ every}\ldots} {}_{p}\varnothing v_f\}} \quad \text{(OT13)}$$

Le résultat de « i every j where e » est disponible après le calcul de j et sa valeur dépend point à point de celle de e.

II. Vérification du typage

Une fois que le graphe des dépendances a été calculé, il faut vérifier qu'il admet un ordonnancement complètement statique. Pour cela il suffit de vérifier que tout cycle dans le graphe vérifie (tout arc confondu) :

– un cycle ne contient pas un arc de dépendance totale;

– un cycle contient au moins un arc de dépendance positive.

Le typage d'une expression 81/2 est correct si le graphe d'ordonnancement qui lui est associé vérifie cette propriété. En effet, si on s'intéresse à la production de la valeur du point i d'un tissu qui est associé à un sommet présent dans un cycle, alors cette valeur dépend d'elle-même si et seulement si le cycle contient un arc de dépendance totale ou bien si le cycle ne contient pas d'arc positif (Cf. figure 4).

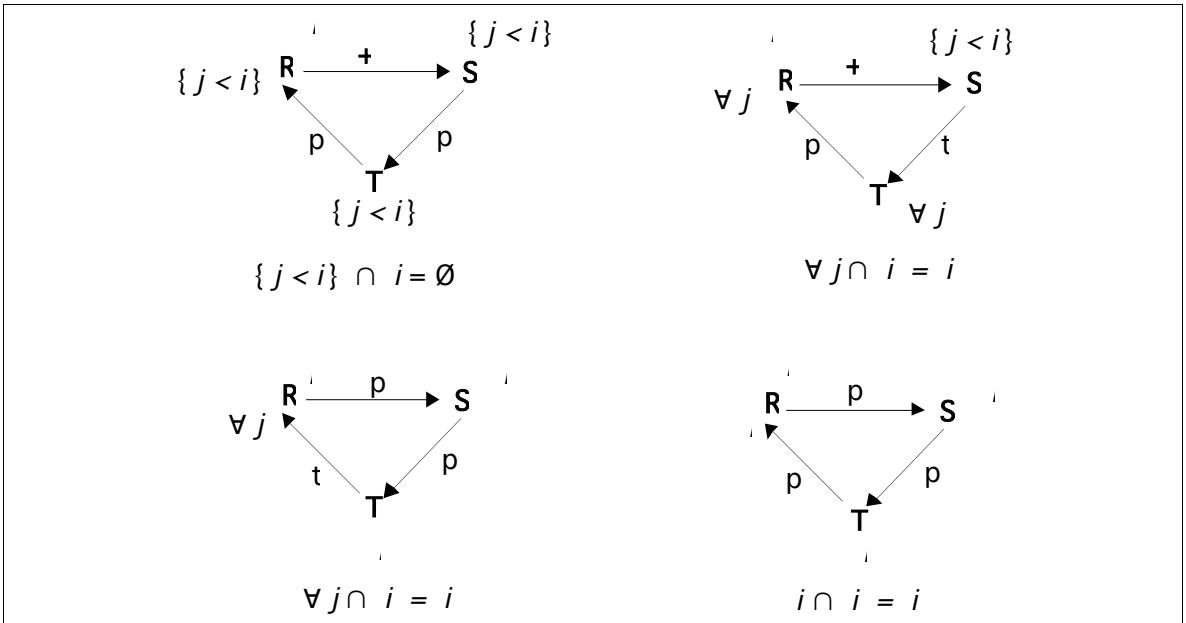


Figure 4 : Vérification de la correction de l'ordonnancement. La présence d'un cycle représente la dépendance d'un tissu avec lui-même. Cette dépendance n'empêche pas un ordonnancement correct si les points du tissu ne dépendent pas d'eux mêmes. Les quatre schémas représentés figurent les quatre possibilités des dépendances entre trois tissus (à un renommage près). Les formules en italiques indiquent les indices des points dont dépend le calcul du point *i* de R. Si le calcul du point *i* de R dépend de lui-même, alors l'expression est incorrecte. On voit que le seul cas acceptable de cycle est celui contenant au moins une dépendance positive *et* pas de dépendance totale.

III. Exemples

III.1. Un exemple d'ordonnancement statique avec recouvrement

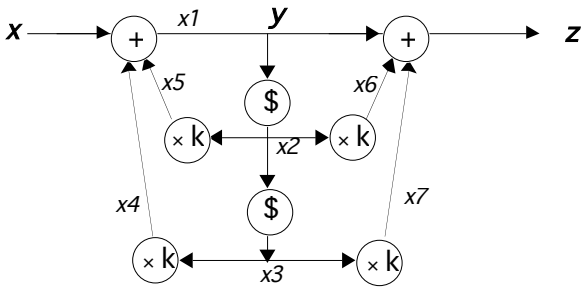
Le fragment de programme suivant correspond à un filtre digital du second ordre utilisé en traitement du signal :

$y = x + k*\$y + k*\$\$y$;
$z = y + k*\$y + k*\$\$y$;

Ce filtre est habituellement présenté sous la forme du circuit ci-dessous. Pour plus de commodité, nous avons

identifié chaque branche du circuit par une variable. Les équations 81/2 correspondantes sont :

$z = x1 + x6 + x7$;
$x1 = x + x5 + x4$;
$x2 = \$x1$;
$x3 = \$x2$;
$x4 = k * x3$;
$x5 = k * x2$;
$x6 = k * x2$;
$x7 = k * x3$;



Pour obtenir le graphe des dépendances des variables, il suffit de répliquer le graphe data-flow ci-dessus en oubliant les arcs entrant dans un délai. Chaque réplication correspond à un tic. Il faut ensuite ajouter les dépendances entre opérations de tics différents et qui correspondent aux délais.

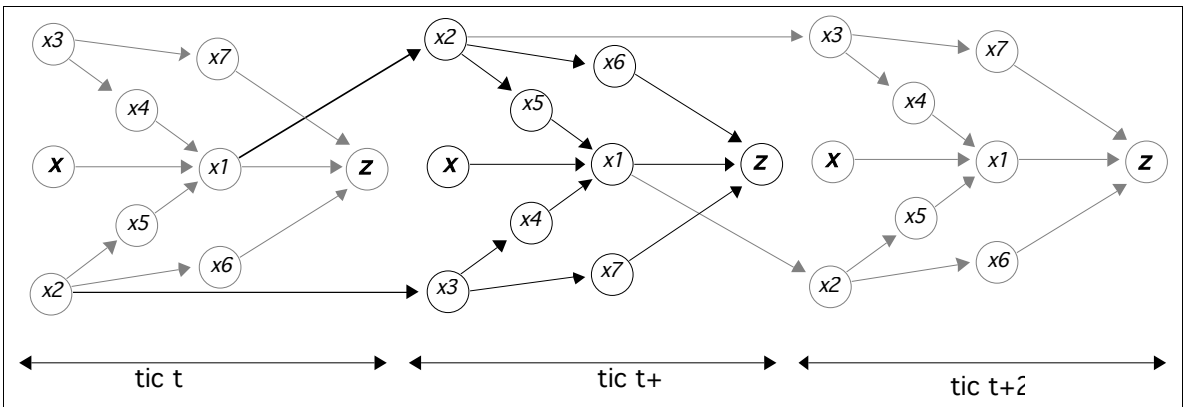


Figure 5 . : Une partie du graphe des dépendances des opérations du programme précédent. Ce graphe des dépendances est la représentation sagittale d'une relation d'ordre partiel. Cette relation est indépendante des calculs effectués : elle est statique. Par contre cette relation peut présenter des recouvrements : la valeur de *x3* au tic t+1 peut être calculée sans attendre la production de la valeur de *z* au top t par exemple.

L'ordre des opérations du programme peut présenter un recouvrement, par exemple la valeur de *z* au tic t n'est pas comparable aux opérations du tic t+1. Sur un calculateur qui dispose de suffisamment de processeurs, il est possible de tirer parti de cette caractéristique afin de commencer les calculs concernant le tic t+1 pendant le déroulement des calculs du tic t : c'est l'effet *pipe-line*.

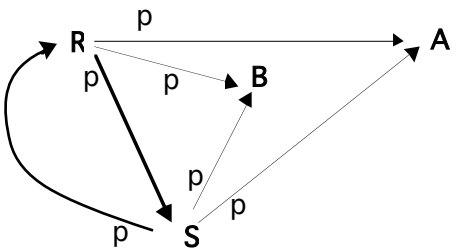
Il est bien sûr possible de « dégrader » l'ordre obtenu, en rajoutant des dépendances artificielles, afin d'obtenir un ordre sans recouvrement. On peut par exemple ajouter une opération entre les opérations du tic t et celle du tic t+1. Cette opération supplémentaire dépend de toutes les opérations du tic t, et toutes les opérations du tic t+1 dépendent de cette dernière. L'opération que nous ajoutons devient un point d'articulation du graphe et correspond à une *barrière de synchronisation* entre tics.

III.2. Exemples de typage

III.2.1. Un exemple de programme non-statique

Nous allons donner quelques exemples de typage correct et incorrect. Reprenons pour commencer l'exemple donné en introduction :

R = si *A* alors *S* sinon *B* fi;
S = si *A* alors *B* sinon *R* fi;



(Le graphe associé à un programme est l'union des graphes des expressions du programme.) Ce graphe présente un cycle inacceptable. Ce programme est donc rejeté. Il correspond à un programme qui n'admet pas d'ordonnancement statique.

Une autre façon d'obtenir des programmes non statiques est l'utilisation des initialisations. Nous n'avons pas donné de règles explicites mais celles-ci sont très simples : les dépendances de

T@*n* = … ; T = … ;

sont les unions des dépendances engendrées par chaque équation. Cette règle (statique) peut éliminer des programmes qui admettent un ordonnancement dépendant de l'exécution :

T@0 = U ;
U@0 = 0 ;
U = T ;
T = … ;

Les dépendances pour le calcul du premier top sont $T \xrightarrow{p} U \xrightarrow{p} 0$, alors que pour les autres tops elles sont : $U \xrightarrow{p} T$. On a donc un cycle quand on considère l'union des deux dépendances. Dans ce cas l'ordonnancement dépend de l'exécution mais pas de la valeur des données (l'ordonnancement dépend du tic).

III.2.2. Exemple de correction d'un schéma d'équations itératif

Voici un deuxième exemple qui implique une dépendance entre points d'un même tissu (calcul de factorielle de manière itérative) :

f (x, y) = si (x == 0) alors 1 sinon x * y fi;
Q [10] = +\\ 1 ;
T [10] = 0 # *f* ^ (*T*, *Q*) : [10] ;

Le graphe (simplifié) associé au programme est représenté figure 6 (on convient de ne pas faire figurer les labels « p » des arcs point à point, ni les dépendances vers les constantes). Il y a deux cycles qui contiennent chacun une dépendance positive. Le programme est correct.

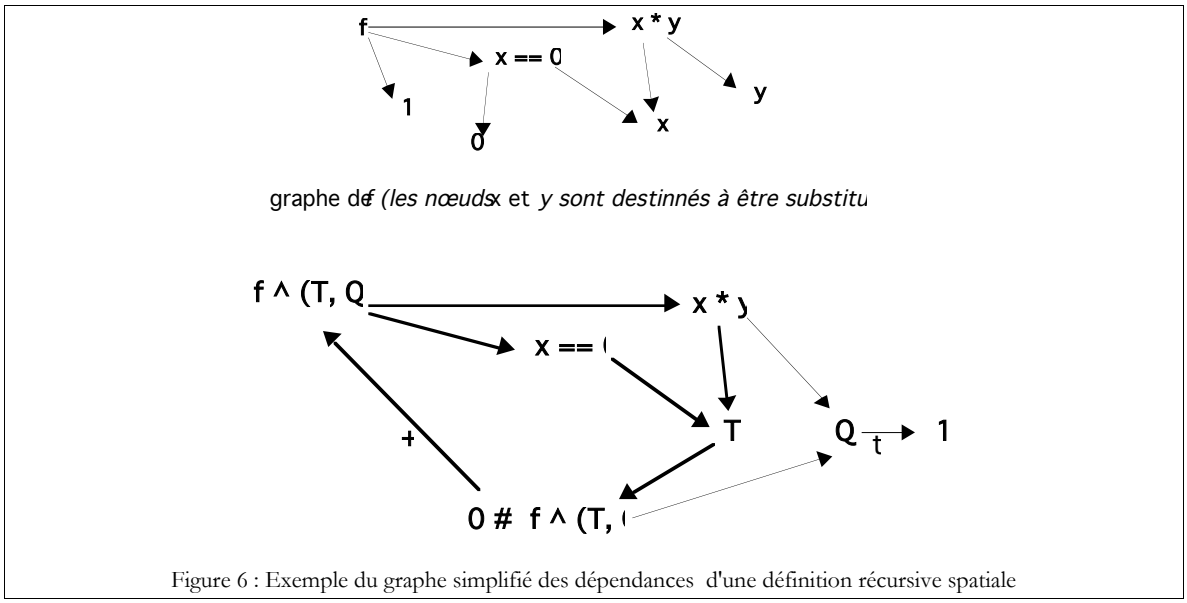
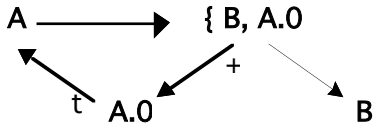


Figure 6 : Exemple du graphe simplifié des dépendances d'une définition récursive spatiale

III.2.3. Exemple d'approximation trop grossière

L'exemple suivant montre que le mécanisme de typage n'est pas très fin : il écarte des programmes qui admettent un ordonnancement correct.

$A = \{ B, A.0 \}$;



Le graphe contient un cycle contenant une dépendance totale et le programme est rejeté comme incorrect. Pourtant il existe un ordonnancement complètement statique : $B < A.0 < A.1$.

Il faut noter que suivant notre convention, les programmes que nous considérons ont été débarrassés du « sucre hiérarchique », i.e. le programme 81/2 :

$A = \{ a0 = B; a1 = a0; \}$

est un programme transformé en :

$A\_a0 = B; A\_a1 = A\_a0;$

programme qui est déclaré correct du point de vue de l'ordonnancement. L'échec de l'ordonnancement de « $A = \{ B, A.0 \}$; » est dû à la perte d'information au niveau de la création d'un bloc et de l'accès à un point : la valeur de $A.0$ ne dépend pas réellement de toutes les valeurs de A mais uniquement de la première.

IV. Généralisation

Notre approximation de la dépendance entre points par trois types de dépendance entre tissus peut être raffinée pour pallier aux inconvénients soulignés plus haut. Le problème est de choisir des types de dépendance permettant de modéliser les dépendances entre points d'un programme 81/2 tout en restant suffisamment abstrait pour éviter la représentation explicite de toutes les dépendances. De plus les calculs impliqués par la manipulation de ces dépendances doivent être facilement automatisables.

De manière générale, on notera

$A \xrightarrow{F} B$ avec $F : \mathbb{N} \to \Upsilon\mathbb{N}$

si $A.i$ dépend des points $F(i)$. Les fonctions primitives dont nous disposons sont :

$p(i) = \{ i \}$
$t(i) = \mathbb{N}$

On définit la composition de fonctions de dépendance par :

$$(F, G)(i) = \bigcup_{j \in F(i)} G(j)$$

Notons que p est un élément neutre pour la composition et t un élément absorbant. Les notions de bloc et d'agrégation motivent l'introduction du combinateur suivant :

$(P:F)(i) =$ si $P(i)$ alors $F(i)$ sinon $\varnothing$

Les trois prédicats P qui nous intéressent seront notés par :

$=\mathbf{n} = \lambda i.i == n$ $\quad <\mathbf{n} = \lambda i.i < n$ $\quad \geq\mathbf{n} = \lambda i.i \geq n$

Avec ces définitions, nous étendons le typage des dépendances en remplaçant les règles (OT5), (OT12) et (OT13) par

$$\frac{EO \trianglelefteqslant e \varnothing G}{EO \trianglelefteqslant e.i \varnothing G \approx \{v_{e.i\ =i:p} \varnothing v_e\}} \qquad (OT5')$$

$$\frac{\begin{array}{c} EO \trianglelefteqslant e_1 \varnothing G_1 \\ \dots \\ EO \trianglelefteqslant e_n \varnothing G_n \end{array}}{EO \trianglelefteqslant \{ e_1, \dots, e_n, \} \varnothing G_1 \approx \dots \approx G_n \approx \{v_{\{\dots\}\ =1:t} \varnothing v_{e1}\} \approx \dots \approx \{v_{\{\dots\}\ =n:t} \varnothing v_{en}\}} \qquad (OT12')$$

$$\frac{\begin{array}{c} EO \trianglelefteqslant e \varnothing G \\ EO \trianglelefteqslant f \varnothing F \end{array}}{EO \trianglelefteqslant e \# f \varnothing G \approx F \approx \{v_{e \# f\ <|e|:p} \varnothing v_e\} \approx \{v_{e \# f\ \geq|e|:p} \varnothing v_f\}} \qquad (OT13')$$

Un typage est correct si pour tout cycle $A \xrightarrow{F1} B \xrightarrow{F2} \dots \xrightarrow{Fn} A$, la fonction

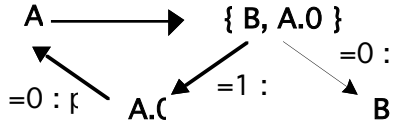
$F = (F1, (F2, \dots (Fn\text{-}1, Fn)\dots)$

a la propriété suivante :

$\forall i, F(i) \cap \{i\} = \varnothing$

On peut vérifier que cette condition recouvre bien la condition donnée dans le paragraphe précédent. Si on revient à l'exemple qui motive cette généralisation, on obtient :

$A = \{ B, A.0 \}$;



On doit donc calculer :

$F = (=0 : p), p, (=1 : t)$

(on évalue le cycle en partant de A.0, mais on pourrait partir de n'importe quel autre sommet du cycle). Il est facile de montrer que pour tous prédicats P et Q et pour toutes fonctions G et H :

$H = \lambda i\,.\,\mathit{true} : H$ (R0)

p, G = G, p = G (R1)

G, t = t (R2)

(P : G), H = P : (G, H) (R3)

P : (Q : G) = (P ∧ Q) : G (R4)

et donc :

F = (=0 : p), p, (=1 : t)
= (=0 : p), (=1 : t) (par R1)
= =0 : (p, =1 : t) (par R3)
= =0 : (=1, t) (par R1)
= (=0 ∧ =1) : t (par R4)
= ($\lambda i\,.\,\mathit{false}$) : t (par l'évaluation de =0 ∧ =1)
= $\lambda i.\varnothing$

ce qui conclut à la correction de l'expression car F(i) ∩ { i } = ∅ ∩ { i } = ∅.

La méthode peut être automatisée : pour manipuler les fonctions F il faut être capable de calculer $F(i) \cap \{i\}$ pour $0 \leq i \leq c$, c étant le cardinal d'un tissu. L'utilisation des règles R0 à R4 (orientées de la gauche vers la droite), permet de mettre F sous l'une des quatre formes suivantes :

F est d'une des formes $\begin{cases} p \\ t \\ (P\ \ Q\ \ \ldots\ \ Z) : p \\ (P\ \ Q\ \ \ldots\ \ Z) : (t, H) \end{cases}$

où H est de la même forme que F et les prédicat P, Q, …, Z sont d'un des trois types donnés plus haut.

Le test pour savoir si $F(i) \cap \{i\}$ est vide pour $0 \leq i \leq c$ quand F est une fonction de la forme précédente peut donc se faire en commençant par examiner quand P∧Q∧…∧Z est vraie (dans les deux autres cas, la réponse est immédiate). Cela se fait en calculant les intersections d'un nombre fini d'ensembles de la forme :

[0, a[(prédicat <a)
[b, ∞[(prédicat ≥b)
IN (prédicat *true*)

avec a et b des constantes produites par le typage géométrique. Une intersection vide correspond au cas où P∧Q∧…∧Z = *false* et on a résolu le problème. Sinon, il faut déterminer si (t, H)(i) ∩ {i} est vide pour les i qui appartiennent à l'intersection non vide. Cela revient à déterminer si H(j) ∩ {i} est vide pour tout j de IN et pour tout i de [0, c] tels que (P∧Q∧…∧Z)(i) = *true*. H étant de la même forme que F celà conduit à calculer l'ensemble des k tels que (P'∧Q'∧…∧Z')(k) = *true*. Il faut à nouveau à calculer l'intersection d'un ensemble fini de segments. On répète cette procédure autant de fois que nécessaire (le nombre de répétition est bornée par la longueur du cycle).

L'approche est donc automatisable, même si la vérification de l'existence d'un ordonnancement est beaucoup plus lourde que pour le jeu de dépendances précédentes.

VI. Le typage des horloges

On peut regarder un programme 81/2 comme un réseau de processus : un processus est attaché à chaque définition de tissu, consomme les valeurs envoyées par les processus identifiés en partie droite de sa définition et calcule une valeur émise vers les autres processus. De ce point de vue, un tissu dont la valeur est réduite à ⊥ s'interprète comme un processus souffrant de *famine*. Dans l'exemple suivant

T = \$1; (E1)

T attend une valeur en entrée sans jamais avoir une chance d'en recevoir. Il est donc vital de pouvoir détecter automatiquement de telles expressions. Remarquons qu'on peut donner des exemples non triviaux comme :

U = *false* after *false* ; (E2)

V = *Clock* after \$$W$; W = *Clock* after \$$V$; (E3)

pour lesquels on peut parler d'*inter-blocage*. La difficulté principale est de préciser ce que l'on entend par « les expressions dont la valeur se réduit de manière certaine à ⊥ ». Par exemple, est ce que « U = *false* when *false* » est une expression incorrecte ? Il faut noter que pour déterminer que U se réduit à ⊥, on a besoin d'examiner la valeur de ses sous-expressions. Cela n'est pas nécessaire pour V dans (E3) pas plus que pour T dans (E1) : dans ces deux exemples on a besoin d'une information sur la structure de l'horloge des sous-expressions, mais on n'a pas besoin d'informations sur la valeur des sous-expressions.

Nous voulons capturer dans un système de typage des informations sur la structure de l'horloge d'une expression e. Toutes les expressions qui ont la même structure d'horloge auront le même type. Il sera alors facile de comparer le type d'une expression avec le type d'horloge de ⊥.

L'interprétation d'un type consistera à associer à chaque type (à l'aide d'une fonction *Texp*) un sous-ensemble de *TISSU* représentant les instances du type. Évidemment, pour que le typage soit d'une certaine utilité, il faut que l'interprétation d'un type soit cohérente avec l'évaluation d'une instance d'un type. C'est ce qui est illustré par la figure ci-dessous (fig. 1).

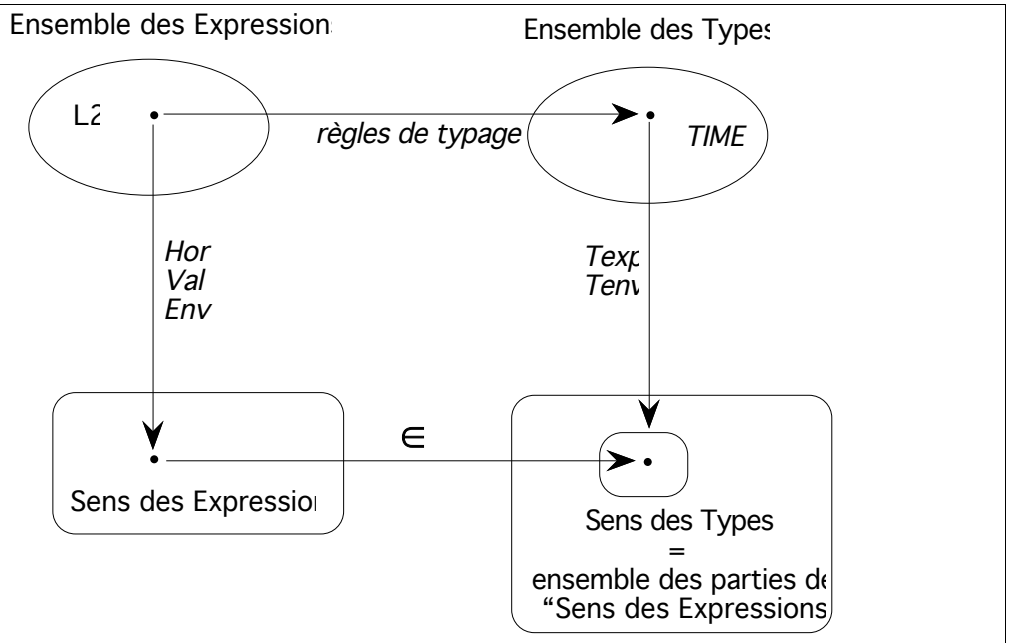


Figure 1 : Le typage. On associe un type à chaque expression. Un type représente un ensemble de valeurs « ayant certaines propriétés en commun ». Le typage est correct vis à vis de la sémantique d'une expression si la sémantique d'une expression (son sens) appartient à la sémantique de son type (i.e. le sens d'une expression fait partie des instances d'un type).

Les techniques impliquées sont assez lourdes, aussi nous avons restreint ce système de typage à un sous-ensemble L2 de 81/2 ne contenant pas les collections. Les opérations temporelles when, until, after et at y figurent. L2 est donc suffisamment représentatif pour que le système de typage développé soit caractéristique des techniques à mettre en œuvre sur 81/2.

I. Calcul d'un type temporel

I.1. Les domaines du typage

I.1.1. Le sous-langage L2

Dans ce qui suit, e, f, g ∈ L2 et id ∈ ID :

e ::=n | **Clock**
| id | id **=** e
| f **binop** g
| f **fby** g | **$** f | f **when** g | f **until** g | f **after** g | f **at** g

I.1.2. L'ensemble des types TIME

Il nous faut à présent définir l'ensemble des types temporels : TIME. Il est défini par les équations suivantes où t parcourt les variables de type TID et où F, G et H parcourent TIME. Nous avons indiqué à droite des constructions de TIME, à quels éléments de L2 un type temporel va correspondre intuitivement.

H ::=	*0*	*horloge d'une constante scalaire*
|	*clock*	*horloge de la constante clock*
|	t	*identificateur d'horloge*, t ∈ TID
|	$ H	*horloge d'un retard*
|	H + G	*horloge d'une opération arithmétique*
|	H fby G	*horloge d'une séquence*
|	(∃t ⊆ G after F).H	*horloge d'un trigger*
|	(∃t ≥ F after G).H	*horloge d'un filtre passe-haut*
|	(∃t ≤ F after G).H	*horloge d'un filtre passe-bas*
|	(∃t ∈ F after G).H	*horloge d'un dirac*

Les éléments de TIME sont les éléments d'un domaine syntaxique et il ne faut pas attacher de signification particulière aux notations choisies. Cependant, les quatre dernières constructions font apparaître un symbole de quantification existentielle. Ce choix n'est pas innocent et sera éclairé un peu plus tard. L'identificateur t est lié par le symbole de quantification dans l'expression H. Cela veut dire que les deux expressions suivantes (s, t ∈ TID) :

(∃t ⊆ G after F) . H

et

(∃s ⊆ G after F) . H[s/t]

sont équivalentes (sous la condition que s n'apparaisse pas comme variable libre dans H). Autrement dit, deux types sont équivalents s'ils sont structurellement équivalents et non pas s'ils sont équivalents par leur nom.

I.2. Les règles de typage

Les règles du typage temporel sont données sous la forme de formules « ETH ⋦ e ∴ H », ce qui se lit : « avec les hypothèses ETH l'expression e a pour type temporel H ». L'environnement ETH permet de donner un type à chaque variable d'une expression. Il faut remarquer que dans la règle ETH ⋦ e ∴ H il est implicitement supposé que ETH définit un type pour au moins chaque variable libre de e. Nous suivrons les conventions suivantes :

variable	*type d'objet*
ETH	environnement des types d'horloge
id	identificateur
F, G, H	types d'horloge
t, u	variables de type, t, u ∈ TID
e, f, g	expressions

Typage d'une variable

La première règle donne le type d'un identificateur; c'est le type qui lui est donné dans l'environnement :

Type d'un identificateur : ETH, id ∴ H ⋦ id ∴ H (RT0)

Liaison variable - valeur : **Erreur !** (RT1)

(avec id n'apparaissant pas dans ETH). La notation « ETH, id ∴ H » indique l'environnement qui assigne le type H à la variable id et qui assigne tout autre identificateur du domaine de ETH au même type que ETH.

Expressions arithmétiques

Les constantes : ETH ⋦ n ∴ 0 et ETH ⋦ Clock: ∴ *clock* (RT2,3)

L'arithmétique : **Erreur !** (RT4)

Expressions temporelles

La séquence : **Erreur !** (RT5)

Le retard : **Erreur !** (RT6)

Le trigger : **Erreur !** (RT7)

Le filtre passe-bas : **Erreur !** (RT8)

Le filtre passe-haut : **Erreur !** (RT9)

Le dirac : **Erreur !** (RT10)

avec u n'étant pas libre dans H pour les équations RT7 à RT10. e[a/t] dénote le résultat obtenu en remplaçant dans e les occurrences de la variable t par l'expression a après renommage si nécessaire de toutes les variables libres afin d'éviter les collisions lors de la substitution.

La forme des règles de typage existentiel (RT7–10) factorise les quantificateurs existentiels en-tête d'une expression de type. L'ordre des sous-expressions F et G est inversé dans la règle RT7 par rapport aux règles RT8–10.

I.2.1. Exemple de typage d'une expression

Le typage d'une expression peut être prouvé valide en donnant une liste de typages finissant par le type en question. Chaque élément de la liste est la conclusion d'une instance de règle dont les prémisses apparaissent plus haut dans la liste. Par exemple, on peut prouver à l'aide du tableau ci-dessous que le type de 1+$T est *0*+$t si t est le type de T :

Environnement ⩹	Expression ∴ Type temporel	Règle
T ∴ t ⩹	1 ∴ *0*	constantes
T ∴ t ⩹	$T ∴ $t	délai
T ∴ t ⩹	1+$T ∴ *0*+$t	arithmétique

I.2.2. Types récursifs

Le langage TIME n'est pas réduit aux expressions de longueur finie. Par exemple un type possible pour l'expression T = 1+$T est le type : *0* + $(*0* + $(*0* + $(*0* + …)…) (pour s'en convaincre, il suffit de substituer cette dernière expression à t dans le tableau précédent et d'utiliser la règle de liaison identificateur–valeur).

Pour pouvoir écrire commodément des expressions de longueur infinie, nous introduisons la construction suivante : si H est un élément de TIME, on note

μt.H

l'expression obtenue en substituant infiniment t par H dans H :

μt.H = H [H/t] [H/t] [H/t]…

Ainsi, l'expression « *0* + $(*0* + $(*0* + $(*0* + …)…) » peut se noter μt.(*0* + $t). Il faut comprendre que l'introduction des types récursifs n'enrichit pas le domaine des types : c'est la notation sous forme finie d'un élément de TIME qui aurait sinon été exprimée par un terme infini.

Nous introduisons cette notation afin de pouvoir désigner le terme infini qui sert de type à certaines expressions (récursives) comme « T = $T + 1 ». Si t est le type temporel de T, i.e. ⋐ T ∴ t, alors : ⋐ 1+$T ∴ `*0*+$t et par suite t = *0*+$t. La solution de cette équation est obtenue en substituant indéfiniment t par *0*+$t et se note donc μt.(*0* + $t).

Nous pouvons alors remplacer (RT1) pour éviter de devoir faire un nombre infini de dérivations par :

Récursion (introduction) : **Erreur !** (RT1)

où t est une variable de type qui n'est pas libre dans H et où id n'apparaît pas dans ETH. Cette règle permet l'introduction de la notation explicite d'un point fixe. Toutes les liaisons variables-valeurs ne génèrent pas un type récursif, aussi introduit-on une règle d'élimination :

Récursion (élimination) : **Erreur !** (RT1 *bis*)

Donnons un exemple sur « T = $Clock » :

Environnement	⋐	Expression	∴	Type temporel	Règle
T ∴ t	⋐	Clock	∴	*clock*	RT3
T ∴ t	⋐	$Clock	∴	$*clock*	RT6
	⋐	T = $Clock	∴	μt.$*clock*	RT1
	⋐	T = $Clock	∴	$*clock*	RT1 *bis*

Des règles concernant les types récursifs sont introduites par exemple dans [Reynolds 85] ou encore dans [MacQueen, Plotkin, Sethi 86]. Dans ce dernier article, les types récursifs sont introduits afin de donner un type à chaque terme du lambda-calcul. En effet, on peut toujours appliquer un terme *x* sur lui même et par suite, si *x* est de type *s*, alors *x*(*x*) est un terme valide qui est aussi de type *s*. Mais le type de *x* doit aussi s'exprimer comme celui d'une fonction *s*→*t* qui s'applique aux éléments de type *s*. Par suite le type de *x* est solution de *s* = *s* → *t* ce qui se note μ*s*.*s*→*t*. On voit que nos motivations pour l'introduction de μ sont tout à fait semblables.

II. Interprétation d'un type

II.1. Une interprétation des types d'horloge

Si e est une expression de type t, nous voulons associer au type t un ensemble représentant l'horloge de e. Plus précisément, nous voulons associer à e un ensemble représentant les tops pour lesquels e a une valeur non réduite à ⊥[1]. Appelons cet ensemble *Def*(e).

À cause de l'opérateur when, il n'est pas possible de calculer *Def*(e) sans effectuer le calcul de e. Par contre il est possible d'associer de manière *statique* (i.e. à la compilation) à chaque expression e un ensemble E(t), qui dépend du type de e et tel que : *Def*(e) ∈ E. E(t) représente toutes les horloges possibles pour une expression de type t. (Cette approche est habituelle pour modéliser l'indéterminisme : quand on est incapable de déterminer des valeurs précises, on travaille avec l'ensemble des valeurs possibles, Cf. par exemple [Scott 89, *p. 23 & suiv.*]).

Évidemment, il peut être lourd sinon impossible de calculer une représentation de toutes les exécutions possibles. *Mais nous n'en avons pas besoin* : si E est réduit à l'ensemble vide, alors *Def* (e) aussi est réduit à ∅ et donc la

[1] Rappelons que ce n'est pas la même chose, Cf. le paragraphe III.4 du chapitre consacré à la sémantique dynamique du langage

valeur de e est ⊥. Remarquons que cette approche a beaucoup à voir avec l'évaluation partielle des programmes : calculer si l'interprétation E(t) d'un type t est réduite à ∅ correspondant à l'évaluation partielle d'une expression de type t.

Cependant, il serait désirable de « récupérer » des informations sur E. Cela permettrait par exemple d'optimiser des programmes en évitant de calculer des valeurs, tant qu'il est su de manière certaine, qu'elles se réduisent à ⊥. C'est pourquoi nous proposerons dans le paragraphe suivant une approximation de E (tout le problème est de trouver une approximation E(t) qui soit assez « fine »).

II.1.1. Définition d'une interprétation des types d'horloge

C'est la fonction *Texp* qui permet d'associer un ensemble E à un type :

$Texp \in \mathrm{TIME} \rightarrow \mathrm{TENV} \rightarrow \Upsilon\Upsilon\mathrm{IN}$: l'interprétation d'un type temporel

$\eta \in \mathrm{TENV} = \mathrm{TID} \rightarrow \Upsilon\Upsilon\mathrm{IN}$: l'environnement des variables de types

Remarquons qu'un type est calculable statiquement, et par suite *Texp* [[t]] aussi. On note par ΥD, l'ensemble des parties d'un ensemble D. $\Upsilon\Upsilon$IN représente donc un ensemble dont les éléments sont des ensembles contenant des parties de IN. En terme d'horloge, un élément de $\Upsilon\Upsilon$IN correspond à un ensemble d'horloges. Pour simplifier la lecture, nous adoptons les conventions suivantes : les lettres K, L… représentent un élément de $\Upsilon\Upsilon$IN, les lettres A, B… dénotent une partie de IN (donc un élément de ΥIN) et les lettres n, m… sont des éléments de IN.

On définit *Texp* par induction sur la structure de l'expression d'un type :

$Texp\,[\![1]\!]\,\eta = \{\{0\}\}$

$Texp\,[\![\mathrm{clock}]\!]\,\eta = \{\mathrm{IN}\}$

$Texp\,[\![t]\!]\,\eta = \eta(t)$

$Texp\,[\![\$H]\!]\,\eta = \{ up(A) \mid A \in Texp\,[\![H]\!]\,\eta \}$

$Texp\,[\![H+G]\!]\,\eta = \{ som(A, B) \mid A \in Texp\,[\![H]\!]\,\eta,\ B \in Texp\,[\![G]\!]\,\eta \}$

$Texp\,[\![H \text{ fby } G]\!]\,\eta = \{ follow(A, B) \mid A \in Texp\,[\![H]\!]\,\eta,\ B \in Texp\,[\![G]\!]\,\eta \}$

La fonction *up* (A) est égale à l'ensemble vide si A est réduit à l'ensemble vide, et sinon est égale à l'ensemble A privé de son plus petit élément. Le plus petit élément d'un ensemble A se note *min* (A) et vaut $+\infty$ si A est réduit à l'ensemble vide. Les fonctions *som* et *follow* sont définies ainsi :

som (A, B) = { n | n ≥ *min*(A) et n ≥ *min*(B) et (n∈A ou n∈B) }

follow (A, B) = *si* (A = ∅) *alors* ∅ *sinon* { *min*(A) } ∪ { n∈B | n > *min*(A) }

L'interprétation d'un trigger « f when g » est simple : il faut exprimer toutes les horloges qui peuvent être obtenues en sélectionnant arbitrairement des tops dans une horloge de g, ces tops devant se produire après le premier top d'une des horloges de f). L'interprétation des autres formes de quantifications existentielles ne présente pas plus de difficultés :

$$Texp\,[\![\exists t \subseteq G \text{ after } F \,.\, H]\!]\,\eta = \bigcup_{A\ Texp[\![G]\!]} \bigcup_{B\ Texp[\![F]\!]} \bigcup_{K\ exist(A,B)} \{ Texp[\![H]\!]\,\eta[K/t] \}$$

$$Texp\,[\![\exists t \leq F \text{ after } G.\ H]\!]\,\eta = \bigcup_{A\ Texp[\![F]\!]} \bigcup_{B\ Texp[\![G]\!]} \bigcup_{K\ préfixe(A,B)} \{ Texp[\![H]\!]\,\eta[K/t] \}$$

$$Texp\ [\![\ [\exists t \geq F \text{ after } G.\ H]\!]\ \eta = \bigcup_{A} \bigcup_{Texp\ [\![F]\!]\ B} \bigcup_{Texp\ [\![G]\!]\ K} \bigcup_{suffixe\ (A,B)} \{\ Texp\ [\![H]\!]\ \eta[K/t]\ \}$$

$$Texp\ [\![\ [\exists t \in F \text{ after } G.\ H]\!]\ \eta = \bigcup_{A} \bigcup_{Texp\ [\![F]\!]\ B} \bigcup_{Texp\ [\![G]\!]\ K} \bigcup_{in\ (A,B)} \{\ Texp\ [\![H]\!]\ \eta[K/t]\ \}$$

avec

$$exist\ (A, B) = \{\ C \mid C \subseteq A \text{ et } min(C) \geq min(B)\ \}$$

$$préfixe\ (A, B) = \bigcup_{b \in B} \{\ \{c \mid c \in A \text{ et } c \leq b\}\ \}$$

$$suffixe\ (A, B) = \bigcup_{b \in B} \{\ \{c \mid c \in A \text{ et } c \geq b\}\ \}$$

$$in\ (A, B) = \bigcup_{b \in B,\ b \geq min(A)} \{\ \{b\}\ \}$$

L'autre cas intéressant est celui de l'interprétation d'un point fixe qui permet de donner une valeur aux termes infinis :

$$Texp\ [\![\ [\mu t.H]\!]\ \eta = FIX_{\Upsilon\Upsilon\mathrm{IN}}\ (\lambda K.\ Texp\ [\![\ [H]\!]\ \eta[K/t])$$

L'expression $FIX_{\Upsilon\Upsilon\mathrm{IN}}\ f(K)$ représente la plus petite solution de l'équation $K = f(K)$ si elle existe, où K prend sa valeur dans $\Upsilon\Upsilon\mathrm{IN}$ et où la relation d'ordre est l'inclusion.

II.1.2. Correction de la définition de Texp

Nous devons montrer que nous avons correctement défini *Texp*. *Texp* est défini par cas sur la structure d'une expression finie (car on utilise μ pour représenter les types infinis). Le seul cas qui pose problème est celui de la définition de l'interprétation de μ : il faut montrer que le point fixe utilisé existe.

L'expression $FIX_{\Upsilon\Upsilon\mathrm{IN}}\ F$ a un sens si la fonction *F* est continue sur le treillis $\Upsilon\Upsilon\mathrm{IN}$. Ici *F* a pour valeur $\lambda K : \Upsilon\Upsilon\mathrm{IN}\ .\ Texp\ [\![\ [H]\!]\ \eta[K/t]$. Cette démonstration, qui demande du travail, est rejetée à la fin du chapitre.

II.1.3. Un type existentiel

La règle de typage d'un trigger fait apparaître un type existentiel. Mitchell et Plotkin ont introduit dans [Mitchell, Plotkin 88] l'usage des types existentiels pour prendre en compte les mécanismes d'abstraction de données.

Nous en faisons ici une utilisation différente. La quantification existentielle est utilisée pour modéliser le fait qu'on ne peut savoir, sans réaliser le calcul, quelle sera l'horloge effective d'un trigger. Un type dénote un ensemble d'horloges; un type existentiel représente le choix d'une horloge prise dans un ensemble d'horloges possibles. Ce choix est déterministe mais non prévisible (sauf en effectuant le calcul complet). Cependant, on possède des informations sur l'ensemble dans lequel l'horloge sera choisie (c'est pourquoi on a une quantification prise dans un ensemble fixé).

Le type « $\exists t \subseteq G$ after F.H » peut donc se lire comme « une horloge H(t) pour une horloge t incluse dans une des horloges de (G after F) ». Le type « G after F » peut se lire comme « une horloge obtenue en restreignant une des

horloges de G aux tops postérieurs au premier top d'une des horloges de F ».

Le terme de type « existentiel » n'est pas justifié ici par une analogie avec la logique constructiviste (comme dans [Mitchell, Plotkin 88]). Il est justifié par la considération suivante. Soit une fonction *f* de $D \times D^n$ dans D, on peut définir suivant [MacQueen, Plotkin, Sethi 86, §*4.3*] la « quantification existentielle de *f* sur $E \subseteq D$ » par

$$(\exists_E f)\ (d_1, \ldots, d_n) =_{def} \bigcup_{e \in E} f(e, d_1, \ldots, d_n)$$

ce qui nous permet de reformuler la définition de *Texp* en :

$$Texp\ [\![\ [\![\exists t \subseteq G \text{ after } F\ .\ H]\!]\ ]\!]\ \eta = \exists_K\ \lambda L.\ \{\ Texp\ [\![H]\!]\ \eta[L/t]\ \}\ )$$

$$\text{avec } K = \bigcup_{\substack{A \in Texp\ [\![G]\!] \\ B \in Texp\ [\![F]\!]}} \{ \text{exist } (A, B) \}$$

La quantification existentielle de *f* est ainsi reliée à la définition d'une somme (éventuellement infinie). Il serait intéressant de poursuivre plus avant et d'éclaircir les relations entre le point de vue « abstraction de donnée » et « non-prévisibilité ».

II.2. La correction du typage

Maintenant que nous avons associé une interprétation à un type, il nous faut montrer que le typage est cohérent avec la sémantique du langage. Nous posons la définition suivante : pour toute expression e, soit type(e) l'élément H de TIME associé à e par les règles de typage RT1 à RT10, *Texp* est cohérent vis-à-vis de la sémantique du langage si et seulement si :

$$\forall\ e \in L2,\quad Def(e) \in Texp\ [\![\ [\![\text{type}(e)]\!]\ ]\!]$$

On a donc en particulier :

$$Texp\ [\![\ [\![\text{type}(e)]\!]\ ]\!] = \varnothing \quad \Rightarrow \quad Val\ [\![\ [\![e]\!]\ ]\!] = \bot$$

propriété qui nous permet de détecter certaines horloges vides dans un programme. Cette conséquence de la cohérence du typage est à rapprocher de la définition usuelle de la « soundness » proposée par exemple par Abadi *& al.* dans [ACPP 89] où un typage est défini comme correct (*sound*) si toute expression bien typée ne s'évalue pas en *wrong* (*wrong* est une valeur similaire à $\bot$ mais spécifiquement ajoutée aux domaines sémantiques pour modéliser les erreurs).

La démonstration de la cohérence de *Texp* se fait par récurrence sur la structure d'une expression. Cette démonstration, technique et fastidieuse, est rejetée à la fin du chapitre.

III. Considérations pratiques

III.1. Un exemple de calcul de type et de son interprétation

Nous allons détailler le traitement de l'exemple (E3). Ce programme dans le langage L2 peut s'écrire :

```
V = Clock after $(Clock after $V) ;
```

Le type temporel de V est le suivant :

Env.	⊑	Expression	∴ Type temporel	Règle
V	∴ v ⊑	Clock	∴ *clock*	RT3
V	∴ v ⊑	$V	∴ $v	RT6
V	∴ v ⊑	Clock after $V	∴ ∃t ≥ *clock* after $v. t	RT5
V	∴ v ⊑	$(Clock after $V)	∴ $(∃t ≥ *clock* after $v. t)	RT6
V	∴ v ⊑	Clock after $(Clock after $V)	∴ ∃u ≥ *clock* after $(∃t ≥ *clock* after $v. t) . u	RT5
	⊑	V = Clock after $(Clock after $V)	∴ μv.∃u ≥ (*clock* after $(∃t ≥ *clock* after $v. t)) . u	RT1

Il faut vérifier que ∅ est ou n'est pas une interprétation de « μv. ∃u ≥ clock after $(∃t ≥ *clock* after $v. t) . u ». Pour cela, il suffit d'interpréter le type en remplaçant v par ∅ et en oubliant μ (cela découle de la preuve de l'existence de *Texp* qui est donnée en annexe à ce chapitre). Ainsi on doit calculer :

suffixe (IN, *up*(*suffixe*(IN, *up*(∅))))

quantité qui est clairement égale à ∅.

III.2. Implémentation du type-checking temporel

D'un point de vue pratique, pour automatiser le calcul du typage temporel, on peut mettre en place la procédure suivante :

1) On associe une variable de type *tid* à chaque variable *id* :

$$\left.\begin{array}{l} T = \$Q + 1 ; \\ Q = \$T + 1 ; \end{array}\right\} \quad \left\{\begin{array}{l} S = \text{Clock} ; \\ 1 ; \end{array}\right\} \quad \Rightarrow \quad \left\{\begin{array}{l} S \varnothing s \\ T \varnothing t \\ Q \varnothing q \end{array}\right\}$$

2) Chaque variable *tid* est définie par l'interprétation *Texp* de l'équation de définition de *id*. On oublie au passage les unions et la paire d'accolades extérieures de l'interprétation des constantes (on considère qu'une seule horloge est associée à chaque tissu) :

$$\left.\begin{array}{l} T = \$Q + 1 ; \\ Q = \$T + 1 ; \end{array}\right\} \quad \left\{\begin{array}{l} S = \text{Clock} ; \\ 1 ; \end{array}\right\} \quad \Rightarrow \quad \left\{\begin{array}{l} s = \mathbb{N} \\ t = \mathit{som}(\mathit{up}(q), \{0\}) \\ q = \mathit{som}(\mathit{up}(t), \{0\}) \end{array}\right\}$$

3) On vérifie que le système obtenu n'admet pas pour certain *tid* une solution réduite à ∅. Pour cela il suffit de remplacer toutes les variables en partie droite par ∅. On obtient de nouvelles valeurs pour les variables. On calcule alors l'ensemble E des variables dont la valeur est ∅ et dont la définition dépend uniquement de la valeur des éléments de E (le calcul se fait par élimination des variables qui ne sont pas ∅ et des variables dépendantes). L'ensemble E correspond alors aux horloges qui acceptent ∅ pour solution. Dans notre exemple,

$$\left\{\begin{array}{l} s = \mathit{clock} \\ t = \mathit{som}(\mathit{up}(\ ), \{0\}) = \\ q = \mathit{som}(\mathit{up}(\ ), \{0\}) = \end{array}\right.$$

le calcul de E donne {t, q}. On sait alors que T et Q sont réduit à ⊥.

Il faut bien noter que les calculs impliqués sont des calculs finis, même si l'ensemble des entiers naturels apparaît : il s'agit de faire un calcul symbolique avec 3 constantes, ∅, {0} et IN. La justification de cette procédure, et les raffinements possibles de celle-ci, reposent sur les techniques d'*interprétation abstraite*.

L'idée qui se cache derrière cette procédure est la suivante : on cherche à obtenir une approximation du calcul du point fixe dans le treillis ϒϒIN plutôt que de calculer les véritables horloges (ce qui se révèle impossible). Diverses approximations peuvent être envisagées. Puisque nous nous intéressons principalement au nombre des éléments d'une horloge, on peut approcher l'ensemble des horloges d'un tissu par leur union. Une deuxième

approximation consiste ensuite à remplacer un sous-ensemble de $\mathbb{N}$ par un des trois ensembles $\emptyset$, $\{0\}$ ou $\mathbb{N}$ (Cf. aussi une justification de la procédure présentée ci-dessus dans le §IV.1.2 de l'annexe de ce chapitre). D'autres approximations plus « fines » sont possibles et offrent un intérêt fondamental pour l'optimisation des programmes (Cf. le chapitre IX, §IV.6) et leur vérification (Cf. par exemple l'analyse statique des communications d'un programme de type OCCAM dans [Mercouroff 90]).

IV. Annexe au chapitre : démonstration des résultats

IV.1. La fonction *Texp* est correctement définie

Dans cette section on note par A, B… des éléments de $\Upsilon\Upsilon$IN, par a, b,… des éléments de ΥIN et par n, m… des éléments de IN.

IV.1.1. Le treillis $\Upsilon\Upsilon$IN

On munit $\Upsilon\Upsilon$IN de la relation d'ordre suivante :

$$A \leq B \;=_{def}\; A \subseteq B$$

le plus petit élément, noté ∅, est l'ensemble vide. Avec cette relation d'ordre, $\Upsilon\Upsilon$IN est un ensemble inductif au sens des chaînes (Cf. [Livercy 78]). Une chaîne est une suite croissante (A_i) et toute suite croissante a une borne supérieure que l'on note

$$\bigcup_i A_i$$

(elle correspond effectivement à l'*union ensembliste* des A_i). Une fonction f *continue au sens des chaînes* doit par définition vérifier :

$$\sup \{ f(A_i) \mid i \in IN \} = f(\bigcup A_i)$$

pour toute chaîne A_i. Toute fonction continue (au sens des chaînes) est croissante. $f(A_i)$ est donc une chaîne ce qui permet d'écrire que f est continue si et seulement si :

$$\bigcup f(A_i) = f(\bigcup A_i)$$

Le théorème qui justifie ces définitions est le suivant : Si (X, ≤) est un ensemble inductif et f une fonction continue de X vers X, alors f admet un plus petit point fixe unique, $FIX_X(f)$ qui vérifie

$$FIX_X(f) = \bigcup_n f^n(\ ).$$

IV.1.2. Continuité de F

Nous devons montrer que nous avons correctement défini *Texp*. *Texp* est défini par cas sur la structure d'une expression finie (car on utilise μ pour représenter les types infinis). Par suite, il suffit que la définition attachée à chaque cas soit correcte. Le seul cas qui pose problème est celui de la définition de l'interprétation de μ : il faut montrer que le point fixe utilisé existe.

Soit *F* définie par

$$F : (TID \times TENV \times TIME) \rightarrow (\Upsilon\Upsilon IN \rightarrow \Upsilon\Upsilon IN)$$

$$F(t, \eta, H) = \lambda K{:}\Upsilon\Upsilon IN.\ Texp\ [\ [H]\ ]\ (\eta[K/t])$$

Pour que la fonction *Texp* soit correctement définie il suffit que pour tout identificateur t, tout environnement η de TENV et pour toute expression H de TIME, la fonction *F* (t, η, H) de $\Upsilon\Upsilon$IN dans $\Upsilon\Upsilon$IN soit continue (attention à ne pas confondre la continuité de *F* avec la continuité de *F* (t, η, H)). Cette démonstration se fait en raisonnant par récurrence sur la structure de H.

Nous allons d'abord montrer la continuité de certaines fonctions auxiliaires :

$f1 = \lambda A.B$, où $B \in \Upsilon\Upsilon I\,N$

est une fonction constante donc continue. La fonction f2

$f2 = \lambda K.\{up(a) \mid a \in A\}$

vérifie

$$f2(\bigcup A_i)) = \{ up(a) \mid a \in \bigcup A_i \} = \bigcup \{ up(a) \mid a \in A_i \} = \bigcup f2(A_i)$$

donc f2 est une fonction continue. La fonction f3

$f3 = \lambda A, B . \{ f(a, b) \mid a \in A, b \in B \}$

est continue pour toute fonction $\mathtt{f}$ de $\Upsilon I\,N$ dans $\Upsilon I\,N$ car f3 vérifie

$$f3\,(\bigcup A_i, \bigcup B_i) = \{ f(a, b) \mid a \in \bigcup A_i, b \in \bigcup B_i \} = \bigcup_{i,j} \{ f(a, b) \mid a \ A_i \ b \ B_j \} = \bigcup_{i,j} f3(A_i, B_j)$$

donc la fonction f3 est continue. La fonction f4

$f4 = \lambda A.\{ f(b) \mid b \subseteq a \in \mathbf{A} \}$

est continue pour toute fonction f de $\Upsilon I\,N$ dans $\Upsilon I\,N$. En effet,

$$f4(\bigcup A_i) = \{ f(b) \mid b \subseteq a \in \bigcup A_i \} = \bigcup \{ f(b) \mid b \subseteq a \in A_i \} = \bigcup f4(A_i)$$

Les fonctions f5, f6 et f7 sont définies par

$f5 = \lambda A.\{ c \mid c \subseteq A \text{ et } max(c) \leq n \}$
$f6 = \lambda A.\{ c \mid c \subseteq A \text{ et } min(c) \geq n \}$
$f7 = \lambda A.\{ \{m\} \mid \{m\} \subseteq A \text{ et } m \geq n \}$

et sont clairement continues pour tout n.

Munis de ces résultats, nous pouvons passer à la démonstration de la continuité de $F(t, \eta, H)$. La longueur d'un terme H est la hauteur de son arbre syntaxique. La continuité de $F(t, \eta, H)$ pour les termes $\mathtt{H}$ de longueur 0 (les atomes de TIME), découle de la continuité des fonctions f1. Supposons que la fonction $F(t, \eta, H)$ soit continue pour tout environnement η et tout terme K de longueur inférieure à un entier n donné, alors nous pouvons montrer que $F(t, \eta, H)$ est continue pour tout environnement η et tout terme H de longueur n+1. En effet, un terme H de longueur n+1 prend une des formes suivantes

a) $H = \$K$
b) $H = K_1 + K_2$
c) $H = K_1 \text{ fby } K_2$
d) $H = \exists s.\subseteq K_1 \text{ after } K_2 . K_3$
e) $H = \exists s.\geq K_1 \text{ after } K_2 . K_3$
f) $H = \exists s.\leq K_1 . K_2$
g) $H = \exists s.\in K_1 . K_2$
h) $H = \mu s.K$

où K, K_1, K_2, K_3 sont des termes de profondeur inférieure ou égale à n. Pour les cas a) à g), la continuité de $F(t, \eta, H)$ se prouve en exprimant F comme composée de la fonction $F(t, \eta, K)$, ayant un argument K de longueur inférieure ou égale à n et donc continue par hypothèse, et d'une ou plusieurs fonctions prises parmi les fonctions continues f2 à f7. La composée de fonctions continues étant continue, on a le résultat cherché.

Si H est de la forme μs.K alors F(t, η, μs.K) peut s'écrire sous la forme suivante :

F(t, η, μs.K)
= λA:ϒϒI N. *Texp*⟦ ⟦μs.K⟧ ⟧ (η[A/t]) (1)
= λA:ϒϒI N. (FIX$_{ϒϒIN}$ (λB. *Texp*⟦ ⟦K⟧ ⟧ (η[A/t]) [B/s])) (2)
= λA:ϒϒI N. FIX$_{ϒϒIN}$ (F (s, η[A/t], K))

L'expression (2) est obtenue en remplaçant *Texp*⟦ ⟦μs.K⟧ ⟧ par sa valeur dans (1). On est assuré de l'existence de

FIX$_{ϒϒIN}$ (F (s, η[A/t], K))

car par hypothèse de récurrence, la fonction F (s, η[A/t], K) est continue (de ϒϒI N dans ϒϒI N pour s, η, A et K fixés). La fonction F(t, η, μs.K) est donc définie. Il faut montrer qu'elle est continue. Soit la suite de fonctions

g_0 = λB:ϒϒI N. ∅
g_{m+1} = λB:ϒϒI N. F(s, η[A/t], K)(g_m(B))

définie quels que soient s, η, A et K fixés. Il est facile de voir que chaque fonction g_m est continue (par récurrence sur m : g_0 est une fonction constante donc continue; supposons que la fonction g_m soit continue, alors g_{m+1} est la composée de deux fonctions continues, F(s, η[A/t], K) et g_m). Intéressons nous à la suite g_m(X) :

g_0(X) = ∅
g_{m+1}(X) = F(s, η[A/t], K)(g_m(X))

autrement dit, si on pose G = F(s, η[A/t], K), il vient que

g_{m+1}(X) = G^m (∅)

avec la convention que G^0 = g_0 = λx.∅. Par hypothèse de récurrence la fonction G est continue (car K est de longueur inférieure ou égale à n). Par conséquent la suite g_{m+1}(X) est une chaîne qui converge vers le plus petit point fixe de G. On a donc pour tout X et pour tout A :

⋃ g_m (X) = FIX (G) = FIX (F(s, η[A/t], K))

en particulier, prenons X = A = ⋃$_i$ A_i , il vient alors

FIX (F(s, η[⋃A_i /t], K))
= ⋃$_m$ g_m (⋃$_i$ A_i) *par* (3)
= ⋃$_m$ (⋃$_i$ g_m(A_i)) *car* g_m e*st continue*
= ⋃$_i$ (⋃$_m$ g_m(A_i))
= ⋃$_i$ FIX (F(s, η[A_i/t], K)) *par* (3)

Ceci montre la continuité de λA:ϒϒI N. FIX$_{ϒϒIN}$ (F (s, η[A/t], K)) et achève la récurrence.

Au passage on peut remarquer comment s'obtient le point fixe correspondant à l'interprétation d'une expression de la forme μt.F : on l'obtient par itération de F à partir de ∅. Si ∅ est une solution, l'itération s'arrête immédiatement puisqu'on à F(…) = ∅. Cela justifie la procédure de type-checking temporel qui a été proposée.

IV.2. Correction du typage des horloges

Def(e) dénote l'ensemble des tops pour lesquels e a une valeur non réduite à ⊥, c'est-à-dire :

Def(e) = *dset*(*Val*⟦ ⟦e⟧ ⟧*Env*⟦ ⟦e⟧ ⟧ε) ∩ *vset*(*Hor*⟦ ⟦e⟧ ⟧*Env*⟦ ⟦e⟧ ⟧ε)

avec *dset*(s) = { i | i ∈ I N et s.i ≠ ⊥ }
vset(s) = { i | i ∈ I N et s.i = *true* }

La fonction type(e) est une fonction qui, à une expression e, associe un élément de TIME tel que :

⋐ e ∴ type(e)

(N.B. : le type de e est obtenu à partir d'un environnement vide).

La propriété de cohérence du typage temporel se définit par :

$\forall$ e∈L2, *Def*(e) ∈ *Texp* [[type(e)]] ξ

où ξ est l'environnement vide (i.e. ξ (tid) = ∅).

Avant de passer à la démonstration proprement dite de la cohérence du typage (qui est assez indigeste), nous allons donner une sémantique formelle de L2 et établir quelques résultats préliminaires. Les résultats intermédiaires sont intéressants en-soi car ils donnent la « structure temporelle » d'un tissu : la suite des valeurs d'un tissu est de la forme ⊥* ; s où s ne contient pas d'éléments indéfinis. De plus, le premier élément de s correspond à un top d'horloge.

IV.2.1. Sémantique de L2

La sémantique de L2 est facilement déduite de celle de L1 en oubliant les collections et les opérations afférentes. Par contre nous introduisons explicitement la dénotation de until, after et at. Nous ne montrerons pas qu'elle est équivalente à l'expression de définition de until after et at en terme de when. On n'a pas fait apparaître d'opérateur de composition des définitions : on est en effet intéressé par le type d'une expression, pas par le type d'un programme. Cela n'exclut pas les tissus dont la définition est obtenue par des équations mutuellement récursives : il suffit de « déplier » en remplaçant un identificateur par sa définition jusqu'à tomber sur un système avec une seule équation récursive. Par exemple,

T = 1 + $Q;
Q = 1 + $T;

devient « T = 1 + $(Q = 1 + $T) ».

e ::=n
| **Clock**
| id
| id **=** e
| f **fby** g
| f **binop** g
| **$** f
| f **when** g
| f **until** g
| f **after** g
| f **at** g

VAL = $SCALAIRE^{\infty}$
HOR = $BOOLEAN^{\infty}$
ENV = ID → VAL × HOR

Val : EXP → ENV → $SCALAIRE^{\infty}$
Hor : EXP → ENV → $SCALAIRE^{\infty}$
Env : EXP → ENV → ENV

Env [[id = e]] *env* = *env* [⟨ *Val* [[id=e]] *env*, *Hor* [[id=e]] *env* ⟩ / id]

si e ne contient pas de liaison variable-valeur (i.e. e n'est pas de la forme id = e) :
Env [[e]] *env* = *env*

Val [[n]] *env* = cte (n)
Hor [[n]] *env* = *vrai* ; cte (*faux*)

Val [[Clock]] *env* = cte (*vrai*)
Hor [[Clock]] *env* = cte (*vrai*)

Val [[id]] *env* = *env* id . 0
Hor [[id]] *env* = *env* id . 1

$Val\,[\,[\,\mathrm{id} = \mathrm{e}\,]\,]\ env =$
$\quad Val\,[\,[\,e\,]\,]\ \mathrm{FIX}_{\mathrm{ENV}}\,(\lambda\rho{:}\mathrm{ENV}.\ (Env\,[\,[\,\mathrm{e}\,]\,]\,env)[\langle Val\,[\,[\,\mathrm{e}\,]\,]\,\rho,\ Hor\,[\,[\,\mathrm{e}\,]\,]\,\rho\rangle\ /$
id**]**)
$Hor\,[\,[\,\mathrm{id} = \mathrm{e}\,]\,]\ env =$
$\quad Hor\,[\,[\,e\,]\,]\ \mathrm{FIX}_{\mathrm{ENV}}\,(\lambda\rho{:}\mathrm{ENV}.\ (Env\,[\,[\,\mathrm{e}\,]\,]\,env)[\langle Val\,[\,[\,\mathrm{e}\,]\,]\,\rho,\ Hor\,[\,[\,\mathrm{e}\,]\,]\,\rho\rangle\ /$
id**]**)

$Val\,[\,[\,\mathrm{i\ fby\ j}\,]\,]\ env$ = fby (hor(i), val(i), hor(j), val(j))
$Hor\,[\,[\,\mathrm{i\ fby\ j}\,]\,]\ env$ = hfby (hor(i), val(i), hor(j), val(j))

$Val\,[\,[\,\mathrm{f\ binop\ g}\,]\,]\ env$ = bop (binop, $Val\,[\,[\,\mathrm{f}\,]\,]\ env$, $Val\,[\,[\,\mathrm{g}\,]\,]\ env$)
$Hor\,[\,[\,\mathrm{f\ binop\ g}\,]\,]\ env$ = hbop ($Hor\,[\,[\,\mathrm{f}\,]\,]\ env$, $Hor\,[\,[\,\mathrm{g}\,]\,]\ env$)

$Val\,[\,[\,\$\mathrm{e}\,]\,]\ env$ = délai ($Val\,[\,[\,\mathrm{e}\,]\,]env$, $Hor\,[\,[\,\mathrm{e}\,]\,]env$, ⊥, ⊥)
$Hor\,[\,[\,\$\mathrm{e}\,]\,]\ env = Hor\,[\,[\,\mathrm{e}\,]\,]\ env$

$Val\,[\,[\,\mathrm{e\ when\ f}\,]\,]\ env$ = trigger ($Val\,[\,[\,\mathrm{e}\,]\,]env$, $Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$, ⊥)
$Hor\,[\,[\,\mathrm{e\ when\ f}\,]\,]\ env$ = htrigger ($Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)

$Val\,[\,[\,\mathrm{e\ until\ f}\,]\,]\ env$ = avant ($Val\,[\,[\,\mathrm{e}\,]\,]env$, $Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)
$Hor\,[\,[\,\mathrm{e\ until\ f}\,]\,]\ env$ = havant ($Val\,[\,[\,\mathrm{e}\,]\,]env$, $Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)

$Val\,[\,[\,\mathrm{e\ after\ f}\,]\,]\ env$ = après ($Val\,[\,[\,\mathrm{e}\,]\,]env$, $Hor\,[\,[\,\mathrm{e}\,]\,]env$, $Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)
$Hor\,[\,[\,\mathrm{e\ after\ f}\,]\,]\ env$ = haprès ($Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)

$Val\,[\,[\,\mathrm{e\ at\ f}\,]\,]\ env$ = dirac ($Val\,[\,[\,\mathrm{e}\,]\,]env$, $Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)
$Hor\,[\,[\,\mathrm{e\ at\ f}\,]\,]\ env$ = hdirac ($Val\,[\,[\,\mathrm{f}\,]\,]env$, $Hor\,[\,[\,\mathrm{f}\,]\,]env$)

Les fonctions auxiliaires sont définies ci-dessous :

fby (h, s, l, t) = si (h.0 ∧ (s.0 ≠ ⊥)) alors s.0 ; fby2 (s.0, *rest*(l), *rest*(t))
sinon s.0 ; fby (*rest*(h), *rest*(s), *rest*(l), *rest*(t))
fby2 (c, h, s) = si (h.0 ∧ (s.0 ≠ ⊥)) alors s sinon c ; fby2 (c, *rest*(h), *rest*(s))

hfb (h, s, l, t) = si (h.0 ∧ (s.0 ≠ ⊥)) alors *true* ; hfby2 (*rest*(l), *rest*(t))
sinon *false* ; hfb (*rest*(h), *rest*(s), *rest*(l), *rest*(t))

hfb2 (h, s) = si (h.0 ∧ (s.0 ≠ ⊥)) alors *true* ; *rest*(h) sinon *false* ; hfb2 (*rest*(h), *rest*(s))

bop (s, t) = binop (s.0, t.0) ; bop (binop *rest*(s), *rest*(t))
hbop(h, l) = (h.0 $\underline{\vee}$ l.0) ; hbop (*rest*(h), *rest*(l))

délai (s, h, current, next) = si (h.0 ∧ (s.0≠⊥))
alors next ; délai (*rest* (s), *rest* (h), next, s.0)
sinon current ; délai (*rest* (s), *rest* (h), current, next)

trigger (i, j, h, v) = si ((j.0 ≠ ⊥) $\underline{\wedge}$ h.0 $\underline{\wedge}$ j.0) alors i.0 ; trigger (*rest* (i), *rest* (j), *rest* (h), i.0)
sinon v ; trigger (*rest* (i), *rest* (j), *rest* (h), v)

htrigger (j, h) = si (j.0 ≠ ⊥) alors (h.0 $\underline{\wedge}$ j.0) ; htrigger (*rest* (j), *rest* (h))
sinon *false* ; htrigger (*rest* (j), *rest* (h))

avant (s, t, h) = si (h.0 ∧ t.0≠⊥) alors cte(s.0) sinon s.0 ; avant (*rest*(s), *rest*(t), *rest*(h))
havant (h, t, l) = si (l.0 ∧ t.0≠⊥) alors h.0 ; cte(*false*)
sinon h.0 ; havant (*rest*(h), *rest*(t), *rest*(l))

après (s, h, t, l) = si (l.0 ∧ t.0≠⊥) alors fby2(⊥, h, s)
sinon après (*rest*(s), *rest*(h), *rest*(t), *rest*(l))
haprès (s, h, t, l) = si (l.0 ∧ t.0≠⊥) alors hfby2(*false*, h, s)
sinon *false* ; haprès (*rest*(s), *rest*(h), *rest*(t), *rest*(l))

dirac (s, t, h) = si (h.0 ∧ t.0≠⊥) alors cte(s.0) sinon ⊥ ; dirac (*rest*(s), *rest*(t), *rest*(h))
hdirac (t, h) = si (h.0 ∧ t.0≠⊥) alors *true* ; cte(*false*) sinon false ; hdirac (*rest*(t), *rest*(h))

IV.2.2. Quelques résultats préliminaires

Avant de passer à la démonstration de la cohérence du typage, nous avons besoin de montrer certains résultats préliminaires sur L2.

On admettra que toute opération binop vérifie la propriété P1 suivante :

∀ x, y ∈ *SCALAIRE*, binop (x, y) = ⊥ ⇒ (x = ⊥ ∨ y = ⊥); (P1)

Du point de vue de l'implémentation cela veut dire qu'une opération arithmétique ne produit pas de valeur indéfinie si ses arguments ne sont pas indéfinis. Pour cela il suffit de faire la distinction entre valeur indéfinie et valeur exceptionnelle (comme le résultat d'une division par 0, Cf. les remarques finales dans le chapitre sur la sémantique de 81/2). Il est donc légitime d'admettre la propriété P1.

Nous introduisons les définitions suivantes :

Un couple ⟨ s, h ⟩ d'éléments de $SCALAIRE^{\infty} \times BOOLEAN^{\infty}$ est un *couple normal* si et seulement si les deux points suivants sont vérifiés :

i) s est de l'une des trois formes suivantes :
s = ⊥ ; ⊥ ; ⊥ ; ⊥ ; ⊥ ; …
$s = \perp^n ; t$ avec $n \in \mathbb{N}^+$ et *dset*(t) = IN (i.e. ⊥ n'est pas un élément de t)
s = t avec *dset*(t) = IN (i.e. ⊥ n'est pas un élément de t)

ii) *min*(*dset*(s)) ∈ *vset*(t)

Un environnement *env* ∈ ENV est un *environnement normal* si et seulement si pour tout identificateur, *env*(id) est un couple normal.

Une expression e est une expression normale si les deux points suivants sont vrais :

i) ⟨*Val*[[e]]*env*, *Hor*[[e]]*env*⟩ est un couple normal pour tout environnement normal *env*

ii) *Env*[[e]]*env* est un environnement normal pour tout environnement normal *env*.

La propriété (P1) étant admise, nous allons montrer que :

les expressions de L2 sont normales (P2)

Cette propriété exprime que les tissus ont tous une certaine structure : un tissu est suite de valeurs définies préfixée par des valeurs indéfinies et la première valeur définie correspond à un top. La démonstration se fait par récurrence sur la longueur d'une expression. La propriété (P2) est vérifiée pour les expressions de longueur 0. Supposons-la vérifiée pour toute expression de profondeur inférieure ou égale à n. Une expression de longueur n+1 est de l'une des huit formes **a)** à **h)** :

a) e est de la forme « id = f »

Soit

$s = Val[[f]]\ \ FIX_{ENV}(\lambda\rho{:}ENV.\ (Env[[f]]env)[\langle Val[[f]]\rho,\ Hor[[f]]\rho\rangle / id])$

h = *Hor*⟦ f ⟧ FIX$_{ENV}$ (λρ:ENV. (*Env*⟦ f ⟧ *env*)⟦⟨ *Val*⟦ f ⟧ ρ, *Hor*⟦ f ⟧ ρ⟩ / id ⟧).

On doit montrer que ⟨s, h⟩ est un couple normal. Puisque f est une expression de longueur inférieure ou égale à n, il suffit de prouver que FIX$_{ENV}$ (λρ:ENV. (*Env*⟦ f ⟧ *env*)⟦⟨ *Val*⟦ f ⟧ ρ, *Hor*⟦ f ⟧ ρ⟩ / id ⟧) est un environnement normal.

Soit σ = FIX$_{ENV}$ (λρ:ENV. (*Env*⟦ f ⟧ *env*)⟦⟨ *Val*⟦ f ⟧ ρ, *Hor*⟦ f ⟧ ρ⟩ / id ⟧). On a

σ = φ(σ) = ⋃$_n$ φ^n (ε) (par définition du plus petit point fixe FIX$_{ENV}$)

avec :

φ = λρ:ENV. (*Env*⟦ f ⟧ *env*)⟦⟨ *Val*⟦ f ⟧ ρ, *Hor*⟦ f ⟧ ρ⟩ / id ⟧

ε = λid:ID.⟨⊥, ⊥⟩

⋃$_n$ ρ$_n$ = borne supérieure de la suite ρ$_n$

Pour i ∈ ID et i ≠ id, σ(i) = *env*(i) qui est un couple normal puisque *env* est par hypothèse un environnement normal. Il reste à établir que σ(id) est un couple normal.

Il est facile de montrer par récurrence que φ^n (ε) est un environnement normal pour tout n : ε est normal; soit ρ un environnement normal, alors φ(ρ) = (*Env*⟦ f ⟧ *env*)⟦⟨ *Val*⟦ f ⟧ ρ, *Hor*⟦ f ⟧ ρ⟩ / id⟧ est un environnement normal si (*Env*⟦ f ⟧ *env*) est un environnement normal et si ⟨ *Val*⟦ f ⟧ ρ, *Hor*⟦ f ⟧ ρ⟩ est un couple normal. Comme on peut appliquer l'hypothèse de récurrence à f, on a le résultat. Montrons à présent que si σ n'est pas un environnement normal, alors il n'est pas le plus petit des majorant de φ^n (ε). Il s'en suivra que σ est normal.

Si σ(id) = ⟨ *Val*⟦ f ⟧ σ, *Hor*⟦ f ⟧ σ⟩ n'est pas un couple normal alors soit la contrainte i) de la définition est violée, soit la contrainte ii). Supposons que ce soit la contrainte i) alors *Val*⟦ f ⟧ σ doit être de la forme ⊥^{p} ; a$_1$; … ; a$_q$; ⊥ ; s avec a$_1$, …, a$_q$ ≠ ⊥.

On a par définition de σ, φ^n (ε)(id) ≤ σ(id). Par suite *Val*⟦ f ⟧ φ^n (ε) ≤ *Val*⟦ f ⟧ σ = ⊥^{p} ; a$_1$; … ; a$_q$; ⊥ ; s. Or on vient de montrer que φ^n (ε) est un environnement normal et par suite, pour tout n, *Val*⟦ f ⟧ φ^n (ε) doit être de la forme ⊥$^{p+q+1}$; t (en effet, l'inégalité montre que le p+q+1 *ième* élément de *Val*⟦ f ⟧ φ^n (ε) doit être inférieur à ⊥, donc est égal à ⊥, ce qui permet d'affirmer que *Val*⟦ f ⟧ φ^n (ε) doit débuter par au moins p+q+1 éléments indéfinis). Il est donc facile de construire une suite ρ qui vérifie pour tout n : *Val*⟦ f ⟧ φ^n (ε) ≤ ρ < σ. Il suffit de prendre ρ = ⊥$^{p+q+1}$; s. Ceci contredit le fait que σ est la borne supérieure des φ^n (ε).

Supposons à présent que ce soit la contrainte ii) qui soit violée. On procède de la même façon (il existe un φ^m (ε) qui fixe le premier top ayant une valeur non ⊥; le préfixe de φ^m (ε) jusqu'à ce top doit aussi être un préfixe du majorant des φ^m (ε)).

Par suite σ(id) est un couple normal et par conséquent, σ est un environnement normal. D'où il vient que ⟨s, h⟩ est un couple normal.

Il reste donc à prouver que *Env*⟦ id = f ⟧ *env* est un environnement normal pour tout *env* normal. *Env*⟦ id = f ⟧ *env* = *env*⟦⟨ *Val*⟦ id=e ⟧ *env*, *Hor*⟦ id=e ⟧ *env*⟩ / id ⟧ qui est un environnement normal si ⟨ *Val*⟦ id=e ⟧ *env*, *Hor*⟦ id=e ⟧ *env*⟩ = ⟨s, h⟩ est un couple normal. Or c'est ce que l'on vient de montrer.

Les autres expressions ne modifient pas leur environnement (i.e. *Env*⟦ e ⟧ *env* = *env* pour tout e n'étant pas de la forme « id = f »). Dans la suite il suffit donc de prouver que ⟨ *Val*⟦ e ⟧ *env*, *Hor*⟦ e ⟧ *env*⟩ est un couple normal pour tout environnement normal *env*.

b) e est de la forme « f **fby** g »

Rappelons que l'on note val(e) = *Val*⟦ e ⟧ *env* et hor(e) = *Hor*⟦ e ⟧ *env*. Par hypothèse de récurrence ⟨val(f), val(g)⟩ est un couple normal. À partir de là, il est aisé de montrer en examinant les fonctions fby et fby2 que val(f fby g) est de la forme ⊥ ; ⊥ ; … ; ⊥ ; s.0 ; s.0 ; … ; s.0 ; r avec :

val(f) = ⊥ ; … ; ⊥ ; s et s libre de ⊥
val(g) = ⊥ ; … ; ⊥ ; t et t libre de ⊥
r un suffixe de t (i.e. t = t.0 ; t.1 ; … ; t.n ; r)

ce qui montre que val(e) a la forme appropriée. L'examen de la définition de hfb montre que le premier top de « f fby g » correspond au premier top de f pour lequel f a une valeur définie. Avec le résultat précédent sur la forme de val(e), cela suffit à montrer que : $min(dset(\text{val(e)})) \in vset(\text{hor(e)})$.

c) e est de la forme « f **binop** g »

Les fonctions binop vérifiant l'hypothèse donnée dans l'énoncé du résultat, il est facile de montrer que

bop (⊥ ; …; ⊥ ; s, ⊥ ; …; ⊥ ; t) = ⊥ ; …; ⊥ ; r

avec r = bop(s′, t′), s′ et t′ suffixes de val(f) et de val(t) respectivement, et avec la suite t ne contenant pas d'élément ⊥. Par ailleurs les valeurs pour lesquelles « f binop g » est défini sont, par suite de (P1), l'intersection des valeurs définies de f et de g. De plus, l'ensemble des tops de « f binop g » est obtenu par union des tops de f et g. D'où le résultat.

d) e est de la forme « **$**f »

Il est simple de montrer par examen de délai que val(e) = ⊥ ; … ; ⊥ ; val(f). Par ailleurs la suite val(e) prend une nouvelle valeur sur l'horloge de f qui est aussi celle de e. D'où la normalité de ⟨val(e), hor(e)⟩.

e) e est de la forme « f **when** g »

val(e) est obtenu en choisissant une suite d'éléments d'indice croissant dans val(f), ces indices correspondant à des valeurs à vrai dans hor(e). Par suite val(e) change de valeur sur les éléments à vrai de hor(e). D'où le résultat.

f) e est de la forme « f **until** g »

g) e est de la forme « f **after** g »

h) e est de la forme « f **at** g »

Les 3 opérateurs until, after et at peuvent se réécrire en des expressions ne contenant que les 5 opérateurs précédents. Le résultat suit. Ceci achève la démonstration de P2. ◊

Dans la suite on décide de noter :

val(e) = $Val[\![e]\!]Env[\![e]\!]\varepsilon$
hor(ε) = $Hor[\![e]\!]Env[\![e]\!]\varepsilon$

d'après P2 $Env[\![e]\!]\varepsilon$ est un environnement normal et par suite il est immédiat que :

$Def(e) = \{ n \mid n \in vset(\text{hor(e)}) \text{ et } n \geq min(dset(\text{val(e)})) \}$ (P3)

$min(dset(\text{val(e)}) \cap dset(\text{val(f)}) = min(\{min(dset(\text{val(e)}), min(dset(\text{val(f)})\})$ (P4)

$min(dset(\text{val(e)})) \in vset(\text{hor(e)})$ (P5)

Ces résultats seront utilisés lors de la démonstration par récurrence de la cohérence de *Texp*.

IV.2.3. Cohérence du typage

Nous posons les définitions suivantes :

La fonction *Def*(e, *env*) est définie par :

$Def(e, env) = dset(Val[\![e]\!]Env[\![e]\!]env) \cap vset(Hor[\![e]\!]Env[\![e]\!]env)$

On a donc : *Def*(e) = *Def*(e, ε).

La fonction type (e, ETH) est une fonction de L2×(ID → TIME) → TIME tels que :

ETH ≦ e ∴ type(e, ETH)

Les environnements *env* ∈ ENV, ETH ∈ TID → TIME et *tenv* ∈ TID → ϒϒIN sont cohérents pour une expression e si et seulement si *Def*(id, *env*) ∈ *Texp* [[type(id, ETH)]] *tenv*

Nous allons démontrer la propriété suivante :

Si les environnements *env*, ETH et *tenv* sont cohérents, alors pour tout e de L2,

Def(e, *env*) ∈ *Texp* [[type(e, ETH)]] *tenv* (P6)

La cohérence de *Texp* découle immédiatement de (P6). Nous allons démontrer (P6) pour une expression e par récurrence sur le nombre de symbole « = » dans e.

Nous allons d'abord montrer que le résultat est vrai pour une expression ne contenant pas de ce symbole. Nous prouvons ce premier résultat par induction sur la structure d'une dérivation de type. On suppose la cohérence pour les règles au numérateur pour déduire la cohérence de la règle en dénominateur. Il y a un cas pour chaque règle de typage (sauf (RT0) qui n'est pas une vraie règle mais une méta-règle permettant de spécifier ETH et (RT1) qui s'applique à une expression contenant un « = »).

(RT2, 3) :

Pour tout environnement *env*, ETH et *tenv* on a :

Def(n, *env*) = {0} ∈ {{0}} = *Texp* [[type(n)]] *tenv*
Def(Clock, *env*) = IN ∈ {IN} = *Texp* [[type(Clock)]] *tenv*

d'où immédiatement le résultat.

(RT5) :

Soient *env*, ETH et *tenv* cohérent pour f et g. On note F = *Texp* [[type (f, ETH)]] *tenv* l'interprétation du type de f et G l'interprétation du type de g. L'interprétation H de « f fby g » est égale (par RT5) à { *follow*(x, y) | x∈F, y∈G }. On veut donc montrer que :

dset(val(f fby g)) ∩ *vset*(hor(f fby g)) ∈ { *follow*(x, y) | x∈F, y∈G }

sachant que

a = *dset*(val(f)) ∩ *vset*(hor(f)) ∈F et b = *dset*(val(g)) ∩ *vset*(hor(g)) ∈G.

Pour cela, il suffit de montrer *dset*(val(f fby g)) ∩ *vset*(hor(f fby g)) = *follow*(a, b). Deux cas se présentent. Si a = ∅ alors f est réduit à ⊥ et il est facile de voir que « f fby g » aussi; cela est cohérent avec *follow*(∅, b) = ∅. Si a ≠ ∅ il est aisé de montrer successivement les propriétés suivantes en examinant les fonctions fby et fby2 :

- *min*(*dset*(val(f fby g)) ∩ *vset*(hor(f fby g))) = *min*(*dset*(val(f)) ∩ *vset*(hor(f))) = *min*(a)
- *dset*(val(f fby g)) ∩⊇ *vset*(hor(f fby g)) = *min*(*dset*(val(f)) ∩ *vset*(hor(f)))
  ∪ { n > *min*(*a*) | n ∈ *dset*(val(f fby g)) ∩ *vset*(hor(f fby g)) }
- { n > *min*(*a*) | n ∈ *dset*(val(f fby g)) ∩ *vset*(hor(f fby g)) }
  = { n > *min*(*a*) | n ∈ *dset*(val(g)) ∩ *vset*(hor(g)) }

d'où il découle : *dset*(val(f fby g)) ∩ *vset*(hor(f fby g)) = *min*(a) ∪ { n∈b | n>*min*(a) } = *follow*(a, b) ce qui montre le résultat.

(RT4) :

Soit F l'interprétation de f, G l'interprétation de g et H celle de « f binop g ». On doit montrer que :

$H = \{ n \mid n \in vset(\text{hor}(e)) \text{ et } n \geq min(dset(\text{val}(e))) \} \in \{ som(x, y) \mid x \in F,\ y \in G \}$

avec par hypothèse de récurrence (en utilisant P3) :

$a = \{ n \mid n \in vset(\text{hor}(f)) \text{ et } n \geq min(dset(\text{val}(f))) \} \in F$

$b = \{ n \mid n \in vset(\text{hor}(g)) \text{ et } n \geq min(dset(\text{val}(g))) \} \in G.$

Les opérations arithmétiques étant des fonctions strictes, on a : $dset(\text{val}(f \text{ binop } g)) = dset(\text{val}(f)) \cap dset(\text{val}(g))$. De par la définition de hbop, on a $vset(\text{hor}(f \text{ binop } g)) = vset(\text{hor}(f)) \cup vset(\text{hor}(g))$. En reportant dans l'expression de H, il vient que : $H = \{ n \mid n \in (vset(\text{hor}(f)) \cup vset(\text{hor}(g))) \text{ et } n \geq min(dset(\text{val}(f)) \cap dset(\text{val}(g))) \}$. En appliquant (P2) on a : $H = \{ n \mid n \in (vset(\text{hor}(f)) \cup vset(\text{hor}(g))) \text{ et } n \geq min(dset(\text{val}(f))) \text{ et } n \geq min(dset(\text{val}(g))) \}$. En appliquant (P5) on obtient : $H = \{n \mid n \in (vset(\text{hor}(f)) \cup vset(\text{hor}(g))) \text{ et } n \geq min(a) \text{ et } n \geq min(b) \} = som(a, b)$. D'ou le résultat.

(RT6) :

Soit F l'interprétation f. L'interprétation H de « $f » est égale (par RT6) à $\{ up(x) \mid x \in F \}$. On veut donc montrer que :

$\{ n \mid n \in vset(\text{hor}(e)) \text{ et } n \geq min(dset(\text{val}(e))) \} \in \{ up(x) \mid x \in F \}$

sachant que $a = dset(\text{val}(\$f)) \cap vset(\text{hor}(\$f)) \in F$. Or $vset(\text{hor}(\$f)) = vset(\text{hor}(f))$ et il est facile de montrer d'après la définition de délai que $min(dset(\text{val}(\$f))) = min(up(dset(\text{val}(f))))$. On a donc ce que l'on cherche.

(RT7) :

Le résultat est immédiat puisque $Def(e) \subseteq Def(g)$ et que $\bigcup_{A \in exist(G, F)} \{A\}$ contient tous les x tels que : $x \subseteq G = Def(g)$.

(RT8, RT9, RT10) :

La démonstration pour les trois autres opérateurs temporels utilise un argument semblable au cas « when ».

Ceci achève la démonstration de P6 dans le cas des expressions ne contenant pas de symbole « = ». ◊

Supposons la propriété P6 vraie pour toute expression contenant au plus *n* symboles « = » et montrons le résultat pour une expression e contenant *n*+1 « = ». À nouveau la démonstration se fait par induction sur la dérivation du type de e. Il suffit de considérer le cas de la règle (RT1).

Hypothèse de récurrence *(H1)* : Supposons que $Def(e, env') \in Texp[[\text{type}(e, ETH')]]\ tenv'$ pour tout environnement env', ETH' et $tenv'$ tels que env', $(ETH', id \therefore t)$ et $tenv'$ soient cohérents. Nous devons montrer que $Def(id = e, env) \in Texp[[\text{type}(i = e, ETH)]]\ tenv$ pour tout environnement *env*, ETH et *tenv* cohérents. Rappelons que la notation « ETH, id ∴ H » désigne un environnement semblable à ETH pour tout identificateur différent de id et de valeur H pour l'argument id.

En appliquant les définitions, il vient :

$Texp[[\text{type}(i = e, ETH)]]\ tenv'$

$= Texp[[\mu t.H]]\ tenv$

$= FIX_{\gamma\gamma\iota N}(\lambda K.\ Texp[[H]]\ tenv[K/t])$ (par définition de *Texp*)

$= Texp[[H]]\ tenv[FIX_{\gamma\gamma\iota N}(\lambda K.\ Texp[[H]]\ tenv[K/t]) / t]$ (par définition de FIX)

par ailleurs,

$Def(id = e, env) = Def(e, \sigma(env))$

avec $\sigma(env) = FIX_{ENV}(\lambda\rho{:}ENV.\ (Env[[e]]\ env)[\langle Val[[e]]\ \rho,\ Hor[[e]]\ \rho\rangle\ /\ id])$

on peut donc appliquer l'hypothèse *(H1)* et conclure si pour tout environnement *env*, ETH et *tenv* cohérents pour « id

= e », les environnements $env' = \sigma(env)$, $ETH' = ETH$ et $tenv' = tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t]$ soient tels que env', $(ETH', id \therefore t)$ et $tenv'$ sont cohérents. On a donc à montrer que pour tout i :

$$Def(i, \sigma(env)) \in Texp[\![type(i, (ETH, id \therefore t))]\!]\ tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t]$$

sachant que pour tout i :

$$Def(i, env) \in Texp[\![type(i, ETH)]\!]\ tenv$$

Posons $def(\langle s, h \rangle) = dset(s) \cap vset(h)$. Alors $Def(i, env) = def(env(i))$. Pour $i \neq id$, on a en appliquant les définitions :

$$env(i) = \sigma(env)(i) \qquad (a)$$

$$type(i, (ETH, id \therefore t)) = ETH(i) = type(i, ETH) \qquad (b)$$

$$Texp[\![type(i, ETH)]\!]\ tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t]$$
$$= Texp[\![type(i, ETH)]\!]\ tenv \qquad (c)$$

Cette dernière égalité est vraie car t n'apparaît pas dans type(i, ETH) : il est précisé dans la règle (RT1) que t n'est pas une variable libre dans H ce qui veut dire ici que, ou bien i n'apparaît pas dans e, ou bien i est une des variables libres de e (ensemble noté VL(e)). On utilise le résultat : si t n'apparaît pas dans H, alors pour n'importe quel K, $Texp[\![H]\!]\ tenv = Texp[\![H]\!]\ tenv[K/t]$. Cela demanderait en toute rigueur à être démontré (ce qui se ferait par induction sur l'expression d'un type). En utilisant *(H1)*, (a), (b) et (c) il vient immédiatement pour $i \in VL(e)$, la cohérence de env', $(ETH', id \therefore t)$ et $tenv'$.

Il reste donc à montrer que :

$$Def(id, \sigma(env)) \in Texp[\![type(id, (ETH, id \therefore t))]\!]\ tenv[FIX_{\gamma\gamma N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t]$$

or en appliquant les définitions il vient :

$$Def(id, \sigma(env)) = def(e, \sigma(env)) \qquad \text{(par les équations sémantiques)}$$

$$Texp[\![type(id, (ETH, id \therefore t))]\!]\ tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t]$$
$$= Texp[\![t]\!]\ tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t]$$
$$= tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])\ /\ t] \quad (t)$$
$$= FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t])$$
$$= Texp[\![H]\!]\ tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t]\ /\ t]$$

Pour démontrer l'égalité de ces deux quantités, nous allons les écrire comme limite d'une suite :

$$def(e, \sigma(env)) = def(e, \bigcup_n \varphi^n(\rho))$$
$$Texp[\![H]\!]\ tenv[FIX_{\gamma\gamma\iota N}\ (\lambda K.\ Texp[\![H]\!]\ tenv[K/t]\ /\ t] = Texp[\![H]\!]\ (\bigcup_n \phi^n(\eta))$$

avec φ et ϕ définies par :

$$\varphi(\rho) = (Env[\![e]\!]\rho)[\langle Val[\![e]\!]\rho, Hor[\![e]\!]\rho \rangle\ /\ id]$$
$$\phi(\eta) = \eta\,[Texp[\![H]\!]\ \eta\ /\ t]$$

Supposons à présent que l'on peut établir l'implication suivante :

$$def(e, \rho) \in Texp[\![H]\!]\ \eta \quad \Rightarrow \quad def(e, \varphi(\rho)) \in Texp[\![H]\!]\ \phi(\eta) \qquad (P7)$$

alors on aura aussi montré que :

$$def(e, \bigcup_n \varphi^n(env)) \in Texp[\![H]\!]\ (\bigcup_n \phi^n(tenv))$$

puisqu'en effet, on a par hypothèse *(H1)* : $def(e, env) \in Texp[\![H]\!]\ tenv$. On a donc pour tout n :

$$def(e, \varphi^n(env)) \in Texp[\![H]\!]\ (\phi^n(tenv))$$

et par suite

$$\bigcup_n def(e, \varphi^n(env)) \in \bigcup_n Texp[\![H]\!]\ (\phi^n(tenv))$$

Comme λx.def(e, x) et λy. *Texp*[[H]] y sont continues (nous ne le montrerons pas) on a :

$$\bigcup_n \text{def}(e, \varphi^n(env)) = \text{def}(e, \bigcup_n \varphi^n(env))$$
$$\bigcup_n \textit{Texp}[\,[H]\,](\phi^n(tenv)) = \textit{Texp}[\,[H]\,](\bigcup_n \phi^n(tenv))$$

d'où le résultat voulu :

$$\text{def}(e, \sigma(env)) = \textit{Texp}[\,[H]\,]\,tenv[\text{FIX}_{\gamma\gamma\text{IN}}\,(\lambda K.\ \textit{Texp}[\,[H]\,]\,tenv[K/t] / t]$$

La démonstration de (P7) se fait par induction sur la structure de l'expression e : si e est de la forme « id′=e′ » on a le résultat en utilisant l'hypothèse de récurrence. Si e est d'une autre forme, on retombe sur les démonstrations déjà faites à l'étape précédente.

Ceci achève la démonstration par récurrence de P5 d'où l'on déduit la cohérence du typage. ◊

VII. Un modèle d'exécution dynamique

Ce chapitre et les suivants abordent le problème de l'exécution d'un programme 81/2. Un modèle d'exécution d'un langage data-flow sur une machine parallèle doit répondre à trois questions : la **partition** des programmes ou *qu'est ce qu'une tâche* ? le **placement**, à savoir *sur quel processeur réside une tâche* ? et l'**ordonnancement**, c'est-à-dire *comment sont activées les tâches*? On peut classer les réponses à ces questions suivant l'approche adoptée : l'approche *statique* consiste à répondre à ces questions le plus tôt possible, à la compilation; l'approche *dynamique* retarde le moment de la réponse jusqu'à l'exécution.

Un programme 81/2 correspond à un graphe data-flow. Ce graphe est statique : les fonctions en 81/2 ne pouvant être récursives, il est possible à la compilation d'obtenir le graphe data-flow complet du programme. La *partition* d'un programme 81/2 est donc statique : un mécanisme dynamique de partition des programmes n'est pas nécessaire car tous les calculs à effectuer sont connus a priori. Pour simplifier l'exposé, nous allons supposer qu'une tâche est associée à chaque point d'un tissu. La quantité de calculs que l'on regroupe dans une tâche est connue sous le nom de *granularité*. L'association d'une tâche à chaque point d'un tissu produit des tâches de granularité très fine. Nous examinerons dans le chapitre IX le problème de savoir s'il faut augmenter cette granularité.

Ce chapitre examine les réponses « dynamiques » aux questions du placement et de l'ordonnancement. Nous commençons par décrire les stratégies d'ordonnancement dynamique qui correspondent à une *évaluation data-flow*. Ce schéma d'exécution a été très étudié. Nous ne l'avons pas implémenté, notre but étant simplement d'estimer les coûts dus à la gestion dynamique. Nous évoquons ensuite les problèmes du placement dynamique sur une architecture MIMD. Les conclusions motivent le développement d'un modèle d'exécution statique (ordonnancement et placement), ce qui est fait dans les deux prochains chapitres.

I. Les quatre variantes de l'exécution data-flow

Du point de vue de la sémantique dénotationnelle du langage, une implémentation data-flow peut se voir comme une *implémentation directe* de la dénotation d'un programme dans laquelle les fonctions sémantiques gardent en mémoire les dernières valeurs qui ont été évaluées (de manière analogue aux *mémo-fonctions* des lispiens). Une exécution data-flow correspond aussi à un mode de gestion dynamique des tâches.

Deux critères permettent de distinguer le comportement d'un nœud data-flow. Ces deux critères sont

orthogonaux et conduisent à des implémentations différentes. Le premier critère permet de fixer le comportement d'un arc du graphe data-flow : *statique* ou *dynamique*; le second critère permet de fixer le comportement d'un nœud : *data-driven* ou *demand-driven*. Cela conduit à quatre variantes du modèle d'exécution data-flow. Malheureusement, les adjectifs « statique » et « dynamique » sont utilisés pour qualifier le comportement d'un arc data-flow, mais dans tous les cas, il s'agit bien d'un ordonnancement dynamique (c'est-à-dire réalisé pendant l'exécution du programme).

Les implémentations existantes du data-flow correspondent principalement au data-flow dynamique data-driven (Cf. les tutoriels [Thakkar 87], [Sansonnet 91] et les différents articles déjà cités à propos du data-flow). Cependant cette approche présente des inconvénients, principalement dans le traitement des fonctions non strictes. Aussi proposons nous une implémentation de type demand-driven dans lequel un mécanisme de crédit permet de diminuer le coût des messages de requête. La stratégie d'évaluation finalement obtenue doit être assez proche (?) du mécanisme d'évaluation hybride data et demand-driven évoqué dans [Ashcroft 85].

I.1. Data-flow statique et dynamique

Un nœud data-flow correspond à une tâche. Ce nœud calcule l'horloge et la valeur d'un point. Chaque nœud se caractérise donc par deux fonctions qu'il applique à ses entrées : l'une des fonctions correspond au calcul de l'horloge et la seconde au calcul de la valeur. Les arcs du nœud data-flow correspondent aux communications entre tâches (un arc est associé à chaque variable d'une expression).

Un arc dans un graphe data-flow peut être vu comme une file d'attente servant de buffer entre deux nœuds. Les données envoyées sur un arc ressortent suivant une politique « premier entré, premier sorti ». Si cette file d'attente a une longueur limitée à 1, on est dans le cas du data-flow statique [Dennis 91].

Un autre comportement possible est celui d'un tampon : Les données envoyées sur l'arc sont simplement stockées au fur et à mesure de leur arrivée. Dans ce cas on a pas nécessairement besoin de préserver l'ordre d'arrivée. C'est le cas par exemple si la messagerie est asynchrone (les messages arrivent dans un ordre arbitraire). Le nœud récepteur doit cependant accéder aux données suivant un certain ordre. Cet ordre est réalisé en associant un *tag* à chaque donnée (*tagged token*). Ce tag représente une position dans la séquence des données produites par un nœud. C'est le data-flow dynamique.

La différence entre les deux approches consiste en ce que l'allocation mémoire de toutes les données a été effectuée dans le data-flow statique *à la compilation*. Le data-flow statique économise un tag et correspond à la stratégie naturelle dans le cas où la communication entre nœuds se fait sans tamponnement (i.e. un nœud source ne peut envoyer une donnée tant que la précédente n'a pas été acceptée). Mais l'activation d'une tâche est dynamique, même dans un modèle data-flow statique, comme nous le verrons un peu plus loin.

Le data-flow dynamique est intéressant lorsque la communication entre nœuds est tamponnée. La bufferisation des messages (i.e. l'émetteur n'a pas à attendre la réception du message par le receveur) présente deux avantages :

- Le tamponnement des messages permet d'adapter les débits en entrée et en sortie des nœuds, ce qui permet un meilleur rendement des ressources de calculs.
- Dans certains cas, il n'est pas nécessaire de traiter les données dans l'ordre. Si la messagerie est asynchrone, on fait alors l'économie du réordonnancement des messages[1].

Cependant le coût de la gestion du tamponnement peut être élevé : il faut gérer le fait que les buffers n'ont pas une longueur infinie. (Ce coût de gestion peut parfois être réduit s'il est implicitement réalisé par le placement des tâches. En effet, la plupart des nœuds ne produisent une sortie qu'à partir d'une entrée. Si le graphe est cyclique, alors un

[1] Les opérateurs dont le comportement est indépendant de l'historique des valeurs comprennent par exemple les nœuds qui appliquent une fonction à leurs entrées mais aussi les nœuds dont la sortie est une combinaison associative et commutative de la succession des entrées.

placement judicieux des nœuds permet de contrôler la production des messages sur un processeur P1 par la production des messages sur un processeur P2, et vice-versa).

I.2. Data-flow demand-driven et data-driven

Le comportement d'un nœud data-flow peut être glouton ou paresseux. C'est-à-dire que son comportement peut-être déclenché uniquement par la présence des données à traiter (mode data-driven) ou bien être activé par une requête (mode demand-driven).

Dans le mode d'évaluation demand-driven, l'activation d'un nœud est conditionnée par une demande provenant de l'un des nœuds successeurs. Si les données nécessaires à l'élaboration de la réponse ne sont pas présentes, une demande est propagée vers les prédécesseurs. Chaque nœud réalise donc l'algorithme suivant :

```
pour toujours
    attendre les requêtes
    propager les requêtes (si nécessaires et aux arguments dont on a besoin)
    attendre les valeurs arguments
    faire le calcul de l'horloge
    si c'est un top faire le calcul de la nouvelle valeur
    propager le résultat
```

I.3. Les problèmes du data et du demand-driven

Le comportement data-driven doit être modulé afin de tenir compte des opérateurs non stricts. Un opérateur non strict est un opérateur qui ne requiert pas tous ses arguments pour avoir une valeur. La conditionnelle est un exemple d'opérateur non strict : si la condition s'évalue à vrai, les arguments correspondant à l'évaluation de l'expression « fausse » ne sont pas requis. Pis, il ne faut pas évaluer cette expression car le calcul correspondant peut boucler ou produire de « fausses erreurs » :

if $x == 0$ then 0 else $1/x$ fi (E1)

est une expression qui a une valeur pour tout x en entrée mais qui provoque une erreur si son implémentation conduit au calcul systématique de $1/x$ pour chaque x. Dans un mode d'évaluation data-driven, le comportement de la conditionnelle ne doit pas anticiper sur la valeur de la condition (parallélisme spéculatif) mais doit retarder la propagation des calculs jusqu'à l'arrivée de cette valeur. Pour cela la conditionnelle est traditionnellement implémentée, [Dennis 74], en utilisant des nœuds de blocage et de fusion (ou de sélection et de distribution, Cf. figure 1).

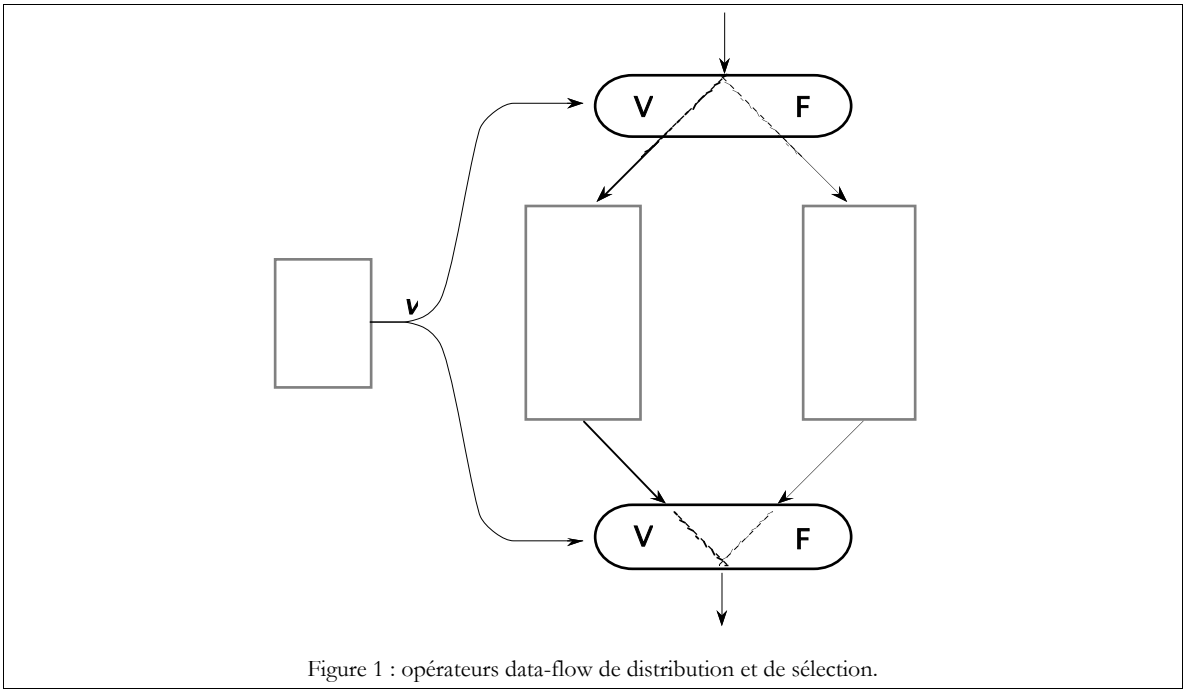


Figure 1 : opérateurs data-flow de distribution et de sélection.

D'autres opérateurs non-stricts en 81/2 posent des problèmes à l'évaluation data-driven : les opérateurs until et at sont au mieux inefficaces. En effet, le sous-graphe data-flow correspondant aux arguments de until et at continue à produire des valeurs qui ne sont plus utilisées après la première valeur à vrai de la condition. Enfin, la production de constantes (nœuds sans antécédents) est un point délicat. Si le taux de production est trop grand il y a « éblouissement » du réseau (phénomène de *hot-spot*). S'il est trop faible, il y a perte de parallélisme.

Les arguments précédents plaident pour un mode d'évaluation demand-driven qui ne présente pas ces problèmes. Il est cependant difficile de renoncer au parallélisme apporté par le data-driven. En effet, l'exécution demand-driven détruit en première approche le parallélisme entre opérateurs : un nœud n'effectue un calcul que quand le calcul correspondant à la demande précédente est complètement terminé.

Cet effet indésirable peut disparaître si le flot des demandes est indépendant du flot des réponses (i.e. on n'attend pas la fin du calcul d'une valeur avant de demander le calcul de la valeur suivante). Cela n'est possible que pour les parties du graphe qui supportent sans danger une évaluation data-driven. Ces parties sont traditionnellement détectées par une analyse de *stricticité* (grossièrement, il ne faut pas évaluer les conditionnelles en data-driven). Il faut alors contrôler le débit des demandes afin de ne pas surcharger le réseau. E. Payan propose dans [Payan 91] d'anticiper simplement les demandes sur les calculs (Cf. figure 2). Mais cette solution ne donne de bons résultats que quand le rapport temps de communication/temps de calcul de chaque nœud est proche de 1. Une généralisation consiste à utiliser un mécanisme de *crédit* : chaque nœud demande les *n* prochaines valeurs. Cela implique l'utilisation de file d'attente et donc de tag. L'autorisation suivante de crédit a lieu quand *m* valeurs ont été consommées, avec *m* proche mais inférieur à *n*. La valeur de *m* est ajustée suivant le temps d'attente du processeur élémentaire (PE) et le taux de bufferisation des messages.

C'est pourquoi nous combinons les deux modes d'exécution en espérant économiser sur le coût de propagation des demandes.

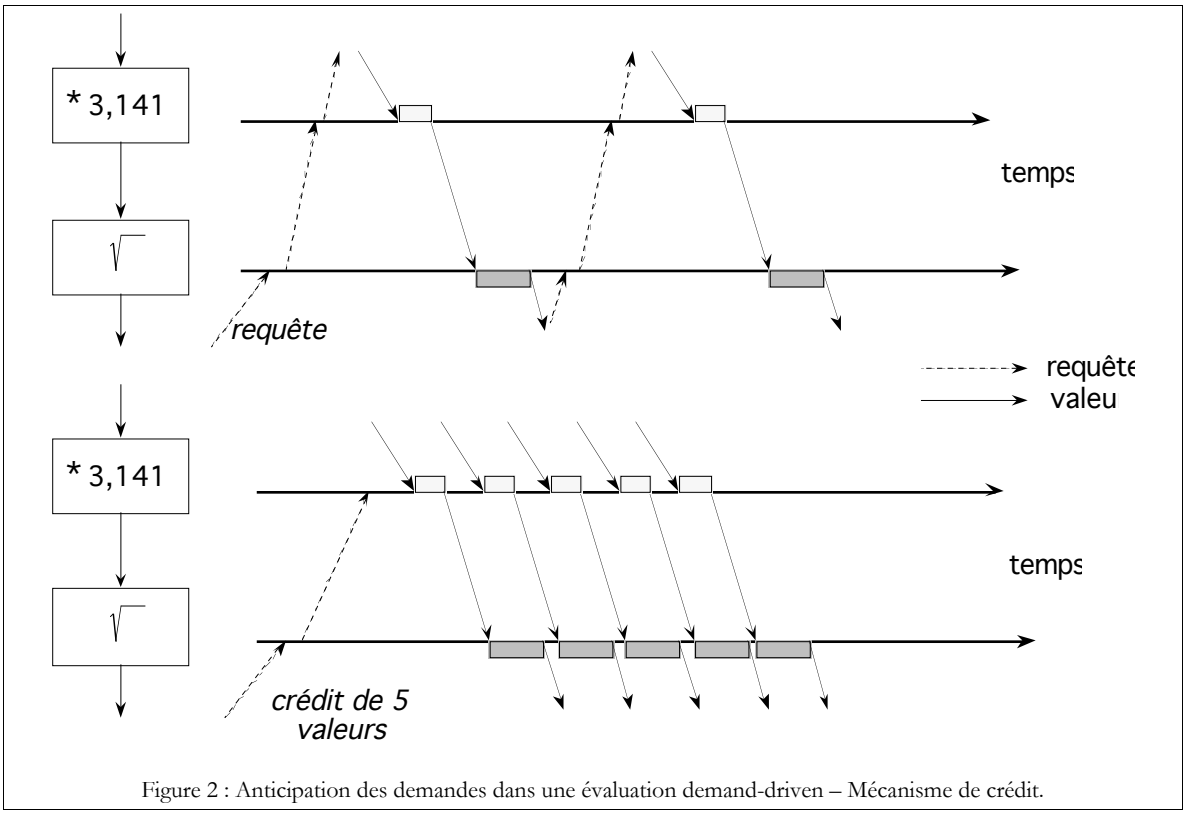


Figure 2 : Anticipation des demandes dans une évaluation demand-driven – Mécanisme de crédit.

II. Évaluation mixte

Eazyflow est une architecture hybride basée sur LUCID, qui essaie de mixer explicitement les deux modèles [Ashcroft 85]. L'idée est de distinguer des nœuds ayant un comportement data-driven et des nœuds ayant un comportement demand-driven. L'interaction entre les deux types de nœuds est assez confuse. Un nœud demand-driven doit fixer une limite au nombre de valeurs qu'un nœud data-driven lui envoie. Ainsi un nœud data-driven doit réagir différemment en fonction de ses sorties. De plus un nœud data-driven dont une entrée est demand-driven se transforme en nœud demand-driven.

Nous préférons généraliser le mécanisme de crédit présenté plus haut en restant ainsi dans la ligne d'une évaluation demand-driven (Le mécanisme de demande de crédit présente des analogies avec le mécanisme de *sponsors* que l'on peut trouver dans certains langages acteurs [Giavitto 86]). Chaque nœud transmet à ses *prédécesseurs* une **demande impérative**, une **autorisation de crédit** ou bien une **annulation**. Il peut aussi transmettre à ses *successeurs* un **testament**.

II.1. Demande impérative, autorisation de crédit et annulation

Il y a trois sortes de messages adressés par un nœud à ses prédécesseurs :

Demande impérative de tic :

Le receveur est requis de fournir la valeur d'un tic donné. Cette requête correspond au modèle demand-driven pur. La réponse fournie peut être l'indication d'un tic ou bien un top (auquel cas il faut fournir la valeur du top).

Autorisation de crédit de tic :

Le receveur est autorisé à envoyer au plus *n* valeurs à partir de la dernière valeur précédemment autorisée (l'envoi des valeurs correspondant au tic 0 est implicitement autorisé).

Le crédit *n* correspond à un nombre de tics que le receveur R à le droit d'exécuter. R enverra des valeurs vers l'émetteur du crédit uniquement si le calcul d'un tic donne lieu à un top. Le numéro du tic courant est joint à toutes les valeurs envoyées.

Annulation :

Le receveur est averti qu'il ne faut plus envoyer de valeurs. Il se peut que certaines valeurs soient déjà en chemin (cas de la communication asynchrone). L'émetteur de la demande (qui va recevoir ces valeurs) doit donc être prêt à traiter la réception d'un nombre de valeurs au plus égal à la dernière autorisation de crédit (dans le cas d'une autorisation limitée). Une annulation est définitive : aucune autorisation ne peut suivre une annulation.

Une autorisation ou annulation de crédit est émise par :

- *L'exécutif* 81/2 *:*

  Le résultat d'un programme consiste en *la suite des valeurs d'un seul tissu* (mais ce tissu peut être un bloc). Ce tissu correspond « à une sortie » qui est traitée par le système d'exploitation (on peut la visualiser, l'enregistrer dans un fichier, etc). C'est l'exécutif qui se charge de transmettre l'autorisation de crédit afin d'enregistrer les *n* prochaines valeurs. La détermination de *n* est un problème lié à la taille des buffers de communications. Si *n* est trop grand, les buffers de communications risquent de s'engorger. Si *n* est trop petit, l'ordinateur risque d'être sous-exploité. L'exécutif réémet une autorisation de crédit chaque fois que *m* valeurs ont été produites. La détermination de *m* pose un problème.

- *Un nœud* until *ou* at *sur réception d'une valeur vraie*

  Les nœuds until et at envoient à leurs prédécesseurs une annulation quand ils reçoivent la première commande ayant une valeur « vraie ».

- *Un nœud quelconque sur réception d'une autorisation ou d'une annulation*

  Quand un nœud reçoit une autorisation ou une annulation de crédit il est habilité à propager la demande :

  Propagation d'une annulation :

  Sur réception d'une annulation un nœud peut décider de propager l'annulation vers ces prédécesseurs et de se suicider. Cette décision est prise s'il n'y a pas d'autres nœuds successeurs (chaque nœud peut tenir compte du nombre de ses successeurs). On implémente ainsi une sorte de *récupération* des tâches mortes (garbage) par comptage des références. Le suicide est retardé jusqu'à la réception du dernier message des prédécesseurs.

  Propagation d'une autorisation :

  Sur réception d'une autorisation de crédit, un nœud peut retransmettre cette autorisation à ses successeurs (s'il en a). Cette retransmission est retardée jusqu'à ce que le nombre de messages reçus par le nœud atteigne une certaine valeur *m* (inférieure à la valeur *n* qu'il a accordée lors de la précédente autorisation).

II.2. Testament

Les nœuds until et at ne font plus de calculs utiles après l'occurrence d'une commande à vrai. Ils avertissent alors leurs successeurs en signalant que la dernière valeur envoyée est un *testament*. Dans le cas où le nombre de messages envoyés par le nœud avant son testament excède le seuil m des successeurs (ce seuil a été communiqué avec leur autorisation de crédit), le suicide du nœud est retardé jusqu'à réception de la nouvelle autorisation de crédit. Sinon le suicide a lieu immédiatement.

Sur réception d'une valeur testamentaire, un nœud receveur est assuré de ne plus recevoir de valeurs ultérieures. Il peut à son tour envoyer un testament (s'il n'a pas d'autres prédécesseurs).

II.3. Synchronisation

Le crédit n reçu par un nœud N correspond à un nombre de tics que N à le droit d'exécuter. N enverra des valeurs vers l'émetteur du crédit uniquement si le calcul d'un tic donne lieu à un top. Le numéro du tic courant est joint à toutes les valeurs envoyées. En effet, les opérateurs binaires et ternaires demandent d'apparier des valeurs de même tic.

Supposons que B soit un nœud correspondant à un opérateur binaire (une addition ou bien un trigger). Quand B reçoit une valeur (correspondant à un top), il a besoin de connaître parfois l'état correspondant de son autre argument. Par exemple

$$B = T + Q \qquad \text{(E2)}$$

$$B = T \text{ when } Q \qquad \text{(E3)}$$

dans l'exemple (E2) l'arrivée d'une valeur pour T sur un tic t demande à connaître la valeur correspondante de Q sur t. Ce n'est pas le cas dans l'exemple (E3) (mais par contre l'arrivée d'une valeur pour Q requiert la valeur correspondante de T).

Plaçons nous dans l'hypothèse d'une communication synchrone ou déterministe. Deux cas sont alors possibles (Cf. Figure 3, où on étudie le comportement de $C = A + B$) :

1) La valeur requise (correspondant au tic t) est disponible sur place. Ce cas se produit si la valeur d'un tic $t' > t$ a été reçue[1]. Dans ce cas on est assuré de connaître la valeur requise. Afin de conserver cette valeur, il suffit de conserver toutes les valeurs correspondant à des tics ultérieurs au plus petit des tics reçus en entrée. La production de la valeur de c ne pose aucun problème. La production d'une valeur correspondant au plus petit des tics conservés déclenche l'élimination des valeurs correspondant à ces tics.
2) La valeur requise n'est pas disponible. Cela est causé par l'une des trois raisons suivantes :
   - la valeur requise n'a pas encore été calculée;
   - la valeur requise a été calculée mais n'a pas été communiquée car elle correspond à un tic et le calcul de tic ultérieur n'a donné lieu à aucun un top
   - la valeur requise a été calculée et la valeur d'un tic supérieur ou égal au tic requis a été communiquée, mais le message est en chemin.

   Quelle que soit cette raison, le nœud C émet une demande impérative. Dans le deuxième et troisième cas, le prédécesseur de C qui reçoit cette requête n'a aucun mal à y répondre. Dans le premier cas il y a anticipation des calculs.

[1] Dans le cas de la communication déterministe, les messages ne se doublent pas.

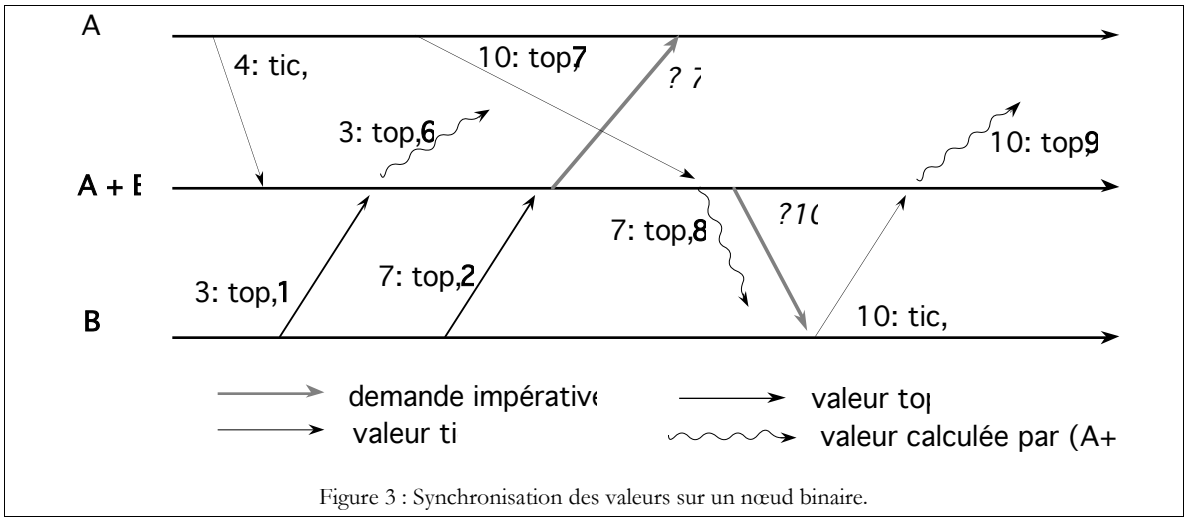


Figure 3 : Synchronisation des valeurs sur un nœud binaire.

Quand un nœud reçoit à la fois une demande impérative et une demande crédit, il satisfait les deux demandes (le nœud n'est pas obligé de répondre une valeur à une autorisation de crédit si toutes les valeurs subséquemment obtenues sont des tics, par contre une réponse est requise dans le cas d'une demande impérative). Si par chance il a déjà répondu à la demande impérative (à travers un message envoyé précédemment *pour le même* tic), il n'a pas besoin de répondre une deuxième fois (le message finira par arriver).

Il y a bien sûr intérêt à minimiser les demandes de synchronisations qui sont cause d'émissions de demandes impératives. Pour cela il peut y avoir intérêt à grouper les communications de valeurs. Par exemple un nœud peut stocker les valeurs qu'il calcule jusqu'à la réception d'une demande impérative. Il émet alors les valeurs déjà calculées (en profitant incidemment de la diminution relative du coût de la communication due au groupage) puis il émet la réponse à la demande impérative). Cela n'est pas toujours possible, en particulier quand le programme est séquentiel (Cf. l'exemple donné un peu plus loin).

Dans le cas où l'on dispose uniquement d'une communication asynchrone, il faut s'assurer que les messages concernant un tic donné ne sont pas reçus avant les messages correspondant à des tics antérieurs. Cela est possible en utilisant une méthode d'estampillage ou de réordonnancement des messages. Une solution simple peut consister à émettre un message pour chaque tic et à ordonner les messages reçus en fonction de cette datation.

II.4. Implémentation des délais

L'implémentation des délais est implicite. Elle correspond simplement à ajuster l'appariement des messages reçus : la valeur du top t est émise pour le top t+1 sur réception de celui-ci.

II.5. Implémentation des conditionnelles

L'implémentation des conditionnelles se fait habituellement à travers plusieurs nœuds de sélection et de diffusion. La propagation des crédits pose un problème dans ce cas : à quelle branche faut-il propager la requête? Nous voulons éviter le parallélisme spéculatif et les problèmes qui s'y attachent[1]. Il est plus simple alors de

[1] Le problème du calcul d'une valeur erronée (comme dans l'exemple E1) ne présente pas trop de difficultés : l'erreur est rattrapable en introduisant une valeur indéfinie. Mais il peut se poser un problème plus grave qui n'est pas facilement résolu : le calcul déclenché peut très bien boucler. C'est possible en 81/2 dans le cas d'un every dans une branche de conditionnelle (on peut par exemple imaginer un every dont la condition d'arrêt est l'expression qui conditionne le choix de la branche contenant le every).

considérer que la conditionnelle constitue un opérateur atomique (un seul nœud). L'autorisation de crédit est propagée aux entrées correspondant au calcul de la condition. Au fur et à mesure de la réception des valeurs de la conditionnelle, une *demande impérative* est propagée aux entrées nécessaires aux calculs de l'expression sélectionnée par la condition.

Si on tient au parallélisme spéculatif, il est toujours possible d'analyser les expressions afin de déterminer si elles présentent un danger (de bouclage) on non. C'est l'analyse de stricticité [Mycroft] [Cousin 90]. Si les expressions sont inoffensives, alors on peut utiliser le mécanisme de crédit.

II.6. Implémentation des every

L'implémentation des every demande de collecter toutes les valeurs correspondant à un top donné, de dérouler le calcul correspondant à l'activation du réseau puis de récupérer le résultat. Cela peut se faire de la manière suivante :

- les nœuds correspondant au corps de every sont activés par un message de contrôle émis par le nœud calculant l'horloge du trigger de l'every.
- Sur réception de ce message, chaque nœud demande impérativement les valeurs correspondant au tic de réveil.
- L'exécutif 81/2 génère les crédits de tics nécessaires jusqu'à l'obtention du top d'arrêt. Sur réception de ce top, un testament est émis vers chaque nœud de l'every. L'effet de ce testament n'est pas le suicide des nœuds mais leur endormissement.

Ce comportement est semblable au comportement demand-driven pur des conditionelles.

II.7. Génération des crédits : problèmes et optimisations

Le problème suivant peut se poser : l'exécutif a émis une autorisation de crédit et ne voit rien venir. Deux cas sont possibles : le programme boucle ou bien le crédit a été épuisé sans qu'il y ait eu production d'un top.

L'exécutif peut alors émettre une demande impérative. Si elle aboutit, c'est qu'aucun top n'avait été émis et l'exécutif peut réémettre une autorisation de crédit. Si la demande n'aboutit pas, on est dans le cas douloureux des programmes dont on ne sait s'ils sont en train de boucler ou bien si le calcul est très long. L'équivalent 81/2 d'un programme qui boucle est un programme comportant un every qui ne s'arrête pas.

Un autre problème est la détermination du seuil à partir duquel une nouvelle autorisation de crédit est émise. Ce seuil commande le recouvrement des demandes et par suite a un effet sur le trafic des demandes. Un ajustement adaptatif, tenant compte des taux d'occupation des CPU est sans doute nécessaire.

Enfin, il est possible d'optimiser la propagation des crédits. Supposons qu'un nœud *N* a deux arguments et que tous les deux sont absolument requis pour le calcul du nœud (le nœud réalise une fonction stricte). Alors les successeurs de *N* n'ont pas besoin de lui envoyer une autorisation de crédit, car elle résulterait simplement en sa retransmission vers les prédécesseurs de *N*. Les successeurs de *N* peuvent envoyer leur autorisation directement vers les prédécesseurs de *N* : à l'arrivée des données, *N* effectuera son calcul de manière data-driven. D. Skillicorn présente dans [Skillicorn 91] une méthode pour analyser si une fonction LUCID est stricte ou pas.

II.8. Un scénario d'exécution

Nous proposons d'illustrer le modèle d'exécution dynamique par un scénario correspondant à une exécution possible du programme :

T@0 = 0 ; T = $V ;
V = T + (1 when H) ;

Ce programme correspond à trois tâches T, V et H (Cf. figure 4). L'exemple n'est pas choisi en fonction du parallélisme qu'il exhibe (le programme est purement séquentiel) mais parce qu'il illustre l'implémentation d'une équation récursive.

Les autorisations de crédit se font en respectant l'ordonnancement des valeurs à calculer. On ne s'intéresse pas au calcul effectué par H. Le calcul de l'horloge de V correspond à $h \vee t$, où h est l'horloge de H et t l'horloge de T. V doit donc recevoir les tops de H et de T. L'horloge de T correspond à h : T doit donc recevoir les tops de H. Évidemment il est possible de simplifier ces expressions (Cf. par exemple [BHSV 90]) et d'éviter une propagation supplémentaire de messages entre T et V. Cependant nous allons conserver ces expressions telles quelles dans le déroulement du scénario.

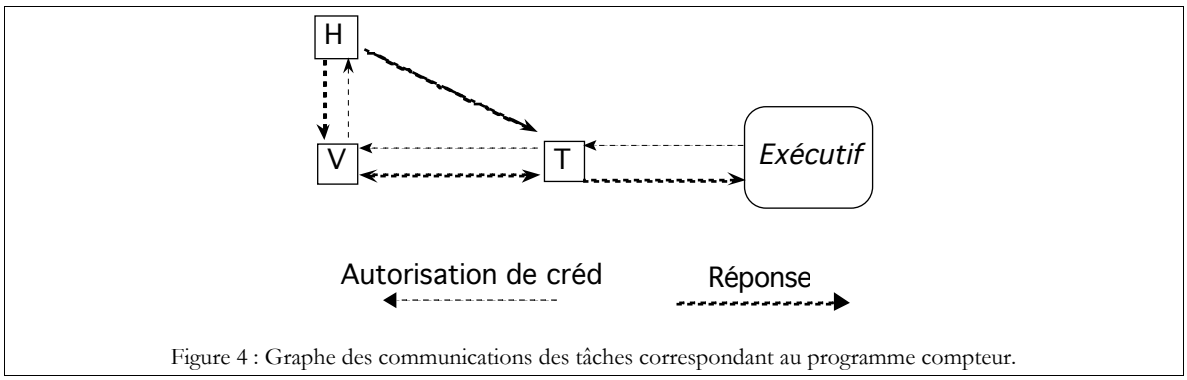


Figure 4 : Graphe des communications des tâches correspondant au programme compteur.

Le scénario est détaillé à la page suivante. On suppose que les messages issus d'un même émetteur et à destination d'un récepteur donné arrivent dans l'ordre d'envoi. On utilise les conventions suivantes :

X → c *n*	réception en provenance de X d'un crédit de *n* tics	**X ← c *n***	émission vers X d'un crédit de X de *n* tics
X → *t, v*	réception en provenance de X de la valeur *v* de tic *t*	**X ← *t,v***	émission vers X de la valeur *v* de tic t
X → d *t*	réception d'une demande impérative en provenance de X de la valeur du tic t	**X ← d *t***	émission vers X d'une demande impérative de la valeur du tic t

H	V	T
		• exécutif → c 10 • V ← c 10 • V, exécutif ← top = 0, val = 0
	• T → c 10 • H ← c 10	
• V → c 10 • T, V ← top = 0, val = false		
	• T → top = 0, val = 0 • H → top = 0, val = false • T ← top = 0, val = 1	• H → top = 0, val = false
…		• V → top = 0, val = 1 *sur réception du prochain top de H, cette valeur deviendra la valeur de T pour ce top là*
• T, V ← top = 3, val = true		• H → top = 3, val = true • V, exécutif ← top = 3, val = 1
• T, V ← top = 7, val = true	• T → top = 3, val = 1 *V ne connaît pas la valeur de H au top 3 et décide d'envoyer une demande impérative (en fait le message est en chemin)* • H ← d 3	
• V → d 3 *H a déjà envoyé un message correspondant au tic 3 : il décide de ne rien faire*	• H → top = 3, val = true *le message concernant le tic 3 arrive* • T ← top = 3, val = 2	• H → top = 7, val = false *T ne sait pas s'il existe des valeurs de V entre le top 3 et le top 7 car il n'a rien reçu. Il décide d'envoyer une demande impérative à V* V ← d 7
	• H → top = 7, val = false • T → d 7 *V peut élaborer la réponse :* • T → top = 7, val = 3	• V → top = 3, val = 2 *La réception de ce message ne répond pas à la demande impérative, T attend toujours*
		• V → top = 7, val = 3 *Maintenant T est fixé : le seul top qui précède le tic 7 est le top 3* • V, exécutif ← top = 7, val = 2
		• exécutif → c 10 *La réception du top précédent (7) par l'exécutif a déclenché une nouvelle autorisation de crédit*

II.9. Granularité des tâches

La gestion des demandes, crédits, annulations et testaments entraîne un nombre de messages négligeable si on choisit un *n* assez grand. Par contre la gestion de la synchronisation peut entraîner un coût non négligeable. Il y a donc tout intérêt à avoir des nœuds les plus « gros » possible afin de réduire autant que possible le ratio temps de communication/temps de calcul. En effet, dans l'état actuel de la technologie, le débit des unités de calcul est supérieur à celui des réseaux de communication. Un ratio temps de communication/temps de calcul proche de 0 correspond à une application « compute bound » qui utilise au mieux les ressources de la machine (si toutefois tous les PE sont saturés).

Si on désire faire grossir une tâche, deux voies sont possibles : grouper les expressions, afin de générer des tâches plus grosses, ou bien regrouper les tâches qui correspondent au calcul de différents points du même tissu. Dans ce dernier cas on perd du parallélisme mais ce parallélisme n'était peut être pas utilisé (car la machine était déjà saturée). Cette question sera examinée plus en détail dans le chapitre IX.

III. Placement et ordonnancement dynamique

L'exploitation du parallélisme demande à ce que l'on place les nœuds pouvant s'exécuter en parallèle sur des processeurs différents. Les nœuds qui résident sur un même processeur se partagent l'utilisation de la CPU. Nous devons donc préciser comment un nœud est placé sur un processeur et comment les nœuds d'un même processeur concourent entre eux pour l'obtention de la CPU. Encore une fois, le critère statique/dynamique nous permettra de distinguer entre les différentes stratégies. Le tableau ci-dessous combine les deux stratégies possibles pour le placement et l'ordonnancement et donne des exemples de chaque approche.

ordonnancement \ *placement*	**statique**	**dynamique**
statique	machines SIMD, circuits systoliques, VLIW	—
dynamique	Transputer/Occam [Hoare 88]	*machine data-flow :* LAU [Plas *& al*, 76], MIT [Dennis 80] *système distribué + migration de tâche*

III.1. La répartition des tâches

La question « sur quel processeur réside une tâche ? » correspond au problème du *placement*. Encore une fois nous classerons les différentes réponses possibles en *statiques* et *dynamiques*. La réponse statique consiste à assigner un processeur donné à une tâche avant l'exécution de celle-ci. La décision est prise à la compilation. L'affectation des processus Occam à un transputer est un exemple de répartition statique.

L'approche dynamique consiste à retarder la prise de décision jusqu'à l'activation de la tâche. C'est ce qui se passe dans une architecture data-flow où une instruction prête est exécutée par un des processeurs libres à ce moment là. La situation est légèrement différente quand on considère des tâches dont la durée de vie n'est pas limitée (ce qui est notre cas). Dans ce cas, l'aspect dynamique consiste à pouvoir faire migrer une tâche d'un processeur à un autre, au cours de l'exécution du programme (*dynamic load-balancing*).

Des approches qui combinent les deux stratégies existent bien sûr : dans la machine data-flow de Manchester [Watson, Gurd 82] les tâches sont assignées statiquement à un groupe (*ring*) de 15 processeurs mais l'affectation à

l'un de ces processeurs est dynamique.

III.1.1. La répartition de charge dynamique

Nous envisageons des calculateurs cibles qui ne sont pas des architectures pour le data-flow. Le placement dynamique correspond donc à la migration des tâches. Le problème de la migration des tâches dans un système distribué est un domaine de recherche très actif. On peut citer [Powell 83] [Walker 83] [Theimer 85] [Fowler 86] [Douglis 87] [Zayas 87] [Maguire 88] [Ravi 88] [Smith 88] [Vautherin 88] [Ming 90] etc. La migration des tâches n'est pas seulement vue comme une solution au problème de la répartition de charge mais comme une structure de contrôle distribué permettant la programmation des applications réparties (voir par exemple [BFLPS 85]).

Dans notre cas, l'attrait de l'approche dynamique réside dans sa possibilité à utiliser au mieux les ressources de calcul. Elle est rendue nécessaire dans le cas où l'activité des tâches dépend fortement des données et ne peut être prévue à la compilation. Ce n'est a priori pas le cas d'un programme 81/2. Par ailleurs dans le cadre où nous nous plaçons, nous considérons des machines ayant plusieurs milliers de PE et plusieurs centaines de milliers de tâches. Il est alors difficile d'imaginer un contrôle centralisé de la migration. Il y a alors de bonnes raisons de penser que dans ce cas, la migration des tâches ne peut pas être un facteur d'efficacité.

III.1.2. Une modélisation de la migration des tâches

Pour justifier ce jugement, nous nous reposons sur un modèle étudié par B. Huberman et T. Hogg dans [Huberman, Hog 88]. Ces auteurs veulent modéliser le comportement d'une *grande* population d'agents. Chaque agent doit choisir une stratégie pour maximiser un gain. En supposant que seule une fraction α des agents réévalue sa stratégie pendant un intervalle de temps donné, il est possible de donner une équation différentielle gouvernant l'évolution de n_i, (nombre d'agents qui ont choisi la stratégie i). On introduit dans le modèle une notion de délai correspondant à une connaissance retardée des paramètres globaux du système qui sont nécessaires au choix de la stratégie. On introduit aussi une notion de connaissance imparfaite (un paramètre est connu par un agent avec une certaine probabilité d'erreur). Dès que ces deux notions sont introduites, les n_i n'évoluent plus vers une valeur stable. L'évolution d'un n_i présente des oscillations où même un comportement chaotique (le comportement chaotique apparaît même dans un modèle sans délai si le comportement d'un agent dépend du comportement des autres agents).

Ce modèle peut se réinterpréter avec des tâches et des processeurs : un agent correspond à une tâche et une stratégie correspond à la migration vers un processeur donné. La notion de délai prend en compte le fait que la valeur instantanée d'un paramètre global dans un système distribué (comme le rendement total de la machine) ne peut être connue partout instantanément. La connaissance imparfaite d'un paramètre rend compte de l'estimation du taux de charge d'un PE par une tâche distante. Et bien évidement, la décision de migrer une tâche n'est pas indépendante de la stratégie des autres tâches.

Il en ressort que le nombre de tâches sur un PE est en constante variation. Cela implique que des tâches migrent continuellement dans le système. Or le coût d'une migration de tâche est très lourd (en terme de messages) car il faut avertir toute les tâches filles et toutes les tâches mères du changement d'adresse (ou alors mettre en place un système de redirection des messages [Delaplace, Giavitto 91]). En conséquence de quoi la répartition dynamique de charges n'apparaît pas, dans ces hypothèses, comme viable.

III.2. L'ordonnancement des tâches

Nous supposons ici que le placement des tâches est statique. L'activation des tâches sur un PE peut se faire soit dans un ordre fixe, soit dynamiquement (par exemple sur réception d'une valeur). Dans ce dernier cas, une tâche

peut obtenir l'usage de la CPU au détriment du nœud actif courant : l'ordonnancement est *préemptif*. Dans le cas d'un ordonnancement statique, le compilateur détermine l'ordre d'exécution de chaque tâche (le placement doit donc aussi être statique, puisqu'il faut connaître les tâches assignées à chaque PE). Un ordonnancement est *complètement statique* [Lee 91] si le compilateur détermine aussi l'instant d'activation des tâches. L'alternative est l'ordonnancement que nous avons appelé *réactif* (*self-timed* dans [Lee 91]) où un processeur attend la disponibilité des données nécessaires à la prochaine tâche qui doit être activée.

L'ordonnancement statique consiste à activer séquentiellement les tâches présentes sur un PE. Dans tous les cas d'ordonnancement statique, l'ordonnancement des tâches résidant sur un même PE doit respecter l'ordonnancement total calculé au chapitre V sous peine d'étreintes fatales. On peut imaginer un séquencement statique même en présence de communications déterministes où asynchrones : au moment de son activation, la tâche consulte la file d'attente des messages qui lui sont adressés. Mais dans ce cas l'activation d'une tâche est décorrélée de la réception d'un message (le séquencement n'est plus réactif[1]).

À nouveaux des approches hybrides existent : [GKS 87] essaie d'intégrer à l'architecture data-flow dynamique du MIT, un mécanisme d'ordonnancement statique à travers un système de priorités des instructions. Mais le choix d'un séquencement statique ou dynamique dépend surtout du support existant sur l'architecture cible. Le tableau ci-dessous donne des exemples :

statique	*réactif*	*préemptif*
CM, réseau de cellules asynchrones [Payan 91], PTAH	Cosmic Cube [SEI 85], MEGA, réseau de cellules asynchrones [Payan 91]	iPSC, Transputer

IV. Conclusion

Ce chapitre a pour but de montrer combien l'approche dynamique complique la vie : il faut faire appels à des mécanismes coûteux et sophistiqués et on peut se poser à bon droit la question de leur utilité. En effet, une des contraintes dans le design du langage a été de privilégier l'aspect statique. On dispose donc à la compilation, d'un grand nombre d'information qu'il est possible d'utiliser. C'est que nous allons faire dans les deux chapitres suivants.

[1] Un système d'exploitation développé pour le Cosmic Cube [SEI 85] a proposé d'activer les tâches sur réception d'un message la concernant. L'ordonnancement n'est pas préemptif : une tâche active rend la main à sa terminaison.

VIII. Un modèle d'exécution statique

Le chapitre précédent nous a permis de juger de la complication d'un modèle d'exécution dynamique. Dans ce chapitre nous allons proposer une réponse statique à la question *qu'est ce qu'une tâche*. Le chapitre suivant traitera du problème de l'ordonnancement statique.

Ce chapitre débute par la description du modèle d'exécution statique. Le modèle statique permet la compilation de l'exécution. Il a été en partie motivé par les capacités de la machine PTAH [Cappello, Béchennec 91] mais il peut se transposer au cas des machines séquentielles, des machines SIMD et des réseaux de transputers. Dans tout ce qui suit nous faisons l'hypothèse que la machine cible est une machine MIMD disposant d'une communication synchrone (comme en OCCAM) ou déterministe (i.e. le délai de communication entre deux PE est borné et les messages émis par un PE à destination d'un autre PE sont reçus dans l'ordre d'émission). Nous supposons aussi que le nombre de tâches générées par un programme 81/2 est *grand* devant le nombre de PE, bien que le nombre de PE soit *grand* lui aussi. Cette hypothèse est justifiée par le parallélisme à grain très fin des tissus.

Nous poursuivons par la description de ce qu'est une tâche, et plus précisément par les calculs impliqués par l'évaluation d'un programme 81/2. Ces calculs correspondent à la résolution d'un système d'équations entre collections à chaque tic d'horloge. Un examen attentif de la structure de ce système nous permettra de montrer qu'il suffit pour le résoudre, de calculer les équations dans un certain ordre. La résolution d'une équation de ce système correspond alors à l'application d'une fonction, ce qui correspond à une tâche. Ces tâches sont ensuite réparties dans la machine, afin de paralléliser et de distribuer la résolution du système, problème qui est traité dans le chapitre suivant.

I. Le modèle d'exécution statique

L'intérêt d'un modèle d'exécution statique est double :

- il réduit au maximum le coût de gestion;
- il permet l'ordonnancement statique des opérations.

Appliquée à l'ordonnancement des opérations, l'approche statique tire parti de la connaissance de la structure temporelle du calcul (le graphe des dépendances).

Comme dans le modèle d'exécution dynamique, une tâche a pour charge de calculer la valeur d'un point pour chaque tic. L'activation d'une tâche correspond au calcul de la valeur d'un point à un tic donné. Le modèle d'exécution statique consiste à prévoir à la compilation, les dates d'activation de chaque tâche. Pour cela, on fait des hypothèses permettant de calculer ces dates :

i) le nombre de données consommées à chaque activation d'une tâche est connu *a priori*;

ii) le nombre de données produites en sortie d'une tâche (chaque fois que des données se présentent en entrée) est connu *a priori*;

iii) le temps de traitement de chaque tâche est connu *a priori*.

Ces hypothèses correspondent à des programmes data-flow qui sont appelés *synchrone* dans [Lee, Messerschmitt 87a & 87b]. Ces programmes synchrones interviennent par exemple dans le domaine du traitement du signal et du traitement numérique [PM 89] [PM 91] [Lee 91]. Il faut remarquer que les conditions i) à iii) ne sont pas satisfaites par un programme 81/2 : l suffit de considérer les opérateurs if, when, until, after, at et every.

Ce qui rend le data-flow synchrone si intéressant, ce sont les techniques dont on dispose pour calculer les dates d'activation. En effet, quand un graphe data-flow vérifie les propriétés précédentes, l'ordonnancement de l'activité des nœuds correspond à un domaine très étudié en Recherche Opérationnelle, l'*affectation de ressources*. Certains résultats sont passés en revue dans le chapitre IX. La conclusion qui se dégage est la suivante : l'ordonnancement statique « optimal » de tels graphes est une activité coûteuse sinon impossible (le problème est NP-complet); mais il existe des heuristiques qui s'exécutent en temps polynomial et *dont on peut prévoir les bonnes performances*. Or nous verrons un peu plus loin que l'ordonnancement d'un programme 81/2 implique la manipulation d'un très grand nombre n de données, en conséquence de quoi nous ne voulons pas envisager d'algorithmes de complexité supérieure ou égale à $O(n^2)$. Par ailleurs nous pensons que toute heuristique doit être justifiée par un résultat minimum garanti. Ces diverses raisons militent fortement en faveur d'une approche statique.

I.1. Un modèle data-flow synchrone étendu

Les contraintes du data-flow synchrone correspondent aux propriétés nécessaires pour déterminer *à la compilation* la date d'activation de chaque nœud. Un nœud peut être activé quand ses données sont présentes. Il faut donc prévoir *quand* les données sont présentes, ce qui implique en particulier de savoir *si* elles sont présentes. Dans tous les cas nous demanderons de connaître *a priori* les communications entre nœuds, ce qui correspond aux hypothèses i) et ii). On peut relâcher l'hypothèse iii) suivant les primitives de communication dont on dispose :

– *Dans le cas où la communication est synchrone*
  Il suffit de savoir que la communication doit avoir lieu. Le récepteur est par hypothèse en attente de la réception des données.

– *Dans le cas où la communication est déterministe*
  Sachant que la communication doit avoir lieu, il suffit de connaître la date d'émission au plus tard pour connaître la date de réception au plus tard (le temps de communication est borné). Pour cela il suffit de connaître une borne supérieure au temps de traitement de chaque nœud et une borne supérieure du délai de communication entre nœuds.

– *Dans le cas où la communication est asynchrone*
  Nous n'envisagerons pas ce cas.

I.1.1. Comment savoir à priori si une communication aura lieu

Les opérateurs when, until, after et at peuvent consommer des données sans en produire. Cela se traduit par une incertitude sur la réception d'une donnée par les nœuds successeurs. Il est facile de remédier à ce défaut en adoptant la stratégie suivant : *un opérateur* 81/2 *doit produire une valeur à chaque tic.* L'opérateur when devient ainsi l'équivalent temporel du *where* qui existe pour les langages de collections.

La transformation est simple : il suffit de conditionner le comportement de chaque nœud par l'horloge du nœud. Si *O* représente une opération correspondant à un nœud, la nouvelle opération *O′* se déduit de *O* par :

O′ = si c'est un top alors *O* sinon émettre l'ancienne valeur

(*O* est de la forme « calcul d'une valeur ; émission de cette valeur »). **Il faut donc calculer *explicitement* l'horloge d'une expression.** C'est à quoi va s'employer la section suivante.

I.1.2. Comment borner supérieurement le temps de traitement

Le modèle d'exécution statique, quand on dispose d'une communication déterministe, requiert que l'on puisse borner le temps d'exécution de chaque opération. L'opérateur every pose donc un problème puisqu'il peut boucler.

Afin d'implémenter l'opérateur every, nous allons demander une primitive de synchronisation globale. Celle-ci servira à synchroniser la poursuite du calcul après un every. L'implémentation est exposée en détail dans le chapitre suivant.

I.1.3. Le modèle d'exécution statique est-il un modèle efficace ?

La question qui vient naturellement à l'esprit est de savoir si l'exécution déterministe d'un programme 81/2 est plus ou moins coûteuse que l'exécution dynamique. Cette réponse dépend de la machine cible et de l'application. Néanmoins les éléments suivants rendent le modèle statique très attractif :

- Un placement/ordonnancement qui s'applique aux tâche est de-facto statique (on ne peut pas faire migrer tant de tâche aussi « légère », ainsi que nous l'avons examiné au chapitre précédent).
- L'ordonnancement statique est déterministe et réduit au minimum les coûts de gestion du calcul. De plus le modèle d'exécution statique est indépendant de la valeur des données qui sont calculées.
- En particulier, l'implémentation statique du when correspond au cas pire où la condition est toujours vraie. Dans ce cas une exécution dynamique ne fait pas mieux.
- Dans le cas d'une expression évaluée à chaque tic, le modèle statique utilise moins de messages qu'une approche dynamique.
- Le tamponnement des messages et les acquittements deviennent inutiles car l'interaction producteur-consommateur est prise en charge statiquement, lors de la compilation.
- L'implémentation des after, until et at peut être optimisée comme nous le verrons un peu plus loin. Dans un cas idéal, mais qui correspond à un style de programmation standard, leur implémentation n'est pas plus coûteuse que dans le cas dynamique.

II. Le calcul d'une expression

Un programme 81/2 consiste en la définition d'un tissu. On veut convertir cette définition en un ensemble de tâches permettant de calculer la valeur des points de ce tissu au cours du temps. Nous allons décrire ici cette

conversion.

Le principe de la conversion d'un programme en tâches a déjà été évoqué : une tâche est associée à un ensemble d'expressions et permet de calculer la valeur d'un ou plusieurs points du tissu défini par l'expression. Afin de simplifier l'exposé, nous allons associer une tâche au calcul de la valeur d'un point d'une *définition* de tissu. Autrement dit nous utilisons un critère syntaxique, la division d'un programme en équations, pour décider de la granularité des tâches à générer. Cette approche doit bien sûr être ajustée en fonction des caractéristiques de l'architecture cible (Cf. la discussion dans le chapitre suivant consacrée à la granularité des tâches).

II.1. Comportement d'une tâche

Une tâche correspond aux calculs de la valeur d'un point et ces calculs sont itérés dans le temps. Le comportement d'une tâche est cyclique :

Répéter
1) obtenir les valeurs nécessaires au calcul de l'expression
2) calculer le résultat correspondant
3) tenir à disposition ce résultat

Il nous faut donc répondre à deux questions :

– Quelles sont les valeurs nécessaires au calcul d'une expression ?
– Quel est le moyen d'obtention et de mise à disposition des valeurs ?

La réponse à la deuxième question correspond au problème de la synchronisation et de la communication. Il est traité dans le chapitre suivant. Nous allons ici nous concentrer sur la réponse à la première question et plus précisément déterminer *comment calculer la valeur d'une expression*.

II.2. Calcul de la valeur d'une expression

La valeur d'une expression est définie par les fonctions sémantiques *Val*`[ [ • ] ]` et *Hor*`[ [ • ] ]`. L'application des fonctions sémantiques conduit à la recherche de la plus petite solution d'un système d'équations (sur des suites de collections). Par exemple au programme

T@0 = 0 ; *T* = $*T* + 1 when *Clock* ;

correspond le système d'équations[1] :

$$T = \begin{cases} t = true \ ; \ rest(t, cte(true)) \\ fby\ (true \ ; cte(false), cte(0), t, bop\ (+, delay(T), cte(1))) \end{cases} \quad (E1)$$

(avec t l'horloge du tissu *T* et T la suite des valeurs de *T*; on suivra dans la suite la convention de noter en capitale droite la suite des valeurs d'un tissu et en minuscule, son horloge).

Nous avons déjà remarqué que les fonctions qui définissent les équations sémantiques sont définies par induction à partir du constructeur ";", i.e. elles prennent la forme suivante : $F_1\ (a\ ;\ t) = g_1\ (a)\ ;\ F(a, t)$ où $F(a, t)$ est aussi une fonction de cette forme pour tout a (a est une collection et t une suite de collections). Par suite, si $a;t$ est la solution cherchée de l'équation $x = F_1\ (x)$ alors,

$$a;t = F_1\ (a;t) = g_1\ (a)\ ;\ F(a, t) \quad (1)$$

et en particulier,

$$a = g_1\ (a) \quad (2)$$

[1] Cf. les exemples donnés dans le chapitre sur la sémantique dynamique.

Par suite pour résoudre (1) il suffit donc de résoudre le système (2) puis de répéter le procédé en posant $F_2 = \lambda x.\ F\ (g_1\ (\mathit{false})\ ,\ x)$ pour le tic suivant, etc. Le système en un tic donné ne dépend pas des valeurs des tics suivants et les valeurs des tics précédents sont connues.

Si on applique cela à l'exemple (E1), on doit résoudre à chaque tic le système :

$$\textit{pour le tic 0} \begin{cases} t = \mathit{true} \\ T = 0 \end{cases} \quad \textit{pour les autres tics} \begin{cases} t = \text{true} \\ T = \$T + 1 \end{cases}$$

Cet exemple appelle plusieurs remarques :

- $T dénote une constante connue au moment où on résout le système et qui dépend du tic sur lequel on évalue ce système d'équations (c'est la valeur du dernier top, i.e., la valeur de T la dernière fois que t valait *vrai*).
- On a simplifié en ne se posant pas le problème du choix entre les différents systèmes d'équations de définition de T; en réalité chaque système différent doit être résolu et une valeur choisie en fonction du nombre de tops déjà écoulés et des solutions obtenues).
- La résolution en parallèle de ce système peut se faire en associant une tâche à chaque équation. La tâche calcule la valeur de la variable désignée à gauche de « = ». Pour cela elle a besoin des valeurs des variables qui apparaissent à droite de « = ».
- Cet exemple peut faire croire que le système à résoudre à chaque tic est simple et exhibe en quelque sorte une « forme triangulaire ». Nous voulons dire par là que comme il existe une relation d'ordre entre les tissus, on peut croire qu'il est possible de classer les équations de telle manière que la valeur et l'horloge d'une variable ne dépendent que des variables définies par les équations précédentes (ce qui rend très simple la résolution du système d'équations). Par suite il suffit à chaque tâche d'attendre les valeurs nécessaires et d'émettre le résultat de son calcul vers les tâches consommatrices. Il n'en est rien : l'existence d'une relation d'ordre implique uniquement que « $T » est une constante[1].

Examinons plus en détail un exemple un peu plus compliqué (c'est à dessein que l'exemple n'a aucune signification particulière) :

$$B = A \text{ when } \$C \qquad (E2) \qquad \begin{cases} A = \$B \text{ when } \$C\ ; \\ C = B \text{ when } A\ ; \end{cases}$$

génère le système (S2)

$$\begin{cases} a = c \underline{\wedge} \$C \\ A = \mathit{si}\ a\ \mathit{alors}\ \$B\ \mathit{sinon}\ \$A \\ b = c \underline{\wedge} \$C \\ B = \mathit{si}\ b\ \mathit{alors}\ A\ \mathit{sinon}\ \$B \\ c = a \underline{\wedge} A \\ C = \mathit{si}\ c\ \mathit{alors}\ B\ \mathit{sinon}\ \$C \end{cases}$$

Dans le système d'équations (S2), les variables qui apparaissent précédées d'un « $ » représentent en réalité des constantes (qui dépendent du tic). **Attention**, le « *si* … *alors* … *sinon* … » qui est employé ici n'est pas le « si … alors … sinon … » usuel : si la condition est indéfinie, alors la valeur de l'expression est celle de la branche *sinon*. Cela nous évite de mettre explicitement le test « $\neq \bot$ » qui apparaît dans les équations sémantiques. L'opérateur $\underline{\wedge}$ correspond à la version non stricte de $\wedge$ qui traite $\bot$ de manière équivalente à *false*.

Le programme (E2) admet un ordonnancement : $C \xrightarrow{p} B \xrightarrow{p} A$, mais le système (S2) associé ne prend pas une « forme triangulaire » car les horloges créent une dépendance cyclique :

$$A \rightarrow a \rightarrow c \rightarrow A.$$

La question que nous nous posons est : comment résoudre *simplement* et *efficacement* le système d'équations (S2), bien qu'il ne soit pas triangulaire ?

[1] Dans le cas d'une définition récursive spatiale, il faut envisager la résolution de n systèmes d'équations successivement, qui ne correspondent plus à des tics successifs, mais à des points successifs. La relation d'ordre assure que la valeur des points précédents est connue.

II.2.1. Résolution des équations associées à un tic

En toute généralité on doit résoudre un système de la forme :

$$\begin{cases} v = g\,(v, V) \\ V = G\,(v, V) \end{cases} \qquad (3)$$

où v est le vecteur des variables booléennes (correspondant aux horloges *et* aux tissus booléens qui apparaissent dans la commande d'un when) et V le vecteur des autres variables. On sait que la solution que l'on recherche s'obtient par itération :

$$\begin{cases} v_0 = \bullet\textit{false}, \ldots, \textit{false}® \\ V_0 = \bullet\bot, \ldots, \bot® \end{cases} \qquad \begin{cases} v_{n+1} = g\,(v_n, V_n) \\ V_{n+1} = G\,(v_n, V_n) \end{cases}$$

La suite $\langle v_n, V_n \rangle$ obtenue est *stationnaire* après au plus $p+q$ itérations, où p est la dimension de v et q la dimension de V (i.e. $p+q$ correspond au nombre de variables du programme). En effet, les fonctions g et G sont continues comme composées de fonctions continues (au sens des chaînes) et par suite, croissantes. La suite $\langle v_n, V_n \rangle$ est donc croissante. La longueur du préfixe strictement croissant de cette suite est bornée par la longueur de la plus longue suite strictement croissante de $BOOLEAN^p \times SCALAIRE^q$. *SCALAIRE* est un domaine plat et la plus longue suite strictement croissante de *SCALAIRE* est donc de longueur 2. La plus longue suite strictement croissante de $SCALAIRE^q$ a donc une longueur d'au plus q (un élément parmi q doit être strictement croissant d'une itération à une autre). Il en va de même pour le domaine *BOOLEAN*. On a alors facilement que la longueur de la plus longue suite strictement croissante de $BOOLEAN^p \times SCALAIRE^q$ est $p+q$ (voir par exemple [Cai, Page 89, *section 5*] pour une généralisation du procédé).

Ceci nous donne un premier moyen pour obtenir les valeurs cherchées. Mais cela nous oblige à évaluer $p+q$ fois les fonctions g et G. Il est possible de mieux faire.

Considérons le graphe des dépendances du système (3) : un nœud correspond à une définition de variable et un arc entrant correspond à une variable apparaissant en partie droite de la définition. On peut séparer les nœuds du graphe en composantes fortement connexes maximales. Il existe une relation d'ordre partiel strict entre celles-ci : $C_1 < C_2$ si il existe un arc d'un sommet de C_1 vers un sommet de C_2 (la relation est bien une relation d'ordre strict car si il existe un arc de C_2 vers C_1 cela veut dire que C_1 et C_2 ne sont pas des composantes maximales). La figure suivante représente la situation pour l'exemple (E2) :

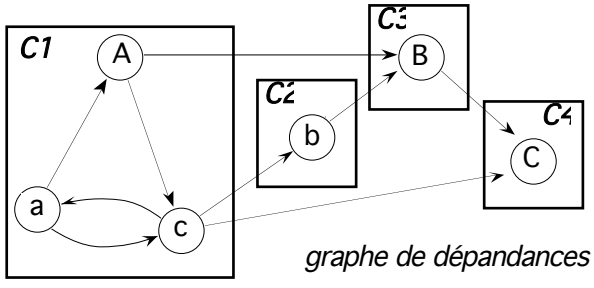


Figure 1 : Graphe des dépendances du système d'équations associé au programme (E2) et graphes des composantes fortement connexes maximales. Entre 2 sommets N et M d'une composante fortement connexe, il existe un chemin de M à N et un chemin de N à M. Une composante fortement connexe C est maximale si pour tout sommet N, C + N fortement connexe $\Rightarrow$ N $\in C$. *C1* est un élément minimal du graphe des composantes fortement connexes maximales car elle n'a pas de prédécesseur.

On va résoudre le système entier en résolvant successivement les sous-systèmes correspondant aux composantes connexes maximales. Appelons *racine* un élément minimal de la relation d'ordre entre composantes fortement connexes maximales. Une racine est un sommet qui n'a pas de prédécesseur. Une racine représente un sous-système d'équations dans lesquelles toutes les variables qui apparaissent en partie droite sont définies par une équation de ce sous-système. Supposons qu'on sache résoudre un tel sous-système, alors on sait résoudre le système entier : une fois connues les valeurs des nœuds de la racine, on propage ces valeurs dans les autres équations et on retire les nœuds correspondants du graphe; on obtient un nouveau graphe sur lequel on itère le procédé, cela jusqu'à exhaustion de tous les nœuds.

Soit C une racine. Chaque nœud de ce graphe correspond à une variable. Pour simplifier, on fera l'hypothèse que toutes les définitions sont des expressions de profondeur 1 (i.e. on a donné un nom à chaque (sous-)expression). Dans une racine, les équations prennent obligatoirement une des formes suivantes :

0) x = *constante* : constante du langage
0′) X = *constante* : constante du langage

1) x = y : horloge d'une opération unaire
2) X = *si* x *alors* f(Y) *sinon constante* : opération unaire

3) x = y $\underline{\vee}$ z : horloge d'une opération binaire
4) X = *si* x *alors* f(Y, Z) *sinon constante* : opération binaire

5) x = y $\underline{\wedge}$ Y : horloge d'un when
6) X = *si* x *alors* Y *sinon constante* : when (identique à la forme 2) avec f = λx.x)

7) x = y $\underline{\vee}$ z $\underline{\vee}$ w : horloge d'une conditionnelle
8) X = *si* x *alors* (si Y alors Z sinon W) *sinon constante* : conditionnelle

x, X, y, Y, z, Z, w, et W sont des « méta-variables », ce qui veut dire qu'on peut très bien avoir une équation de la forme a = a par exemple. Nous ne montrerons pas que ces équations correspondent bien à l'application des fonctions *g*, il suffit de se reporter aux équations sémantiques pour s'en convaincre. Nous employons encore la notation «*si* x *alors* … *sinon* …» permettant d'éviter d'expliciter «si ((x ≠⊥) ∧ x) alors … sinon …». Par ailleurs $\underline{\vee}$ (resp. $\underline{\wedge}$) dénote une opération de *ou logique* (resp. de *et logique*) non stricte dans laquelle ⊥ est équivalent à *false*. Cela correspond aux définitions des fonctions hbop et htrigger dans les équations sémantiques.

La solution recherchée du (sous-)système s'obtient, comme pour le système général, par itération à partir des valeurs initiales ⊥. Nous voulons montrer qu'*une seule itération suffit*. Après la première itération, le résultat des équations 1), 3), 5) et 7) ne peut être que ⊥, ce qui ne modifie pas la valeur initiale des variables d'horloges. La valeur des équations 2), 4), 6) et 8) est celle de la constante correspondante. Cette valeur diffère éventuellement de ⊥.

À la deuxième itération, les variables d'horloge ont toujours la valeur ⊥. Par suite seul le cas 5) est susceptible de générer un résultat différent. En effet, au lieu d'évaluer « ⊥ $\underline{\wedge}$ ⊥ », on sera amené à évaluer éventuellement « ⊥ $\underline{\wedge}$ *true* » ou bien « ⊥ $\underline{\wedge}$ *false* » (suivant la valeur des constantes). Les différentes évaluations donnent le même résultat : *false*. Il n'y a donc pas de différence entre la première et la deuxième itération. L'évaluation répétée des équations 0) et 0′) donne toujours le même résultat. Et donc une itération suffit.

Une fois qu'on a obtenu les valeurs de la racine, on peut propager dans les expressions restantes. Supposons qu'on ait obtenu une valeur pour y, alors on a aussi obtenu une valeur pour Y. La propagation des valeurs de y et Y dans les équations de type 1) et 2) permet d'obtenir la valeur de x et X. La propagation dans les équations de type 3) et 4) transforme ces équations en des équations du type 1) et 2) : x = z et X = *si* x *alors* f′(Z) *sinon constante*, avec f′ = λx.f(Y, z). On a le même résultat si c'est z = ⊥ et Z qu'on propage. Enfin si c'est y = z = ⊥ qui est propagé, les équations 3) et 4) se transforment en x = ⊥ et X = *constante*. La propagation d'une valeur dans 7) et 8) transforme ces équations en des équations du type 3) et 4). La propagation de deux valeurs dans 7) et 8) transforme ces équations en

des équations du type 1) et 2). La propagation de trois valeurs permet d'éliminer l'équation. La propagation de y et Y dans les équations du type 5) et 6) a pour effet l'élimination de ces équations : si y = ⊥ ou y = *false* alors on obtient x = ⊥ et X = *constante*, sinon on a x = *true* et X = Y.

Après propagation des valeurs obtenues, on élimine les équations résolues et on simplifie les autres. On obtient à nouveau un graphe dont on peut tirer à nouveau des racines dont les équations ont la forme 1) à 8). Et on itère.

Les valeurs obtenues par concaténation des solutions des racines successives constituent clairement une solution du système général. On montre facilement que c'est la plus petite (car la solution obtenue sur chacun des sous-systèmes est la plus petite).

II.2.2. Remarques

Ce résultat appelle quelques commentaires. Nous n'avons pas pris en compte les opérateurs until, after et at. Comme ils peuvent se réécrire uniquement en terme de when, ils ne posent aucun problème particulier.

Nous n'avons pas non plus pris en considération l'opérateur fby correspondant aux équations de définitions multiples. Il ne pose aucun problème particulier puisqu'on peut le réécrire en termes de when, until et de after uniquement. En effet, le programme

X@0 = A ; X = B ;

est équivalent à :

a@0 = true synchro A ; a = false synchro B ;
b = true synchro A ;
X = if a then (A until a) else (B after b) fi ;

(on peut vérifier qu'il n'y a pas de top parasite dans la réexpression de X). La formulation générale de « X@t » est un peu plus complexe car il faut faire intervenir un compteur.

La valeur des horloges des tissus faisant partie d'une composante fortement connexe **et** qui sont définis par une équation de la forme 1) à 8) est indéfinie. Une horloge indéfinie correspond à un tic, ce qui apparaît implicitement dans les équations sémantiques (si on remplace ⊥ par *false* dans une horloge, on obtient la même suite de valeur).

Nous savons à présent comment calculer la valeur des expressions d'un programme 81/2. Il faut commencer par classer les équations en composantes fortement connexes, et ordonner ces composantes. Dans chacune des racines on sépare les équations de la forme 0) ou 0′) des autres. On sait que les variables x des autres équations auront pour valeur ⊥. Cela permet d'éliminer ces variables des équations, et par suite d'ordonner les calculs. En effet, une fois que les variables d'horloges qui causaient un cycle ont été supprimées, il ne peut plus y avoir de boucle puisque le programme admet un ordonnancement statique. On trace ensuite l'effet de la valuation des variables des racines afin de déterminer les racines suivantes. Tout ce travail est réalisé à la compilation. À l'exécution il suffit de propager les valeurs selon l'ordonnancement obtenu par la trace de la propagation des variables des racines.

L'existence d'un ordonnancement statique (définie dans le chapitre consacré à l'ordonnancement statique) n'intervient pas vraiment dans la démonstration ci-dessus : on pourrait considérer des programmes avec des boucles temporelles instantanées de type « *X* = *X* » qui généreraient deux équations du type : « X = X » et « x = x ». Mais on est certain alors que la solution est « X = ⊥ » et « x = ⊥ ». En fait l'ordonnancement statique ne prend en compte que le calcul des valeurs. Quand on veut une exécution complètement statique, il faut en plus calculer les horloges (dans une exécution dynamique ce calcul se fait de facto par la stratégie d'évaluation par nécessité). Cela introduit des contraintes supplémentaires qui correspondent aux précédances induites par le calcul de l'horloge des retards.

II.2.3. Exemples

Commençons par un exemple simple :

A = 1 when *Clock* ;
B = 1 + $*A* ;

On s'attend à ce que l'horloge de B soit aussi celle de *A*. Pourtant, remarquons bien qu'il n'y a pas de dépendance directe entre la valeur de *A* et celle de *B*. ce qui donne le système d'équations (pour un tic ≠ 0)

$$\begin{cases} \text{A} = \textit{si}\ \text{a}\ \textit{alors}\ 1\ \textit{sinon}\ \$\text{A} \\ \text{B} = \textit{si}\ \text{b}\ \textit{alors}\ 1 + \$\text{A}\ \textit{sinon}\ \$\text{B} \\ \text{a} = \textit{true} \\ \text{b} = \text{a} \end{cases} \qquad \text{(E3)}$$

Rappelons que $X désigne une constante, la valeur de la variable X au tic précédent. Les dépendances calculées par l'ordonnancement statique sont réduites à un graphe vide (plus exactement *A* et *B* dépendent de constantes). L'algorithme de transformation en tâche rajoute une dépendance à *B* car il contient un délai : la dépendance qui s'ajoute correspond au calcul de b qui lui-même dépend de a. On a donc les dépendances : B → b → a, A → a. Cela fournit un ordonnancement des équations du système (E3). Le code de la tâche associé au tissu *B* se déduit directement des équations de b et B :

```
pour toujours
   oldA = NIL;
   attendre a
   b = a, émettre b
   si a alors
        B = oldA + 1, émettre B
        attendre A, oldA = A
   sinon
        émettre B
```

Revenons à l'exemple (E2). Il y a au départ une seule racine, ***C1*** (Cf. la figure 1), correspondant aux variables a, A et c. On a donc le sous-système :

$$\begin{cases} \text{a} = \text{c} \_ \$\text{C} \\ \text{A} = \textit{si}\ \text{a}\ \textit{alors}\ \$\text{B}\ \textit{sinon}\ \$\text{A} \\ \text{c} = \text{a} \_ \text{A} \end{cases}$$

les équations sont de la forme 5) et 6) et on a donc a = c = ⊥ (ou de manière équivalente a = c = *false*) et A = $A. En propageant à la composante ***C2***, on obtient b = *false*, ce qui élimine cette équation. On continue en propageant à la composante ***C3*** : B = $A puis à la composante ***C4*** : C = $C. Arrivé là, on a résolue tout le système sans avoir à former de nouvelle racine. Au passage a = b = c = *false* ce qui montre que les tissus *A*, *B* et *C* sont réduit à ⊥. Ce résultat aurait été détecté par la technique proposée au chapitre VI.

Enfin examinons une variante de l'exemple classique du compteur :

T@0 = 0 ; *T* = $*V* ;
V = *T* + (1 when *Clock*) ;

On commence par nommer « 1 when *Clock* » en *tmp*. On obtient le système

$$\begin{cases} \text{clock} = \textit{true} \\ \text{CLOCK} = \textit{true} \\ \text{tmp} = \text{clock} \_ \text{CLOCK} \\ \text{TMP} = 1 \\ \text{t} = \text{v} \\ \text{v} = \text{t} \_ / \text{tmp} \end{cases}$$

Il y a 5 composantes fortement connexes maximales : *C1* = { clock }, *C2* = { CLOCK }, *C3* = { tmp }, *C4* = { TMP }, *C5* = { t, v }. *C1*, *C2* et *C4* constituent des racines. On peut les traiter dans n'importe quel ordre. Les équations de ces racines sont toute de la forme 0) et 0′). On propage leur valeur à *C3* ce qui donne la valeur de tmp.

On peut à nouveau propager ce qui élimine *C5*. L'ordre de calcul obtenu est donc : (*C1*, *C2*, *C4*) → *C3* → *C4*.

Les tâches correspondant à *C1*, *C2* et *C4* peuvent se dérouler en parallèle. Ici il y a une seule tâche par composante, car il y a une seule équation (par composante). Dans le cas général, les *n* tâches correspondant au calcul des *n équations* d'une racine peuvent se dérouler en parallèle.

IX. Placement et ordonnancement statique des programmes 8$_{1/2}$

Dans le chapitre VII nous avons montré le peu d'intérêt du placement et de l'ordonnancement dynamique dans le cadre de l'implémentation de 81/2. Nous proposons ici une famille d'algorithmes d'ordonnancement statique. Afin de profiter de la connaissance de la structure temporelle des calculs, nous choisissons d'unifier les phases de placement et d'ordonnancement.

La principale difficulté, qui doit rester présente à l'esprit, est que le nombre de tâches à ordonner est très grand et qu'il n'est pas possible de les représenter toutes explicitement. Il faut donc trouver une représentation « compacte » de l'ordre à respecter entre tâches, ce qui nous conduit à considérer les ordonnancements répétitifs complètement statiques. Une représentation judicieuse des tâches transforme le problème en un problème de *bin-packing*. Une revue des résultats du domaine permet de choisir un algorithme. Cet algorithme permet l'ordonnancement des programmes statiques, au sens défini dans le chapitre précédent. Il faut ensuite adapter cet algorithme afin de pouvoir l'appliquer à des programmes réels et en particulier, il faut traiter le cas de l'opérateur every. Nous passons ensuite en revue plusieurs problèmes pratiques d'implémentation. Le chapitre se termine par une discussion sur la granularité des tâches.

I. Placement et ordonnancement statique

Pour réaliser le placement et l'ordonnancement des tâches, on dispose d'un graphe des dépendances. Ce graphe est le résultat de l'analyse du système d'équations entre les valeurs et les horloges des tissus du programme. Cette analyse a été présentée dans le chapitre précédent. Dans ce graphe, un arc (`r`, `s`) entre une tâche `r` et une tâche `s` exprime deux types d'information :

- *une information temporelle* : l'activation de `r` doit précéder l'activation de `s`;
- *une information spatiale* : la tâche `r` (localisé sur un PE) doit communiquer un résultat à la tâche `s` (qui est localisé sur un autre PE).

Il est possible de ne tenir compte, lors du placement, que de l'information spatiale : le graphe de dépendances se transforme alors en un graphe de communication entre tâches. Nous appellerons cette approche le *placement pur*.

I.1. Le placement pur

Pour transformer un graphe de dépendances en un graphe de communication, il suffit d'oublier l'orientation des arcs dans le graphe des dépendances. Chaque nœud du graphe de communication possède un « poids » qui représente la charge de travail de la tâche. La diminution des coûts de communication passe par le rapprochement des tâches qui interagissent entre elles. Des tâches placées sur le même processeur ont un coût de communication négligeable, mais augmentent le taux d'occupation du PE.

Le problème est alors de trouver une affectation des tâches aux PE qui minimise le temps global de communication et qui répartit la charge de travail sur tous les PE. En première approche, on peut distinguer deux classes dans les méthodes proposées dans la littérature : les méthodes basées sur la partition du graphe des communications et les méthodes basées sur l'optimisation d'une fonction d'affectation. Y. Belhamissi et M. Jégado proposent dans [Belhamissi, Jégado 91] une revue des différentes stratégies.

L'approche du placement pur est intéressante dans le cadre d'une exécution dynamique d'un programme 81/2. En particulier, dans le cadre d'une messagerie asynchrone, le coût de la communication rend attractives les méthodes dédiées à sa minimisation. De plus l'indéterminisme des communications asynchrones plaide pour une approche d'optimisation globale. Enfin, les graphes de communication des programmes 81/2 ont des propriétés topologiques intéressantes, liées à la transparence référentielle, qu'il est sans doute possible d'exploiter (se référer par exemple à [Schlag 87]).

I.2. L'ordonnancement répétitif

Cependant, un des intérêts principaux du data-flow est la connaissance de la structure fine des dépendances entre opérations. Il serait donc dommage de ne pas en tenir compte, surtout quand l'architecture cible offre des communications déterministes ou synchrones. Dans la suite, quand nous parlerons d'ordonnancement, on comprendra qu'il s'agit de déterminer à la fois le PE sur lequel une tâche va s'exécuter et les dates d'activation de cette tâche. Parmi tous les ordonnancements statiques qui sont possibles, nous allons considérer uniquement les ordonnancements *répétitifs complètement statiques*. Nous allons expliquer ce que nous entendons par là.

Dans notre modèle d'exécution statique, il s'agit d'activer chaque tâche de manière répétitive. Comme il y a une infinité d'activations et que l'on veut prévoir à la compilation la date de chaque activation, on va considérer que les dates d'activations sont *périodiques*.

L'activation de toutes les tâches du programme, une fois et une seule, est traditionnellement appelée une *itération*. Une itération correspond au calcul des valeurs des tissus d'un programme pour un tic. Le temps mis pour exécuter une itération s'appelle la *période temporelle*. Si un programme 81/2 ne comporte pas de délai, alors il n'y a pas de dépendance entre itérations et on peut réduire la période temporelle en fonction du nombre de PE dont on dispose. À la limite, si on dispose d'une infinité de PE, la période temporelle est nulle : on affecte un PE à l'exécution d'une itération.

La présence d'un délai correspond à une dépendance entre itérations : avant d'exécuter une itération, il faut disposer des valeurs produites à une itération précédente. La présence de délai impose donc une borne inférieure à la période temporelle [Kung, Lewis, Lo 87] [Parhi, Messerschmitt 89] [Parhi, Messerschmitt 91] : c'est la *borne d'itération* (*itération bound*).

Il est souhaitable que le temps de la simulation « avance le plus vite possible ». Un ordonnancement est donc d'autant meilleur que la période temporelle est courte. Un ordonnancement périodique est *optimal* (*rate-optimal*) si sa période temporelle est égale à la borne d'itération.

Il faut à présent tenir compte du PE sur lequel s'effectue une opération. Ce paramètre supplémentaire donne un

degré de liberté en plus : les opérations d'un même point peuvent très bien s'effectuer sur un processeur différent suivant le tic (cette assignation variable devant néanmoins être prévue à la compilation). Schwartz introduit dans [Schwartz, Barnwell 85] [Foren, Schwartz 87] ce type d'ordonnancement sous le nom d'ordonnancement *cyclo-statique*.

I.2.1. Ordonnancement cyclo-statique

La figure 1 donne un exemple d'ordonnancement cyclo-statique. Les ordonnancements cyclo-statiques sont caractérisés par une période spatiale *K* et une période temporelle *T*. Si l'opération correspondant au tic *t* d'un nœud donné se déroule sur le processeur *n*, alors l'opération *(t +1)* de ce même nœud se déroulera sur le processeur *(n+K) modulo N* au temps *(t + T)* (*N* représente le nombre total de PE).

Nous ne considérerons pas dans la suite les ordonnancements avec recouvrement. En effet :

– Le recouvrement est intéressant quand les opérations à effectuer dans un tic ne saturent pas les PE (i.e. il reste des PE libres une fois que toutes les opérations d'un tic ont été assignées). Ce n'est pas ce qui est attendu dans le cas d'un programme 81/2 (il y a une opération par point, des centaines de points par tissu et des centaines de tissus).

– À notre connaissance les méthodes proposées pour l'établissement d'un ordonnancement cyclo-statique sont trop coûteuses : [Schwartz, Barnwell 85] propose par exemple une méthode énumérative.

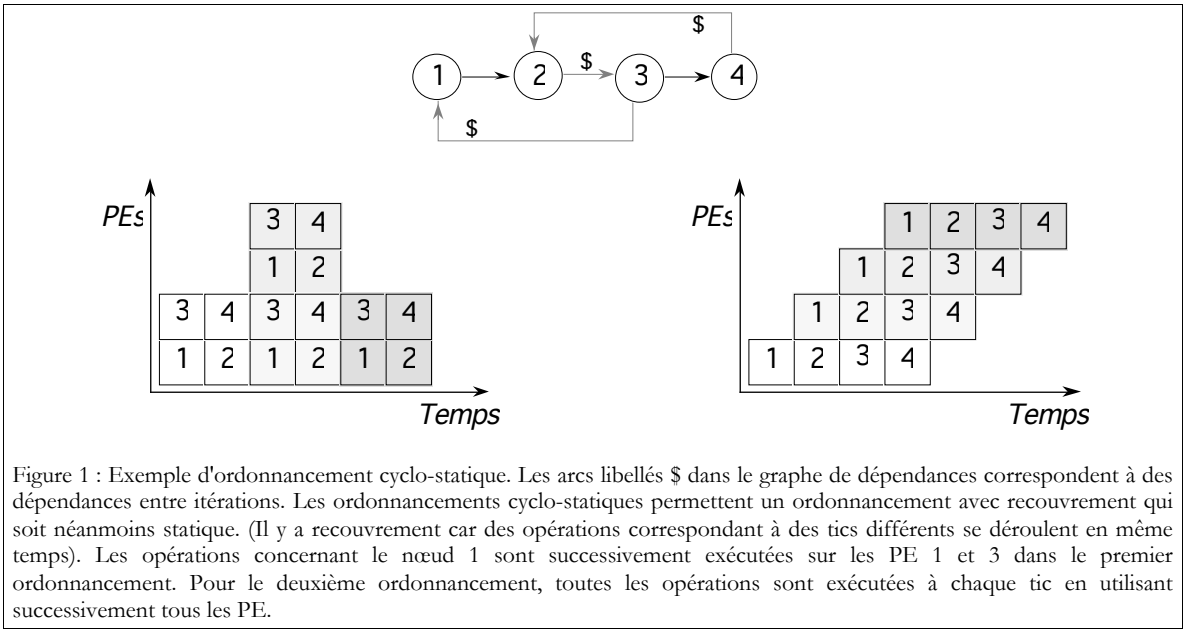


Figure 1 : Exemple d'ordonnancement cyclo-statique. Les arcs libellés $ dans le graphe de dépendances correspondent à des dépendances entre itérations. Les ordonnancements cyclo-statiques permettent un ordonnancement avec recouvrement qui soit néanmoins statique. (Il y a recouvrement car des opérations correspondant à des tics différents se déroulent en même temps). Les opérations concernant le nœud 1 sont successivement exécutées sur les PE 1 et 3 dans le premier ordonnancement. Pour le deuxième ordonnancement, toutes les opérations sont exécutées à chaque tic en utilisant successivement tous les PE.

I.2.2. Ordonnancement optimal et recouvrement

L'intérêt d'un ordonnancement cyclo-statique est de permettre des ordonnancements statiques avec recouvrement : un recouvrement existe si des activations correspondant au tic *t* se déroulent en même temps que des activations du tic *t+1*. Le recouvrement permet d'augmenter l'utilisation des PE, et par là, permet de diminuer la période temporelle.

La question qui se pose est alors de savoir s'il existe un ordonnancement sans recouvrement qui soit optimal. La réponse est non. Cependant, Parhi et Messerschmitt proposent dans [Parhi, Messerschmitt 89] et [Parhi, Messerschmitt 91] une méthode permettant d'obtenir un ordonnancement optimal dans le cas où on dispose d'un

nombre suffisant de PE.

L'idée est de montrer qu'un ordonnancement répétitif sans recouvrement est optimal dès que le graphe des tâches possède une certaine propriété. En fait, cette propriété est celle de *systolicité* qui est introduite dans [Leiserson, Saxe 83] : le nombre de délais sur chaque cycle doit être exactement de 1. On montre ensuite que le « dépliage » d'un graphe quelconque permet de transformer un graphe de façon à ce qu'il possède ces propriétés. Le dépliage d'un graphe correspond à agréger des itérations consécutives en remplaçant des expressions retardées par leurs définitions.

La méthode proposée par Parhi et Messerschmitt est séduisante. Elle correspond à un ordonnancement sans recouvrement d'un graphe équivalent à celui de départ (mais qui se traduit en terme du graphe de départ comme un ordonnancement avec recouvrement). Cependant cette méthode ne garantit un résultat optimal que dans le cas où on dispose d'autant de PE que nécessaire à l'algorithme. Rien n'est garanti quand les PE constituent une ressource limitée. Par ailleurs le nombre de dépliages à faire peut être très grand (c'est le plus petit commun multiple des nombres de registres utilisés dans chaque cycle).

C'est pourquoi nous allons nous restreindre aux ordonnancements répétitifs sans recouvrement. Si nous ne considérons pas les ordonnancements avec recouvrement, nous notons néanmoins que le dépliage de graphe permet d'augmenter le nombre de tâches à effectuer. En terme de tics, cela revient à ordonnancer plusieurs tics successifs en même temps. Cette approche permet, si besoin est, de toujours s'assurer qu'il y a plus de tâches que de PE, hypothèse que nous faisons dans ce chapitre.

I.2.3. Techniques d'ordonnancement classique et d'ordonnancement répétitif

La problématique de l'ordonnancement répétitif (avec ou sans recouvrement) a été traitée en Recherche Opérationnelle sous l'angle de l'existence d'un ordonnancement sous diverses contraintes (on trouvera une large bibliographie dans [Serafini, Ukovich 89] et [KALW 91]). La réponse cherchée est celle répondant à la question « Existe-t-il un ordonnancement répétitif vérifiant telle contrainte et dont la période est p ? ». La période p est un paramètre du problème. La contrainte est de vérifier un certain ordre entre l'activation des tâches, ou bien d'utiliser un nombre limité de ressources dans une fenêtre temporelle donnée.

L'ordonnancement répétitif en Recherche Opérationnelle ne se pose donc pas le problème de la minimisation de la période. Par contre des algorithmes ont été proposés qui, étant donnée une période p, permettent de produire éventuellement un ordonnancement satisfaisant p ([Serafini Ukovich 89] se concentre sur le traitement des contraintes de précédance entre tâches, [KALW 91] sur l'utilisation d'un nombre limité de ressources). Une approche possible, pour l'ordonnancement d'un programme 81/2, est d'utiliser ces algorithmes avec une recherche dichotomique de la période (Cf. [Hanen 91]). Cette approche risque d'être coûteuse en temps de calcul et est pour le moins indirecte. C'est pourquoi nous nous tournons vers les algorithmes développés dans le cadre de l'ordonnancement classique (non répétitif).

Dans le domaine de la conduite de projet (*project scheduling*), le problème consiste à établir un ordonnancement d'un certain nombre d'activités sujettes à des contraintes. Des techniques du type PERT permettent d'obtenir des solutions qui minimisent la durée totale du projet. Les algorithmes correspondent à des algorithmes de programmation dynamique de complexité polynomiale. Ces algorithmes ne prennent hélas pas en compte la contrainte de ressources que nous avons : il faut utiliser au mieux les PE qui sont en nombre fixé.

I.2.4. Ordonnancement répétitif complètement statique

Un ordonnancement *répétitif complètement statique* est un ordonnancement périodique sans recouvrement. Le problème que nous nous posons est donc d'affecter un PE à chaque tâche et de trouver un ordonnancement au

mieux des activités d'une itération.

Dans certains cas particulier, il existe des algorithmes de complexité polynomiale, même si on tient compte d'un coût de communication [Nicol, Hallaron 91]. Mais il faut pour cela se restreindre à certaines classes d'architectures : architectures multi-processeurs à mémoire partagée, pipe-line ou bien maîtres/esclaves (communication de 1 vers n et de n vers ce 1 là).

On trouvera dans [Veltman, Lageweg, Lenstra 90] une synthèse des résultats portant sur la complexité de diverses variations du problème de l'ordonnancement répétitif avec contraintes de précédances et délais de communication. La conclusion est toujours la même : avec les hypothèses qui nous intéressent (le nombre de processeurs n'est pas réduit à deux, le nombre de tâches n'est pas inférieur ou égal au nombre de PE, les communications sont quelconques) la détermination d'un ordonnancement de période minimale est un problème est NP-difficile [Rayward-Smith 87].

C'est pourquoi nous allons nous intéresser à des algorithmes approchés et en particulier à ceux qui ont été développés pour le *bin-packing*. On trouvera dans [GGJ 78] et dans [CGJ 84, §7] une présentation du domaine du *bin-packing* et une abondante bibliographie.

I.3. Le *bin-packing* : un modèle abstrait pour l'ordonnancement des tâches

le problème du *bin-packing* est de minimiser l'espace nécessaire à la juxtaposition sans superposition d'une collection d'objets. Par le choix d'un espace et d'objets appropriés, l'ordonnancement se ramène à un problème de *bin-packing*.

Dans notre cas l'espace correspond au produit PE × *temps* et les objets correspondent aux opérations à exécuter. Le premier problème qui se pose est celui de la représentation des opérations (il peut y en avoir des centaines de milliers). Nous commençons par proposer une représentation symbolique. Puis nous passons en revue un certain nombre de résultats concernant le bin-packing.

I.3.1. La représentation des tâches 81/2

Si on considère un programme d'une centaine de tissus, chacun ayant un millier de points, on obtient une centaine de millier de tâches. Il est donc exclu de les manipuler explicitement. Le formalisme des tissus va permettre permet de les manipuler sous une forme symbolique : un tissu représente un ensemble de tâches qu'on va pouvoir manipuler globalement. Les tâches associées à un tissu sont indépendantes sauf dans le cas des tissus spatialement récursifs (Cf. le chapitre V). *Dans ce qui suit, nous ne considérons pas les tissus spatialement récursifs, pas plus que les tissus complexes* : mais nous reviendrons dessus à la fin de ce chapitre.

Du point de vue de l'exécution, on peut représenter un tissu comme une région rectangulaire dans l'espace *temps* ×PE (Cf. la figure 2). La longueur de ce rectangle (suivant l'axe des PE) correspond au nombre de tâches associées au tissu. Idéalement, c'est-à-dire si on a assez de PE, elles peuvent toutes se dérouler en parallèle. C'est pourquoi la longueur du rectangle correspond au nombre de tâches (i.e. de points).

La hauteur du rectangle correspond à une *durée* : c'est le temps nécessaire au calcul de la valeur d'un point du tissu. Toutes les tâches utilisent le même temps de calcul : c'est pourquoi la région associée à un tissu est rectangulaire.

On aurait pu tout aussi bien associer deux, trois, … ou bien *n* rectangles à un tissu, chacun représentant un sous-ensemble des points (*n* représentant le cardinal du tissu). Mais on veut bien sûr minimiser le nombre des objets à manipuler. Aussi, plutôt que de considérer plusieurs rectangles plus petits, on va permettre en cas de besoin, de découper le rectangle initial en rectangles plus petits. Cependant, on ne peut pas décomposer un rectangle n'importe

comment. La principale contrainte est que les sous-rectangles doivent avoir la même hauteur que le rectangle initial : cela correspond au fait qu'on ne considère pas d'ordonnancement préemptif. Par suite on ne peut découper l'activité d'une tâche en morceaux plus petits, ce qui rend incompressible la durée d'occupation d'un PE. La deuxième contrainte est bien évidement de conserver la surface du rectangle initial.

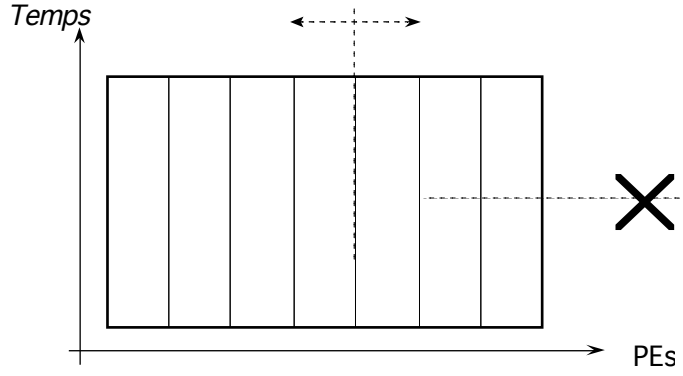


Figure 2 : Un modèle des tissus du point de vue de l'exécution. Un tissu est un ensemble de tâches qu'on peut effectuer en parallèle. L'ordonnancement des tâches n'est pas préemptif ce qui veut dire qu'une tâche a *une certaine hauteur*, insécable, correspondant au temps qu'elle utilise pour s'exécuter sur un PE. Elle a aussi une longueur qui correspond au nombre de PE utilisés. On peut couper une tâche en deux dans le sens de la longueur.

Cependant, même en regroupant les tâches correspondant à un tissu, il faut encore compter une centaine de tissus, sans compter que les architectures cibles ont de l'ordre du millier de PE. Il est donc exclu d'utiliser pour l'ordonnancement des algorithmes dont la complexité n'est pas polynomiale. En effet, on veut conserver des temps de compilation de l'ordre de la minute car un compilateur doit rester un outil de développement.

Plus récemment a apparu en recherche opérationnelle le problème de l'ordonnancement de tâches requérant plusieurs processeurs pour s'exécuter [Du, Leung 89]. Ce modèle ne correspond pas aux tissus 81/2 qui ne sont qu'un moyen pour regrouper des ensembles d'opérations qui peuvent se dérouler indépendamment. En particulier il est toujours possible de transformer un rectangle en deux (sous) rectangles indépendants (qui auront des dates d'activation différentes). Cela est à opposer aux tâches multi-processeurs qui sont contraintes à s'exécuter simultanément.

I.3.2. Ordonnancement sans contrainte de précédance et sans coût de communication

Un modèle simple de *packing* à deux dimensions sans contrainte est le suivant. On se donne une capacité N et un ensemble L de tâches $\{t_1, t_2, \ldots, t_n\}$. Chaque tâche t a une certaine surface $h(t) \times l(t)$ avec $0 \leq l(t) \leq N$. Le problème est de déterminer un arrangement des tâches dans un rectangle $D \times N$ qui minimise D. Les rotations des tâches ne sont pas permises. Cela peut se représenter par la figure 3 suivante :

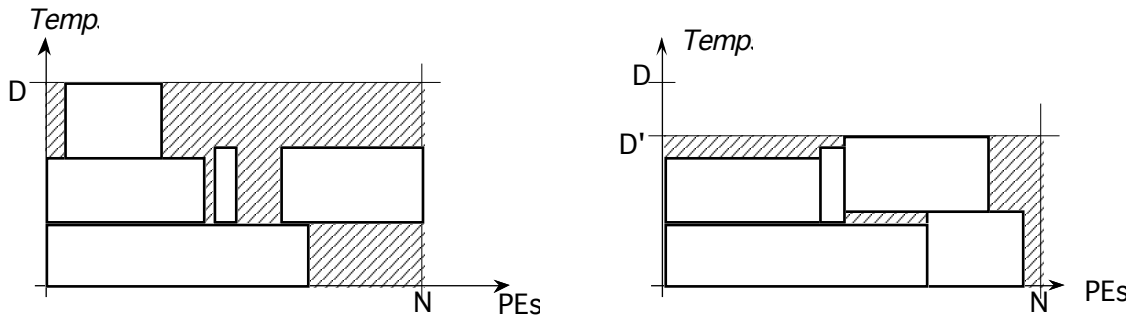


Figure 3 : Deux arrangements possibles des mêmes tâches. Le deuxième arrangement utilise une surface moins importante, ce qui veut dire que la période de l'ordonnancement est plus petite. La partie grisée représente une inactivité des PE correspondants.

Ce modèle peut s'interpréter de la manière suivante :

- N représente le nombre de processeurs
- D correspond à la période de l'ordonnancement, quantité que nous cherchons à minimiser.
- h(t) représente le temps d'exécution d'une tâche. Ce temps est incompressible : l'ordonnancement est non-préemptif, une fois commencée, la tâche se poursuit jusqu'à sa terminaison.
- l(t) représente le nombre d'opérations qui peuvent se dérouler en parallèle.
- Le problème du placement des tâches correspond au cas particulier de rectangles ayant tous une largeur de 1.

Une grande famille d'algorithmes s'exprime comme l'ordonnancement successif des tâches à partir d'une liste pré-ordonnée (*list-scheduling*). Quelle que soit la stratégie d'élaboration de la liste, on peut montrer que quand le nombre de rectangles à placer tend vers l'infini, alors

$$D_\infty \leq (2 - 1/N)\, O_\infty$$

où O_∞ est la durée de l'ordonnancement optimal et D_∞ la durée de l'ordonnancement généré par un algorithme de list-scheduling. Le comportement asymptotique de l'algorithme nous intéresse pour plusieurs raisons : si on a beaucoup de tâches, on se rapproche de ce comportement. De plus les performances asymptotiques permettent de « gommer » les mauvaises performances dues à des cas très particuliers.

On peut illustrer les algorithmes de list-scheduling par l'algorithme FFDH (first-fit, decreasing height). Il consiste à trier la liste des tâches par ordre de durées décroissantes. Les rectangles sont ensuite placés par niveau. Le premier niveau correspond au temps 0. On place les rectangles de gauche à droite, en choisissant séquentiellement dans la liste le premier rectangle qui « tient » dans l'espace restant. Quand plus aucun rectangle ne tient, on commence un nouveau niveau. La figure 4 donne un exemple.

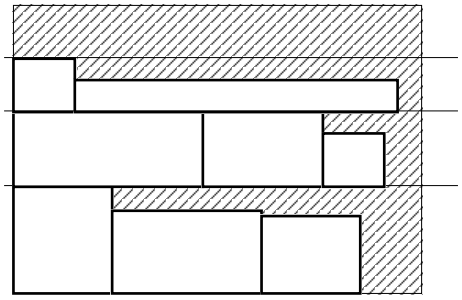


Figure 4 : un exemple de FFDH

On peut montrer que cet algorithme produit au pire une durée totale qui est inférieure à trois fois la durée optimale. Mais si on s'intéresse au comportement asymptotique de l'algorithme, alors on trouve une borne de 17/10 (cette borne améliore le coefficient 2 donné précédemment pour tous les algorithmes de list-scheduling). Des algorithmes plus sophistiqués permettent d'améliorer cette borne (on peut descendre à 5/4 au lieu de 17/10).

Les rectangles associés aux tissus 81/2 ont des longueurs différentes mais il est toujours possible de les couper en sous-rectangle. Quand un rectangle R a été sélectionné, il se peut qu'il soit trop long. On le coupe alors en deux sous-rectangles R1 et R2. La longueur de R1 est telle qu'elle « tient » dans l'espace restant. Le rectangle R2 est inséré en tête de la liste *d'ordonnancement* : c'est possible car R2 a même hauteur que R; l'insertion en tête de R2 respecte donc le tri de la liste). La figure 5 donne un exemple. Tout se passe donc comme si on avait ordonnancé chacune des tâches explicitement.

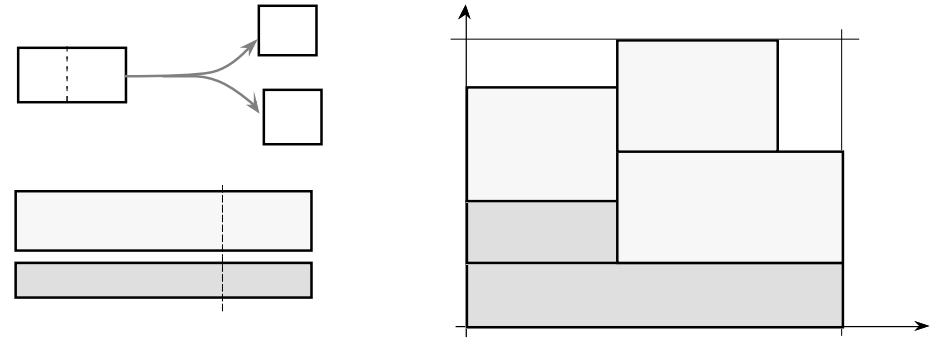


Figure 5 : On peut couper en deux un rectangle 81/2 afin d'obtenir des sous-rectangles utilisant moins de PE. La transformation doit se faire à surface constante Les rectangles obtenus après transformation sont soumis aux mêmes contraintes que le rectangle initial. Du point de vue de l'ordonnancement des tâches, tout se passe comme si on avait explicitement ordonnancé les tâches une à une.

I.3.3. Ordonnancement avec contrainte de précédances et sans coût de communication

L'hypothèse que les tâches peuvent s'exécuter dans n'importe quel ordre n'est pas correcte : il faut tenir compte de l'ordonnancement qui a été calculé dans le chapitre précédent. Dans ce cas la généralisation d'un algorithme de list-scheduling redonne la même performance asymptotique.

Remarquons que dans le cas de 81/2 on a plusieurs sortes de dépendances mais qui s'expriment *in fine* comme des dépendances point à point entre les opérations des tâches correspondantes (Cf. chap V). La figure 6 ci-dessous donne un exemple :

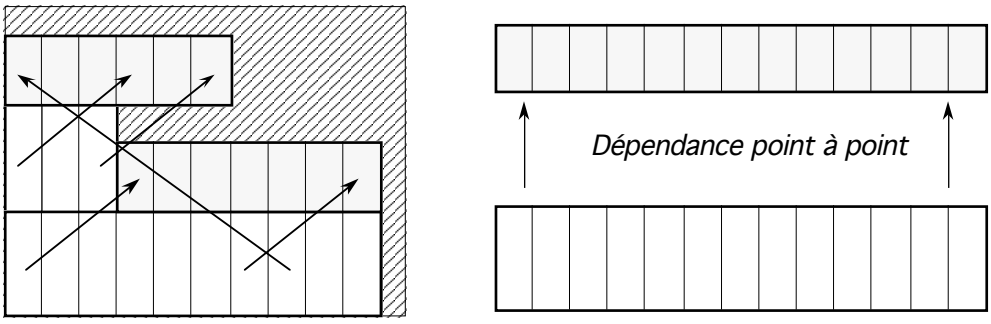


Figure 6 : Deux tâches dont les dépendances sont point-à-point peuvent se retrouver en parallèle si les opérations respectent cette dépendance. Dans la figure, les 7 premières opérations de la deuxième tâche s'exécutent en parallèle avec les trois dernières opérations de la première tâche.

I.4. Ordonnancement avec contrainte de précédances et coût de communication

I.4.1. Modélisation des communications

Nous modélisons à présent le coût de communication entre tâches. Pour qu'une tâche puisse démarrer, il faut que ses prédécesseurs soient terminés et que les messages à destination de la tâche soient arrivés. Nous ne parlerons de communication que dans le cas où la tâche émetteur et la tâche récepteur sont sur des PE différents (sinon le coût de la communication est réduit à 0).

Dans le cas d'une communication déterministe, on suppose implicitement que quand une tâche démarre, ses paramètres sont tous présents. L'ordonnancement doit assurer cette propriété, tout en minimisant l'inutilisation des PE.

Dans le cas d'une communication synchrone, l'attente de l'arrivée des messages est implicite. L'ordonnancement a juste pour effet de répartir les tâches sur les PE à partir de la connaissance de leur interaction (i.e. deux tâches qui peuvent se dérouler en parallèle ont une chance de se trouver sur des PE différents, et deux tâches qui sont séquentielles ont une chance de se retrouver sur le même PE). Cependant il faut aussi réduire les temps d'attente. Pour cela on supposera, comme dans le cas déterministe, que les temps de communication d entre deux PE sont bornés par une formule du type

$$C = f(x)$$

où x représente la quantité de données à transmettre. Ainsi on fait l'hypothèse que le délai de communication ne dépend pas des processeurs émetteur et récepteur et qu'il n'y a pas de contention.

On se place donc dans le cas idéal où on a pas besoin de considérer une topologie particulière sur l'axe des PE dans un diagramme de Gantt (les diagrammes de Gantt correspondent aux diagrammes « *temps* × PE » que nous avons utilisés). On peut considérer que les points de cet axe sont répartis par une fonction de hashage de manière uniforme dans l'espace réel des PE. Nous allons cependant rapidement examiner comment un modèle plus fin, qui tienne compte de la topologie du réseau de communication, peut être développé.

La prise en compte de la topologie du réseau de communication peut se faire soit en considérant une fonction de coût plus complexe, soit dans le processus même de *packing* en considérant que l'espace à remplir a des propriétés topologiques qui influent sur le remplissage. Dans le premier cas, rien de ce qui va suivre n'est fondamentalement modifié.

Un exemple de la deuxième approche est présenté dans [Li, Cheng 89] [Li, Cheng 90]. Les auteurs proposent une extension du *bin-packing* au cas où l'espace à remplir est tri-dimensionnel. Cela permet de modéliser un espace de PE connectés en grille 2D (avec la dimension temporelle, on obtient bien un espace tri-dimensionnel). Les algorithmes de remplissage classiques présentent alors de mauvais comportements. Par exemple il n'est pas possible de borner le comportement asymptotique d'une stratégie FFDH (mais les auteurs proposent des algorithmes admettant une borne asymptotique).

Cette approche nous intéresse dans le cas de PTAH. Elle semble pourtant souffrir de plusieurs faiblesses :

– Le réseau de communication est une grille 3D. L'espace de remplissage est donc un espace de dimension 4, ce qui complique les algorithmes de remplissage.

– Il n'est plus possible de représenter une tâche symboliquement par une partie connexe (cube 4D). En effet, deux cubes adjacents ne représentent pas une communication optimale entre les opérations des deux tâches. Un exemple est donné dans la figure 7, avec un espace mono-dimensionnel (qui correspondrait à des PE en *pipe-line*).

Par ailleurs, en ce qui concerne PTAH64k, la variation de la durée de communication entre PE est de 1 à 3. Il nous semble donc raisonnable de considérer un modèle sans topologie et d'équilibrer le ratio communication/calcul à travers la grosseur des tâches, comme cela sera exposé plus avant dans ce chapitre.

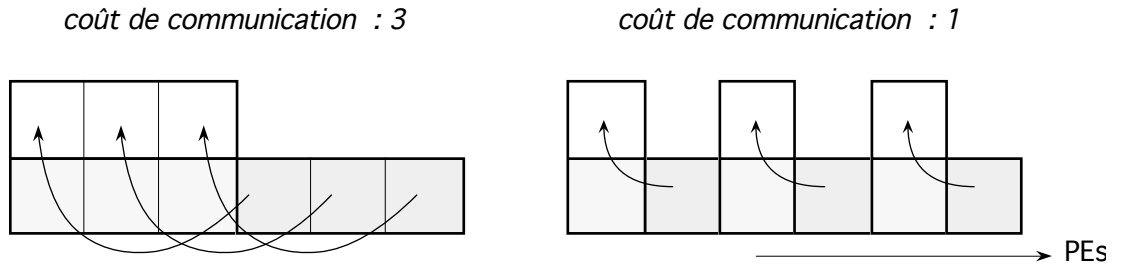


Figure 7 : Du moment où la topologie du réseau entre en considération, il devient difficile de représenter tout un ensemble d'opérations par une région connexe de l'espace de *mapping*. Comme le montre la figure (qui représente le calcul par la tâche blanche de la somme point à point des résultats des tâches grises), on réduit le coût de communication par n (avec n le cardinal d'un tissu) quand on peut entrelacer spatialement les opérations.

I.4.2. Un algorithme général

Beaucoup d'heuristiques proposées dans la littérature peuvent se voir comme des variantes de l'algorithme analysé dans [HCAL 89]. Lui-même est une extension naturelle des stratégies de list-scheduling pour prendre en compte les délais de communication. L'algorithme analysé, appelé ETF pour *Earliest Task First*, étend un algorithme présenté dans [Rayward-Smith 87] et restreint aux cas où les délais de communication et le temps d'exécution des tâches sont unitaires. Dans l'algorithme ETF initial, une tâche réside sur un seul PE. C'est-à-dire qu'il correspond à un problème de bin-packing où tous les rectangles ont une longueur de 1. Nous le présentons tel quel avant de l'adapter à notre cas.

L'algorithme ETF est un algorithme qui applique une stratégie *gloutonne* : il ordonne la tâche qui est disponible au plus tôt. Quand plusieurs tâches sont disponibles au plus tôt, il faut en choisir une arbitrairement. Par exemple [Sih, Lee 90] choisit celle qui se trouve sur le chemin critique. Une tâche T est disponible au temps t si :

- les tâches qui la précèdent ont été ordonnées et se sont achevées à une date $t' < t$;
- $(t - t')$ représente un délai suffisant pour acheminer un message d'un processeur sur lequel réside une des tâches qui précède T à un processeur libre à l'instant t.

Afin de décrire plus précisément l'algorithme, nous introduisons les notations suivantes :

pré (T)	représente l'ensemble des tâches qui précèdent immédiatement T
f (T)	représente la durée d'une tâche T
p (T)	représente le processeur sur lequel se déroule la tâche T
d (T, p)	représente le délai de communication des résultats de T (qui réside sur p(T)) au processeur p
p (t)	représente la liste des processeurs libres au temps t

Avec ces notations, une tâche T est disponible à un instant t si :

$$t \geq \text{Max} \{ f(S) + d(S, p) \mid S \in \text{pré}(T), p \in p(t) \}$$

La tâche $\underline{T}$ disponible au plus tôt à partir d'un instant t minimise donc la quantité

$$\text{Max} \{ f(S) + d(S, p) \mid S \in \text{pré}(\underline{T}), p \in p(t'), t' \geq t \}$$

Évidemment, on ne veut pas calculer le maximum pour n'importe quel temps $t' \geq t$: on va calculer cette quantité à

chaque date correspondant à la fin d'une tâche (ce n'est qu'à ces dates qu'une nouvelle tâche est susceptible de devenir disponible ou que le parc de processeurs libres s'agrandit).

Attention : il ne suffit pas de considérer l'ensemble des tâches disponibles à la terminaison d'une tâche pour déterminer la prochaine tâche à ordonner. La figure 8 ci-dessus montre pourquoi. L'algorithme doit donc retarder l'ordonnancement effectif d'une tâche disponible jusqu'à ce qu'on soit certain que cette tâche est la tâche disponible au plus tôt. C'est le cas si aucune fin de tâche ne survient avant la fin de la tâche candidate. Afin d'en tenir compte, on va utiliser une variable nt représentant la prochaine fin de tâche. La variable t représente la date courante, p les processeurs libres à la date t. La variable « disponible » dénote les tâches disponibles à la date t. L représente la liste des tâches qui ont été ordonnées.

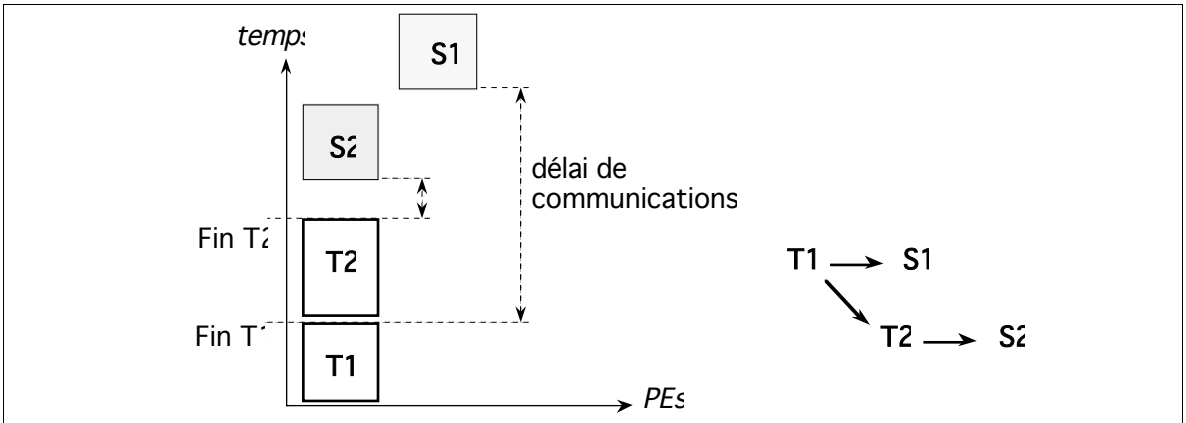


Figure 8 : Une tâche S2 qui n'est disponible qu'à la fin d'une tâche T2 disponible après T1 peut devenir disponible plus tôt qu'une tâche S1 immédiat successeur de T1. Il suffit pour cela que la quantité de données à échanger soit suffisamment importante.

On peut donc exprimer l'algorithme ETF ainsi :

```
1) P ← { tous les PE }, disponible = {T | pré(T) = ∅ }, t = 0, nt = ∞
2) Tant qu'il reste des tâches non ordonnées
   2.1) Tant que P ≠ ∅ et disponible ≠ ∅
              T, q ←    un couple ∈ disponible × P qui
                        minimise temps = Max { f (S) + d(S, q) | S ∈ pré(T)
}
              début = max (t, temps), fin = début + f(T)
              Si début ≤ nt alors
                   // T se déroulera sur q
                   L ← L + T
                   disponible ← disponible – T
                   P = P – q
                   Si nt ≥ fin alors nt ← fin sinon sortie de la boucle 2.1
   2.2) // Y avait plus de PE libres, ou bien plus de tâches disponibles
        // ou bien de nouvelles tâches et de nouveaux PE vont être rendus
        // disponibles par une fin de tâche
        t = nt
        nt ← la plus petite date de terminaison > t d'une tâche de L
   2.3) P ← les PE libres à l'instant t
        disponible = disponible + {T | ∀S ∈ pré(T), S ∈ L}
```

I.4.3. Performances de l'algorithme ETF

Les résultats établis dans [HCAL 89] sont les suivants.

La complexité de l'algorithme est de $O(Nm^2)$ où N est le nombre de PE et m le nombre de tâches.

La durée D d'un ordonnancement produit par ETF vérifie :

$D \leq (2 - 1/N)\ O' + C$

où O′ est la longueur d'un ordonnancement optimal *ignorant la communication entre PE*, et C est une constante bornée supérieurement par C_m :

$C_m = \text{Max} \{ C_{T1, \ldots, Tn} \mid \text{pour toutes chaînes T1, …, Tn qui existent dans le programme} \}$

avec

$C_{T1, \ldots, Tn} = \sum f(x_i)$ où x_i est la quantité de données à transmettre entre T_i et T_{i+1}

Cette borne supérieure, C_m, peut être améliorée car la plupart du temps, la communication prend place pendant un calcul et n'induit pas de délais supplémentaires. L'algorithme suivant calcule une meilleure borne :

```
// algo pour calculer la meilleure borne C d'un ordonnancement ETF
1) C ← 0, T ← une tâche avec la plus grande date de fin.
2) u = Sup {t | ???? et il y a au moins un PE de libre à l'instant t }
3) Si u = 0 alors stop et C contient la nouvelle borne
4) Si u ≤ début (T) alors
        LT ← les tâches qui précèdent T et qui vérifient début(T′) ≥ u
             ainsi que T si début(T*)<u pour tout T* immédiat prédécesseur
de T
5) T′ ← une tâche de LT qui finit le plus tard.
   Si fin (T′) = u alors T ← T′, aller en 2)
6) Trouver un processeur P libre pendant [u - ε, u], pour un certain ε > 0 .
   Trouver une tâche T′ ∈ pré(T)
   C ← C + d(T′, P)
   aller en 2)
```

En général la borne C calculée par l'algorithme est bien meilleure que la borne C_{max}.

I.4.4. Extensions de l'algorithme au cas 81/2

La complexité théorique de l'algorithme est encourageante : elle est globalement polynomiale, et linéaire en le nombre de PE. La complexité quadratique en le nombre de tâches n'est certainement pas raisonnable si on considère les opérations, mais plus qu'acceptable si on considère les tissus.

Les performances théoriques sont très attractives : si on néglige la constante, comme on peut le faire pour une analyse asymptotique, on obtient la garantie que le rendement de la machine n'est pas inférieur à 50%. C'est un résultat excellent : ce résultat n'est par exemple pas atteint par une machine SIMD comme la Connection-Machine. De plus il faut tenir compte que la borne donnée est une borne au pire. Par ailleurs l'ordonnancement fourni par l'algorithme peut n'être qu'un ordonnancement initial, soumis à des optimisations ultérieures ([LHF 91] propose un exemple d'optimisation d'un placement initial fait à partir d'une analyse de chemin critique).

L'algorithme présente la particularité de pouvoir incorporer diverses heuristiques de choix des tâches à ordonner, quand il y en a plusieurs disponibles. [Sih, Lee 90] propose une version de l'algorithme qui choisit les tâches disponibles sur le chemin critique (obtenu par une technique PERT). Des mesures, faites à partir de graphes aléatoirement générés, montrent des ordonnancements bien meilleurs qu'avec un algorithme HLFET (qui consiste à placer les tâches sans tenir compte des communications, puis à retarder les dates de début des tâches afin de rendre admissible l'ordonnancement obtenu).

Par ailleurs ETF accepte parfaitement des fonctions de coût plus compliquées, à condition que celles-ci ne dépendent que du placement des tâches (la contention n'est pas prise en compte). C'est nous qui avons restreint les fonctions de coût à avoir cette forme afin de généraliser plus facilement l'algorithme au cas des tâches multi-processeurs.

Pour toutes ces raisons, nous avons adopté ETF comme la base de la stratégie d'ordonnancement du compilateur 81/2. Plusieurs adaptations sont nécessaires :

– L'algorithme ETF est utilisé pour l'ordonnancement des parties « data-flow synchrones » des programmes 81/2. L'ordonnancement des every, until, after et at, ainsi que l'ordonnancement des tissus définis par une récursion spatiale, demandent un traitement particulier qui est décrit plus loin.

– Nous voulons ordonnancer les opérations, sans les manipuler toutes explicitement (il y en a trop). On va donc adapter l'algorithme afin qu'il manipule des groupes d'opérations (les tâches). Les assignations à des PE se feront par intervalle de PE libres (éventuellement un intervalle est réduit à un seul PE, par exemple dans le cas d'un tissu scalaire). Quand un tissu présente plus d'opérations que de PE disponibles, on crée deux sous-rectangles, ainsi qu'il a été montré plus haut dans le cas du *packing*. Le sous-rectangle restant est placé parmi les rectangles disponibles, exactement comme dans le cas du FFDH.

– Le calcul du coût de la communication ne dépend pas des PE. Par ailleurs, la quantité de données échangées entre tâches de deux rectangles est indépendante des tâches (et ne dépend que des rectangles). Par suite le coût de communication entre deux rectangles est aussi celui de la communication entre tâches. Le critère « rectangle disponible » correspond donc bien au critère « tâches disponibles »[1].

L'implémentation de l'ordonnancement passe donc par la manipulation d'un très grand nombre de rectangles « sécables ». Nous avons développé à cet effet un ensemble de classes C++ permettant une « algèbre des rectangles » ainsi qu'une « algèbre d'arbre de rectangles » : union, intersection, etc. Ces modules constituent le noyau logiciel à partir duquel a été implémenté l'ordonnanceur 81/2 : **ROMA** (Rectangular Object Management Algebra). La figure 9 présente un exemple d'ordonnancement sur 8 PE du programme 81/2 suivant :

S [10] = $S + A ;	*temps d'exécution*		1
P = $P * A ;	—		2
I = $I + 1 when A ;	—		1
M = S / I ;	—	4	
G = P / S	—		4

[1] La prise en compte d'un modèle plus fin, en particulier qui tiendrait compte de la distance entre PE, peut rendre certaines tâches d'un rectangle disponible alors que d'autre non.

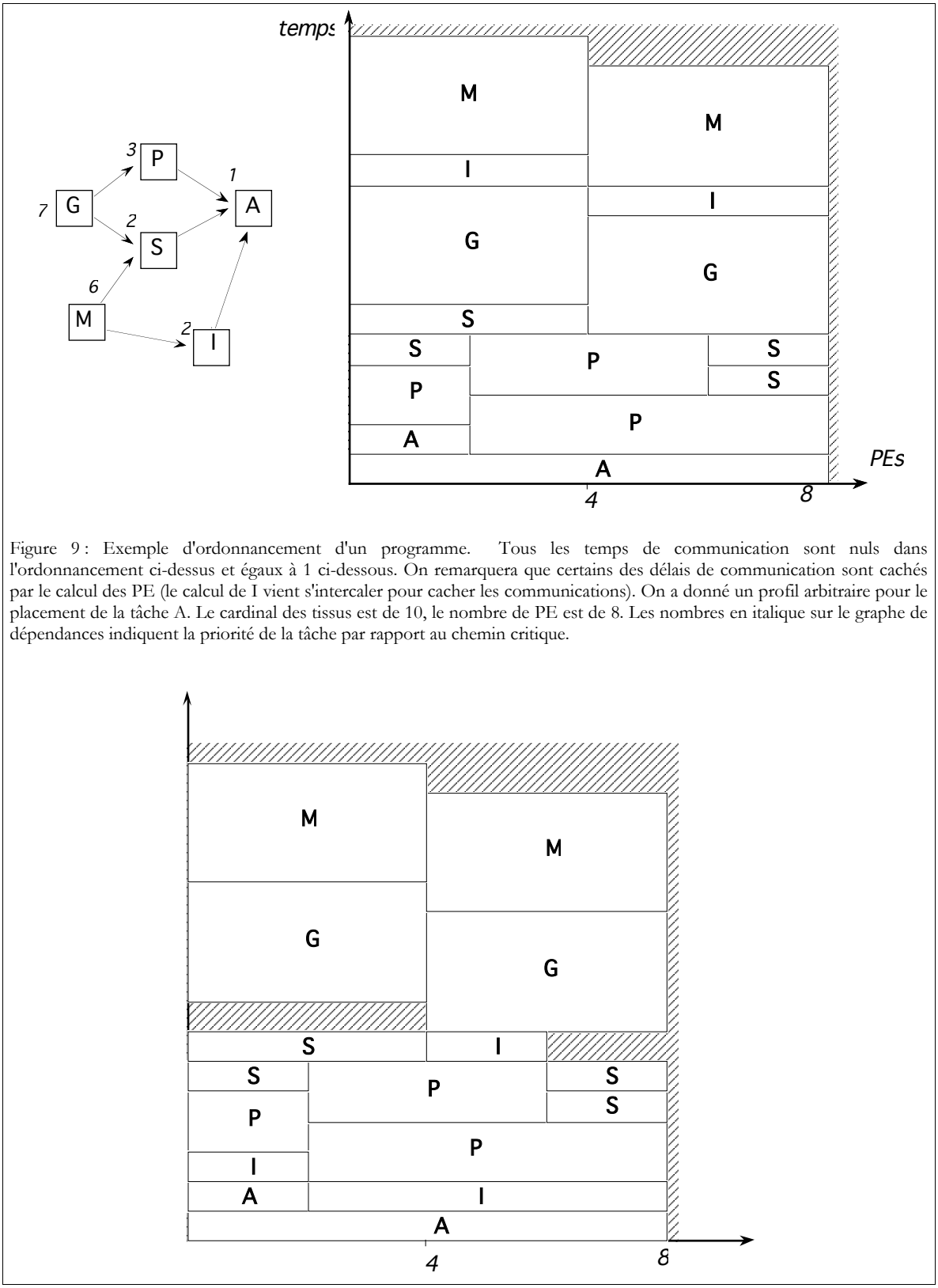


Figure 9 : Exemple d'ordonnancement d'un programme. Tous les temps de communication sont nuls dans l'ordonnancement ci-dessus et égaux à 1 ci-dessous. On remarquera que certains des délais de communication sont cachés par le calcul des PE (le calcul de I vient s'intercaler pour cacher les communications). On a donné un profil arbitraire pour le placement de la tâche A. Le cardinal des tissus est de 10, le nombre de PE est de 8. Les nombres en italique sur le graphe de dépendances indiquent la priorité de la tâche par rapport au chemin critique.

II. Compilation des programmes 81/2

Nous allons à présent décrire plus précisément comment les techniques évoquées plus haut sont utilisées pour la compilation d'un programme 81/2.

II.1. Ordonnancement des tics

Chaque tâche au moment où elle est activée s'attend à trouver ses données en mémoire. Ces données sont rangées à des adresses calculées à la compilation (les communications étant statiques, il est possible de réserver l'espace mémoire à la compilation).

Dans le cas d'une communication déterministe, il faut éviter qu'une tâche d'un tic t+1 vienne écraser les données d'une tâche d'un tic t (Cf. figure 10) : il faut donc mettre une barrière de synchronisation entre le calcul des tics. Cette barrière n'est pas nécessaire dans le cas d'une communication synchrone où l'émetteur est en attente de l'acquittement du récepteur. Dans tous ce qui suit, on fera l'hypothèse d'une communication déterministe, puisque c'est elle qui pose problème.

Une première idée pour implémenter la barrière de synchronisation est la synchronisation statique par insertion de *nops* : on suppose qu'il existe une horloge globale permettant de mesurer l'écoulement du temps dans la machine, on calcule la date de fin au plus tard des tâches et sur chaque PE on insère l'attente correspondante. Il faut remarquer que pour cela le calculateur n'a pas besoin d'être synchrone : par exemple sur un iPSC/2, architecture MIMD, il est possible de connaître sur chaque PE une date globale avec une exactitude de l'ordre de la micro-seconde.

Cependant une synchronisation statique ne suffit pas : nous allons voir que nous avons besoin de synchroniser un groupe de PE sur une condition calculée. En conséquence, nous allons demander qu'il existe sur le calculateur un moyen de synchronisation globale. Dans le cas idéal, on peut synchroniser une partition quelconque de PE (c'est le cas pour PTAH). Sinon, on sera obligé de synchroniser inutilement tous les PE.

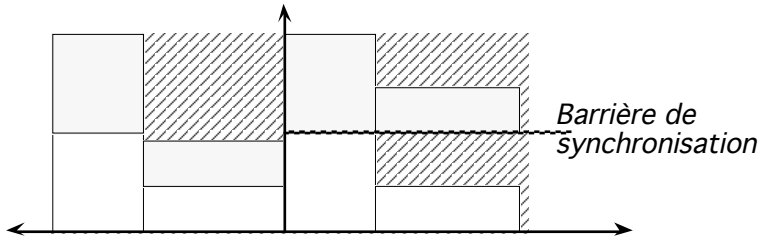


Figure 10 : Exécution du code avec et sans barrière de synchronisation. Sans barrière de synchronisation, les opérations de deux tics différents peuvent se recouvrir (ici on a recouvrement des opérations blanches et grises) ce qui peut conduire à l'écrasement des données envoyées. Une barrière de synchronisation « naturelle » existe quand la communication est synchrone.

II.2. Ordonnancement des *every*

La synchronisation dynamique est nécessaire à l'implémentation des *every*. En effet, un *every* est une structure de contrôle qui engendre un nombre non connu de calculs. En fait le corps d'un every correspond exactement à un programme 81/2 en soi. Le principe d'implémentation est donc le suivant :

- On réserve une partie de la machine au calcul de l'every. Si le calcul de l'every peut occuper toute la machine, alors c'est ce choix qui est utilisé. Sinon, le déroulement de l'every laisse libre une partie de la machine, partie libre qu'on peut utiliser pour l'exécution des tâches parallèles à l'every.
- Le corps de l'every est calculé sur la partie qui a été réservée, exactement comme si c'était un programme 81/2 qui s'exécutait sur une machine plus petite. En particulier, si le corps de l'every contient un autre every, la procédure est répétée.
- Le reste de la machine est dédié au calcul des expressions du programme qui peuvent se dérouler en parallèle.

– Une barrière de synchronisation dynamique permet de synchroniser les processeurs sur la terminaison des deux groupes de tâches.

La figure 11 représente le déroulement d'un tic en présence d'un every. Nous avons représenté les tâches correspondant à l'exécution de l'every sur des PE contigus pour des raisons de commodité. Il n'en est rien : les n PE alloués à un every sont les n premiers PE libres le plus tôt, et ils n'ont aucune raison d'être les uns à coté des autres. De manière générale, la contiguïté sur l'axe des PE n'a aucune signification.

L'ordonnancement des différents tics internes à l'every demande aussi une synchronisation. S'il n'y a pas d'imbrication des every, cette synchronisation peut être statique (insertion de nops). Si les every sont imbriqués, il faut à nouveau une synchronisation dynamique. C'est pourquoi nous supposerons qu'il existe un moyen de synchroniser dynamiquement un groupe quelconque de PE.

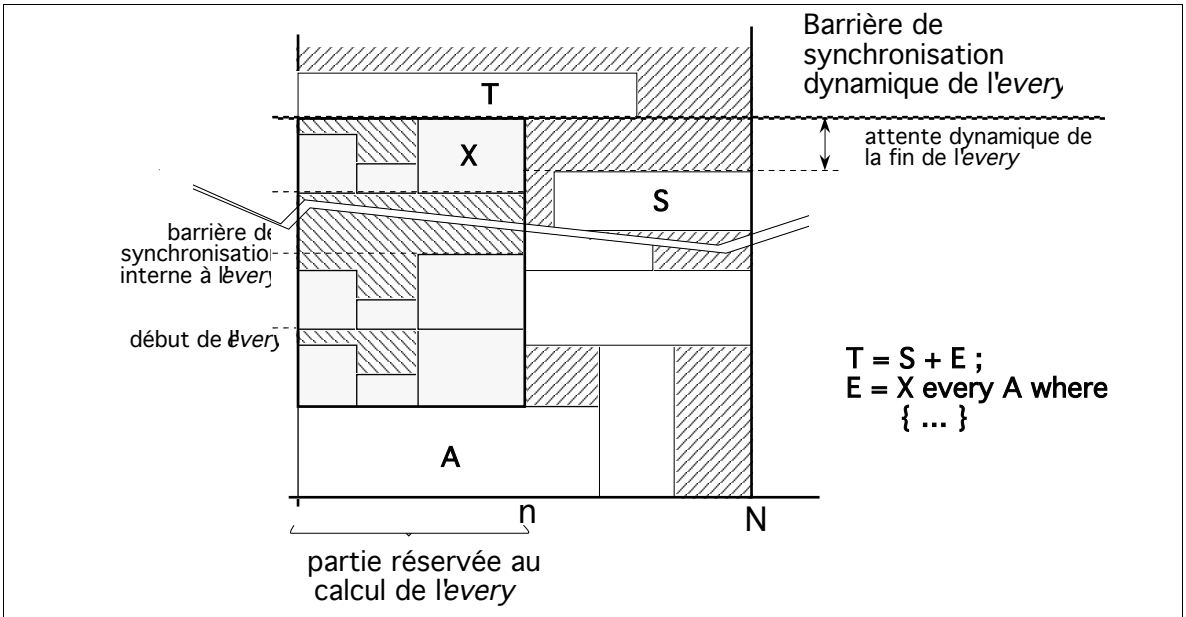


Figure 11 : Implémentation des *every*. T est une tâche qui a besoin du résultat de l'every pour s'exécuter. S est une tâche indépendante. Quand toutes les tâches indépendantes ont été exécutées, il y a synchronisation sur la fin de l'every. Il faut noter que le cas contraire peut-être vrai : le *every* peut se terminer avant la fin des tâches du programme extérieur, auquel cas il faut se synchroniser sur la dernière de ces tâches.

L'implémentation d'un every entraîne donc une attente correspondant à la synchronisation des deux parties de la machine. Le seul paramètre sur lequel nous pouvons agir pour minimiser cette attente est le nombre n de PE qui sont alloués à l'exécution du corps de l'every.

Nous allons présenter une méthode pour la détermination d'un tel n. Elle est basée sur une connaissance probabiliste du nombre de tics entraînés par le calcul du corps de l'every. Cette connaissance peut provenir de deux sources principalement :

– Le compilateur peut utiliser des statistiques issues de profil d'exécutions typiques (à la manière du *trace-scheduling* dans la compilation VLIW [Bulldog]).

– Le programmeur peut fournir, à travers une annotation du programme, le nombre d'itérations attendues. Il peut aussi fournir la probabilité qu'un every débute et la probabilité qu'il continue à la fin d'un tic. Si on traduit un every en une structure de contrôle plus usuelle, cela revient à fournir la probabilité que *x* soit vrai et la loi de probabilité de *y* dans la boucle du programme : if (*x*) { do … until (!*y*) }.

Cette méthode s'appuie sur le concept de longueur moyenne d'une exécution [Martin Estrin, 69] qui est souvent

utilisé dans les stratégies d'allocations (Cf. par exemple [LHF 91]). Une stratégie similaire est évoquée dans [Lee 91 *citant* [Ha, Lee 89?]] pour les itérations dépendantes des données (boucle *tant que*).

II.2.1. Détermination du nombre de PE alloués à un every

Avant de déterminer n, il nous faut un modèle de la variation du temps d'exécution d'un ensemble de tâches en fonction des processeurs alloués. Ce modèle est décrit par la figure 12 :

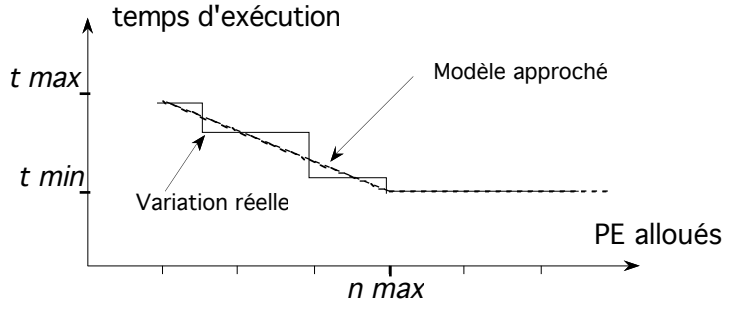


Figure 12 : Variation du temps d'exécution d'un ensemble de tâches en fonction du nombre de processeurs. Cette variation doit être une fonction décroissante (quand on ajoute un PE, le temps d'exécution ne peut pas augmenter) . On approxime la fonction réelle par une fonction linéaire entre 1 et n_{max} processeurs.

Le modèle approxime la variation réelle du temps d'exécution par deux morceaux de fonction affine sur [1, n_{max}] et [n_{max}, ∞[. En effet, au delà d'un certain nombre n_{max}, l'ajout d'un PE ne permet pas de gagner sur le temps d'exécution : tout le parallélisme possible est utilisé et le temps d'exécution est contraint par les dépendances entre tâches. Il est facile de calculer n_{max} et le temps d'exécution t_{min} associé : il suffit d'ordonner les tâches en supposant un nombre infini de PE. Cela correspond à faire la somme des longueurs des rectangles de même niveau[1]. Par ailleurs il est aussi facile de calculer t_{max} le temps d'exécution des tâches sur un PE : cela correspond à la somme des surfaces des rectangles correspondant aux tâches.

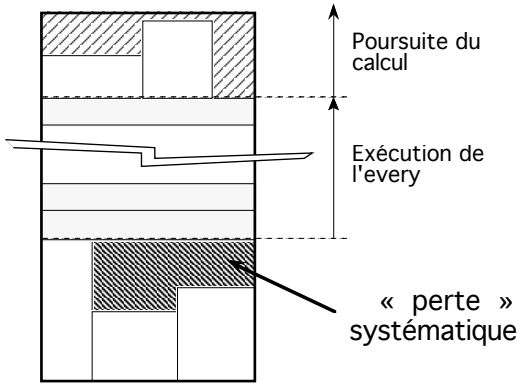


Figure 14 : Ci-dessus, *perte* systématique induite par l'allocation de tous les PE à un every. La perte peut être importante car il faut attendre que tous les PE soient libres. Par contre cette perte est calculable à la compilation et il n'y a pas de perte due à la synchronisation des PE à la fin de l'every.

Afin de déterminer le nombre de processeurs à allouer à l'exécution du corps d'un every, on commence par calculer les paramètres (t_{max}, n_{max}, t_{min}) de la fonction t(n) des variations des temps d'exécution en fonction du nombre de processeurs alloués. L'ensemble des tâches considérées est celui des tâches du corps de l'every. Deux cas

[1] On peut associer à chaque tâche (ou à chaque rectangle correspondant) un niveau. Le niveau d'une tâche est la plus petite longueur d'un chemin entre un sommet sans prédécesseur et la tâche.

peuvent se produire (Cf figure 13 et 14) :

$N \leq n_{max}$

Dans ce cas on alloue tous les PE de la machine à l'exécution de l'every. Il faut donc une première barrière de synchronisation (statique) pour attendre la fin de la dernière tâche qui a été ordonnée, à la suite de quoi l'exécution de l'every débute.

Dans ce cas l'erreur commise est indépendante du nombre d'itérations de l'every et correspond uniquement à l'attente de la fin de la dernière tâche. Si cette erreur (qui est calculable à la compilation) est considérée comme trop importante, alors la deuxième option peut être choisie.

$N > n_{max}$

Dans ce cas on détermine un n suivant l'une des deux stratégies que nous allons exposer. L'erreur commise se décompose en une erreur fixe (attente qu'il y ait n PE libres) plus une erreur qui dépend du nombre d'itérations de l'every : l'erreur dynamique (Cf. fig. 14).

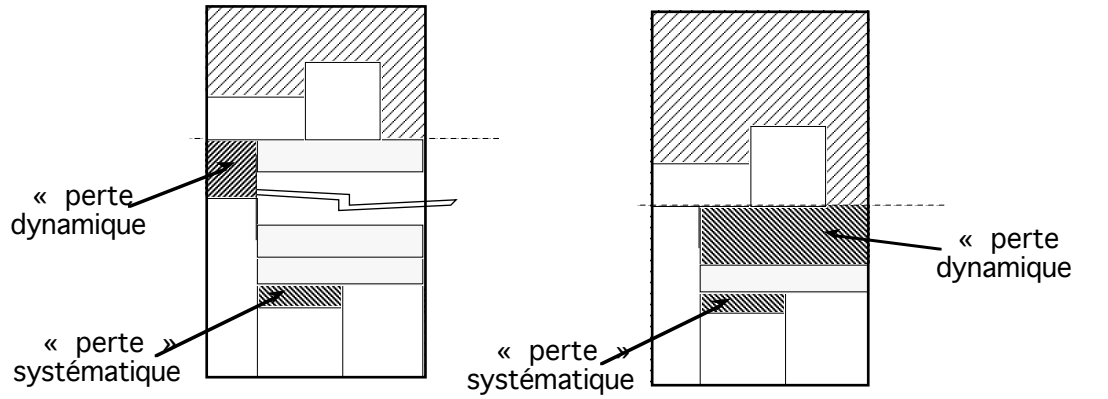


Figure 14 : La figure ci-dessus représente la perte due à l'allocation d'une partie seulement des PE à l'exécution d'un every. Cette perte se décompose en une partie fixe (attente que n PE soient libres) et une partie dynamique qui dépend du nombre d'itérations (synchronisation entre la fin de l'every et la fin des tâches qui ne font pas partie de l'every).

La stratégie la plus simple pour déterminer n est de considérer que celui-ci est une constante fixée à la compilation. On veut minimiser la perte. Deux cas sont possibles suivant les informations dont on dispose.

Si on dispose uniquement du nombre d'itérations attendues *l*, alors la perte s'exprime comme

$$R(m) = e(m) + |l t(m) - T(m)|$$

avec

t(m) le temps d'exécution d'un tic de l'every sur m PE, $1 \leq m < N$

T(m) le temps d'exécution d'un tic de ce qui n'est pas l'every, sur (N - m) PE

e(m) la perte systématique due à l'utilisation de m PE

On connaît l'expression des fonctions t, T et on sait calculer e(m) pour chaque m. Il est donc possible d'optimiser E_A par exemple par une méthode de gradiant. On peut aussi modéliser e(m) de manière analogue à la modélisation de t (fig. 15) :

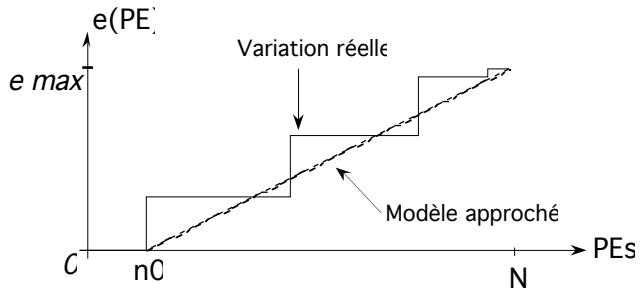


Figure 15 : Modélisation de e(m). La fonction e(m) est une fonction croissante. La quantité e_{max} correspond à la perte si on attend que tous les PE soient libres. Cette quantité se calcule facilement. La quantité n_0 correspond au nombre de PE de la tâche qui se termine au plus tôt. Cette quantité est aussi facilement accessible.

Cela permet de poursuivre explicitement les calculs. R(m) se présente sous la forme d'une fonction linéaire en deux morceaux.

$$R(m) = \begin{cases} m\,(a - a' + la'') + b - Na' + lb'' & si\ l \geq T(m)/t(m) \\ m(a + a' - la'') + b + Na' - lb'' & si\ l < T(m)/t(m) \end{cases}$$

avec

$$a = \frac{e_{max}}{N - n_0} \qquad a' = \frac{t'_{min} - t'_{max}}{n'_{max} - 1} \qquad a'' = \frac{t_{min} - t_{max}}{n_{max} - 1}$$

$$b = \frac{n_0\, e_{max}}{n_0 - N} \qquad b' = t'_{max} - a' \qquad b'' = t_{max} - a''$$

$n'_{max}, t'_{max}, t'_{min}$: paramètres de la fonction $t'(m)$ décrivant les variations de durée de l'exécution des tâches qui ne sont pas celles de l'every

$n_{max}, t_{max}, t_{min}$: paramètres de la fonction $t(m)$ décrivant les variations de durée de l'exécution des tâches de l'every

Ces quantités sont toutes calculables à la compilation, au moment de l'ordonnancement de l'every. Il est facile ensuite de calculer la valeur de n qui minimise R(m). La figure 16 représente les différents cas possibles.

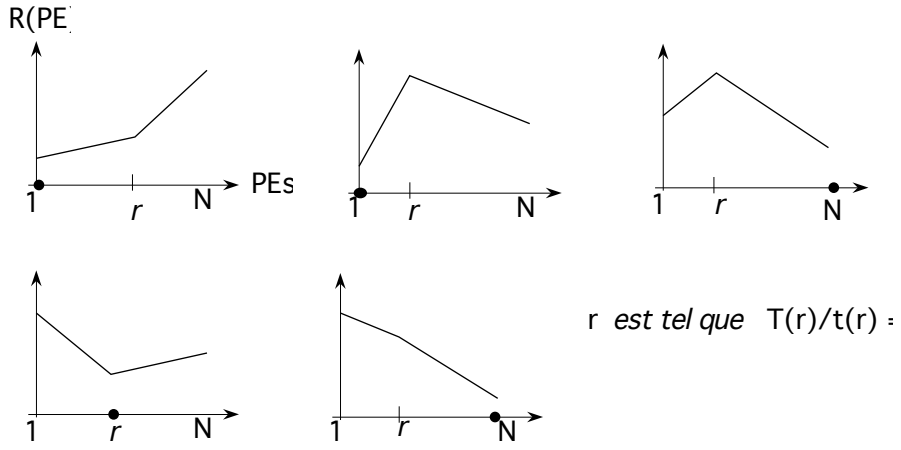


Figure 16 : Les différents cas possibles dans la recherche de n. Après évaluation des coefficients, on détermine dans quel cas on se trouve et on calcule la valeur n correspondante. Nous n'avons pas représenté les cas où par suite de la valeur de l, le point *r* n'appartenait pas à l'intervalle [1, N]. Dans ce cas, la recherche de n donne 1 ou N suivant la pente positive ou négative de la droite.

Si on dispose de la loi de probabilité q décrivant la probabilité qu'une boucle continue à la fin d'un tic, alors on

peut faire une modélisation plus fine. Prenons un exemple où p est décrit par une constante (i.e. la probabilité qu'un every continue à la fin d'un tic suit une loi géométrique). On dispose des éléments suivants :

p probabilité que le every ait lieu

q probabilité que la boucle continue à la fin d'un tic :
pour le moment, une distribution uniforme : q est une constante.

Le calcul du nombre d'itérations moyen de l'every en fonction de p et q est un calcul simple :

P(1) = probabilité qu'il y ait 1 itération = p
P(2) = — 2 itérations = $p\,q\,(1 - q)$

En effet, il a fallu entrer dans la boucle avec une probabilité p. À la fin du premier tic, la probabilité que l'on en commence un second est de q. Enfin, à la fin de la seconde itération, on s'est arrêté avec une probabilité de (1 -q). De manière générale,

P(3) = probabilité qu'il y ait 3 itérations = $p\,q^2\,(1 - q)$
…
P(i) = — i itérations = $p\,q^{i-1}\,(1 - q)$

(On vérifie bien que $\sum P(i) = p$ = *probabilité qu'il y ait au moins une itération.*) Le nombre moyen d'itérations l est alors facilement obtenu par :

$$l = \sum iP(i) = p\,(1 - q) \sum i\,q^{i-1} = p\,/\,(1 - q).$$

À partir de là on peut calculer un n qui minimise la perte. La perte R(k, m) est fonction du nombre d'itérations k effectivement réalisées et du nombre m de processeurs alloués :

$$R_1\,(k, m) = e(m) + |k.t(m) - T(m)|$$

La présence de la valeur absolue ne permet pas de conduire les calculs jusqu'au bout. On peut cependant trouver une autre expression de la perte. On commence par négliger le terme e(m) sur l'hypothèse que ce terme est petit devant la perte due à la synchronisation (On peut toujours vérifier statiquement l'ordre de grandeur de e(m) à la compilation pour voir si cette approximation est acceptable). Dans ce cas la perte R qu'on cherche à minimiser peut s'exprimer par :

$$R\,(k, m) = (k.t(m) - T(m))^2$$

puisque minimiser une quantité toujours positive ou bien son carré revient au même. L'espérance de cette quantité est alors :

$$E(R)(m) = E(k^2.t(m)^2 - 2kt(m)T(m) + T(m)^2)$$
$$= t(m)^2\,E(k^2) - 2E(k)t(m)T(m) + T(m)^2$$

$$E(k) = \sum_i iq^{i-1} = 1/(1 - q)^2$$

$$E(k^2) = \sum_i i^2q^{i-1} = (1 + q)/(1 - q)^3$$

(on obtient E(k) en dérivant $\sum q^i$ et on obtient $E(k^2)$ en dérivant $\sum iq^i$). Par suite, on veut minimiser :

$$E(R)(m) = (1+q)t(m)^2 - 2(1-q)t(m)T(m) + (1-q)^3T(m)^2$$

L'expression de t(m) et de T(m) est linéaire en m, et par suite E(m) est une expression du second degré, que l'on sait parfaitement minimiser sur un intervalle [1, N]. Nous ne conduirons pas les calculs plus avant car les formules obtenues n'ont rien de remarquable.

II.2.2. Extension du schéma

Les calculs que nous avons effectués peuvent se reprendre pour diverses lois de probabilité. En particulier une

loi de Poisson permettrait de modéliser les every dont on connaît le nombre moyen d'itérations (gaussienne centrée en ce nombre). Une autre situation « standard » correspond au calcul de :

while ($\|a\| > \varepsilon$) do a = op(a) ;

sachant que $\|op(a)\| \leq k \|a\|$ (op est lipschitzienne). Dans ce cas, le problème de la détermination du nombre d'itérations revient au problème de la répartition de $\|a\|$.

À la terminaison des tâches à effectuer parallèlement à l'every, tous les PE de la machine sont allouables à l'every (même si celui-ci ne peut utiliser toute la machine). Il est donc possible de tester, à la fin de chaque cycle d'un every, si les tâches parallèles sont achevées. Dans ce cas, il y a un nouvel ordonnancement des tâches de l'every afin de profiter des processeurs libérés. Ce nouvel ordonnancement diminue la durée des tics ce qui permet de réduire la perte de PE.

Cette amélioration conduit à l'idée d'une allocation qui ne serait pas constante au cours du temps. Par exemple pour les 100 premiers cycles, l'every disposerait de la moitié de la machine, pour les 100 cycles suivant du quart, etc. Cette approche demande cependant de connaître plus exactement le type de loi de probabilité que l'on peut rencontrer.

II.2.3. Allocations de multiples every

Dans le cas général, il faut ordonnancer i every en même temps. On se retrouve alors à minimiser la quantité :

$R(k, m_1, m_1, \ldots, m_i) = (iT(m_1) - k.t_1(m_1) - \ldots - kt_i(m_i))^2$

$t_j(m_j)$ la loi de variation du temps d'exécution du corps de l'every numéro j en fonction de la quantité m_j de PE qui lui est accordé. La quantité m_1 correspond au nombre de PE dédiés aux tâches à faire en parallèle. On a la contrainte $m_1 + m_2 + \ldots + m_n = N$. La détermination des valeurs $m_1, \ldots, m_i$ peut se faire à l'aide d'un algorithme de programmation dynamique (le nombre i de variable à déterminer est petit et $1 \leq m_i \leq N$, ce qui doit rendre acceptable le coût d'une telle approche.

II.3. Ordonnancement des récursions spatiales et des tissus complexes

Les récursions spatiales correspondent à des itérations dont on connaît exactement le nombre de répétitions : c'est le cardinal du tissu. Les opérations correspondant aux calculs du tissu s'implémentent donc comme pour un every dont on connaîtrait le nombre d'itérations, ainsi qu'il a été vu plus haut.

Les tissus complexes se gèrent de la même façon : ils correspondent à des itérations imbriquées dont on connaît chaque borne. Il y a un niveau d'imbrication par niveau de hiérarchie du tissu. Cette hiérarchie étant statique, il n'y a aucune difficulté à calculer le temps pris par le calcul du tissu complexe et à générer l'ordonnancement correct.

Les blocs sont « aplatis » avant la compilation et donc ne posent aucun problème.

II.4. Implémentation des sélections

La version statique des sélections, où les destinations sont fixées a priori, ne génère aucune tâche. Le compilateur est simplement chargé d'assurer les connections correctes entre le tissu source et les références au tissu sélectionné (Cf. figure 17).

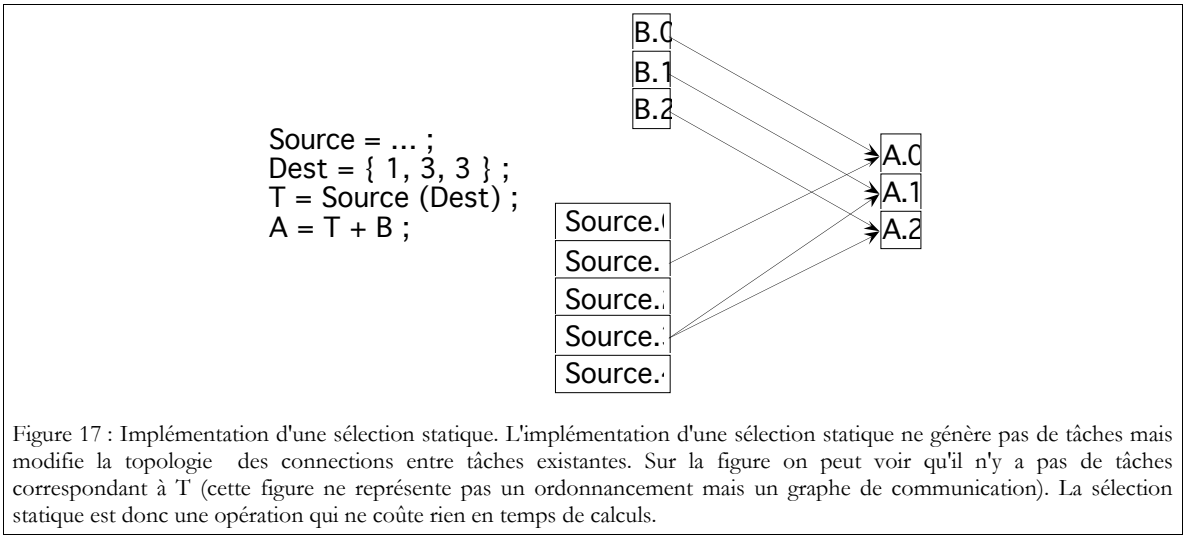


Figure 17 : Implémentation d'une sélection statique. L'implémentation d'une sélection statique ne génère pas de tâches mais modifie la topologie des connections entre tâches existantes. Sur la figure on peut voir qu'il n'y a pas de tâches correspondant à T (cette figure ne représente pas un ordonnancement mais un graphe de communication). La sélection statique est donc une opération qui ne coûte rien en temps de calculs.

Dans le cas d'une sélection dynamique, la machine cible doit offrir un système de routage dynamique (remarquons que c'est le seul endroit où un routage dynamique est nécessaire). L'implémentation de la sélection dynamique

source (destination)

est le suivant : on suppose que le cardinal du tissu destination est plus petit que celui du tissu source; alors

- Une tâche T est associée à chaque point du résultat.
- Une tâche supplémentaire S est associée à chaque point du tissu source.
- S reçoit la valeur correspondante du tissu source.
- S reçoit aussi l'ensemble des valeurs du tissu destination, dans un ordre donné.
- Quand cette valeur correspond à son index, S émet à destination de T sa propre valeur.
- L'émission se fait vers un processeur qui dépend de l'index de la destination. La relation processeur/index de la destination est établie de manière statique par le compilateur. Cette relation n'a pas besoin d'être explicitement représentée : les PE étant alloués par segments contigus, la représentation de cette relation correspond à une fonction linéaire par morceaux.

Dans le cas où le cardinal du tissu destination est supérieur à celui du tissu source, on inverse les rôles : ce sont les valeurs du tissu destination qui reçoivent les valeurs du tissu source. On peut donner un exemple :

Dest = { 1 3 3 } ;
Source = { 10 11 12 13 } ;
T = Source (Dest) ;

Chaque point de Source correspond à une tâche chargée de calculer la valeur du point. La valeur de Source.*i* étant connue, elle est communiquée à une tâche S.*i*. Cette tâche reçoit aussi la suite de valeurs { 1, 3, 3 } en provenance des tâches chargées de calculer Dest. Quand la tâche S.1 reçoit la valeur 1, elle envoie la valeur 11 à T.0. Le processeur alloué à T.0 est connu à la compilation. La tâche S.3 émet deux fois la valeur 13 à destination des tâches T.1 et T.2.

L'implémentation proposée est une implémentation statique, qui fonctionne dans un cas pire. Une implémentation dynamique, i.e. dont on ne connaît pas a priori le temps d'exécution, est possible : chaque tâche du nœud destination émet son index vers le nœud source correspondant à sa valeur. Sur réception de cet index, le nœud destination réémet sa valeur vers le point correspondant du tissu T (calculé à partir de l'index reçu). Dans le cas pire une tâche source doit émettre sa valeur à destination de toutes les tâches T.*i*, ce qui est fait de manière séquentielle.

Mais dans un cas moyen, le temps d'exécution est beaucoup plus petit. Il faut alors traiter la sélection à la manière d'un every dont on connaîtrait le temps d'exécution maximal.

II.5. Ordonnancement des conditionelles

Une expression conditionnelle correspond à deux tâches : la première évalue la condition et la deuxième reçoit la valeur de la condition et tous les arguments nécessaires à l'évaluation de la branche vraie aussi bien que de la branche fausse. Cette dernière tâche effectue uniquement le calcul de la branche sélectionné par la condition. Si on compare cette implémentation à l'implémentation SIMD d'une conditionnelle, on a remplacé l'exécution consécutive des deux branches prenant un temps $t1 + t2$, par un code prenant un temps $max(t1, t2)$. Il n'y a pas de parallélisme spéculatif et le seul parallélisme utilisé est celui-qui provient des données.

Une tâche doit avoir une date de fin déterminée : dans le cas des conditionelles, cette date de fin est la date de fin de la plus longue des deux branches (la plus courte est complétée par des nops). Dans le cas ou une au moins des deux branche contient une construction dont on ne sait pas borner l'exécution, la deuxième tâche se transforme elle-même en une construction dont on ne sait pas borner le temps d'exécution. Si f est la loi de probabilité du temps d'exécution de la branche en cause, la loi de probabilité de la conditionnelle devient $f(t - \tau)$ où τ représente les calculs additionnels impliqués par le traitement de la condition (généralement τ est très petit devant t puisqu'il s'agit juste d'un saut en fonction de la valeur de la condition, et on peut donc le négliger).

II.6. Ordonnancements contraints

Nous avons modélisé les tâches comme des rectangles sécables verticalement. Cette modélisation repose sur l'idée que le calcul d'un tissu occupe les CPU de manière identique et indifférente au point considéré. Cette idée ne reflète pas l'activité réelle de certaines tâches comme par exemple celles qui doivent implémenter une beta-réduction ou un scan.

II.6.1. Ordonnancement des réductions

C'est pourquoi, afin de ne pas forcer inutilement des PE à être inactifs, il peut être nécessaire de représenter une beta-réduction par un ensemble de tâches plus élémentaires. Une réduction du tissu T peut se voir comme la succession de $\log_2(|T|)$ étapes chaque étape nécessite deux fois moins de PE que l'étape précédente. Il est donc possible de représenter une beta-réduction par $\log_2(|T|)$ tâches (la progression en logarithme du cardinal du nombre de point rend cette approche possible).

II.6.2. Contraintes sur un ordonnancement

De manière plus générale, certaines opérations peuvent avoir un ordonnancement prédéfini, parce qu'il dépend par exemple de particularités architecturales de la machine. C'est pourquoi l'ordonnanceur ROMA accepte des contraintes sur les tâches qu'il doit ordonnancer :

- La dépendance entre tâches peut être point-à-point ou bien totale (il faut alors attendre la fin de la dernière opération avant de commencer la première opération de la tâche qui succède).
- Une tâche peut requérir certains PE nommément désignés
- Une tâche peut requérir les mêmes PE qu'une autre tâche
- Une tâche peut être insécable. Elle correspond alors à une boîte noire représentant un ordonnancement ad-hoc.
- Un groupe de tâches peut être inséparable. Ce groupe de tâches correspond au séquencement ad-hoc de

sous-tâches élémentaires. ROMA voit ce groupe de tâches comme un *trapèze* ce qui permet l'imbrication de tâches externes. La beta-réduction peut par exemple être vue comme trapèze. L'intérêt de la notion de groupe de tâches est que l'ordonnancement des différentes tâches entre elles est fixé.

Ces contraintes sont indiquées par le programmeur par des annotations du programme 81/2. Ces annotations sont attachées à une définition de tissu.

II.7. Ordonnancement des quantifications, des *until*, *after* et *at*

Dans cette section nous voulons examiner le surcoût qui est entraîné par la gestion statique de l'exécution, par rapport au coût d'une exécution dynamique. Nous avons déjà examiné au début de ce chapitre les conséquences de la gestion synchrone du when. Nous aimerions discuter plus avant de l'implémentation des définitions multiples, du until, after et at.

II.7.1. Traitement des définitions quantifiées.

Une définition quantifiée (ou *multiple*) permet généralement de donner une valeur initiale à une définition comportant un opérateur de délai :

$T@0 = 0;$ (1)
$T = \$T + E;$

Ce type d'utilisation peut être détecté par le compilateur à partir de l'examen des horloges. En effet, il est possible de donner une date à l'occurrence du top de « 0 » qui donne sa valeur initiale à T : il s'agit là de détection et de propagation des constantes. Ainsi la définition de T peut se transformer en une seule tâche comportant des initialisations adéquates. Ce n'est pas le cas de l'exemple suivant :

$T@0 = 0$ when *une-condition-compliquée* (2)
$T = \$T + E;$

Dans ce cas il faut générer deux tâches, l'une correspondant au calcul de « 0 when *une-condition-compliquée* » et l'autre recevant cette valeur, calculant « $T + E; » et choisissant entre les deux. Le code de cette dernière tâche est par exemple le suivant :

```
Si arrivé alors
   si e alors $T = $T + E
   émettre $T
sinon
   si T_at_0 != ⊥ alors arrivé ← vraie
   émettre 0, top
   $T = 0
```

T_at_0 correspond à la case mémoire dans laquelle est rangée la valeur de « 0 when *une-condition-compliquée* », $T correspond à l'ancienne valeur de T, e à l'horloge de *E*, E à la valeur de *E*, etc. « arrivé » correspond à un drapeau interne. Après mise à vrai de ce drapeau, il n'y a en fait plus besoin de calculer « 0 when *une-condition-compliquée* » mais ce calcul continuera à se dérouler car il n'est pas possible de prévoir statiquement son inutilité.

Dans le cas d'un schéma d'exécution dynamique, cette tâche aurait reçu un testament qui aurait provoqué son suicide. Cependant, l'utilisation attendue des définitions multiples correspond plutôt au cas (1) qu'au cas (2). De plus l'utilisation d'une approximation plus fine dans le calcul d'horloge permet d'espérer pouvoir détecter ce genre de cas : on peut supposer à bon droit qu'il existe dans un programme seulement un petit ensemble de variable de contrôle qui permettent d'arrêter ou de démarrer des activités (c'est le cas de *une-condition-compliquée* qui correspond au démarrage de T). Avec un treillis adéquat, on peut espérer pouvoir représenter la structure du calcul comme un ensemble limité de segments temporels aux frontières desquelles certains tissus « démarrent » et d'autres sont « gelés ». Il est alors possible de traiter efficacement ces situations (Cf. le traitement de until et de at).

II.7.2. Traitement de after

L'observation de l'exemple :

T@0 ; T = $T + 1 when Clock ;
Z = …
Q = T after (Z> 25) ;

montre que pour calculer la valeur du tissu à un moment donné, il faut généralement avoir calculé la valeur du tissu à des moments antérieurs (à cause des délais). Par suite, l'implémentation statique d'un after n'est pas plus coûteuse que l'implémentation dynamique.

II.7.3. Traitement de until et at

L'opérateur until correspond à une tâche qui n'a plus rien à faire après le déclenchement de sa condition. C'est également le cas du at (le calcul des valeurs précédent le déclenchement du at condition relèvent de la discussion menée à propos de l'opérateur after).

Cette implémentation est indubitablement plus coûteuse que l'implémentation dynamique. Nous invoquerons cependant comme pour les définitions multiples, un *schéma d'utilisation standard* qui peut être plus efficacement traité. L'exemple est le suivant :

T = … until C ;
Q = … after C;

Un couple (until, after) est utilisé pour réaliser un séquencement sur une condition dynamique : un calcul doit se dérouler jusqu'à l'occurrence d'une certaine condition puis un autre calcul doit prendre la place.

Avec ce schéma de programmation, on sait que les calculs de T et de Q sont exclusifs. Il est donc possible de calculer 2 ordonnancements, l'un avec T, l'autre avec Q, et de choisir l'ordonnancement courant à la fin de chaque tic. La viabilité d'une telle solution dépend de la possibilité d'analyse des programmes afin de détecter de telles situations et du coût du changement d'ordonnancement à chaque tic. Dans le cas où la condition correspond à un nom de tissu cette analyse est facile (et cette contrainte peut être incluse dans la grammaire du langage). La deuxième condition (un support matériel permettant d'implémenter efficacement le changement de contexte), existe sur les machines reconfigurables, et en particulier sur PTAH.

II.8. Gestion de la mémoire

L'ordonnancement des programmes 81/2 ne tient pas compte de l'occupation de la mémoire sur chaque PE. On peut donc se demander si la stratégie d'allocation choisie ne risque pas de saturer la mémoire de certains PE.

L'utilisation de la mémoire dépend directement du nombre de variables qui apparaît dans une expression. Le principe de l'ordonnancement est de répartir « au mieux » le calcul des expressions. Il ne peut donc y avoir d'hétérogénéité dans l'usage de la mémoire que si :

- il y a des expressions dont la durée d'exécution est très longue et qui utilisent peu de variables;
- il y a des processeurs qui exécutent beaucoup de petites expressions et d'autres qui n'exécutent que de grandes expressions.

La réunion de ces deux conditions en même temps semble difficile : le nombre de références mémoires faites par un code dépend directement de sa longueur. Et on peut supposer, puisqu'il y a beaucoup plus de tâches que de PE, que celles-ci sont uniformément distribuées sur les PE.

Enfin, si nécessaire, il est toujours possible de rajouter un critère permettant de favoriser lors de

l'ordonnancement, le choix de PE à faible taux d'occupation mémoire. Cette préférence sera faite au détriment du critère « minimisation de la durée d'un tic » mais permettra de répartir plus uniformément l'occupation de la mémoire.

II.9. Compilation pour les architectures SIMD partitionnables

L'architecture cible, dans ce qui précède, est une architecture MIMD synchronisable à communication déterministe. Mais il est tout à fait possible de traiter le cas des architectures SIMD partitionnables. La génération des tâches reste la même, mais on considère en place des PE, les séquenceurs de l'architecture partitionnable. Chaque séquenceur joue le rôle d'un PE avec en plus la possibilité de traiter *n* points d'un tissu simultanément. Une tâche n'est donc plus un rectangle sécable par unité, mais sécable en sous-rectangles dont la longueur correspond au nombre de PE d'un séquenceur. Si le nombre *n* de PE associés à un séquenceur est variable (comme par exemple pour PASM [PASM 81]), alors on utilise une stratégie gloutonne allouant le plus de PE possible à la tâche à ordonnancer. La bonne gestion de la mémoire peut demander une attention particulière, dans le cas où *n* est variable, puisqu'il faut que les variables accédées par le code déroulé sur un séquenceur soient implémentées aux mêmes adresses sur tous les PE de ce séquenceur. Une technique permettant d'éviter la fragmentation est proposée dans [NSDQN 90].

II.10. Le problème de la génération effective du code

La génération effective du code de chaque PE pose un problème quand il y a plusieurs milliers de séquenceurs ou de PE MIMD. Cependant, une fois l'ordonnancement réalisé, et celui-ci est d'une complexité raisonnable comme nous l'avons vu, il est parfaitement possible de dupliquer le résultat symbolique de l'ordonnancement afin de générer les programmes des PE en parallèle.

Les communications entre opérations sont elles aussi représentées sous une forme symbolique dans l'ordonnancement : chaque tâche est découpée en sous-tâches allouées à des PE contiguës sur l'axe des PE. L'adresse (sur l'axe des PE) du PE qui évalue la valeur du point *i*, est donc calculée par une fonction linéaire par morceaux (les morceaux correspondant à chaque sous-tâche de la tâche initiale). Cette fonction doit être composée avec la projection de l'axe des PE dans l'espace des PE réels, pour obtenir une adresse physique.

III. La granularité des tâches

Nous avons déjà évoqué l'intérêt du rapport communication/calcul. En particulier si la communication est coûteuse, il est possible de « cacher » le coût de la communication par des tâches prenant plus de temps de calcul [Inmos 91]. Par ailleurs l'architecture spécifique de la machine peut rendre avantageux certains groupements de calculs (par exemple si chaque PE dispose d'unité MAC). Nous finissons ce chapitre en présentant certaines considérations qui président au choix de la granularité des tâches.

III.1. Le rapport communication / calcul

Jusqu'à présent nous avons considéré, dans ce chapitre, des tâches correspondant au code nécessaire au calcul de la valeur en un tic d'une expression. Ce découpage des calculs à effectuer est un découpage correspondant à un parallélisme à grain fin. En conséquence le nombre de messages échangés par rapport au temps de calcul est grand et par suite l'activité principale de chaque PE peut se réduire à l'attente des valeurs nécessaires aux calculs.

Il est nécessaire de diminuer autant que faire se peut l'inactivité de chaque PE. Il est possible à cette fin de

d'augmenter la quantité de calculs correspondant à une tâche. Ce grossissement des tâches s'accompagne bien sûr d'une perte du parallélisme mais aussi d'une réduction du temps d'attente : les messages transitent dans le réseau pendant que les PE calculent. Le deuxième effet peut compenser le premier.

Il y a trois façons de faire grossir une tâche :

– calculer la valeur de plusieurs tics en même temps (ce qui correspondrait en quelque sorte à un dépliage de boucle);

– calculer la valeur de plusieurs points du même tissu à un tic donné;

– calculer la valeur des points d'un index donné pour plusieurs tissus.

Nous ne retenons pas la première solution car elle va à l'encontre du schéma d'exécution par tics que nous avons retenu. C'est cependant une approche qu'il conviendrait d'examiner avec attention car elle présente des avantages analogues à ceux de la deuxième solution.

III.2. Groupage en largeur

Calculer la valeur de plusieurs points sur le même PE présente le désavantage de rendre séquentiel des calculs a priori parallèles, sans apparemment diminuer la quantité d'informations échangées. Il y a cependant trois avantages possibles. Le premier consiste en l'augmentation du temps de calcul permettant par là de masquer du temps de communication.

Le deuxième avantage possible est la réduction du temps de calculs. Par exemple, si l'architecture interne du PE est pipe-liné, il peut être avantageux d'exécuter une boucle séquentiellement sur un PE plutôt que de la répartir sur plusieurs PE.

Le regroupement de plusieurs tâches parallèles (c'est pourquoi nous qualifions cette opération de groupage *en largeur*) peut aussi présenter l'avantage de diminuer le coût de communication. En effet, le groupement en largeur des tâches *diminue le nombre de messages échangés* entre PE.

Nous avons modélisé le coût de la communication par une formule

$$C = f(x)$$

où x représente la quantité de données à transmettre. Cette formule, où pour des raisons de simplicité on a supposé que le temps de communication ne dépend ni de l'émetteur ni du receveur, est (approximativement) vraie pour l'iPSC. On peut même aller plus loin et supposer que la formule est linéaire avec la quantité de données à transmettre, $C = ax + b$, où b est une constante qui représente le temps d'initialisation de la communication, et a le temps pour transmettre une quantité unitaire de données. Souvent b est plus grand que a. Par exemple pour le Supernode, une machine à base de transputers, $C = 1.1x + 8{,}5 + 9.5(q-1)$, q étant le nombre de liens de communication actifs simultanément (C est exprimé en micro-secondes). Ainsi il vaut parfois mieux minimiser le nombre de messages plutôt que la quantité d'informations échangées [Saltz, Naik, Nicol 87]. La détermination du facteur g de groupage est un problème ouvert. Une heuristique possible est de grouper les points d'un tissu proportionnellement au temps d'inactivité des PE car dans une première approximation, le facteur de groupage g permet de gagner un temps $(g - 1)b$ sur la communication alors que le temps de calcul est multiplié par g.

III.3. Groupage en profondeur

Le groupage en profondeur consiste à grouper des nœuds successifs dans le graphe de dépendances. Cela consiste par exemple à utiliser une seule tâche pour implémenter les deux tissus :

```
T@0 = 1; T = $T + 1 when S ;
U = S / T;
```

Puisqu'on a la dépendance T → U, ce groupement est appelé groupement *en profondeur*. La tâche correspondante doit recevoir la valeur et l'horloge de S ce qui fait une communication par tic. L'implémentation naturelle avec deux tâches requiert la communication de l'horloge et de la valeur de S aux deux tâches chargées de calculer T et U, et la communication de la valeur et de l'horloge de T à la tâche qui calcule U. Cela fait en tout 2 communications. On a donc gagné une communication. Par ailleurs on n'a perdu aucun parallélisme car le calcul de T doit précéder le calcul de U.

Ce fait est général : si on groupe deux expressions consécutives, on élimine au moins une communication (la communication qui fait que ces deux expressions sont reliées) et on ne perd aucun parallélisme. La transformation est facile mais demande à préciser deux paramètres : « combien » faut-il grouper et comment choisir entre les différents groupements possibles.

Le choix entre les différents groupements possibles peut se faire à partir d'une analyse de chemin critique : on regroupe les expressions qui sont sur le chemin critique. Cette heuristique se base sur le fait que les performances de l'ordonnancement total sont fortement conditionnées par les performances de l'ordonnancement de ces expressions là. En effet, le délai apporté par les communications à l'évaluation d'une expression non-critique est, dans une certaine mesure, absorbé par le temps d'exécution total du graphe. Par contre le retard d'exécution d'une expression critique est propagé le long du chemin critique et augmente le temps d'exécution global. Il est donc nécessaire de réduire autant que possible les délais de communication des expressions critiques

Implicitement nous avons supposé qu'une tâche disposait de toutes ses données au moment de son activation, et qu'elle émettait ses valeurs à la fin de son exécution. Il faut, à présent que les tâches peuvent devenir importantes, affiner ce modèle : une donnée peut arriver pendant l'exécution d'une tâche à condition qu'elle arrive avant son utilisation effective. Et une donnée peut être émise avant la fin d'une tâche. Cela complique le travail de l'ordonnanceur qui doit gérer plusieurs dates d'utilisation et plusieurs dates d'émissions par tâches. Par contre cela permet des regroupements de longueur arbitraires, sans introduire des délais de communication artificiels.

La stratégie de découpage des tâches peut donc consister à calculer les expressions critiques du graphe de dépendances. Le calcul de ces expressions constituera le travail des *n* premières tâches (une tâche par point tissu, le regroupement des expressions se faisant tant qu'elles ont la même géométrie). Le processus est répété jusqu'à exhaustion des nœuds.

X. 8,5 : la plate-forme $8_{1/2}$

Ce chapitre décrit le système **8,5** qui permet de développer interactivement des programmes 81/2 et de les compiler. Les desiderata pour une telle plate-forme et l'état courant de la maquette sont exposés.

Le développement interactif des programmes 81/2 implique en particulier de pouvoir évaluer au vol n'importe quelle expression. La plate-forme 8,5 est cependant plus proche d'un compilateur que d'un interprète. En effet, les phases d'analyses du compilateur sont effectivement réalisées par 8,5 et l'évaluation d'une expression conduit à la génération et à l'exécution au vol d'un macro-code correspondant à la gestion des collections. La phase d'interprétation est réduite et le modèle d'exécution adopté est le modèle statique simplifié au cas mono-processeur. La plate-forme *version 1* traite un sous-ensemble du langage 81/2 qui correspond aux tissus plats sans récursion spatiale. Elle est écrite en C++ et comporte environ 25000 lignes de code.

I. Une plate-forme de compilation

Pour qui a utilisé les premières cartes d'évaluation du transputer (1985), la découverte de l'environnement de programmation accompagnant la Connection-Machine a été un véritable soulagement. Le programmeur dispose en effet d'un interprète *LISP avec tous les avantages de ce mode d'exécution des programmes : exécution pas à pas, traceur, visualisation au vol de la valeur des variables, cycle correction/évaluation raccourci, etc. Cet environnement comprend de plus un simulateur *complet* permettant de se passer de la CM. Cet environnement sert aussi de *plate-forme* de compilation : la compilation d'un programme *LISP est complètement intégrée à l'interprète du langage.

Cette expérience nous a convaincu que, plus un langage était « exotique », plus il était nécessaire de fournir un environnement confortable et qui permette une expérimentation aisée. Par ailleurs la compilation d'un langage pour une machine parallèle doit répondre à des questions qui ne se posent pas dans le cas d'une machine séquentielle : exhiber, caractériser et quantifier le parallélisme présent dans un programme, l'exploiter à travers l'ordonnancement des tâches et la gestion de la distribution des données, réaliser les synchronisations nécessaires, etc. Ces activités requièrent une grande quantité d'informations concernant les caractéristiques architecturales du calculateur cible, les capacités du système d'exploitation et le profil typique d'exécution du programme. En conséquence, l'interaction entre le programmeur et le compilateur est beaucoup plus poussée que dans le cas de la programmation séquentielle.

Le contrôle du système *Parafrase* [KKLW 84] peut par exemple contenir plus d'une centaine de commandes, chaque commande correspondant à la spécification d'une passe de compilation et de ses paramètres[1].

Ces deux raisons nous ont conduit à développer un interprète séquentiel servant de plate-forme au compilateur. L'interprète 81/2 intègre les différentes fonctions que l'on attend d'un environnement d'expérimentation et de programmation d'un langage parallèle (Cf. aussi [Zima 91, *chap. 8*]) :

Interprétation

Toute expression 81/2 doit pouvoir être évaluée dynamiquement. L'évaluation d'une expression consiste à calculer successivement la suite de ses valeurs. Une expression peut faire référence à des variables *outside*. Dans ce cas, l'évaluation consiste à demander interactivement à l'utilisateur la valeur du tissu à chaque top. La définition attachée à une variable doit pouvoir être modifiée à tout moment. L'ensemble des définitions impliquées par une expression doit pouvoir être sauvé dans un fichier. Inversement, les définitions stockées dans un ou plusieurs fichiers doivent pouvoir être internées dans l'interprète. Enfin, des langages comme LISP et PROLOG, qui reposent essentiellement sur des interprètes, ont popularisé les nécessaires facilités de traces et d'exécution pas à pas.

Analyse statique

L'analyse statique permet d'établir la correction d'un programme : détermination de la géométrie, détermination d'un séquencement statique, détection (partielle) des famines et des étreintes fatales. Les différents outils d'analyse statiques doivent être accessibles en-ligne. Le résultat des analyses doit être conservé afin de pouvoir être réutilisé lors des phases de transformation de programme et de génération de code.

Outils de visualisation et d'édition

Diverses commandes doivent permettre d'inspecter les définitions attachées aux variables. L'acquisition des informations ne se réduit cependant pas à la visualisation textuelle des expressions 81/2. L'information produite par les analyses statiques peut se représenter synthétiquement par des graphes de dépendances. Le placement des données et l'ordonnancement des tâches font appel à des cartes colorées et des diagrammes de Gantt. La suite des valeurs produites par une exécution se visualise avantageusement de manière graphique à l'aide de graphomètres (affichage de courbes de valeurs en fonction du temps), d'histogrammes ou encore d'afficheurs analogiques.

Environnement d'exécution

L'application privilégiée de 81/2 est la simulation des systèmes dynamiques discrets. Ces applications requièrent souvent des quantités très importantes d'entrée/sortie. Si les sorties d'un programme peuvent se représenter de manière très synthétique à l'aide de représentations graphiques, il n'en va pas de même avec les entrées. Les entrées d'un programme 81/2 correspondent à la spécification de la trajectoire des tissus « outside ». Cette trajectoire doit pouvoir être enregistrée dans un fichier (afin de pouvoir être réutilisée), être calculée par une fonction ou un processus concurrent, ou encore provenir d'une interaction directe avec l'utilisateur. F. Legrand conclut à la fin de son stage [Legrand 91] que les facilités d'entrée/sortie sont un point fondamental pour l'utilisabilité de 81/2.

Transformations de programme

Les transformations de programme sont des outils de structuration permettant de normaliser la forme d'un programme en vue d'une amélioration des performances du code généré. La normalisation peut être [KKLW

[1] Parafrase est un système qui transforme un programme FORTRAN77 en un programme équivalent en FORTRAN étendu correspondant à un type de machine. Les architectures cibles comprennent les machines vectorielles registre/registre et mémoire/mémoire, les multi-processeurs à mémoire partagée et les machines cellulaires (array-processors).

84] : indépendante de l'architecture cible (propagation des constantes, resynchronisation, etc), spécifique à une classe d'architecture (par exemple vectorisation ou parallélisation) ou spécifique à un calculateur donné (allocation des registres, ordonnancement des instructions, …).

Génération de code

L'utilisateur doit pouvoir décrire les caractéristiques connues du programme (nombre moyen d'itérations attendues pour un « every », ratio valeurs vraies/valeurs fausses des expressions booléennes, intervalle de variation de la valeur des variables, …) et de l'architecture cible (nombre de PE, moyen de synchronisation, modèle de coût des communications, …). Il doit aussi être capable de forcer le schéma d'exécution (séquentiel, vectoriel, parallèle) d'un tissu donné. À partir de ces informations, le compilateur doit générer le code correspondant à chaque PE.

Simulation et analyse de l'exécution

L'analyse de l'exécution d'un programme doit permettre d'examiner le flot des communications et les taux de charge des processeurs afin de pouvoir améliorer le comportement du programme.

Cette analyse de l'exécution peut se faire à partir d'un simulateur de la machine cible ou bien par espionnage du calculateur pendant une exécution réelle. Le code généré doit éventuellement comporter les instructions nécessaires pour la mesure de paramètres inconnus du programmeur et permettant d'améliorer le code produit lors d'une future compilation (à la manière du trace-scheduling). Il faut aussi pouvoir insérer le code permettant de vérifier au cours de l'exécution, la validité des hypothèses faites lors de la transformation du programme et de la génération de code.

Bibliothèques

Le programmeur doit disposer d'une bibliothèque de programmes standards : algèbre linéaire, solveurs, tris, routines de communications dynamiques, comportements standards, etc.

Gestion de projet

Les programmes, leurs compilations et les traces d'exécution doivent pouvoir être documentés, regroupés en module et être archivés.

II. La plate-forme 8$_{1/2}$ *version 1*

La description précédente énumère les éléments d'un environnement idéal. L'état d'avancement courant de la maquette de la plate-forme 81/2 (version 1) est détaillé dans les paragraphes qui suivent. On trouvera aussi des détails d'implémentation dans [Legrand 91]. Le langage C++ [Stroustrup 86] a été choisi comme langage de développement car il permet un style de programmation orienté objet, une gestion fine de la mémoire et une exécution compilée. Il faut noter que **la version 1 est restreinte aux tissus plats *et* aux blocs**. Cela représente néanmoins un travail non négligeable, ainsi que le montre le tableau 1 ci-dessous (les chiffres sont approximatifs).

fonctions	*lignes de code*	*classes C++*
lecteur		lex : moins de 100 tokens yacc ≈ 140 règles non terminales
écrivain	4000	7
représentation des expressions	5000	40 classes dérivant d'une classe abstraite définissant 50 méthodes de services qui seront héritées et 20 méthodes virtuelles qui seront redéfinies par les sous-classes
expansion des fonctions	1300	
gestion de la géométrie	3000	
typages scalaires	1300	
détection des cycles	1500	
ordonnancement mono-PE	1300	1
évaluation (macro-code collection)	1600	3
ordonnancement multi-PE (ROMA, *en cours*)	*2000*	*4*
total	25000	55

Table 1 : Nombre de classes, de méthodes et de lignes de codes de chaque fonction de la plate-forme.

II.1. L'interface

L'interprète 81/2 se lance à partir d'un shell UNIX à l'aide de la commande « **8,5** ».

II.1.1. Le lecteur

Le *lecteur* est le gestionnaire des entrées 81/2. Ces entrées proviennent du clavier ou d'un fichier. Elles peuvent être de trois types :

- une expression 81/2;
- une commande;
- la spécification de la valeur d'un tissu externe.

Au passage, le lecteur gère les inclusions multiples de fichiers et les annotations laissées par cpp (cpp est le préprocesseur C; il est possible d'appliquer une passe de pré-traitement à un programme 81/2 quand on utilise la plate-forme en batch). Le code du lecteur est basé sur deux automates générés par FLEX (une version de LEX) et YACC.

Une grammaire « yaccable » du langage 81/2 a été développée. La principale difficulté a été de spécifier une grammaire supportant une entrée interactive (entre autre, la terminaison d'une définition doit être détectée sans lookahead ce qui motive l'utilisation du « ; » comme terminateur).

II.1.2. L'écrivain

L'*écrivain* est chargé de gérer les sorties de la plate-forme. Ces sorties correspondent à des résultats des commandes de l'utilisateur. Le flot de sortie standard correspond au shell de lancement de la plate-forme, mais peut être dupliqué en cours de session vers un fichier.

II.1.3. Les commandes

Les commandes actuellement disponibles sous l'interprète sont les suivantes :

QUIT **^D**	quitte la session courante (**^D** représente le signal UNIX SIGQUIT émis par le shell sur une commande clavier contrôle-D).
^C	interrompt l'interprète et reprend au top-level.
LOAD ***file***	le contenu du fichier *file* est inséré dans le flot des entrées.
STORE [*id***]** ***file***	sauvegarde la définition du tissu *id* (par défaut de tous les tissus définis dans l'interprète) dans le fichier *file*. Les tissus nécessaires pour que la définition soit complète, seront sauvés aussi.
? ***id*** **??** ***id*** **? all**	permet d'afficher la définition du tissu *id* (commande « ? »), et aussi de tous les tissus s'y rapportant (commande « ?? ») ou bien de tous les tissus qui sont définis dans l'interprète (commande « ? all »).
SAVE [on|off] ***file***	permet de dupliquer les sorties dans un fichier *file*.
CLEAR [*id***]**	permet d'oublier une définition (par défaut, toutes les définitions).
CHECK ***id***	effectue les analyses statiques sur le tissu *id* et sur les tissus afférents.
EVAL ***id*** **[***n***]**	effectue les analyses statiques sur le tissu *id* et ceux qui s'y rapportent puis évalue *id* jusqu'au tic *n* (par défaut, pour toujours).
COMPIL ***id***	génère le code C correspondant à une compilation séquentielle de *id* (commande non complètement implémentée au 1er Septembre 1991).
SH ***string***	permet de lancer une commande shell.
VERIFY … **DEBUG …**	commandes de debug permettant la vérification d'assertions ou l'affichage de traces lors des phases d'analyse et d'évaluation.

II.2. Représentation des expressions

L'intégration des différentes fonctions dans l'interprète se fait à travers une représentation commune des expressions 81/2. Une expression 81/2 correspond à un graphe construit à partir de l'arbre syntaxique abstrait obtenu par le parsing de l'expression. Ce graphe est implémenté à travers un ensemble de classes C++.

II.2.1. La forme interne des expressions

Un nœud du graphe des expressions correspond à une instance d'une classe dérivant de la classe **Web**. La classe Web est une classe abstraite dont les descendants sont des implémentations (classe *virtuelle* dans la terminologie C++). Les méthodes virtuelles déclarées sur Web sont généralement définies sur les feuilles de la hiérarchie des classes.

Par exemple, les trois premiers descendants de Web correspondent aux déclarations, aux définitions, et aux (autres types d') expressions. Ces classes sont elles aussi des classes abstraites qui se raffinent à leur tour. On trouvera dans [Legrand 91] une description de la hiérarchie.

La liaison identificateur/définition se fait à l'aide de tables de hash-code. Ces tables sont associées aux expressions correspondant à des blocs et représentent un scope. Avant les phases d'analyses et d'évaluation, ces liens sont résolus et remplacés par des pointeurs. Ce chaînage est transparent à l'utilisateur et intervient systématiquement

après une modification des définitions. En conséquence, un utilisateur peut à tout moment redéfinir des tissus.

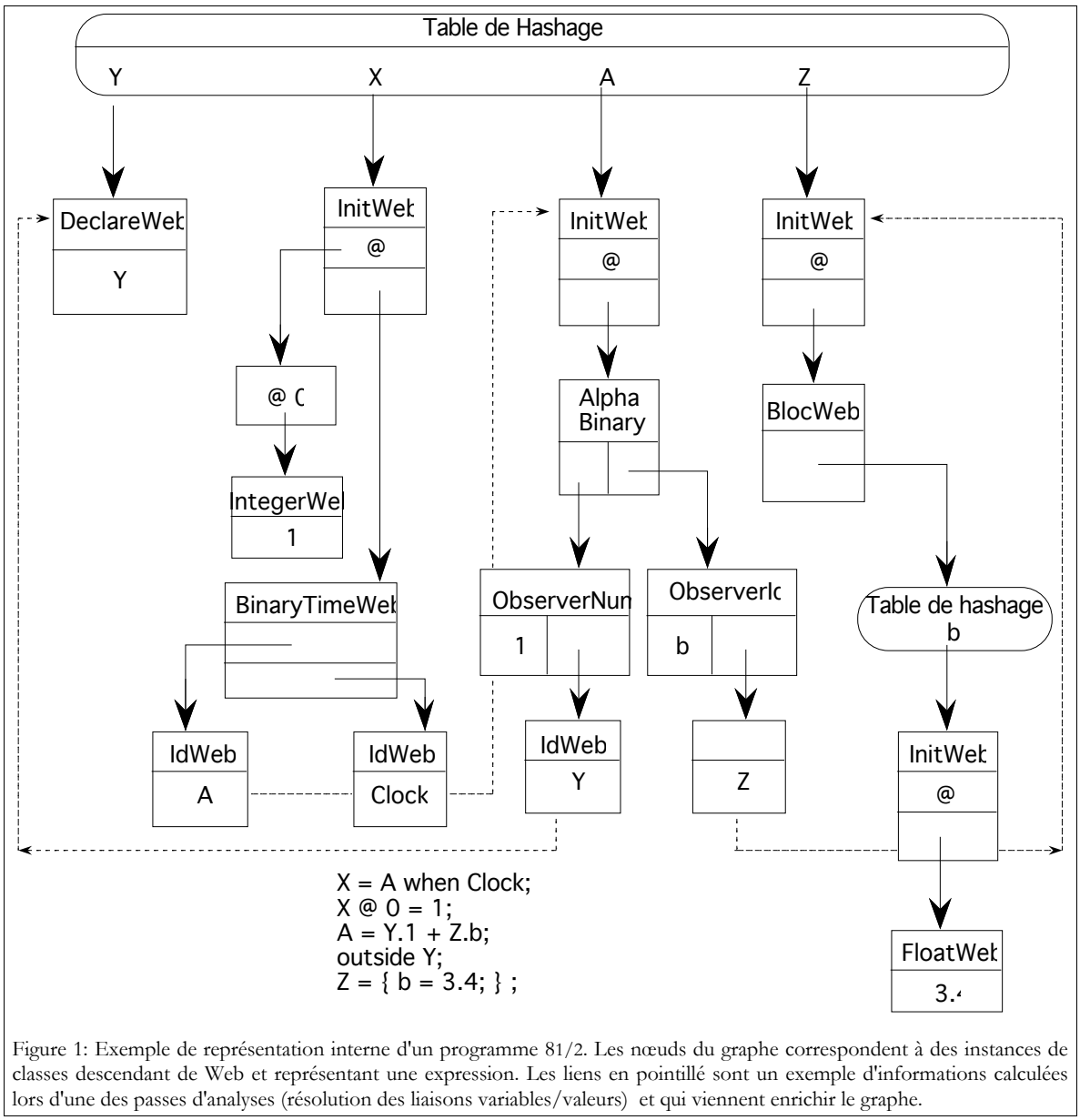


Figure 1: Exemple de représentation interne d'un programme 81/2. Les nœuds du graphe correspondent à des instances de classes descendant de Web et représentant une expression. Les liens en pointillé sont un exemple d'informations calculées lors d'une des passes d'analyses (résolution des liaisons variables/valeurs) et qui viennent enrichir le graphe.

La figure 1 illustre de manière simplifiée la structure de données correspondant à la représentation de l'ensemble d'équations suivant :

```
int outside Y [10];
X@0 = 0;  X = A when CLOCK
A = Y.1 + Z.b;
Z = { b = 3.4; };
```

II.2.2. Vers une représentation externe des expressions

La représentation des expressions par un ensemble de classe C++ est une représentation interne à l'interprète. Dans l'état actuel, toute fonction nouvellement développée doit s'intégrer dans l'interprète, avec les avantages et les

inconvénients de cette approche. On peut noter entre autres que :

- La taille de l'exécutable s'accroît avec les fonctions réalisées;
- Il est très difficile d'intégrer des fonctions demandant une interaction graphique complexe. En effet, la boucle de gestion des événements graphiques est difficilement intégrable à la boucle de lecture/évaluation/écriture de l'interprète.
- Le travail réalisé sur un ensemble d'expressions (par exemple l'analyse statique) est perdu à la sortie de l'interprète.

Ces inconvénients motivent le développement d'une représentation externe permettant l'échange d'informations entre des « outils » travaillant sur les expressions 81/2.

PRESTO (acronyme approximatif de : PeRsistent Elementary Swappable and Tiny Objects) est un système de gestion d'objets persistants [GDR 88]. PRESTO permet de créer des classes d'objets rémanents. Ces objets ont un identificateur unique qui permet d'y accéder d'une invocation de programme à une autre. Le système est transparent au programmeur C++ qui définit et utilise les classes rémanentes comme n'importe quelles autres classes C++ (le système se restreint cependant aux hiérarchies de classes qui sont des arbres, i.e. à l'héritage simple).

PRESTO a été utilisé pour supporter le système d'information d'A*dage* [GDRM 90]. Le modèle de données utilisé est basé sur des graphes attribués typés. Il est donc certain que l'approche adoptée par PRESTO est tout à fait utilisable pour représenter les graphes des expressions 81/2 et les annotations calculées par les différents outils. Cette approche présente de plus l'avantage de conserver la représentation adoptée par les fonctions existantes. C'est pourquoi dans une version ultérieure, nous nous proposons, grâce à PRESTO, de rendre rémanentes les classes utilisées dans la représentation des expressions.

II.3. Analyses statiques

Les analyses statiques qui ont été implémentées dans la version courante du prototype sont des simplifications des techniques qui ont été exposées dans les chapitres IV et V.

II.3.1. Détermination du type scalaire

Nous n'avons pas parlé jusqu'à présent de la détermination du type des valeurs des feuilles d'un tissu. Nous appellerons ce type, *type scalaire* d'un tissu. Le type scalaire d'un tissu peut être omis. Dans ce cas il sera inféré par 8,5. 8,5 doit aussi inférer le type de chaque expression afin d'insérer les conversions nécessaires (par exemple dans « int x = …; z = x + 3,14159 », la valeur de x doit être convertie en flottant lors du calcul de z).

Dans la version courante, un type scalaire appartient à {entier, flottant, booléen}. Les deux conversions suivantes sont possibles :

entier → booléen *et* flottant ↔ entier

L'algorithme utilisé est un algorithme ad-hoc. Il n'est pas capable d'inférer dans tous les cas de figure le type d'un tissu quand ceux-ci sont implicites. Mais il couvre une très grande part des cas rencontrés et n'est pas difficile à mettre en œuvre. De plus, si chaque variable est typée, on est assuré de pouvoir calculer le type de toute expression (comme en C par exemple).

L'algorithme consiste à propager les types connus dans le graphe des expressions afin d'attribuer un type à chaque nœud. Le type scalaire d'une expression est connu si :

- il est explicitement déclaré;
- l'expression est le résultat de certains opérateurs primitifs (par exemple « 'A » est obligatoirement un tissu

d'*entiers*);

– le type des antécédents du nœud sont connus.

La passe de propagation calcule le type de chaque expression à partir du type des arguments connus. Par exemple le type de « A when B » est le type de A. Cette propagation *en avant* ne suffit pas à inférer le type de chaque expression puisque le type des variables n'est pas obligatoirement explicite. La propagation *en avant* se double donc d'une propagation *en arrière* permettant de propager des contraintes : par exemple, à partir de « A when B » on peut propager sur B le fait que B est entier (tout entier est aussi un booléen, mais un booléen n'est pas un entier : le type le plus général pour B est donc le type entier). Au cours de cette phase de propagation, on vérifie que le typage de chaque expression est admissible (on ne peut avoir *entier* + *booléen*). À la fin de cette passe, on vérifie que chaque variable possède un type. Si ce n'est pas le cas, la variable non typé prend par défaut le type entier et on propage les conséquences. On itère jusqu'à ce que chaque variable ait un type.

II.3.2. Détermination de la géométrie d'un tissu

La méthode utilisée pour calculer la géométrie d'un tissu dérive de la technique proposée au chapitre IV, restreinte aux cas des tissus plats (pas de dimension à calculer, seulement le cardinal de chaque tissu). Une première passe est effectuée pour trouver les sommets du graphe des expressions qui possèdent une géométrie connue. Cette géométrie provient soit d'une indication explicite du programmeur, soit résulte d'un opérateur prédéfini (par exemple « ' », CLOCK, etc). Ces géométries connues sont ensuite propagées.en fonction des sommets rencontrés. Au cours de cette propagation, des erreurs sont éventuellement détectées. À la fin des propagations il peut rester des sommets sans géométrie (cas d'un système avec une infinité de solutions). Dans ce cas, on affecte la géométrie « [1] » a un sommet et on recommence la passe de propagation. Cette stratégie correspond à donner par défaut une géométrie de scalaire.

II.3.3. Détection des dépendances temporelles

Le calcul de l'ordonnancement statique est une version simplifiée de l'algorithme présenté dans le chapitre V. En effet, les cas de récursion spatiale ne sont actuellement pas traités. Le problème se réduit donc à détecter dans le graphe des expressions les cycles qui ne comportent pas d'opérateur de délai. Certains autres opérateurs jouent un rôle similaire à l'opérateur de délai en inhibant un cycle : ce sont la quote et le cardinal. En effet, des expressions comme

T [10] = 'T ; *ou* U = |U| ;

ne sont pas des expressions instantanées, puisque 'T et |U| sont en fait des constantes calculées à la compilation.

II.3.4. Autres traitements

Nous ne ferons qu'évoquer certaines passes de compilation qui se traduisent par des parcours plus ou moins complexes dans le graphe des expressions :

Inlining

les applications de fonctions sont substituées par les expressions correspondantes. C'est à ce moment que sont détectées d'éventuelles fonctions récursives.

Résolution des références

C'est dans cette phase que la définition correspondant à chaque variable est retrouvée. En cas de définition manquante, le tissu est automatiquement promu au rang de tissu externe.

Initialisation/chaînage

Afin d'accélérer le traitement de certains algorithmes, des redondances sont créées dans la représentation des expressions. En particulier certains chaînages sont doublés d'un lien arrière afin de simplifier les parcours du graphe. Ces informations redondantes deviennent obsolètes chaque fois qu'il y a une modification du graphe des expressions. Il faut donc aussi gérer des phases de réinitialisation.

II.4. Interprétation et compilation

II.4.1. Évaluation séquentielle

L'évaluation d'une expression 81/2 consiste à déterminer pour chaque tic :

– l'horloge de l'expression (est-ce un tic ou un top)

– la valeur de l'expression (uniquement si c'est un top).

Le calcul de la valeur d'une expression se fait à l'aide d'un *macro-code* permettant de manipuler des collections. Ce macro-code se présente sous la forme d'un *code trois adresses* :

destination ← *op* ($source_1$, $source_2$)

La destination et les arguments correspondent à des collections : ce sont des zones mémoires qui ont été réservées à l'avance. Éventuellement il y a 2 ou 4 arguments dans le cas des opérateurs monadiques (par exemple la négation) et triadiques (la conditionnelle).

Nous avons vu dans les chapitres précédents comment générer l'horloge d'une expression et ordonner statiquement les calculs. On utilise alors un tri topologique pour ordonner complètement les expressions dans un tableau. L'évaluation séquentielle consiste simplement à itérer sur les éléments de ce tableau. Une itération correspond aux calculs des valeurs d'un tissu pour 1 tic :

```
TabExpr [n] = …  le tableau ordonné des expressions
pour toujours
   pour i = 0 à n            calcul des valeurs des tissus pour un tic
        e ← TabExpr[i]
        hor = horloge(e)
        si hor alors valeur(e)
```

La fonction `valeur()` correspond à une macro-opération. La fonction `horloge()` correspond au calcul d'une expression booléenne. Ce schéma d'exécution est statique car l'ordonnancement des calculs est déterminé avant l'évaluation. L'argument des fonctions `valeur()` et `horloge()` sert uniquement d'index (i.e. il sert à sélectionner une expression d'horloge et une macro-opération).

II.4.2. Compilation pour les calculateurs séquentiels et SIMD

La compilation d'un programme 81/2 pour une machine séquentielle consiste à générer le code de cette boucle plutôt que de l'exécuter au vol en mémoire. Les phases de génération du code séquentiel seront donc disponibles très prochainement (début 92).

La compilation pour une architecture SIMD ne pose aucun problème supplémentaire : il suffit de remplacer le macro-code SISD permettant de manipuler séquentiellement les collections par un macro-code SIMD permettant de manipuler les collections en parallèle. C'est pourquoi nous prévoyons la génération de code pour Connection-Machine dans les mêmes délais que pour le cas séquentiel.

II.4.3. Compilation pour les architectures MIMD synchronisables

L'exploitation à la fois du parallélisme de données et du parallélisme de contrôle qui sont exprimables 81/2 ne pourra se faire que sur un calculateur MSIMD ou sur un ordinateur MIMD synchronisable. C'est le cas de l'architecture PTAH développée dans le cadre de l'équipe « Architecture des ordinateurs et conception des circuits intégrés » du LRI [Cappello, Béchennec 91]. Dans le cadre de ce projet, un simulateur complet devrait être prochainement disponible ce qui nous permettra d'émuler l'exécution de programme 81/2 sur PTAH. Certaines des fonctions spécifiques à la compilation pour cible MIMD ont d'ailleurs déjà été implémenté, comme par exemple les prémisses de l'ordonnanceur ROMA.

XI. Exemple d'optimisations

Nous avons évoqué dans le premier chapitre les possibilités offertes par le cadre data-flow pour raisonner sur les programmes. Nous voudrions pour achever cette étude du langage 81/2, illustrer cette capacité en présentant deux techniques d'optimisation du code généré.

La première technique est fondée sur une méthode d'optimisation développée par Leiserson et Saxe pour les circuits VLSI synchrones. Elle consiste à transformer le programme afin de raccourcir la longueur du chemin critique. La deuxième optimisation est analogue à la propagation de constantes : à partir des techniques présentées dans le chapitre VII, il est possible de détecter les expressions qui n'ont qu'un seul top en 0. La valeur de ces expressions est calculable à la compilation.

I. Minimisation du chemin critique par transformations de programme

I.1. Introduction

Nous avons vu dans le chapitre précédent l'ordonnancement statique des programmes 81/2. Nous avons mentionné que les ordonnancements répétitifs complètement statiques n'étaient pas optimaux. Parhi et Messerschmitt proposent une technique intéressante permettant d'obtenir théoriquement un ordonnancement optimal. Cette technique consiste à transformer le programme de départ en un programme sémantiquement équivalent qui sera plus efficacement ordonnancé (si l'on dispose d'un nombre non borné de PE).

La transformation de programme utilisée a pour but de mettre les programmes sous une « forme normale » (un seul opérateur de délai dans chaque boucle du graphe data-flow du programme). Cette forme correspond au critère de *systolicité* donné par C. Leiserson et J. Saxe dans [Leiserson, Saxe 83]. Ces auteurs explorent l'effet du *retiming* sur la période de l'horloge d'un circuit VLSI synchrone (la période de l'horloge est l'analogue de la période temporelle d'un programme data-flow). Le retiming consiste à déplacer les registres dans le circuit VLSI (un registre correspond à un opérateur de délai).

Le but du jeu est le suivant : il faut déplacer les registres de façon à raccourcir le chemin critique tout en préservant la sémantique du circuit. En effet, lors de l'ordonnancement, un arc comportant un registre n'intervient pas dans le graphe de dépendances des calculs. La période de l'horloge du circuit est donné par la longueur du chemin critique. En déplaçant les registres, il est possible de minimiser la longueur de ces chemins. Les auteurs proposent une méthode permettant de déterminer l'emplacement des registres [Leiserson, Saxe 83] [Leiserson, Rose, Saxe 83]. La systolicité d'un circuit est un critère permettant d'affirmer qu'un circuit a une horloge minimale (c'est un critère suffisant).

I.2. Comment déplacer les délais

Plaçons nous dans le cadre d'un programme data-flow. La transformation de programme qui permet de déplacer les délais est la suivante : il est possible sous certaines conditions de déplacer les délais qui sont sur les entrées d'un nœud à la sortie de ce nœud. Donnons un exemple :

```
a = $ …
B = a + $C;
C = a;
```

peut se réécrire en

```
A = …
B = $A + $C;
C = $A;
```

Pour comprendre pourquoi il suffit de partir du deuxième programme, de poser a = $A et d'utiliser la transparence référentielle qui permet de remplacer une variable par sa définition.

Si nous regardons les dépendances du deuxième programme, nous voyons qu'on peut mener en parallèle le calcul des valeurs de A, de B et de C alors que le calcul de a doit précéder celui de B et de C dans le cas du premier programme (par contre le calcul des horloges de B et de C doit quand même suivre celui de A).

Il faut donc exhiber des définitions du type « a = $ … » afin de pouvoir transformer les programmes. Une définition est généralement de la forme :

```
a = f(x, y, z, …);
```

On peut la mettre sous la forme

```
a = $f(X, Y, Z);
```

si on a les deux propriétés suivantes

a) x = $X, y = $Y, z = $Z, …

b) f est *distributive* par rapport au délai, i.e. f($X, $Y, $Z, …) = $f(X, Y, Z, …)

Intéressons nous plus précisément à cette propriété de distributivité : existe-t'il des fonctions 81/2 qui ne soit pas distributives ? La réponse est clairement oui :

```
f(X) = 0;
f($…) = 0  ≠  $(f(…)) = $0 = NIL
```

Il nous faut donc un moyen pour déterminer les fonctions qui sont distributives. Cela se fait à partir des propriétés suivantes :

– La composée de deux fonctions distributives est distributive. Si f est une fonction à n variables et si g est une fonction à m variables, les n fonctions h_i à n+p variables définies par

$h_i(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+p}) = f(x_1, \ldots, x_{i-1}, g(x_{n+1}, \ldots, x_{n+p}), x_{i+1}, \ldots, x_n)$

sont distributives. Si f(x, y) est distributive, g(y, x) = f(x, y) est distributive. Et ainsi de suite…

– Les fonctions spatiales sont distributives (i.e. les fonctions qui s'appliquent élément par élément sur l'axe temporel).

– La fonction délai est distributive.

– La fonction « f(X) = X when Clock » est distributive (ce n'est pas le cas de f(x, y) = x when y).

Ces propriétés permettent de calculer une condition suffisante pour qu'une fonction soit distributive. Cette condition peut se calculer par exemple à l'aide d'une grammaire attribuée.

I.3. Un exemple

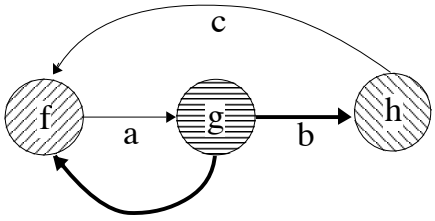
On trouvera dans [Leiserson, Saxe 83] et [Leiserson, Rose, Saxe 83] un critère pour savoir où et combien de registres il faut déplacer dans un circuit VLSI synchrone. Ce critère ne peut pas être utilisé directement dans le cadre de 81/2 car :

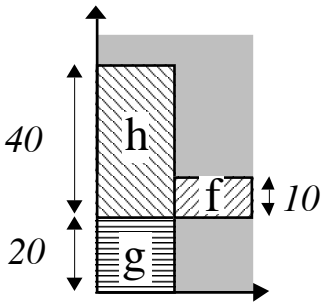
– Dans un circuit VLSI synchrone tous les éléments sont distributifs, ce n'est pas le cas de 81/2.

– La méthode est développée afin de préserver la suite de valeur arrivant à un élément distingué du circuit, le *host*. Dans le cas de 81/2, il peut y avoir plusieurs hosts (correspondant aux tissus outside et aux tissus dont on observe l'évolution).

Notre but n'est pas d'adapter complètement la méthode, mais d'illustrer les effets du retiming, ce que nous ferons sur un exemple. Soit le programme :

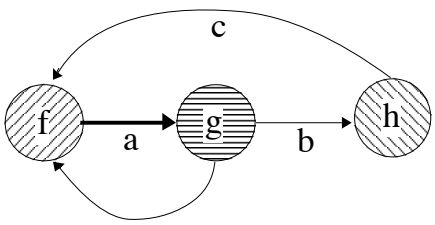
```
a = f($c, b) when Clock;
b = g($a);
c@0 = c0; c = h(b);
```

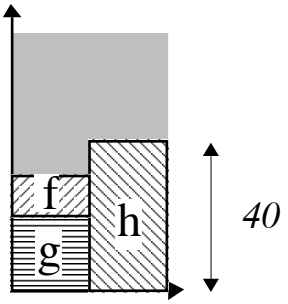




On suppose que f, g et h sont distributives. Nous avons figuré à coté du programme un graphe data-flow simplifié. Les arcs fins correspondent à des délais. À droite on a figuré un ordonnancement possible. On a supposé que le temps de calculs de f est de 10, celui de h, 40 et celui de g, 20; on néglige les temps de communication. On peut réécrire ce programme ainsi :

```
a@0 = f(c0, b);
a = f($c, $b′) when Clock;
b′ = g(a);
c = h($b′);
```





On remarque que le programme transformé à une période temporelle de 40 (contre 60 pour le premier). Il faut bien remarquer que nous n'avons pas tenu compte des délais de communications.

I.4. Extension

Si une fonction est distributive, alors on peut déplacer les délais sur les variables de l'entrée à la sortie de la fonction et vice-versa. Toutes les fonctions n'étant pas distributives, on peut être amené à calculer des formes équivalentes de façons à étendre le domaine des transformations. Par exemple, soit f une fonction distributive à deux variables et une fonction g définie par :

g(x) = f(a, x);

pour un a donné. La fonction g n'est pas en général distributive : g($x) = f(a, $x) ≠ $f(a, x) = $g(x). Par contre on peut utiliser la règle de transformation :

g($x) → $h(x) avec h définit pat h(x) = f(b, x) et a = $b

Une autre transformation utile permet de manipuler les définitions multiples (cette transformation n'est pas un retiming mais est utile pour déplacer des délais) :

```
        b@0 = b0;
   b = …;
a@0 = a0; a = f(b);
```

→

```
            b@0 = b0;
      b = …;
   a′ = f(b);
c@0 = g(a0); c = g(a′);
```

Cette transformation permet d'éliminer la définition multiple de la variable a (mais l'introduit sur la variable c) ce qui permettra par exemple la propagation ou l'élimination d'un retiming.

II. Propagation des constantes

La propagation de constantes est une technique qui permet de détecter certaines expressions dont le calcul peut être effectué à la compilation (Cf. [Zima 91]). Dans le cadre de 81/2 nous voulons déterminer les expressions du type

5 + 2

qui peuvent être calculées à la compilation, ainsi que les expressions du type

x when Clock

qui bien que n'étant pas nécessairement à valeur constante, correspondent à une horloge constante à vraie (i.e. il n'y a pas de calcul d'horloge à implémenter, on sait à la compilation que le calcul de valeur doit se faire à chaque tic).

Pour détecter des expressions de ces deux types, on se place dans le cadre défini par le chapitre VI : il faut déterminer les horloges {0} et I N. Le calcul de l'horloge d'une expression correspond à la recherche d'un point fixe dans le treillis ϒϒI N. Ce calcul peut se faire par itération à partir de ∅, plus petit élément de ϒϒI N, mais l'itération peut très bien ne pas aboutir (i.e. le point fixe n'est pas atteint en un nombre fini d'itérations).

Nous reprenons l'idée esquissée à la fin du chapitre VI et qui est d'exhiber un treillis plus simple qui soit une approximation de ϒϒI N mais dans lequel on trouve nécessairement un point fixe en un nombre fini d'itérations.

II.1. Analyse sémantique approchée

Un treillis complet (E, ≤) est un ensemble ordonné tel que toute partie possède une borne inférieure et une

borne supérieure. Si A $\subseteq$ E, on note $\cup$A la borne supérieure de A et $\cap$A sa borne inférieure. On note $\varnothing = \cap E$ et $\top = \cup E$. Dans le cas du treillis des parties d'un ensemble $\mathcal{U}$, ordonné par la relation d'inclusion, les opérations de borne inférieure et de borne supérieure correspondent aux opérations d'intersection et d'union. Le plus petit élément est l'ensemble vide et le plus grand élément est $\mathcal{U}$.

Un treillis complet (F, $\leq$) est une approximation supérieure de (E, $\leq$) s'il existe une paire d'applications adjointes (α, γ), appelées abstraction et concrétisation, tels que :

α : E $\rightarrow$ F surjective,

γ : F $\rightarrow$ E,
$\forall$ (e, f) $\in$ E×F, $e \leq \gamma(f) \Leftrightarrow \alpha(e) \leq f$

Soit *op* une fonction monotone de E dans E. Une fonction $\underline{op}$ de F dans E est une approximation supérieure de *op* si $\alpha \; op \; \gamma \leq \underline{op}$.

Le résultat qui motive ces définitions est le suivant : si $\underline{op}$ est une approximation supérieure de *op* alors LFP(*op*) $\leq \gamma$ LFP($\underline{op}$).

II.2. Approximation des horloges

Considérons le treillis complet suivant : F = {$\varnothing$, 0, 1, IN, $\top$} avec $\varnothing \leq x \leq \top$ pour tout x $\in$ F (0, 1 et IN sont incomparables). Intuitivement,

- 0 représente un tissu NIL,
- 1 représente une constante,
- IN une expression dont l'horloge est IN,
- $\varnothing$ est une expression dont on ne connaît pas encore l'horloge,
- $\top$ une expression qui n'est pas 0, 1 ou IN.

Les fonctions d'abstraction et de concrétisation se définissent par :

α :	YYIN	$\rightarrow$	F	γ :	F	$\rightarrow$	YYIN
	$\varnothing$	$\rightarrow$	0		$\varnothing$	$\rightarrow$	$\varnothing$
	{0}	$\rightarrow$	1		0	$\rightarrow$	$\varnothing$
	IN	$\rightarrow$	IN		1	$\rightarrow$	{0}
	pour tout autre x, x $\rightarrow$ $\top$				IN	$\rightarrow$	IN
					$\top$	$\rightarrow$	IN

(il est facile de vérifier que ces deux fonctions ont les propriétés voulues).

La fonction *op* dont on cherche le point fixe s'exprime comme composée des fonctions som, up, follow, etc (Cf. chap. VI). Une approximation supérieure de ces fonctions est obtenue en considérant $\alpha \; op \; \gamma$. Il suffit donc de chercher le point fixe de l'approximation dans le treillis approché. Par exemple, l'horloge t de

T@0 = 1; T = 3 + 5;

est obtenue comme la solution de (Cf. chap. VI) : t = follow ({0}, {som({0}, {0}})). Dans le treillis approché on a donc à résoudre :

t = $\underline{follow}$ (1, {$\underline{som}$(1, 1)})
= follow (1, 1)
= 1

d'où le fait que T est une constante.

L'horloge IN est affectée au tissu outside dont on sait que c'est vrai (par exemple Clock) Par extension, on peut aussi affecter cette horloge aux expressions du type « … when Clock ». La technique d'approximation du point fixe ne peut en effet capturer assez finement le comportement de telles expressions car il faudrait connaître la valeur de Clock et non pas seulement son horloge.

III. Conclusion

Nous n'étudierons pas plus avant les deux techniques d'optimisations que nous avons esquissées. L'objectif de ce chapitre est de convaincre le lecteur que le modèle d'exécution que nous avons développé à travers 81/2 facilite le processus de compilation et qu'il susceptible de supporter de nombreuses optimisations.

XII. Conclusion

Chaîne : *Objet (concret ou abstrait) composé d'éléments successifs. (XIII^e) Ensemble des fils parallèles disposés dans le sens de la longueur d'un tissu.*

Trame : *(XII^e) Ensemble des fils passés au travers des fils de chaîne, dans le sens de la largeur, pour constituer un tissu. Fig.. ce qui se déroule comme un fil.*

Tissu : *Sociol. (v. 1968) Ensembles d'éléments de mêmes fonctions, organisés en un tout homogène. Fig. Suite ininterrompue…*

Tissu, trame et chaîne

Dans cette thèse, nous avons essayé de jeter les bases d'un environnement permettant de *très grandes simulations digitales*. La réalisation de grandes simulations s'appuie sur l'utilisation efficace des grands calculateurs dont nous disposons ou dont nous disposerons dans un avenir plus ou moins proche. Les architectures des ordinateurs massivement parallèles ont été examinées brièvement dans le premier chapitre. Cela nous a conduit à choisir un modèle d'exécution hybride, combinant les modèles d'exécution MIMD et SIMD. Ce modèle permet de plus la représentation directe de toute une classe de systèmes dynamiques : les systèmes dynamiques discrets déterministes décrits par des relations fonctionnelles.

Les opérateurs 81/2 correspondent à une algèbre particulière ne permettant d'exprimer que certaines relations fonctionnelles entre les variables d'un système. Le choix de cette algèbre particulière a été motivé par des raisons idéologiques (refus des opérateurs non-causals, Cf. chap. I), pratiques (inclusion des opérateurs largement reconnus comme utile, Cf. chap. I & II) et implémentatoires (refuser les opérateurs impliquant une gestion dynamique, Cf. chap. I, II, VIII & IX).

Cependant ces choix n'ont pas pu être validés par l'implémentation de nombreuses applications. De manière générale, les exemples de programmes 81/2 manquent : dans le temps qui nous a été imparti, nous avons préféré développer les prémisses d'un environnement effectif de programmation (Cf. chap. X) plutôt que d'écrire sur le papier des exemples qui pourraient se révéler à terme, faux.

Par ailleurs nous pensons que 81/2 est plus une façon d'organiser et d'exploiter les ressources d'une machine

parallèle hybride qu'un langage de programmation à la disposition de tout un chacun. De ce point de vue, c'est avec la génération de code pour la machine PTAH que l'ordonnancement des programmes 81/2 peut réellement être testé en vraie grandeur. Ces travaux se poursuivent avec pour première borne l'exécution de programmes 81/2 sur un simulateur de PTAH. Des outils comme ROMA, qui permettent le placement et l'ordonnancement statique d'un graphe de tâches data-flow, seront appelés à jouer un *rôle analogue* et aussi important que celui de l'*éditeur de liens* qui existe pour les machines séquentielles. L'implémentation de ROMA est un travail en cours. Les algorithmes à implémenter (Cf. chap. IX) sont simples mais requièrent une attention très soigneuse afin d'être efficaces. En particulier, le maintien d'une représentation redondante permet d'accélérer énormément les divers calculs, mais nécessite, comme nous avons pu le constater, une gestion pénible de la cohérence.

La génération de code pour une machine parallèle existante, la Connection-Machine 2, sera disponible au début de l'année 92. Cela permettra de d'exécuter de grands programmes 81/2. Cependant il ne faut pas oublier que la CM2 est un calculateur SIMD : la comparaison d'un programme 81/2 et du même programme en *LISP (en C*) ne testera donc pas l'efficacité de l'exploitation du parallélisme présent dans le programme mais uniquement l'expressivité des styles de programmation induits par les deux langages. Nous pouvons tenter de prédire les résultats d'une telle comparaison en extrapolant les enseignements tirés des quelques exemples de programmes existants : 81/2 se révélera particulièrement bien adapté à l'expression de programmes possédant une structure de contrôle répétitive et relativement homogène. C'est par exemple le cas pour la simulation de réseaux comportementaux (Cf. [Legrand 91]). Son emploi sera plus difficile dans le cas de programmes comportant beaucoup de phases très diverses et dont le séquencement est arbitraire.

La sémantique du langage (Cf. chap. III) a été développée dans un style purement dénotationnel présentant l'avantage de techniques connues et suffisamment simples à mettre en œuvre. En particulier, ce cadre élude le caractère parallèle des calculs. En contrepartie, l'étude de la structure temporelle des streams (Cf. chap. VI) qui découle de cette sémantique est particulièrement pénible.

Une autre approche, et qui pourrait se révéler très fructueuse, serait de considérer les valeurs produites à chaque top par une expression comme une trace. On se placerait alors dans un cadre classique pour l'expression explicite d'une sémantique parallèle. Or une sémantique permettant une expression explicite du parallélisme et de l'asynchronisme est nécessaire. En effet, nous avons envisagé dans cette thèse une interprétation *synchrone* des programmes 81/2. Concrètement, cela se traduit par une communication qui doit être déterministe et des barrières de synchronisations. Mais que se passe-t-il si on supprime ces barrières de synchronisations, ou si la communication est indéterministe ? La confrontation des deux modèles d'exécution est intéressante à plus d'un titre. Par exemple l'itération asynchrone est le mode de fonctionnement primitif des réseaux de Hopfield. C'est aussi le schéma utilisé en physique par beaucoup de simulations dites de *Monte-Carlo*. Ces deux modes de fonctionnement correspondent aussi au fonctionnement synchrone ou asynchrone des circuits VLSI.

Les deux modes d'exécution correspondent au *modèle des itérations synchrones* et *au modèle des itérations asynchrones* (ou *chaotiques*). Ainsi qu'on l'a vu au chapitre IX, un programme 81/2 correspond à l'itération d'une fonction :

$$\forall\, t \in \mathbb{N},\ 1 \le i \le n, \qquad x_i(t+1) = f_i(x_1(t), \ldots, x_n(t))$$

pour toutes les variables x_i d'un programme. On appelle ce schéma de fonctionnement une *itération synchrone* car la valeur de x_i à l'instant t+1 est calculée à partir de la valeur des variables à l'instant t.

On peut maintenant imaginer un mode de calcul *asynchrone*. On suppose que pour chaque variable x_i, il existe un ensemble de dates T_i correspondant au changement de valeurs de ces variables; de plus, au moment du calcul de x_i à un instant t, on ne dispose que d'une valeur antérieure des autres variables. Autrement dit :

$$\forall\, t \in T_i,\ 1 \le i \le n, x_i(t+1) = f_i(x_1(\tau_1^i(t)), \ldots, x_n(\tau_n^i(t)))$$
$$\forall\, t \notin T_i,\ 1 \le i \le n, x_i(t+1) = x_i(t)$$

$$\forall\, t \in \bigcup T_i\ , \qquad 0 \le \tau_j^i(t) \le t$$

Ce schéma d'itération s'appelle asynchrone car la mise à jour de la valeur des variables se fait de manière asynchrone. Les fonctions τ^i correspondent à l'horizon temporel du calcul de la variable x_i : elles décrivent comment ce processus de calcul perçoit la valeur des autres variables au cours du temps. Elles modélisent donc un délai de communication (Cf. [Bertzekas, Tsitsikilis 89] pour un traitement complet). Savoir relier le comportement des deux modèles permettrait de simplifier l'analyse de nombreux systèmes, de faciliter la conception des circuits VLSI, de rendre moins coûteuse l'implémentation des synchronisations des programmes 81/2 (par exemple en les supprimant).

La sémantique que nous avons développée est celle d'un langage essentiellement statique. Cela est en accord avec les applications que nous visons et permet la compilation (qui est présentée dans les chapitres VIII & IX; le chapitre VII qui discute d'une exécution dynamique a essentiellement pour but de montrer par comparaison les avantages de l'approche statique). Cette approche statique nous oblige à vérifier certaines contraintes à travers la sémantique statique du langage (chapitres IV & V).

L'analyse de la structure temporelle des streams est introduite dans le chapitre VI. L'approche adoptée est celle du typage. La notion de type existentiel nous semble permettre l'expression de l'« imprévisibilité » due à l'opérateur when et c'est une voie qu'il faut creuser : comment de nouvelles notions du typage peuvent-elles permettre une meilleure compilation des programmes parallèle. Une première application est proposée avec la détection des étreintes fatales (tissu NIL) et une deuxième application est esquissée dans le chapitre XI avec la propagation des constantes ([Steele 90] et [DJG 91] en proposent d'autres). Le choix d'un modèle plus « fin » pour l'interprétation des types temporels, par exemple permettant de capturer le séquencement « a until b, c after b; » est primordial pour l'implémentation des programmes dont le contrôle est irrégulier. Mais c'est sans doute un problème difficile (dans le strict cadre que nous nous sommes donnés, si on « stocke » trop d'information dans le modèle, on obtient des méthodes d'analyses qui ne sont pas utilisables et si on ne conserve pas assez d'information, on risque des anomalies analogues aux anomalies du type « Brock-Ackerman » (i.e. on a perdu les représentations caractéristiques du calcul, Cf. par exemple [Broy 88])).

Beaucoup de questions intéressantes, intriguantes et utiles restent ouvertes comme par exemple les conditions nécessaires pour obtenir l'équivalence entre une interprétation synchrone et asynchrone des équations du langage, une approche algébrique du placement ou bien encore la structure des types temporels qui soit la plus fine possible tout en restant utilisable. Les approches basées sur la vision d'un programme 81/2 comme un système dynamique particulier n'ont pas été du tout explorées (cette approche est par exemple utilisée dans le calcul d'horloge du langage SIGNAL qui correspond à un système dynamique dans $\mathbb{Z}\ \mathbb{Z}/3\mathbb{Z}\ \mathbb{Z}$). Tout cela sans compter les très nombreux travaux à faire sur la plate-forme *8,5*.

À la fin de ce mémoire, il est difficile de ne pas voir surgir le souriant regret de toutes ces questions irrésolues.

∴

Éloge de la simulation[1]

Nous ne pourrions achever ce travail sans parler, au risque de heurter le lecteur, de cette autre thèse, celle qu'on n'a pas faite mais qui se laisse voir entre les fils de la trame. Nous avons essayé de jeter les bases d'un environnement permettant de *très grandes simulations digitales*. Or, il nous semble que la simulation est au centre d'un courant actuel qui tend à lier l'informatique et les sciences de la matière et du vivant. Ce rapprochement, notre opinion le décèle à travers des ouvrages de vulgarisation, des articles de recherche, l'intérêt des étudiants, les programmes de nouveaux

[1] Nous empruntons le titre de cette conclusion à un livre de Philippe Quéau [Quéau 86]. Dans cet essai, l'auteur défend l'idée qu'il nous faut un nouveau langage, celui des images, et les images de synthèse en serait l'écriture.

colloques… c'est-dire que la question est pour le moins *à la mode*. Mais elle est liée à un double enjeu méthodologique dont nous aimerions illustrer quelques aspects en guise de mot-de-la-fin. Les considérations exprimées ici paraîtront peut-être galvaudées : ce serait alors le signe que ce rapprochement est achevé.

Un nouveau paradis/paradigme

L'informatique a souvent emprunté ses objets et ses méthodes à des domaines scientifiques extérieurs. Parmi ceux-ci, la logique a fourni nombre de modèles, de concepts et de techniques (elle continue à en fournir). C'est la logique qui a structuré notre compréhension de l'informatique : en particulier, les limites du calcul automatique sont définies comme les limites des systèmes formels.

Ces résultats sont acquis et ne sont pas remis en cause. Mais le rôle séminal des systèmes logiques rencontre de sérieux compétiteurs dans les nouveaux domaines de l'informatique. Citons trois exemples parmi d'autres :

– *La conception de nouveaux algorithmes parallèles* :
  Plusieurs nouveaux algorithmes parallèles sont fondés sur des analogies provenant de la physique statistique (le recuit simulé), la génétique (les algorithmes génétiques), la biologie (les réseaux de neurones formels).

– *L'analyse de performances et la conception d'algorithmes répartis sur de grands systèmes informatiques distribués* :
  L'analyse de performances fait souvent appel à une modélisation en terme de processus stochastiques. Par exemple, la convergence du « routage forcé » de la machine MEGA du LRI est obtenue à l'aide d'une modélisation en terme de chaîne de Markov. Mais l'approche stochastique ne se cantonne pas à l'analyse des performances : elle permet même d'inventer de nouveaux algorithmes comme le routage randomisé à la Valiant.

– *L'intelligence artificielle* :
  Le récent intérêt suscité autour de la « vie artificielle » [Artificial Life 89] et des théories connexionistes [PDP 86] peut se comprendre comme un changement de point de vue. L'intelligence et les capacités cognitives ne se définissent plus comme les capacités symboliques d'un expert, mais comme les capacités caractéristiques du vivant : l'*adaptation* et l'*autonomie* [1].

Ces problèmes ont en commun le besoin de prendre explicitement en compte deux facteurs qui sont généralement absents des modèles logiques, à savoir l'*espace* et le *temps*. Ces facteurs sont au centre des systèmes parallèles et des applications interactives ou réactives (les programmes interactifs ou réactifs s'opposent aux programmes transformationnels destinés à calculer le résultat $f(x)$ de l'application d'une fonction f sur une entrée x).

Les systèmes dynamiques sont des modèles qui prennent explicitement en compte les paramètres temps et espace. Il est donc naturel que les techniques et les concepts développés pour ces derniers soient pertinents en informatique et qu'ils deviennent de plus en plus prégnants, au fur et à mesure que ces problèmes temps/espace deviennent plus importants. Autrement dit, l'intérêt pour le modèle des systèmes dynamiques grandit en même temps que l'intérêt pour les applications interactives ou réactives. Par ailleurs, le déroulement d'un programme correspond à l'évolution d'un système dynamique discret particulier. Ceux-ci doivent donc pouvoir nous aider à concevoir et à maîtriser les grands systèmes informatiques (et en particulier les ordinateurs parallèles). C'est pourquoi les prochaines idées et préoccupations en informatique proviendront des systèmes dynamiques plutôt que des systèmes formels (si ce n'est déjà fait).

Pour étayer ce point de vue, nous voudrions détailler plus avant un fascinant article de Many Chandy, intitulé « Reasonning about Continuous Systems ». Dans cet article, [Chandy 90], l'auteur nous propose d'utiliser les

[1] [ECAL 91, *intro.*] : « Arificial Life embodies a recent and important conceptual step in modern science : asserting that the core of intelligence and cognitive abilities is the same as the capacity for living. ».

techniques logiques développées pour la construction correcte des programmes afin de raisonner sur les systèmes (dynamiques) continus. Des analogies sont faites entre les deux domaines et en particulier, la suite des calculs d'un programme est comparée à la trajectoire d'un système. Ce point de vue est appliqué à la démonstration de la convergence d'un réseau de Hopfield à l'aide du système formel UNITY. Cet article semble donc aller a contrario de notre analyse. Cependant, si on y regarde de plus près, afin de montrer la convergence du réseau, M. Chandy doit introduire de manière arbitraire une quantité et une assertion qui ont une interprétation physique simple : c'est une énergie (exactement la même que celle qui est utilisée par Hopfield dans sa démonstration initiale) et cette énergie doit décroître. Dans les remerciements M. Chandy déclare : « (…) the UNITY computationnal model is essentially the same as that used in discrete neural networks. » [Chandy 90, *p 132*]. Autrement dit, les techniques logiques qui sont développées de manière ad-hoc afin de pouvoir traiter l'aspect temporel des programmes, redécouvrent des notions qui ont été développées pour la compréhension et la maîtrise des systèmes dynamiques. C'est pourquoi nous pensons que des concepts comme le hasard, l'entropie, l'énergie… sont capables de renouveler le traitement de concepts comme l'asynchronisme, l'indéterminisme, la distribution, etc.

La simulation ou de l'imitation à l'exploration

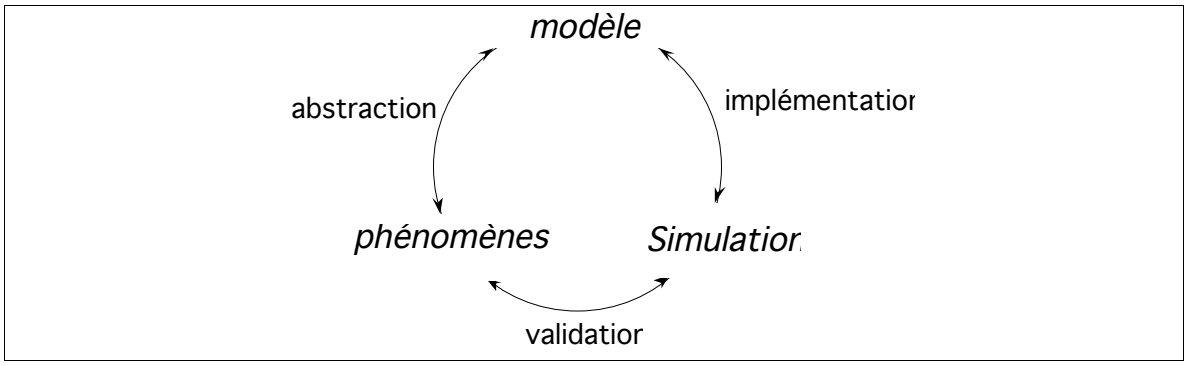
Si les systèmes dynamiques viennent fertiliser le champ informatique, l'informatique vient aussi au secours de l'étude des systèmes dynamiques. Cet apport nous semble principalement centré autour de la notion de simulation.

L'imitation du réel

Quand nous parlons de simulation, il s'agit pour nous de la simulation des systèmes dynamiques sur un ordinateur digital. Cette activité s'est considérablement développée : on a même construit des machines spécialisées et fort coûteuses pour réaliser une simulation particulière (par exemple le GF11 d'IBM pour étudier la chromodynamique quantique). On peut donc à bon droit se demander ce qu'on recherche quand on simule. La réponse la plus immédiate est celle qui a trait à l'*imitation* du réel.

Nous sommes à présent accoutumés à ces images de synthèse représentant l'écoulement d'un fluide par une carte de petits vecteurs. Ceux-ci nous permettent de visualiser un paramètre qui n'était pas immédiatement perceptible dans l'observation du phénomène réel. La simulation correspond donc tout à fait a une *expérience* permettant d'observer et de mesurer. Mais cette expérience est affranchie des contingences matérielles : on peut voir l'invisible mais aussi étudier sans risque des phénomènes inaccessibles ou dangereux comme la fissuration des chaudières nucléaires. Il est aussi possible de la stopper, de la reprendre, éventuellement de la dérouler à l'envers, de la répéter autant de fois que nécessaire, de mesurer chacun des détails, de fixer les conditions initiales et les conditions aux limites, d'en explorer tous les paramètres, de la modifier rapidement, de transformer le modèle… C'est donc une expérience d'une très grande souplesse.

La simulation, en tant qu'imitation du réel, doit donner une *image fidèle* des phénomènes. Elle peut donc jouer le rôle d'une *prédiction* : par exemple la simulation des phénomènes météorologiques permet de prévoir le temps qu'il fera. Mais il faut auparavant atteindre cette adéquation entre la chose réelle et la chose simulée. Avant cela, la simulation est le moyen de confronter un modèle, qui exprime une *hypothèse* donc une *opinion*, et les faits réels. La simulation joue alors le rôle de l'expérience qui permet de valider une théorie (Cf. figure ci-dessous).



Les limites de la simulation

Pourtant il y a une limite au rôle prédictif que peut jouer une simulation. Comme la théorie de la calculabilité fixe une limite à la puissance des systèmes formels, les systèmes dynamiques portent en eux des limitations d'une autre sorte et qui s'opposent à d'autres attentes. Les exhortations de [Thom 83] ou des livres comme [Ekeland 84] nous ont habitué à l'idée qu'il y a une borne à la précision descriptive des modèles. Des phénomènes comme l'instabilité, les comportements chaotiques ou la non-analycité des trajectoires sont des notions qui interviennent presque immédiatement dans l'étude de la dynamique des systèmes complexes et qui empêchent la réduction totale du réel à un ensemble de lois d'évolution.

La simulation ne serait donc utile que dans l'étude d'une classe très restreinte de systèmes ?

Vers une étude abstraite des modèles

Nous ne le pensons pas. La simulation permet d'étudier des objets abstraits. Comme exemple le plus immédiat nous pouvons citer les *mondes virtuels*, ces simulacres créés par le calcul et auxquels on accède par une panoplie encore encombrante de data-gloves, de caméras vidéos, de tubes cathodiques et d'exo-squelettes. Mais qu'on pense aussi à l'étude des attracteurs étranges rendue possible par la simulation numérique intensive. La simulation permet de rendre ces entités abstraites sensibles et vivantes : l'image d'une fractale, bien que fausse, bien qu'échappant à nos moyens de représentation finis, nous apprend quelque chose.

Car quoi, il nous faudrait croire à l'inutilité des simulations quand elles perdent leur qualité prédictive ? Ce serait alors réduire la connaissance d'une chose à la capacité de décrire *exactement* le comportement de cette chose. Or nous nous opposons à cette conception. Pour reprendre le titre servant de manifeste à [Thom 91] : prédire n'est pas expliquer.

Dans *le Sophiste*, Platon utilise la métaphore suivante : la recherche de la connaissance est une chasse. Quand la description n'est plus possible, il faut *ruser* avec la réalité : la chasse (à la connaissance) n'est plus une chasse frappeuse (ou l'on cherche à épingler les phénomènes), mais une chasse qui se pratique à l'aide de nasses, de filets, de lacets, et où on cherche à enclore la connaissance, à délimiter, séparer, trier. C'est la simulation qui joue le rôle de ces clôtures où on essaie de piéger le réel, sachant que les mailles du filet ne seront jamais suffisamment petites.

Un nouvel espace s'ouvre donc au chercheur, qui ne consiste plus à confronter certaines conséquences d'un modèle avec des phénomènes réels, mais qui permet l'*exploration des modèles*. La simulation n'a plus alors pour but la reproduction fidèle de la réalité, mais un moyen permettant d'exhiber certaines propriétés, de chercher les *qualités* de telle ou telle classe de modèles. Il devient alors possible de les comparer, ce qui veut dire par exemple de répondre à des questions comme « que peut-on négliger ? », « quels sont les domaines de validité ? », « quel est l'effet de tels paramètres ? ». La simulation devient l'*instrument* scientifique privilégié d'une étude abstraite des systèmes, semblable en cela au rôle qu'a joué le microscope pour l'exploration du petit. Ainsi, on peut lire en quatrième de couverture du

livre « Dynamique des systèmes complexes » [Weisbuch 89] : « La compréhension du comportement collectif organisé de ces systèmes à partir des propriétés très simples de leurs constituants élémentaires nécessite l'introduction de nouvelles méthodes, *qui s'appuient plus sur la simulation numérique par ordinateur, que sur les méthodes formelles utilisées en mathématiques*. » (c'est nous qui soulignons).

L'étude des systèmes et de leurs propriétés *en-soi*, afin de déterminer leur champ d'application, nous semble être un challenge considérable qui ouvre la voie à une ingénierie de la modélisation **(** enfin un trafic toujours fluide, enfin des usines avec un rendement maximal pour un coût de production et une pollution minimum… **:–)**. Il reste donc beaucoup d'ouvrage sur le métier, beaucoup de *mailles à resserrer*.

Bibliographie[1]

[*LISP 86] Thinking Machine Corporation, *The Essential *Lisp Manual*, Cambridge MA, 1986

[ACPP 89] Martin Abadi, Luca Cardelli, Benjamin C. Pierce, Gordon D. Plotkin, *Dynamic Typing in a Statically Typed Language*, Research Report 47, Digital - Systems Research Center, Palo Alto, June 10, 1989
`Le typage dynamique; très clair exposé des techniques à mettre en œuvre pour prouver la correction d'un système de typage vis à vis de la sémantique d'un langage.`

[Agha 85] G. Agha, *Actors: a model for concurrent computation in distributed systems*, AI tech. rep. 844, MIT, 1985.
`Un modèle classique sur les acteurs (l'autre est celui de Clinger)`

[AHO 83] Alfred Aho, John Hopcroft, Jeffrey Ullman, *Data Structures and Algorithms*, Addison-Wesley series in Computer Science and Information Processing, January 1983.

[Almasi, Gottlieb 89] George S. Almasi and Allan Goettlieb, *Highly Parallel Computing*, The Benjamin Cummings Publishing Company, 1989.
`Un livre qui offre un panorama remarquable des ordinateurs parallèles et un très clair exposé des problèmes architecturaux posés par le parallèlisme`

[Arbib 87] Michael A. Arbib, *Brains, Machines and Mathematics*, Second Edition, Springer Verlag 1987 (ISBN 0-387-96539-4)

[Artificial Life 89] Christopher G LANGTON edt, *Artificial Life*, Addisson Wesley

---

[1] Nous avons simplifié les règles typographiques habituelles; quelque soit le type de la référence, (article, livre, présentation) elle se présente sous la forme suivante : prénom et nom des auteurs dans l'ordre de citation, titre en italique, localisation spatiale et temporelle. L'appel de la référence comporte le nom complet des auteurs tant que ceux-ci sont moins de deux. Au-delà, la référence est constitué de l'initiales de chacun des noms d'auteurs ou d'un acronyme en rapport avec le sujet. Les commentaires ne reflètent que mon appréciation subjective et ne saurait en aucun cas engager mon employeur `:-)`.

[Arvind, Brock 83] Arvind, J.D. Brock, *Streams and Managers,* Proceedings of the 14th IBM Computer Science Symposium, 1983.
`La programmation avec des streams`

[Ashcroft 85] E. A. Ashcroft, *Eazyflow Architecture*, Computer Science Laboratory, SRI Technical Report N° CSL-147, April 1985, reproduit dans [Thakkar 87]
`Une implémentation matérielle possible pour LUCID`

[Ashcroft, Wadge 76] E.A. Ashcroft and W. W. Wadge, *Lucid - A formal system for writing and proving programs*, SIAM Journal on Computing (3):336-354, September, 1976.
`La première référence que je connaisse faîte à LUCID. Depuis le langage à beaucoup évolué (Cf. [Wadge, Ashcroft 85]).`

[Athas 87] W.C. Athas, "*Fine Grain Concurrent Computations*", Tech. Rep 5242, Dep. of Computer Science, California Institue of Technology, May 1987.
`Un modèle dérivé des acteurs.`

[Athas, Seitz 88] W.C. Athas, C.L. Seitz, "Multicomputers : Message-Passing Concurrent Computers", IEEE Computer, vol. 21, n° 8, Août 1988, pp 9-24

[Backus 78] J. Backus, *Can programming be liberated from the Von Neumann style ? A functionnal style and its algebra of programs*, Com. ACM Vol. 21, N° 8, pp 613-641, August 1978 (et aussi dans [Thakkar 87])
`Le manifeste de la programmation fonctionelle`

[BCLM 88] Jean-Pierre Bânatre, Anne Coutant, Daniel Le Metayer, *A parallel machine for multi-set transformation and its programming style*, Future Generation Computer Systems, 4, pp 133-144, Noth-Holland, 1988

[Beckerle, Ekanadham 86] M.J. Beckerle, K. Ekanadham, "*Distributed Garbage Collection with no Global Synchronisation*", IBM research report RC 11667 (#52377) january 1986.
Un `ramasse-miette basée sur les compteurs de références`

[Belhamissi, Jégado 91] Y. Belhamissi, M. Jégado, *Scheduling in distributed Systems : Survey and Questions*, Rapport IRISA/INRIA N°592, Juin 1991.
`Une synthèse de la littérature sur le placement. Plutôt centré sur le placement dynamique`

[Benveniste, LeGuernic 87] Albert Benveniste, Paul LeGuernic, *A denotational theory of synchronous communicating systems*, INRIA research report N° 685, june 1987.
`Une présentation dénotationelle de la sémantique de SIGNAL et le calcul d'horloge (qui n'est pas dans le style dénotationnel)`

[Bergerand 86] Jean-Louis Bergerand, *LUSTRE : un langage déclaratif pour le temps réel*, thèse de l'institut national polytechnique de Grenoble, 8 janvier 1986.
`Une sémantique dénotationelle de LUSTRE et des transformations de programmes.`

[Bernstein 66] A.J. Bernstein, *Analysis of programs for parallel processing*, IEEE Trans. Electronic Computers, 15, p757-62, 1966.
`Cité par Zima comme la première analyse de l'effet du réordonnancement des instructions`

[Berry, Boudol 89] Gérard Berry, Gérard Boudol, *The chemical abstract machine*, Rapport de Recherche N° 1133, INRIA, Décembre 1989.

[Berry, Couronné, Gonthier 87] Gérard Berry, P. Couronné, G. Gonthier, *Synchronous programming of reactive systems, an introduction to ESTEREL*, INRIA technical report 647, 1987.
`Le langage Esterelle et l'hypothèse de fort synchronisme`

[Bertzekas, Tsitsikilis89] Dimitri Bertzekas, John Tsitsikilis, *Parallel and distributed computation – Numerical methods*, Prentice-Hall International Editions, 1989 (ISBN 0-13-648759-9)
`Un must dédié au calcul numérique sur les machines parallèles. Le chapitre 6 et 7 sont consacrés aux méthodes d'itérations asynchrones (parfois encore appelé itérations chaotiques dans la littérature).`

[Bevan 87] D. I. Bevan, "*Distributed Garbage Collection Algorithm using Reference Counts*", ACM trans. on prog. lang. and syst., vol 2, n°3, jully 87.

[BFLPS 85] C Bétourné, M. Fillali, J.M. Lagarde, G. Padiou, A. Sayah, *Méthode de programmation structurée des systèmes répartis*, in "Parallélisme communication et synchronisation", J.P. Verjus & G. Roucairol editeurs, Edition du CNRS, 1985 (ISBN 2-222-03672-0)
`Des processus migrant comme structure de contrôle pour la programmation des applications distribuées`

[Blelloc, Sabot 90] Guy E. Blelloch and Gary W. Sabot, *Compiling Collection-oriented Languages onto Massively Parallel Computers*, Journal of Parallel and Distributed Computing, 8, 119-134 (1990)
`Le concepts de collections et implémentation sur les machines massivement parallèle (centré sur Paralation Lisp).`

[Blelloch 89] Guy E. Blelloch, *Scans primitive Parallel Operations*, IEEE Transactions on Computers, Vol. 38, N° 11, November 1989.
`Un style de programmation utilisant intensivement le scan`

[Bolton 90] Martin Bolton, *Digital Systems Design with programmable logic*, Addison-Welsey Publishing Company, 1990 (ISBN 0-201-14545-6)
`Une introduction au PLD : programmable logic device.`

[Bonabeau, Leboucher 91] E. Bonabeau, L. Leboucher, *81/2 et les L-systèmes*, rapport de mini-projet de DEA d'architecture, Université de Paris-Sud, 1991.

[Bougé 90] Luc Bougé, *On the semantics of language for massively parallel SIMD architectures*, Rapport LIENS, Ecole Normale Supérieure, 22 Juin 1990.
`Equivalence sémantique entre le point de vue microscopique (tableau de processeur) et le point de vue macroscopique (processeur de tableaux) d'un langage SIMD`

[Broy 88] Manfred Broy, *Nondeterministic data flow programms: how to avoid the merge anomaly*, Science of Computer Programming, North-Holland, 10 (1988), pp 65-85,

[Cai, Page 89] Jiazhen Cai, Robert Paige, *Programm Derivation by fixed point computation*, Science of Computer Programming, 11, 1988/1989, pp 197-261
`Transformation de calcul de point fixe en schéma itératif.`

[Calvino 72] Italo Calvino, *Les villes Invisibles*, édition du Seuil, 1974, traduction française de « Le città invisibili », Giulio Einaudi Editore, Torino, 1972
`Marco Polo raconte au grand Kahn mélancolique les villes qu'il a visité dans le cours de ses ambassades. Chaque cité porte le nom d'une femme; il n'est pas dit que Kublai Khan croit à tout ce que Marco Polo lui raconte.`

[Cappello, Béchennec 91] Franc Cappello, Jean-Luc Béchennec, *PTAH : un réseau de proceseeurs à géométrie compilable pour les applications de traitement numérique intensif*, 3ième Symposium National sur les Architectures Nouvelles de Machines, Juin 1991, pp 249-270
`Une présentation succincte du projet PTAH`

[Cardelli 89] Luca Cardelli, *Typeful Programming*, Research Report 45, Digital - Systems Research Center, Palo Alto, May 24 1989
`Du bon usage du typage statique suivant Cardelli`

[Cardelli, Wegner 85] Luca Cardelli and Peter wegner, *On Understanding Types, Data Abstraction and Polymorphism*, ACM Computing Surveys, Vol 17, N°4, December 1985
`Du bon usage des types suivant Cardelli, un classique.`

[Cardone 89] Felice Cardone, *Relationnal Semantics for Recursive Types and Bounded Quantification*, Proc. 16th Colloquium Automata, Languages and Programming, Stresa Italy, July 1989, LNCS 372, pp164-178.
`Une sémantique des quantifications de types`

[Carriero, Gelerntner 89] Nicholas Carriero, David Gelerntner, *Linda in context*, Communications of the ACM, Vol. 32, N° 4, pp 444-458, 1989.
`Une présentation du langage LINDA basée sur la manipulation réparties de tuples.`

[Caspi 91] Paul Caspi, communication personnelle, Novembre 1991.

[CGJ 84] E.G. Coffman, M.R. Garey, D.S. Johnson, *Aproximation algorithm for bin-packing - an updated survey*, in "Algorithm design for computer system design", G. Ausiello, M. Lucertini, P. Serafini eds, CISM Courses and Lectures N°284, Springer-Veralg (ISBN 3-211-8>816-2), 1984
`Une synthèse sur le sujet de packing 2D et ses variations.`

[Chandy 90] K. Mani Chandy, *Reasoning about continuous systems*, Science of Computer Programming, Vol 14, pp 117-132, 1990, North-Holland
`Application des techniques de construction formelle de programmes (et plus particulierement de UNITY) à l'analyse des systèmes continus.`

[Chen 86] Marina C. Chen, *A Parallel Language and its compilation to multiprocessor machines or VLSI*, 30th Annual ACM SYmposium on Principles of Programming Languages, 1986, Florida, pp 131-139.
`Un langage basé sur les équations récurrentes.`

[Clermont 90] Philippe Clermont, *Méthodes de programmation de machine parallèle pyramidale – Applications en segmentation d'images*, Thèse de doctorat de l'université de Paris VII, 22 janvier 1990.
`SPHYNX : une architecture pyramidale pour le traitement d'image, définition d'un langage MSIMD (et plus exactement 8-SIMD).`

[Clermont, Merigot 87] P. Clermont, A. Merigot, *Real time synchronization in a multi-SIMD massively parallel machine*, Proc. of the Workshop on Computer Architecture for Patrtern Analysis and Machine Intelligence, (CAPAMI), Seattle, October 1987.
`La synchronisation sur SPHYNX`

[Clinger 81] W.D. Clinger, "*Foundation of Actor Semantics*", PhD thesis, MIT May 1987 (ai-tr-633).

[CMP 89] A; Couvert, A. Maddi, R. Pédrono *Object Sharing in Distributed Systems - Principles of garbage collection*, IRISA, INRIA report 963, January 1989
`Un état de l'art des ramasses-miettes distribués`

[Collins, Jefferson 90] Robert J. Collins, David R. Jefferson, *Representation of Artificial Organisms*, proc. of the First Internationnal Conference on simulation of adaptative behavior, ed. J.-A. Meyer & S. Wilson, The MIT Press, pp. 382-390

[Coppo, Zacchi 86] M. Coppo, M. Zachi, *Type inference and logical relations*, Proc. Symposium on Logic in Computer Science, IEEE, pp 218-226, 1986.

[Cousot, Cousot 77] P. Cousot and R. Cousot, *Abstract interpretation: a Unified Lattice Model for static Analysis of Programs by construction of approximation of fixed points*, proc. of the 4th ACM Symp. on Principle of Programming Languages, ACM, Los Angeles 1977.
`Un des articles fondateur de l'analyse des programmes par approximation de point fixe.`

[Cousot, Cousot 77b] P Cousot, R. Cousot, *Automatic Synthesis of optimal invariant assertions: Mathematical foundations*, SIGPLAN Notice, Vol. 12, N° 8, pp 1-12, 1977.
`les bases de l'itérations chaotiques dans un treillis (Cf. aussi [Cai, Page 89])`

[CPHP 87] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, *Lustre: a declarative language for programming synchronous systems*, in 14th ACM Symposium on Principles of Programming Languages, january 1987.
`Le langage LUSTRE`

[Dahl 90] E. Denning Dahl, *Mapping and compiled Communications on the Connection-Machine System*, proc. of the 5th Distributed Memory Computing Conference, Charleston, South Carolina, April 8-12, 1990.
`Comment et pourquoi compiler les communications sur une machine SIMD`

[DCF 87] John Darlington, Martin Cripps, Tony Field, Peter G. Harrison, Mike Reeve, *The design and implementation of ALICE : a parallel graph reduction machine* in [Thakkar 87]

[Delaplace, Giavitto 91] Franc Delaplace, Jean-Louis Giavitto, *An Efficient Routing Strategy to Support Process Migration*, CompEuro 91, Vienne, September 1991
`Gestion des tables de routages dans un hypercube afin de minimiser les mises-à-jours nécessaires lors de la migration d'une tâche`

[Denis 80] Jack B. Dennis, *Dataflow Supercomputers*, IEEE Computer Vol 13 N° 11, pp 48-56, November 1980
`Le data-flow et le calcul intensif`

[Dennis 74] Jack B. Dennis, *First Version of a Data Flow Procedure Language*, proc. of the Programming Symposium, Paris, April 9-11, 1974, LNCS 19, Springer.
`Un des premier langage data-flow. Cf. [Dennis 91] pour un historique du data-flow`

[Dennis 91] Jack B. Dennis, *Static Dataflow Architecture*, in [Gaudiot, Bic 91]
`Mise en perspective historique du data-flow par un des pères fondateurs`

[Diderot & d'Alembert] Diderot, d'Alembert, *Encyclopédie – Recueil de planches sur les sciences et les arts libéraux, et les arts méchaniques, avec leur explication - L'art de la soie*, édité à Paris avec approbation et privilège du Roy.
`C'est de la rééditions (chez inter-livres) que sont tirée les illustrations qui se glissent entre les chapitres. Merci à La Folie Balanchine.`

[DJG 91] Vincent Dornic, Pierre Jouvelot, David K. Gifford, *Polymorphic time systems for estimating program complexity*, rapport interne, école des Mines de Paris, 1991.

[Douglis 87] Douglis, Ousterhout, *Process Migration in the Sprite Operating System*, Distribued Computing, Computer Society Press, PP 18-24, 1987

[Du, Leung 89] Jianzhong Du, Joseph Leung, *Complexity of scheduling parallel task systems*, SIAM J. Discr. Math., Vol 2, N° 4, pp 473-487, November 1989.
`Complexité des problèmes d'ordonnancements de tâches requérant de multiples PEs. Toujours NP si on a des précédences entre tâches et au moins deux PEs`

[Dumesnil 91] Stéphane Dumesnil, *Implémentation de réseaux neuronnaux sur l'architecture PTAH*, rapport de DEA option informatique de l'université de PARIS-SUD, LRI, Orsay 1991.
`Comment implémenter l'apprentissage de plusieurs types de réseaux neuronnaux sur la machine PTAH. Un chapitre est consacré au programme 81/2 correspondant.`

[ECAL 91] Paul Bourgine, Francisco Varela (Conference chairmann), *First Europeean Conference on Artificial Life – Towards a practice of autonomous systems*, 11-13 December, Paris 1991 (MIT Press/Bradford Books)

[Eden 58] M. Eden in *Symp. on Information Theory in Biology*, ed. H.P. Yockey, Pergamon Press, New York 1958, p. 359.
`Un modèle de la croissance des tumeurs basé sur les automates cellulaires.`

[Ekeland 84] Ivar Ekeland, *Le Calcul, L'imprévu — Les figures du temps de Kepler à Thom*, éditions du Seuil, 1984
`Un extraordinaire petit livre sur les limites du caluls.`

[EMO 87] R. Cornu-Emieux, G. Mazaré, P. Objois, *A VLSI asynchronous cellular array to accelerate logical simulations*, proc. of the 30th. Midwest Internationnal Symposium on Circuit and Systems, 1987
`Une architecture de réseaux de cellules asynchrones.`

[Feautrier 88] Paul Feautrier, *Parametric integer programming*, RAIRO Recherche Opérationelle, N° 22, pp 243-268, Septembre 1988

[Feautrier 91] Paul Feautrier, *Dataflow Analysis of Array and Scalar References*, rapport interne du MASI, Université P. et M. Curie, 19 avril 1991.

[Feautrier 91b] Paul Feautrier, *Les bases de temps et leur role en programmation parallèle*, communication aux Journée Ordonnancement du Pôle C^3-, 18 Juin 1991, Paris.
`Extraction automatique du parallélisme grâce à des techniques pointues d'analyses de dépendances`

[Foren, Schwartz 87] H. Forren, D. A. Schwartz, *Transforming periodic synchronous multiprocessor programs*, proc ICASSP 87, pp 1406-1409, Dallas, April 1987.
`Ordonnancement répétitifs à la fois dans le temps et dans l'espace des PE.`

[FORTRAN 90] ANSI, *ANSI Fortran Draft S8*, *Version 111*. ANSI
`le Fortran 8X s'est transformé en FORTRAN 90. Ce n'est encore qu'une proposition de standardisation.`

[Fowler 86] R.J Fowler, *The Complexity Using Forwarding Adresses for Decentralized Object Finding*, Proc 5th A.C.M Symposium on Principles of Distribued Computation, Calgary, Canada, August 1986

[Gabriel, McCarthy 84] R. P. Gabriel, J. MacCarthy, *Queue-base multiprocessing LISP*, Conf. Record of the 1984 ACM Symp. on Lisp and Functional Programming, Austin Texas, 1984

[Gajski, Padua, Kuck 82] D.D. Gajski, D.A. Padua and D.J. Kuck, *A second opinion on data flow machines and languages*, IEEE Computer, february 1982 (repris dans [Thakkar 87]).
`Une célèbre critique du modèle data-flow, en particulier de la mauvaise gestion de la mémoire pour les tableaux.`

[Gaudiot, Bic 91] Jean-Luc Gaudiot, Ludomir Bic, editors, *Advanced Topics in data-flow computing*, Prentice Hall, 1991 (ISBN 0-13-006503).
`Les recherches actuelles dans le domaines des architectures data-flow`

[GBES 89] C. Germain, J-L. Béchennec, D. Etiemble and J-P. Sansonnet, *A New Communication Design for Massively Parallel Message-Passing Architectures*, IFIP Working Conf. on Decentralized Systems 1989, North-Holland ed.

[GBES 90] C. Germain, J-L Béchennec, D. Etiemble and J-P. Sansonnet, *An Interconnection Network and a Routing Scheme for a Massively Parallel Message-Passing Multicomputer*, Third Symp. on Frontiers 90 conference on Massively Parallel Computation, October 8-10 College Park, MD

[GDR 88] J.-L. Giavitto, A. Devarenne, G. Rosuel, *Presto: des objets C++ persistants pour le système d'information d'*A*dage*, AFCET Journées d'Etudes Bases de Données Déductives et Bases de Données Orientées Objets, Paris, Decembre 1988.
`La persistence des données dans le langage C++`

[GDRM 90] J.-L.Giavitto, G.Rosuel , A. Devarenne and A. Mauboussin, *"Design Decisions for the Incremental Adage Framework"*, Proceedings of the 12th International Conference on Software Programming, Nice, March 1990.
`Un environnement de développement générique basée sur un modèle de donnée utilisant des graphes rémanent attribués et typés.`

[GGF 91] J.-L. Giavitto, C. Germain, J. Fowler, *OAL: an Implementation of an Actor Language on a Massively Parallel Message-Passing Architecture*, proc. of the Second European Distributed Memory Computing Conference, 22-24 april 1991, Münich, LNCS 492
`Simulation d'un langage d'acteur sur une machine MEGA.`

[GGJ 78] M.R. Garey, R.L. Graham, D.S. Jonson, *Performance Guarantees for Scheduling Algorithms*, Operation research, Vol. 26, N° 1, January-February 1978, pp. 3-20
`Une revue des performances (au pire et asymptotique) de divers algorithmes d'ordonnancements (avec et sans contraintes).`

[GGS 91] C. Germain, J.-L. Giavitto, J.-P. Sansonnet, *Implémentation d'un Paradigme de Programmation Fonctionnelle sur une Machine Massivement Parallèl*e, Journée Francophone des Langages Applicatifs, 28-29 Janvier 1991, Gresse-en-Vercors, actes édités par BIGRE.
`Une implémentation de l'évaluation demand-driven sur une machine MEGA`

[Giavitto 86] J.-L. Giavitto, *La Notion de Sponsor dans Lore - Exemples d'expression de la Sémantique des Langages Parallèles ou Concurrent*, rapport de DEA de l'Université Paul Sabatier de Toulouse, option langage et système, Octobre 1986.

[Giavitto 91] Jean-Louis Giavitto, *81/2 : un modèle d'exécution synchrone pour la machine MEGA*, 3ième Symposium National sur les Architectures Nouvelles de Machines, Juin 1991, pp 229-248

[Giavitto 91b] Jean-Louis Giavitto, *A Synchronous Data-Flow Language for Massively Parallel Computers*, Parallel Computing'91, Londres, Septembre 1991.
`Une présentation succincte de 81/2`

[Giavitto 91c] Jean-Louis Giavitto, *Le langage* 81/2, rapport interne de l'équipe architecture du LRI, février 1991.
`On trouvera une présentation de 81/2 et un des exemples (en particulier un exemple de programmation dynamique).`

[GKS 87] Michael Granski, Israel Koren, Gabriel Silberman, *The effect of operation scheduling on the performance of a data flow computer*, IEEE Trans. on comp. Vol C-36, N°9, September 1987
`Comment introduire de l'ordonnancement statique dans une machine data-flow dynamique à travers une notion de priorité des instructions.`

[Ha, Lee 89?] S. Ha, E. A. Lee, *Compile-Time scheduling and assignment of dataflow program graphs with data-dependent iteration*, Memorandum N° UCB/ERL M89/57, Electronic Research Lab, U. C. Berkeley
`Si quelqun a accès à ce rapport…`

[Halbwachs 90] Nicolas Halbwachs, *Les langages synchrones*, support de la conférence Recherche/Industrie du GRECO de programmation du CNRS, 22 Mai 1990
`Une présentation du domaine de la programmation synchrone des systèmes réactifs.`

[Halstead 85] R. H. Halstead, *MultiLISP : a language for concurrent symbolic computation*, ACM Trans. on programming languages and systems, Vol. 7, N° 4, October 1985.
`Un classique de l'extension de LISP par des primitives explicites pour la programmation paralléle.`

[Hanen 91] Claire Hanen, *Structure et complexité des problèmes d'ordonnancement cycliques*, communication aux Journée Ordonnancement du Pôle C^3-, 18 Juin 1991, Paris.

[HCAL 89] Jing-Jang Hawang, Yuan-Chieh Chow, Frank Angers, Chung-Yee Lee, *Scheduling precedence graphs in systems with interprocessor communication times*, SIAM J. Comp., Vol. 18, N°2, pp 244-257, April 1989
`Analyse théorique des performances asymptotiques d'un algorithme greedy d'ordonnancement en présence de contraintes entre tâches et de délais de communications (extension de [Rayward-Smith 87]).`

[Hillis 85] W.D. Hillis, "*The Connection Machine*", The MIT Press, 1985
`Présentation de la Connection-Machine par son langage. Un classique.`

[Hillis, Steele 86] W. Daniel Hillis, Guy L. Steele, *Data parallel Algorithms*, Communication of the ACM, Vol. 29, N° 12, December 1986
`Un must sur la programmation SIMD (permet en particulier de découvrir que le SIMD n'est pas réduit au calcul numérique, loin de là)`

[Ho 89] Yu-Chi Ho, *Dynamics of discrete event systems*, préface de [IEEE 89]

[Hoare 88] C.A.R Hoare, *OCCAM 2 Reference Manual*, International Series Computer Science, Prentice Hall, 1988

[HOS 85] C. M. Hoffman, M. J. O'Donnel, and R. I. Strandh, *Implementation of an interpreter for abstract equations*, Software Practice and Experience, Vol. 15, N° 12, pp 1185-1204, December 1985
`Définition de la complétude référentielle.`

[HOS 85] C. M. Hoffman, M. J. O'Donnel, and R. I. Strandh, *Implementation of an interpreter for abstract equations*, Software Practice and Experience, Vol. 15, N° 12, pp 1185-1204, December 1985

[Huberman, Hog 88] Bernardo Hunberman, Tad Hog, *The Behavior of Computationnal Ecologies*, in "The Ecology of Computation", B.A. Huberman editor, Studies in computer science and artificiall intelligence, North-Holland, 1988 (ISBN 0-4444-70379-9)
`Recueils d'articles autour du concept de multi-agents : proposition d'un cadre méthodologique et analyse de comportements distribuées.`

[Hudak, Keller 82] P. Hudak, R.M. Keller, *Garbage collection and task deletion in distributed applicative processing systems*, proc. ACM conference on Lisp and Functionnal Programming, 1982 pp 168-178.

[Hughes 85] J. Hughes, *A Distributed Garbage Collection Algorithm* proc. ACM conference on Functional Programming Languages and Computer Architecture, Nancy 1985, LNCS 201.

[IEEE 89] *Special issue on dynamics of discrete event systems,* Proc. of the IEEE, Vol. 77, N°1, pp1-232, January 1989.
Un numéro consacré à la dynamique des systèmes discrets. Les articles sont surtout consacrés à l'analyse des modèles temporelles (réseaux de files d'attentes évaluation de performances), mais voir les articles [Zeigler 89] pour un formalisme de descriptions des systèmes dynamiques discrets et [Righter, Walrand 89] sur le problème de la simulation distribuée.

[Inmos 91] D. May, *The Next generation Transputers and beyond*, proc. of EDMCC2, LNCS 487, April 1991, pp 7-22
Du Transputeur T9000 et de l'importance du ratio communication/calcul et de son ajustement

[INTEL 86] INTEL Scientific Computers,"Intel iPSC System Overview", Order n° 310610-001, 1986

[Iverson 87] K. E. Iverson, *A dictionnary of APL*, APL Quote Quad, 18, 1, September 1987.

[Johnson 83] Steven D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, ACM Distinguished Dissertation 1983, MIT Press 19983 (ISBN0-262-10029-0)
De la conception des circuits VLSI synchrones par la programmation fonctionelle

[Kahn 73] Gilles Kahn, *A preliminary theory for parallel programs*, Rapport de recherche n° 6, IRIA-LABORIA, Janvier 1973.
Un article fondateur de la sémantique des réseaux data-flow.

[Kahn 87] Gilles Kahn, *Natural Semantics*, in Proc. of the Symposium on Theoretical Aspect of Computer Science, Passau Germany, February 1987, also available as INRIA report N° 601, February 1987.
Expression de la sémantique des programmes par des règles de déduction (supporté par le système CENTAUR).

[Kahn, MacQueen 77] Gilles Kahn, David MacQueen, *Coroutines and network of parallel processes*, IFIP 77, North-Holland, 1977, 933-938

[KALW 91] Jan Korst, Emile Aarts, Jan K. Lenstra, Jaap Wessels, *Periodic multiprocessor Scheduling*, proc. PARLE 91, pp166-178
Une méthode pour construire un ordonnancement répétitif de période p donné (ordonnancement cyclo-statique)

[KKLW 84] D.J. Kuck, R.H. Kuhn, B.R. Leasure, M.J. Wolfe, *The structure of an advanced retargetable vectorizer*, in "Supercomputers : Design and Applications Tutorial", Hwang ed., pp967-974, IEEE catalog N° EHO219-6, IEEE Society Press

[Kung, Lewis, Lo 87] S. Y. Kung, P.S. Lewis, S.C. Lo, *Performance analysis and optimization of VLSI data flow arrays*, J. Parallel Distributed Computing, vol. 4, pp 592-618, 1987
Evaluation des performances (temps et espace) et optimisations des graphes data-flow synchrone. Définiton de la borne d'itération.

[Le Guernic, Benveniste 87]
[Le Guernic, Benveniste 87]
P. Le Guernic and Benveniste, *Real-Time Synchronous Dataflow programming : The Language SIGNAL and its mathematical Semantics*, rapport de recherche INRIA 533 ou 620, février 1987.

[Le Guernic, Gautier 90] P. Le Guernic, T. Gautier, *Data-flow to von Neumann: the SIGNAL approach*, IRISA-INRIA internal report n°531, April 1990.

[Lee 91] Edward A. Lee, *Static Scheduling of dataflow programs for DSP*, in [Gaudiot, Bic 91]
`L'approche « compilation statique » des programmes data-flow.`

[Lee, Messerschmitt 87a] Edward A. Lee, David G. Messerschmitt, *Static Scheduling of Synchronous Dataflow programs for Digital Signal Processing*, IEEE Trans. on Comp., Vol C-36, N°1, January 1987.
`L'approche « compilation statique » des programmes data-flow.`

[Lee, Messerschmitt 87b] Edward A. Lee, David G. Messerschmitt, *Synchronous Dataflow*, Proc. of the IEEE, Vol 75, N°9, September 1987.
`L'approche « compilation statique » des programmes data-flow.`

[Legrand 84] B. Legrand, *Apprendre et appliquer le langage APL*, 4ième édition, MASSON 1984 (ISBN 2-225-80297)

[Legrand 91] Fabrice Legrand, *L'interprète* 81/2 *– Utilisation du langage dans la simulation de systèmes adaptatifs*, rapport de DEA option informatique de l'université de PARIS-SUD, LRI, Orsay 1991.
`Description de l'interprète 81/2 version 1 et utilisation du langage pour la simulation de réseaux comportementaux.`

[Leiserson, Rose, Saxe 83] Charles E. Leiserson, Flavio M. Rose, James B. Saxe, *Optimizing Synchronous Circuit by Retiming*, proc. of the Third Caltech Conf. on VLSI, pp. 87-116, Pasadena CA, March 1983
`Une technique d'optimisation des circuits VLSI synchrones. Adaptable aux programmes 81/2, correspond à l'optimisation des programmes par transformations (donnerait un programme sémantiquement équivalent mais avec un chemin critique plus court).`

[Leiserson, Saxe 83] Charles E. Leiserson, James B. Saxe, *Optimizing Synchronous Systems*, Journal of VLSI and Computer Systems, vol. 1, N°1, pp. 41-67, 1983
`Cf. [Leiserson, Rose, Saxe 83]`

[LGBBG 86] P. Le Guernic, A. Benveniste, P. Bournai, T. Gautier, *SIGNAL, a dataflow oriented language for signal processing*, IEEE-ASSP, 34(2):362-374, 1986.
`Le langage SIGNAL`

[LHF 91] Ben Lee, A. R. Hurson, Tse-Yung Feng, *A vertically layered allocation scheme for dataflow systems*, Journal of Parallel and Distributed computing, 11, pp 175-187, 1991
`Une heuristique pour l'ordonnancement des graphes data-flow en deux étapes : découpage du graphe en chemin critique et allocation de ces chemins au PEs, ensuite minimisation du coût des communication par echanges de tâches.`

[LHMRSP 91] D. Litaize, O. Hammami, A. Mzoughi, C. Rochange, P. Sainrat, R. Pulou, *Multiprocesseur massivement parallèle à mémoire multiport série*, recueil des communications du Troisième Symposium sur les Architectures Nouvelles de Machines, pp 155-175, Ecole Polytechnique, Palaiseau, 19-21 juin 1991.
`Description d'une liaison série à ultra haut-débit afin de réaliser un réseau d'interconnexions PE-mémoire permettant une mémoire partagée entre un très grand nombre de PE.`

[Li, Cheng 89] K. Li, K. H. Chen, *Complexity of resource allocation and job scheduling problems in partitionnable mesch connected systems*, Proc. 1st IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society, Dallas TX, May 1989, pp. 358-365
`cité dans [Li, Chen 90]`

[Li, Cheng 90] K. Li, K. H. Chen, *On three-dimensionnal packing*, SIAM J. Comput., Vol. 19, N°5, pp847-867, October 1990
`Packing de cube 3D.`

[Lieberman, Hewitt 83] H. Lieberman, C. Hewitt, "*A real-time garbage collector based on the lifetimes of objects*", CACM vol. 26 n°6, pp 419-428 June 1983.

[LIN 85] F.C.H. Lin, "*Load Balancing and Fault Tolerance in Applicative Systems*", Ph.D. Dissertation, Dep. of Computer Science, Univ. of Utah, 1985.

[Livercy 78] C. Livercy, *Théorie des programmes*, Dunod Informatiques, 1978
`Présentation et utilisation des théorèmes de point fixes dans le cadre de la sémantique des langages de programmation.`

[MacQueen, Plotkin, Sethi 86] David MacQueen, Gordon Plotkin and Ravi Sethi, *An ideal model for recursive polymorphic types*, Information and Control 71, pp 95-130, 1986
`Un modèle d'un système de typage comprenant des types quantifiés. L'interprétation des types quantifiés est indépendant d'une analogie avec la logique constructiviste.`

[Maes 90] Pattie Maes, *A bottom-up mechanism for behavior selection in an artificial creature*, proc. of the First International Conference on Simulation of Adaptative Behavior (SAB 90), pp238-246, MIT Press/Bradford Book, 1990

[Maguire 88] Maguire, Smith, *Process Migration: Effects on Scientific Computation*, A.C.M Sigplan, PP 102-106, March 1988

[MaRS 86] M. Lemaitre, M. castan, M. H. Durand, G. Durrieu, B. Lecussan, *Mechanisms for efficient multiprocessor combinator reduction*, Proc. of the 1986 ACM Conference on LISP and functionnal Programming, Cambridge, Massacchussetts, pp 113-121, août 1986.
`Présentation générale de MaRS, une machine multiprocesseur à réduction de graphe. Il existe une bibliographie commentée du projet, disponible auprès du CERT Toulouse.`

[Martin Estrin, 69] D. F. Martin, G. Estrin, *Path length computations on graph models of computations*, IEEE Trans. on Computers, C-18, pp 530-536, June 1969
`L'article séminal sur l'analyse de la longueur moyenne d'une exécution.`

[MAS 85] J.R. McGraw, S.K. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, R. Thomas, *SISAL : Streams and Iterations in a Single Assignment Language*, Lawrence Livermore National Laboratory, Livermore CA, March 1985. Version 1.2 M-146.
`Un langage data-flow`

[Mauras 89a] Christophe Mauras, *Definition of Alpha: a language for Systolic Programmation*, INRIA research report N° 482, June 1989.

[Mauras 89b] Christophe Mauras, *Alpha : un langage equationel pour la conception et la programmation d'architecture parallèles synchrones*, thèse de l'université de Rennes I, 15 décembre 1989.
`Un langage de programmation systolique basé sur les équations récurentes linéaires.`

[Mazare, Payan 90] Guy Mazaré, Eric Payan, *A new programming method for highly parallel architectures*, proc. of the ISLIP 90 Internationnal Symposium on Lucid and Intensional programming, Kingston, Ontario, 13-15 may 1990.
`Introduction des tableaux dans LUCID afin de programmer un réseaux de cellules asynchrones`

[Mercouroff 90] Nicolas Mercouroff, *Analyse sémantique des communications entre processus de programmes parallèles*, thèse de l'école polytechnique, Septembre 1990.
`Analyse statique des communications dans un programme de type OCCAM grâce à une approximation du calcul d'un point fixe dans un treillis.`

[Milne, Milner 79] George Milne, Robin Milner, *Concurrent processes and their syntax*, J. Assoc. Comput. Mach., **26**(2):302-321, April 1979

[Milner 73] Robin Milner, *Processes: a mathematical model of computing agents*, proc. Logic Colloq. '73, eds Rose and Shepherdson, North-Holland, 1973
`Sémantique des processus basées sur la notion de comportement`

[Milner 78] R. Milner, *A theory of type Polymorphism in programming*, Journal of Computer and System Science, N°17, p. 348-375, 1978

[Ming 90] Ming-Syan Chen, Kang G.Shin, *Subcube Allocation and Task Migration in Hypercube Multiprocessors*, IEEE Transactions on Computers, Vol 39, N°9, PP 1146-1155, September 1990

[Minsky, Papert 69] Marvin Minsky, Seymour Papert, *Perceptrons : an essay in computationnal geometry*, The MIT Press, 1969
`Célèbre analyse de la complexité des perceptrons d'où il ressort que les réseaux à une couche ne peuvent pas faire de la séparation non-linéaire. [Arbib 87 chap 4] est une présentation claire de ce travaux.`

[Mitchell, Plotkin 88] John C. Mitchell, Gordon D. Plotkin, *Abstract Types have existential type*, ACM Transaction on Programming Languages and Systems, Vol. 10; N°3, July 1988, pp470-502.
`Utilisation des types existentiels pour modéliser l'abstraction des données`

[More 79] Trenchard More, *The nested rectangular array as a model of data*, proc. of APL '79 Coonference, pp ((-73, ACM 1979
`Une axiomatisation (à la manière de l'axiomatisation des ensembles) de la notion de tableau multi-dimensionnel`

[Mycrof 83] A. Mycroft, *Abstract Interpretation and Optimization Transformation for Applicative Programs*, University of Edinburgh, Dept. of Computer Science, december 1983.
`L'évaluation abstraite au secours de la programmation fonctionnelle (analyse de stricticité)`

[Nicol, O'Hallaron 91] David Nicol, David O'Hallaron, *Improved Algorithms for mapping pipelined and parallel Computations*, IEEE Trans. on computers, Viol. 40, N°3, March 1991
`Des algorithmes polynomiaux donnant un placement optimal dans le cas d'architecture pipe-line ou à mémoire partagée.`

[NPA 86] Rishiyur S. Nikhil, Keshav Pingali, Arvind, *Id nouveau*, CSG Memo 265, MIT Laboratory for Computer Science, Cambridge MA, July 1986.
`Un langage data-flow`

[NSDQN 90] Mark Nichols, Howard Siegel, Henry Dietz, Russel Quong, Wayne Nation, *Minimizing Memory Requirements for partitionable SIMD/SPMD Machines*, proc. of the 1990 Int. Conf. On Parallel Processing, Vol I, pp 84-91.
`Techniques d'allocation de la mémoire dans les machines SIMD/SPMD partitionnable afin d'éviter la fragmentation`

[Nugent 88] S.F. Nugent, *The iPSC/2 Direct-Connect Communications Technology*, 3° Conf. on Hypercube Concurrent Computers and Applications, 1988

[NUWG 86] Nicolas Ulrich Wan Grehalm, *How to insert somme dummy things in a bibliography (when it becomes boring)*, Int. Conf. of Banalologie, Nov. 1986

[Pagels 88] H.R., *The dreams of reason: The computer and the rise of the sciences of complexity*, Simon and Schuster, New-York 1988 (ISBN ??)
`Une revue des apports de l'ordinateur dans l'étude « expérimentale » des systèmes complexes.`

[Parhi, Messerschmitt 89] Keshab K. Parhi, David G. Messerschmitt, *Fully static rate-optimal scheduling of iterative dataflow programs via optimum unfolding*, proc. 1989 Int. Conf. Parallel Processing, St. Charles, IL, August 1989.

[Parhi, Messerschmitt 91] Keshab K. Parhi, David G. Messerschmitt, *Static rate-optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding*, IEEE Trans. on Comp., Vol 40, N°2, February 1991.
`L'ordonnancement sans recouvrement des graphe data-flow synchrone peut être optimal après dépliage du graphe de départ si on dispose de suffisament de PE`

[PASM 81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, H. E. Smalley, S. D. Smith, *PASM : a partitioanble SIMD/MIMD System for Image Processing and Pattern Recognition*, IEEE Trans. on Computers, Vol C-30, N° 12, December 1981.
`Une machine multi-processeur avec n mémoires, n unités de traitement et q≠n unités de contrôles.`

[PDP 86] David E. Rumelhart, James L. McClelland and the PDP research group, *Parallel Distributed Processing – Explorations in the microstructuire of Cognition*, Volume 1, MIT Press 1986 (ISBN 0-262-18120-7)
`Une exposition des théories connectionnistes`

[Perrot, Zarea-Aliabadi 86] R. H. Perrot, A. Zarea-Aliabadi, *Supercomputer Languages*, ACM Computing Surveys, Vol. 18, N°1, March 1986.
`Revue de différent langages pour les supercalculateur, et en particulier description de Actus.`

[Plas *&* *al*, 76] A. Plas *&* *al*, *LAU system architecture: a parallel data-driven processor based on single assignement*, Proc. 1976 Int. Conf. Parallel Processing, pp 293-302
`A ma connaissance, la première machine data-flow qui a été réellement construite.`

[Plotkin 81] G. D. Plotkin, *A structural Approach to Operational Semantics*, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981

[Powell 83] Powell, Miller, *Process Migration in Demos/MP*, Proc 9th Operating System Principles, PP 110-119, October 1983

[PPSN 90] H.-O. Schwefel, R. Männer, editors, *Parallel Problem Solving from Nature*, Proceedings of the 1st Workshop PPSN I, Dortmund, FRG, October 1990, LNCS 496, Springer-Verlag
`Les processus d'optimisation distribués, leurs applications et leur implémentation parallèle : algorithmes génétiques, recuit simulé, réseaux de neuronnes, systèmes imunologiques…`

[Prusinkiewicz, Hanan 89] P. Prusinkiewicz , J. Hanan, eds, *Lindenmayer Systems, Fractals and Plants*, Spinger-Verlag 1989.
`Une introduction à l'utilisation des systèmes de Lindenmayer pour la représentation graphique des plantes et la modélisation de leur croissance.`

[Quéau 86] Philippe Quéau, *Eloge de la simulation – de la vie des langages à la synthèses des images*. Editions du Champ-Vallon, collection Milieux, 1986.

[Queinnec 90] Christian Queinnec, *Crystal SCHEME : A language for Massively Parallel Machines*, rapport interne (soumis à PPoPP '91), école Polytechnique.
`Une extension de SCHEME vers le parallélisme à travers un mécanisme d'évaluation distant.`

[Quinton 84] Patrice Quinton, *Automatic synthesis of systolic arrays from uniform recurrent equations*, Proc. 11th Annual Internationnal Symposium on Computer Architecture, June 1984, pp. 208-214.

[Ravi 88] Ravi, Jefferson, *A Basic Protocol to Migrating Processes*, International Conference on Parallel Processing, PP 188-197, August 1988

[Rayward-Smith 87] V.J. Rayward-Smith, *UET scheduling with unit interprocessor communication delays*, Discrete Appl. Math., 18, 1987, pp. 55-71
`Complexité du scheduling avec précédence et coût de communication, proposition d'un algorithme greedy d'ordonnancement avec contraintes de précédences et coût unitaire de communications (cf. extension dans [HCAL 89]).`

[REE 87] D.A. Reed, R.M. Fujimoto, "*Multicomputer Networks - Message-Based Parallel Processing*", The MIT Press, 1987

[Reynolds 85] John Reynolds, *Three approaches to type structure*, in Mathematical Foundations of Software Developments, 1985, Springer-Verlag LNCS 185.
`Un article classique sur les techniques de typages`

[Righter, Walrand 89] Rhonda Righter, Jean C. Walrand, *Distributed Simulation of Discrete Event Systems*, in [IEEE 89]
`Introduction à la simulation distribuée des systèmes dynamiques discrets.`

[Robinson 65] J. C. Robinson, *A machine-oriented logic based on the resolution principle*, Journal of ACM, 12, 1, p. 23-49, Janvier 1965
`L'algorithme d'unification`

[Rocheteau, Halbwachs 91] Fréderic Rocheteau, Nicolas Halbwachs, *Implementing reactive programs on circuits - A hardware implementation of LUSTRE*, article en cours de rédaction, 10 mai 1991.
`Les tableaux dans LUSTRE pour la programmation des PAM`

[SAB 90] Jean - Arcady Meyer, editor, *Simulation of Adaptative Behavior - From Animals to Animats*, proceedings of the 1st SAB conf., 24-28 September 1990, Paris
`Intelligence Artificielle et simulation du comportement naturel, Cf. aussi [Legrand 91] pour une utilisation en` 81/2

[Sabot 88] Gary W. Sabot, *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, Cambridge MA, 1988

[Saltz, Naik, Nicol 87] J. Saltz, U. K. Naik, D. Nicol, *Réduction of the effects of communication delays in scientific algorithms on message-passing architecture*, SIAM J. Sci. Stat. Comp., Vol 8 N° 1, pp. 118-134, 1987

[Sansonnet 90] J.-P. Sansonnet, *Concepts d'Architectures Avancées*, Tome 1 et 2, cours de DEA de l'Université d'Orsay, LRI 1991.
`Les transpatrents du maître`

[Schlag 87] Martine Schlag, *The planar topology of Functional Programs* LNCS 274.
`Le graphe data-flow des programmes fonctionels presente des propriétés topologiques qu'il est possible d'utiliser dans le placement du graphe.`

[Schwartz, Barnwell 85] D. A. Schwartz, T.P. Barnwell III, *Cyclo-static processor scheduling for the optimal realization of shift-invariant flow graphs*, Proc. ICASSP 85, pp 1384-1387, Tampa FL, March 1985.
`Ordonnancements périodiques où les activations successives des tâches peuvent se dérouler sur des processeurs différents`

[Scott 89] Dana S. Scott, *Semantic domains and denotationnal semantics,* Working Material for the lectures of D. S. Scott, Internationnal Summer School on Logic, Algebra and Computation, Marktobendorf, Germany, July 25 - August 6, 1989, Technische Universität München.
`Une synthèse des techniques utilisées dans le domaine de la sémantique dénotationelle`

[SDBS 86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky and E. Schonberg, *Programming with sets: An introduction to SETL*, Springer-Verlag 1986.

[SEI 85] C.L. Seitz, "*The Cosmic Cube*", Com. ACM, vol. 28, n° 1, Jan. 1985, pp 22

[Serafini Ukovich 89] Paolo Serafini, Walter Ukovitch, *A mathematical Model for Periodic Scheduling Problems*, SIAM J. Disc. Math., Vol. 2, N°4, pp550-581, November 1989
`Une présentation du problème de l'ordonnancement répétifs (avec contraintes de précédences) et étude d'un algorithme.`

[Shapiro, Taekuchi 83] E. Shapiro, A. Taekuchi, *Object oriented programming in Concurrent PROLOG*, Journal of new Generation Computing, Vol. 1, 1983
`Parallélisme en PROLOG`

[Sih, Lee 90] Gilbert C. Sih, Edward Lee, *Scheduling to account for interprocessor communication within interconnection-constrained processor networks*, 1990 International Conference on Parallel Processing
`Un algorithme d'ordonnancement s'appliquant à un graphe data-flow statique. Peut se voir comme une particularisation utilisant le chemin critique de la stratégie analysée dans [HCAL 89]. Comparaison de performances avec une stratégie HLFET sur des graphes aléatoires.`

[Sipelstein, Blelloch 91] Jay M. Sipelstein, Guy E. Blelloch, *Collection-oriented languages*, proc. of the IEEE, Vol. 79, N° 4, april 1991
`Une revue du concept de collection à travers divers langages.`

[Skillicorn 90] David Skillicorn, Architecture-Independent Parallel Computation, Computer December 1990.
`Plaide pour une expression implicite du parallélisme indépendante de toute architecture. L'article examine le cas des listes et conclu à l'universalité des constructions parallèles qui sont efficaces sur une PRAM.`

[Skillicorn 91] David Skillicorn, *Stream Languages and Data-flow*, in [Gaudiot, Bic 91 *pp 439-454*]

[Smith 85] G. D. Smith, *Numerical solution of partial differential equations : finite difference methods*, third edition, 1985, Oxford Applied Mathematics and Computing science series, Oxford Univeristy Press (ISBN 0-19-859650-2)
`Un livre classique sur la résolution des équations aux dérivées partielles par les méthodes de différences finies.`

[Smith 88] Smith, *A Survey of Process Migration Mechanisms*, Operating Systems Review, A.C.M Sigops,PP 28-40, July 1988

[Stallings 86] W. Stallings Ed., "*Reduced Instruction Set Computers Tutorial",* IEEE Computer Society Press - 1986
`Le recueil des articles fondateurs du concept RISC`

[Steele 90] Guy Steele, *Making Asynchronous Paramllelism Safe for the World*, 17th Annual Symposium on Principle of Programming Languages, San-Francisco, 17-19 January 1990, pp 218-231.
`Comment garder la simplicité du SIMD tout en gagnant la souplesse du MIMD ?`

[Stroustrup 86] B. Stroustrup, *The C++ Programming Language,* Addison-Weslay, Reading Mass. 1986.
`Une introduction à C++ 1.2.`

[TBH 82] Philip C. Treleaven, David R. Brownbridge and Richard P. Hopkins, *Data-driven and demand-driven Computer Architecture*, Computing surveys, vol. 14, N° 1, March 1982, p. 94-143.

[Tesler, Enea 68] L.G. Tesler, H.J. Enea, *A language design for concurrent processes*, AFIPS Conference Proceedings, vol. 32, pp 403-408, 1968
`Première apparition répertoriée des langages à assignation unique`

[Thakkar 87] S. S. Thakkar editor, *Selected reprints in dataflow and reduction architectures*, IEEE Computer Society Press, 1987.

[Theimer 85] Theimer, *Preemptable Remote Execution Facilities for the V-Systems*, Proc of the 10th Operating System Principles, PP 2-12, December 1985

[Thom 83] René Thom, *Paraboles et catastrophes*, Flammarion 1983

[Thom 91] René Thom, *Prédire n'est pas expliqué*, collection La Question, entretien avec E. Noel, 1991.

[Toffoli, Margolus 87] Tommaso Tofooli and Norman Margolus, *Cellular Automata Machine*, The MIT Press, Cambridge MA, 1987

[Vautherin 88] Vautherin, Millot, *Dynamic Creation of Processes on Transputer Network*, Rapport de Recherche du L.R.I N°433, Université de Paris-Sud Orsay, July 1988

[Veltman, Lageweg, Lenstra 90] B. Veltman, B.J. Lageweg, J.K. Lenstra, *Multiprocessor scheduling with communication delays*, Parallel Computing N° 16, pp 173-182, 1990
`Taxonomie des problèmes de placements et ynthèse de la littérature sur leurs complexités`

[Wadge, Ashcroft 85] W. W. Wadge and E.A. Ashcroft, *Lucid, the Data flow Programming Language*, Academic Press U.K., 1985.
`Un livre sur LUCID et ses derniers développement.`

[Walker 83] Walker *& al*, *The Locus Distribued Operating System*, Proc of the 9th Symposium on Operating System Principles, 1983

[Walliser 77] Bernar Walliser, *Systèmes et modèles – Introduction critique à l'analyse de systèmes*, Editions du Seuil, Paris 1977 (ISBN 2-02-004638-5)
`Un exposée introductif à l'idéologie systémique`

[Waters 91] Richard C. Waters, *Automatic Transformation of Series Expressions into Loops*, ACM Trans. on programming Languages and Systems, Vol. 13, N°1, Jannuary 1991, pp 52-98.
`Une algèbre de séquences (i.e. suites), leur optimisatiuons et et leur transformations en des boucles`

[Watson, Gurd 82] I. Watson, J. Gurd, *A practical data-flow computer*, Computer 15 (2), February 1982

[Watson, Watson 87] P. Watson, I. Watson, *An efficient garbage collection scheme for parallel computer*, proc. of PARLE II, LNCS 259.

[Weisbuch 89] Gérard Weisbuch, *Dynamique des systèmes complexes - Une introduction aux réseaux d'automates*, InterEditions/Editions du CNRS, 1986 (ISBN 2729602305)

[Zayas 87] Zayas, *Attacking the Process Migration Bottleneck*, Proc of the 11th Symposium on Operating System Principles, PP 13-24, November 1987

[Zeigler 89] Bernard P. Zeigler, *DEVS Representations of Dynamical Systems : Event-based Intelligent Control*, in [IEEE 89]
`Un formalisme pour la représentation des systèmes dynamiques discrets`

[Zima 91] H. Zima (with B. Chapman), *Supercompilers for Parallel and Vector Computers*, ACM Press, 1991, Addison-Wesley Publishing Company (ISBN 0-201-17560-6).
`Un livre dédié aux techniques d'extraction automatique du parallélisme.`

Bibliographie partielle de l'équipe Architecture du LRI et du projet MEGA

[BCE 91] J.-L. Béchennec, F. Cappello, D. Etiemble, *A 3D Hardware Package for Highly Parallel Architectures*, EuroMicro 91, 3-4 September 1991, Vienna.

[BCN 91] J-L Béchennec, C. Germain, V. Neri, *An Efficient Hardwired Router for a 3D mesh Interconnection Network*, CompEuro 91, Bologne 13-16 May 1991

[BCNE 90] J-L Béchennec, C. Chanussot, V. Neri, D. Etiemble, *VLSI design of a 3D highly parallel message-passing architecture*, Workshop on VLSI for IA and neuronal application, Cambridge, September 1990.

[Cappello 89] F. Cappello, *Conception et simulation de l'unité de traitement d'une machine d'expérimentation des grandes architectures (MEGA)*, DEA d'Informatique de l'université de Paris-Sud, LRI, Juin 1989

[CBE 90] F. Cappello, J-L Béchennec, D. Etiemble, *A RISC Central Processing Unit for a Massively Parallel Architecture*, EUROMICRO 90, Amsterdam, August 90

[CBE 91] F. Cappello, J-L Béchennec, D. Etiemble, *Un réseau de processeurs à géométrie compilable pour les applications de traitement numérique intensif*, 3ème Symposium sur les Architectures Nouvelles de Machines, Palaiseau, 19-22 juin 1991

[CNB 91] F. Cappello, V. Neri, J-L Béchennec, *Design of the dynamic topology communication network for PTAH 64*, Computer Architecture Conference, october 24-26 1991, University of Lecce, Italy

[Delaplace, Giavitto 91] F. Delaplace, J-L Giavitto, *An efficient routing strategy for process migration*, EuroMicro 91, 3-4 September 1991, Vienna.

[Dumesnil 91] S. Dumesnil, *Implémentation de réseaux de neuronnes sur une machine PTAH*, DEA d'Informatique de l'université de Paris-Sud, LRI, Septembre 1991

[ECN 90] D. Etiemble, C. Chanussot, V. Néri, *4-valued BiCMOS Circuits for the Transmission System in a Massively Parallel Architecture*,20th International Symposium on Multiple-Valued Logic", Charlotte, USA, May 90

[Etiemble 90] D. Etiemble, *Design comparison parameters*, in « High Speed Digital IC Technology », Marc Rocchi editor, Artech House, 1990

[Etiemble 91a] D. Etiemble, *Technology impact on CPU performances*, Computer Architecture Conference, october 24-26 1991, University of Lecce, Italy

[Etiemble 91b] D. Etiemble, *Architecture des processeurs RISC*, Armand-Colin, 1991

[Etiemble, Israel 91] D. Etiemble, M. Israel, *Architecture des ordinateurs : Une approche quantitative*, édition française du Henessy, Patterson « Computer Architecture: a quantitative approach», MacGraw Hill

[Fowler 90] J. Fowler, *Study of algorithms adapted to a network of dynamic processes*, rapport de stage européen "Erasmus" Université d'Edimburgh/Université de Paris-Sud, september 1990

[Fuster, Roger 90] P. Fuster, E. Roger, *Simulation Fonctionelle de l'unité de gestion des communications de MEGA*, DEA d'Architecture des Machines Informatiques Nouvelles de l'université de Paris-Sud, LRI, Septembre 1990

[GBES 89] C. Germain, J-L Béchennec, D. Etiemble, J-P. Sansonnet, *A New Communication Design for Massively Parallel Message-Passing Architectures*, IFIP Working Conference on Decentralized Systems, Lyon, December 89

[GBES 90] C. Germain, J-L Béchennec, D. Etiemble, J-P. Sansonnet, *An interconnection network and a routing scheme for a massively parallel message-passing multicomputer*, 3rd Symposium on Frontiers of Massively Parallel Computation, Oct 8-10 1990, College Park, MD

[Germain 89] C. Germain, *Etude des Mécanismes de Communication de la Machine MEGA*, Thèse de l'Université Paris Sud, Decembre 1989

[Germain 90] C. Germain, *Une stratégie de routage pour la machine à parallélisme massif MEGA*, 2ème Symposium sur les Architectures Nouvelles de Machine, Toulouse, Sept 90.

[Germain, Giavitto 90] C. Germain, J-L Giavitto, *A Comparison of two routing strategies for massively parallel computers*, 5° Int. Symp. on Computer and Information Sciences, Cappadocia, Nov 90

[Germain,Sansonnet 89a] C. Germain, J-P. Sansonnet, *L'Architecture des Machines Massivement Parallèles pour l'Intelligence Artificielle*, 11 ème Journées Francophones sur l'Informatique, Février 1989, pp 35-51

[Germain,Sansonnet 89b] C. Germain, J-P. Sansonnet,
"L'Architecture des Machines à Parallélisme Massif",
Entretiens Science et Défense, Simulations Numériques, pp 153-164, 23-24 Mai 89.

[Germain, Sansonnet 91] C. Germain, J.-P. Sansonnet, *Les ordinateurs massivements parallèles*, Armand-Colin, 1991

[GGB 90] C. Germain, J.-L. Giavitto, J.-L. Bechenec, *Rapport intermédiaire du contrat DRET 89/232 : étude d'un système de communications tridimensionnel pour calculateurs massivement parallèle*, Octobre 90

[GGB 91] C. Germain, J.-L. Giavitto, J.-L. Bechenec, *Rapport final du contrat DRET 89/232 : étude d'un système de communications tridimensionnel pour calculateurs massivement parallèle*, Mars 91

[GGF 91] J-L Giavitto, C. Germain, J. Fowler, *OAL : an implementation of an Actors language on a massively parallel message-passing Architecture*, 2° European Distributed Memory Computing Conf., 22-24 Avril 91, Münich, LNCS 487, Springer-Verlag

[GGS 91] C. Germain, J-L Giavitto, J-P Sansonnet, *Implémentation d'un Paradigme de Programmation fonctionnelle sur une Machine Massivement Parallèle*, JFLA 91, Gresse en Vercors, Janvier 1991

[Giavitto 91a] J-L Giavitto, *81/2 : un langage data-flow synchrone pour une machine massivement parallele* 3ème Symposium sur les Architectures Nouvelles de Machines, Palaiseau, 19-22 juin 1991

[Giavitto 91b] J-L Giavitto, *A synchronous data-flow language for massively parallel computers,* Parallel Computing 91, 3-6 September, London 1991

[Giavitto 91c] J-L Giavitto, *Un modèle pour la simulation massivement parallèle des systèmes dynamiques discrets*, Rapport préliminaire DRET, Octobre 1991

[Giavitto 92] J-L Giavitto, Un langage data-flow synchrone pour la simulation des systèmes dynamiques, JFLA'92, Bigre+Globule, Perros-Guirec, 3-4 février 1992.

[Legrand 91] F. Legrand, *Implémentation d'un langage data-flow synchrone pour la simulation des systèmes dynamiques discrets*, DEA d'Architecture des Machines Informatiques Nouvelles de l'université de Paris-Sud, LRI, Septembre 1991

[Mercouroff 90] W. Mercouroff, *Architecture matérielle et logicielle des ordinateurs et des microprocesseurs*, Armand-Colin, 1990

Table Analytique