Introduction à 8½

version 1.0

Un langage parallèle déclaratif de streams et de collections pour la modélisation et la simulation des systèmes dynamiques

Jean-Louis GIAVITTO et Jean-Paul SANSONNET

ÉQUIPE ARCHITECTURES, LRI, U.R.A. 410 DU C.N.R.S., UNIVERSITE DE PARIS-SUD – ORSAY

Pour contacter les auteurs :

LRI, Bâtiment 490
Université de Paris-Sud
91405 Orsay cedex

par courrier électronique:
giavitto@lri.lri.fr
jps@lri.lri.fr

Ce document, « Introduction à 81/2 », a été achevé d'imprimer en avril 1994 par l'Université de Paris-Sud, centre d'Orsay.

Table des matières

∴

A propos de ce document

Avertissement au lecteur

Ce document est une présentation informelle des principaux concepts du langage 81/2. Cette présentation s'adresse à un lecteur informaticien, qui connaît un peu les langages `C` et Lisp et qui possède quelques notions sur les langages de programmation, la simulation et les architectures parallèles (Cf. [SAN 91] et [GIA 93]), comme par exemple un étudiant en DEA. Ce document n'est pas un exposé formel du langage 81/2, pour cela on pourra se référer à [GIA 91b]. Il ne s'agit pas non plus d'un manuel du langage 81/2, on consultera pour cela [MIC 94]. Notre objectif est de présenter le panorama très riche des concepts que l'on rencontre lorsqu'on veut explorer les structures de *stream* et de *collection*, en particulier :

- dans le cadre de la programmation parallèle déclarative,
- pour la simulation des systèmes dynamiques,
- en vue de leur mise en œuvre efficace (compilation) sur les architectures de machines séquentielles et parallèles.

Au cours de notre présentation, nous rencontrerons des notions techniques qui ne pourront qu'être brièvement évoquées. Le lecteur pourra se référer à la bibliographie pour trouver les approfondissements nécessaires.

Synoptique du langage 8½

Le langage 8½ est un *langage parallèle de haut-niveau pour la modélisation et la simulation des systèmes dynamiques*. Nous regroupons sous le nom très général de Système Dynamique (SD) les phénomènes décrits par un *état* qui évolue dans le *temps* et dans l'*espace*. Pour modéliser et simuler les systèmes dynamiques sur les nouveaux ordinateurs parallèles, nous proposons un langage expérimental qui possède les caractéristiques suivantes :

- c'est un langage permettant d'exprimer des relations temporelles et spatiales entre les valeurs des états d'un système dynamique ;
- c'est un langage de haut-niveau ;
- c'est un langage parallèle.

Ces trois caractéristiques se traduisent par les propriétés suivantes :

– 8½ permet de construire des *streams*. Un stream est une suite de valeurs au cours du temps. Un stream implémente la notion de trajectoire d'un SD.

– 8½ permet de construire des *collections*. Une collection est un ensemble de valeurs, simultanément accessibles, et qu'on manipule comme un tout. Une collection implémente la notion d'état d'un SD.

– 8½ est un langage *déclaratif* : un programme est un "paquet" d'équations qui modélise le système dynamique. L'exécution du programme correspond à la simulation du système modélisé : l'exécution consiste à énumérer successivement les états du système au cours du temps (i.e. calculer sa trajectoire).

– 8½ est un langage data-flow et data-parallèle. Le parallélisme de contrôle et le parallélisme de données sont exprimés implicitement à travers les données du programme. Il est donc possible d'utiliser au mieux les capacités des nouvelles architectures parallèles SPMD ou MSIMD.

– L'extraction du parallélisme, le placement et l'ordonnancement des calculs est déterminé automatiquement à la compilation.

– Des techniques modernes de typage sont utilisées pour alléger le travail du programmeur, pour vérifier la cohérence des équations et permettre une compilation efficace.

Organisation du document

Les chapitres de ce document correspondent à la description des applications visées par le langage expérimental 81/2, à la description du langage, à la discussion de l'adéquation d'un tel langage pour la programmation des nouvelles architectures parallèles, et à la présentation des principes de compilation du langage.

Plus précisément, ce document s'organise ainsi :

- **Le premier chapitre** est une introduction à la simulation. La simulation est présentée sous deux aspects : le premier, correspondant à l'idée d'expérience numérique, est le plus traditionnel. Nous exposons un seconde approche de la simulation, où elle peut être vue comme un nouveau style de programmation : ce style de programmation s'appuie sur le fait que la structure interne des phénomènes physiques en font potentiellement des "calculateurs sophistiqués".

  Pour cela, il faut développer des langages permettant la description à un haut-niveau des systèmes dynamiques. Nous proposons de définir la trajectoire d'un système (i.e. la suite de ses états dans le temps et l'espace) dans un langage déclaratif. Nous examinons alors les différents modèles du temps sur lesquels on peut fonder cette notion de trajectoire et nous choisissons un modèle du temps discret et synchrone. Le chapitre se termine par le récapitulatif des propriétés désirées pour 81/2.

- **Le deuxième chapitre** élabore plus précisément la notion de trajectoire. Une trajectoire correspond en 81/2 à un *stream*. Un stream est une suite temporisée définie par des expressions séquentielles à mémoire bornée. La notion de suite temporisée est introduite à l'aide d'un exemple. Différents opérateurs présentant les propriétés de séquentialité et de mémoire bornée sont présentés. Leur calcul d'horloge, notion reliée à l'interprétation temporelle des suites, est introduit.

  Ensuite, la définition des streams par des équations est abordée. Le problème posé par les équations récursives est détaillé. Nous examinons la multiplicité des solutions et indiquons comment choisir une solution canonique sur la base du critère fourni par une relation d'ordre.

- **Le troisième chapitre** présente la notion de collection. Une collection est un ensemble de données manipulées comme un tout. Les opérations sur les collections sont présentées. Celles-ci se divisent en deux catégories : les opérations construites à partir des opérations disponibles sur les éléments d'une collection (alpha-extension scalaire, réduction, etc.) et les opérations géométriques qui manipulent la structure d'une collection (projection, concaténation, etc.). Nous donnons ensuite un aperçu sur les algèbres de Bird-Meertens qui donnent un fondement technique à la notion de collection.

  Nous examinons ensuite la définition des collections par des équations récursives. Si nous voulons que la valeur de chaque élément de la collection soit bien définie, il faut interdire certaines expressions récursives : c'est la notion d'admissibilité d'une définition récursive. Plusieurs exemples sont présentés et la notion de collection récursive est confrontée à la notion de fonction récursive.

- **Le quatrième chapitre** combine la notion de stream et de collection, en une structure unique, le tissu. Un tissu est l'objet construit par un programme 81/2. Le chapitre débute par une présentation de la structure tissu. Puis nous montrons que les tissus, étendus aux collections hétérogènes et explicitement nommées, représentent des "paquets" d'équations. Ces tissus, nommés systèmes,

permettent le regroupement des équations en systèmes hiérarchisés et les débuts d'une algèbre des systèmes sont présentés.

Le chapitre se termine par une série d'exemples qui illustrent la programmation en 81/2. Ces exemples utilisent une première version d'un environnement de programmation 81/2, appelé *8,5* qui est présenté en annexe.

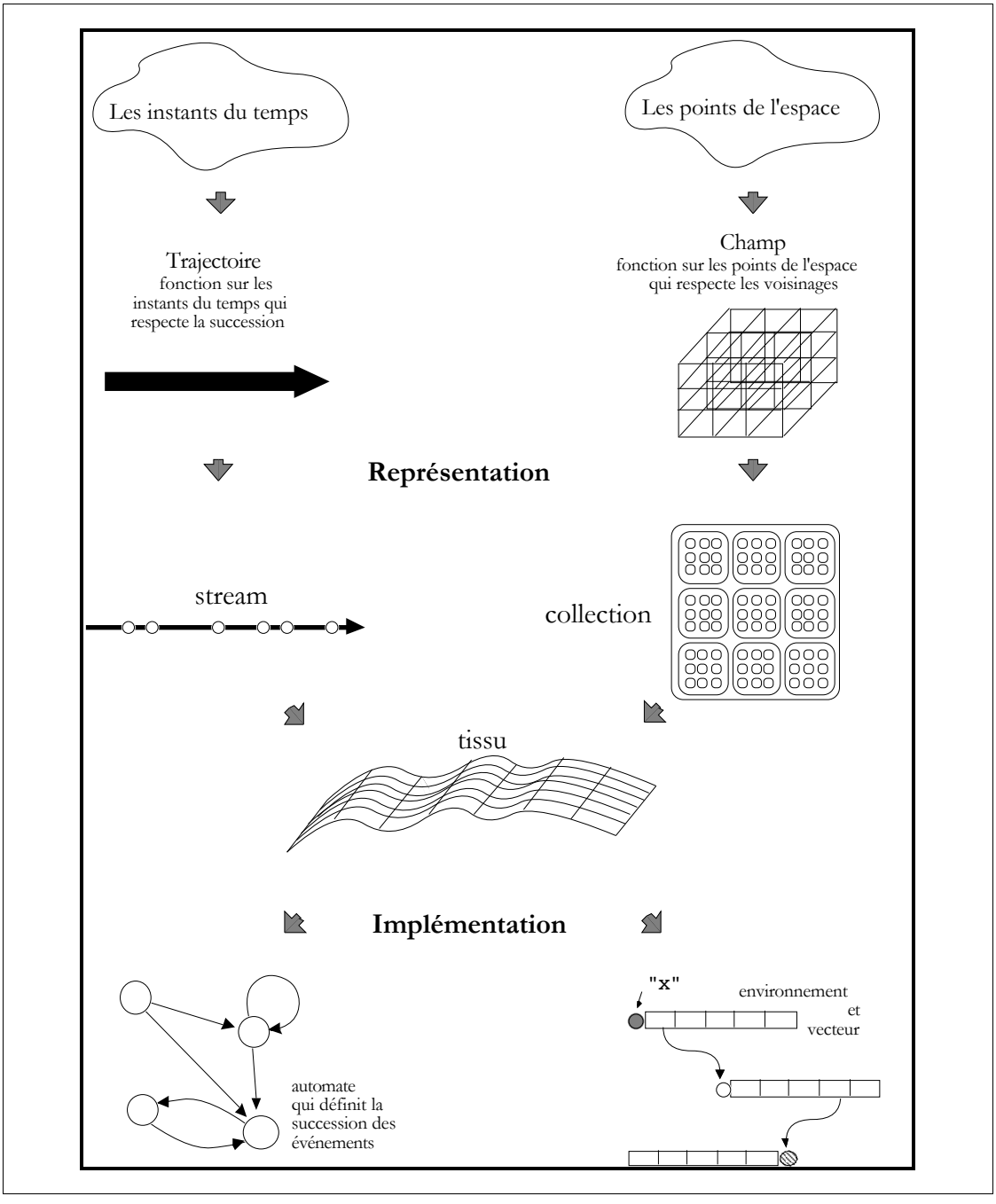
- **Le cinquième chapitre** examine l'adéquation d'un langage déclaratif à la programmation parallèle. Dans le cas de 81/2, l'aspect déclaratif se traduit par un modèle d'exécution statique data-flow. Par ailleurs, le data-parallélisme s'exprime à travers la définition des collections.

  Ce chapitre débute par une classification générale des langages parallèles. Nous examinons ensuite, après une rapide présentation, les avantages et les inconvénients d'une approche data-parallèle séquentielle, tant en termes de sémantique que d'efficacité de l'implémentation. Notre conclusion est qu'il faut exprimer le data-parallélisme à travers la notion de collection plutôt qu'à travers des structures de contrôle ad hoc, et quitter le cadre du contrôle de flot séquentiel. Nous choisissons pour cela le modèle de contrôle data-flow, que nous exposons brièvement. Les liens entre le langage déclaratif 81/2 et le modèle d'exécution data-flow sont alors présenté. Nous décrivons ensuite les inconvénients de l'approche data-flow et concluons à la nécessité de développer des modèles d'exécution data-flow compilés. Nous détaillons alors les bénéfices que l'on peut attendre d'un mariage data-parallélisme plus data-flow, vis à vis des autres approches possibles pour l'expression et l'exploitation du parallélisme, et ceci, au regard des nouvelles architectures massivement parallèles à contrôle hybride.

  La deuxième partie de ce chapitre présente la compilation de 81/2 et montre comment en faire un langage data-parallèle déclaratif compilé. Notre technique de compilation s'appuie sur un modèle d'exécution statique data-flow et data-parallèle, ce qui permet l'exécution très efficace d'un langage de haut-niveau sur des architectures cibles variées. Nous présentons succinctement les différentes phases de la compilation en mettant l'accent sur le calcul des horloges, le calcul et la résolution des dépendances et nous finissons par la présentation de la synthèse d'un placement/ordonnancement sur une architecture massivement parallèle.

- **Le chapitre final** esquisse différentes extensions possibles du modèle 81/2. En particulier, on y discute de représentations plus complexes du temps et de l'espace.

Ce document se complète par une annexe qui présente l'environnement de programmation *8,5* et par une bibliographie.



Organisation de ce document.
Le chapitre I s'attache à décrire les notions de trajectoire et de champ, correspondant aux variations d'une valeur respectivement dans le temps et dans l'espace. Les chapitres II et III vont élaborer une représentation informatique de ces notions et nous définirons les streams 8½ et les collections 8½. Le chapitre IV fusionnera ces deux représentations en une structure informatique unique, le tissu *8½. Un tissu 8½ représente les variations d'une valeur dans l'espace et dans le temps. Les tissus sont implémentés (de manière parallèle, Cf. le chapitre V) en utilisant des environnements, des vecteurs et des automates.*

I. Simulation : programmer le langage de la nature

The Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves.[1]
Ada Lovelace

I.1. Introduction

La puissance de calcul et la souplesse d'utilisation des ordinateurs a permis de rendre réaliste l'idée d'*expérience numérique*. Il s'agit, par la simulation digitale, de calculer des phénomènes plutôt que de les observer dans le monde réel. Cette approche révolutionne la pratique de nombreuses sciences. Il est donc essentiel de développer des langages de programmation permettant d'exprimer directement un modèle et adaptés à la simulation. De ce point de vue, les langages équationnels, fondés sur des descriptions mathématiques des modèles à simuler, offrent un formalisme le plus proche possible des modèles mathématiques de départ.

Il est cependant tentant d'inverser la perspective et de faire de la simulation, non une application qui motive la création de langages spécifiques, mais *un nouveau paradigme de programmation*. Dans cette perspective, les phénomènes du monde physique constituent, sous une version idéalisée, les objets d'un nouveau calcul. Ce point de vue, fertile, motive l'invention de nouveaux algorithmes comme par exemple les procédures de recuit simulé, les algorithmes génétiques, les réseaux neuro-mimétiques, et participe plus généralement d'un domaine de recherche nouveau, identifié sous le nom de "Parallel Problem Solving from Nature" ou **PPSN** [SCH 90].

Ces deux approches s'appuient sur des modèles qui décrivent un fragment de monde physique. Il faut donc pouvoir décrire l'évolution de phénomènes physiques qui prennent place *dans le temps et dans l'espace*. Cette « maquette numérique du monde » est une fin en soi dans le cas de la simulation classique, ou constitue un procédé de calcul dans le cas de la PPSN.

---

[1] « La machine analytique tisse des motifs algébriques de la même manière que le métier Jacquard tisse des fleurs et des feuilles. ». La machine de Jacquard est un métier à tisser automatique, piloté par des cartes perforées lui permettant de produire des motifs variés programmés sur les cartes. Ces paroles, de Ada Lovelace en 1834, décrivent la « machine analytique » que voulait construire Charles Babbage après avoir conçu et essayé de réaliser une machine à calculer les différences finies.

Dans ce chapitre, nous voulons décrire ces deux approches de la simulation et en tirer les implications pour un nouveau langage de programmation. Nous verrons que ce langage doit être déclaratif et qu'il doit permettre la description de valeurs qui varient dans le temps et dans l'espace. Ceci pose la question de la représentation de ces deux concepts dans un langage de programmation, et cela conduit à un excellent support pour la programmation parallèle.

I.2. Modéliser, simuler, programmer

I.2.1. De l'importance de l'expérience numérique

La puissance de calcul et la souplesse d'utilisation des ordinateurs a permis de rendre réaliste l'idée d'*expérience numérique*. Il s'agit, par la simulation digitale, de calculer des phénomènes plutôt que de les observer dans le monde réel. Cette approche révolutionne la pratique de nombreuses sciences. En effet, la modélisation, et la simulation qui permet de d'expérimenter et d'explorer les conséquences d'un modèle, ont un rôle central dans des activités aussi diverses que [AMS 93] :

- *Comprendre*

  Scientifiques et ingénieurs tentent continuellement de modéliser des systèmes qu'ils ne comprennent pas ou qu'ils ne maîtrisent qu'imparfaitement. La simulation du modèle permet d'observer le fonctionnement du modèle. Un modèle et sa simulation peuvent induire des intuitions sur la compréhension du système réel ou peuvent restreindre la recherche de phénomènes particuliers permettant de caractériser le système.

- *Prédire*

  Si le modèle d'un système peut être simulé plus rapidement que l'évolution du système réel, alors le modèle peut être utilisé pour prédire le comportement futur du système, ce qui est utile pour la prévision (météorologie, économie) la prévention (par exemple des tremblements de terre) et le contrôle (par exemple pour maintenir le comportement d'une centrale nucléaire dans des limites prédéterminées).

- *Concevoir et construire*

  Inventer un nouveau mécanisme ou construire un ouvrage d'art sont des activités complexes qui font appel à la modélisation et à la simulation afin de s'assurer que l'objet créé répond bien aux spécifications et exhibe le comportement attendu, avec une efficacité et un coût donnés.

- *Diagnostiquer*

  La détermination de ce qui ne fonctionne pas dans un système est facilitée si l'on possède un modèle du fonctionnement correct du système, mais aussi si l'on dispose d'un modèle de fonctionnement dans divers modes de défaillance.

- *Tester*

  Un modèle facilite le développement des tests permettant de caractériser ou de valider un système.

Il est donc important de développer des langages de programmation permettant d'exprimer directement un modèle et adaptés à la simulation. Dans un tel langage,

programmer = **modéliser** un fragment de monde
exécuter un programme = **simuler**

La modélisation est donc vue comme la "description constructive" d'un fragment de monde physique. Cette description est dite constructive (on dit parfois exécutable ou effective) car il est possible de construire automatiquement l'évolution de l'état du monde à partir de cette description.

Nous reviendrons sur ces notions d'état du monde et d'évolution un peu plus loin, mais nous pouvons déjà noter qu'un langage de programmation réellement adapté à la simulation s'éloigne des langages classiquement utilisés dans ce domaine, comme par exemple FORTRAN. En FORTRAN, programmer est synonyme de commander les actions d'une machine de Von Neumann (modifier le contenu d'une mémoire, contrôler une unité de calcul arithmétique et logique, etc.) et ne permet pas de décrire directement des phénomènes physiques. Un langage permettant d'exprimer directement les équations mathématiques qui décrivent le modèle à simuler, est un langage *de haut-niveau* car il permet l'expression directe des concepts de l'application sans passer par une représentation ou un codage à travers d'autres concepts.

I.2.2. Vers un nouveau paradigme de programmation

L'expérience numérique motive le développement de langages de programmation de haut-niveau dédiés à un domaine d'application. Mais il est tentant d'inverser la perspective. Simuler devient alors *un nouveau paradigme de programmation* et on peut proposer le parallèle :

description d'un monde	**=**	**programme**
paramètres de la description	**=**	**données** du programme
phénomènes qui prennent place dans le monde décrit	**=**	**résultats** d'un calcul

Dans cette approche, la simulation est utilisée pour sa similitude avec le calcul à effectuer. Donnons quelques exemples d'une telle approche :

- *Calcul d'un pgcd* [STE 64] [ROZ 87]

  Sur un billard idéal, dont les côtés sont égaux aux deux nombres dont on cherche le pgcd, une boule lancée depuis un des coins avec un angle de 45°, atteint un des trois autres coins du billard après un nombre fini de rebonds. La distance de l'impact le plus proche du point de départ de la boule, sur le grand côté du billard, est égal au double du pgcd.

- *Calcul d'un optimum*

  La méthode du recuit simulé permet de calculer l'optimum d'une fonction. Cette méthode simule le refroidissement graduel d'un objet physique qui minimise alors l'énergie de son état, pour calculer la valeur qui minimise une fonction donnée. Dans la même ligne, on peut citer les algorithme génétiques qui miment le processus de l'évolution des espèces pour construire un optimum.

- *Le calculateur à spaghetti* [DEN 84]

  Pour trier une liste de nombres, on peut commencer par couper, pour chaque nombre, un spaghetti cru à une longueur égale à ce nombre. On réunit ensuite les spaghettis en une botte dont on plaque brusquement la base contre une table. En raison de la quantité de mouvement de chaque spaghetti, on obtient une botte dont chaque brin aura l'extrémité inférieure en contact avec la table. Il suffit alors de sélectionner le plus long spaghetti qui dépasse, puis le plus long de ceux qui restent, et ainsi de suite. Chaque fois qu'on retire un spaghetti, on mesure et on note sa longueur, ce qui permet d'obtenir les nombres de la liste initiale, triés dans l'ordre décroissant.

- *Une théorie kinesthésique de l'intelligence*

  Un certain nombre de chercheurs en sciences cognitives insistent sur le fait que l'intelligence ne peut être séparée de l'expérience subjective d'un corps [BRO 93]. Par exemple [STE 94] propose une théorie dans laquelle un acte cognitif correspond à la remémoration, par l'imagination, d'une action corporelle dans le monde. Si l'on met en œuvre ce point de vue dans un robot, une action correspond aux interactions d'un ensemble de boucles de rétroaction senseur/actuateur avec un environnement physique et un acte cognitif élémentaire correspond à la simulation (par le robot) d'une action du robot.

Dans ces exemples, la simulation d'un phénomène physique sert à calculer *autre chose que la simulation elle-même*, parce qu'il y a une analogie formelle entre le modèle physique que l'on va simuler et le problème à résoudre. Plus précisément, l'évolution d'un système fournit "naturellement" la solution d'un problème à travers les phénomènes qui s'y déroulent (exemple des chocs de la boule de billard sur la bande latérale), par l'état final qu'il atteint (exemple du recuit simulé), etc. : le résultat (re)cherché doit être "extrait" de la simulation du système.

I.2.2.a. Des phénomènes physiques pour calculer

Emprunter un phénomène physique pour faire un calcul peut sembler inutilement compliqué. Par exemple, simuler un billard pour calculer un pgcd apparaît comme alambiqué à la vue de la simplicité de l'algorithme classique. Cependant, cette méthode vaut aussi pour des côtés qui ne sont pas en rapport entier, ce qui permet d'étendre la notion de pgcd. De plus, un calculateur capable de simuler des billards virtuels peut calculer un pgcd sans avoir la capacité de savoir faire des soustractions. On peut aussi noter que le monde physique étant parallèle (des phénomènes différents ont lieu en même temps), la simulation d'un modèle du monde physique sera naturellement parallèle[1]. Une première raison pour calculer à travers une simulation est donc que cela permet d'obtenir de nouveaux algorithmes, avec d'autres propriétés que les algorithmes classiques : les nouveaux algorithmes sont souvent parallèles et d'une complexité différente des algorithmes standards, même si très souvent leur implémentation réelle sur un ordinateur digital conduit à un résultat de même complexité que les algorithmes standards[2].

I.2.2.b. Calculer les phénomènes physiques sur un ordinateur

Une deuxième raison pour "calculer avec des phénomènes physiques" est qu'intuitivement une bonne partie des problèmes que nous nous posons est issue du monde physique ou s'y rapporte. Il est donc naturel de trouver une solution par la simulation de ces phénomènes. Par exemple, le problème de la prévision peut se ramener à réaliser une simulation plus rapide que le phénomène réel à prédire ; par exemple, la prédiction du temps qu'il fera fait appel à un programme qui simule l'évolution des conditions météorologiques sur un superordinateur.

I.2.2.c. Du "phénomène physique" comme structure fondamentale de calcul

L'approche fondamentalement métaphorique de la PPSN peut paraître inadéquate dans le cas d'un problème qui n'a pas de relation directe avec le monde réel, comme par exemple la gestion d'une base de données. Mais les exemples précédents montrent qu'un phénomène physique peut parfaitement servir de "solveur" et résoudre un problème abstrait (par exemple, trier est une opération faite naturellement par les spaghettis).

---

[1] Par exemple, dans le cas de l'exemple du calcul du pgcd par une boule de billard, il est possible de simuler la physique du billard par un automate cellulaire : voir par exemple la « programmation plane » [ROZ 87]. Sur cet automate, une boule de billard est représentée par une certaine configuration des cellules. Si les boules de billard ne se croisent pas, on peut faire plusieurs calculs en même temps, puisque l'espace du billard permet de faire évoluer en même temps plusieurs boules. On peut régler le problème des chocs des boules entre elles en utilisant des configurations qui peuvent se croiser sans interagir (comme des solitons). On peut rendre ces boules sensibles à des bords différents (i.e. les bandes latérales, qui correspondent à un certain état de certaines cellules, ne sont visibles que pour un type donné de boules, et il y a autant de type de boules que de pgcd à calculer). On peut ainsi calculer autant de pgcd en parallèle que l'on veut.

[2] Par exemple, on peut distinguer trois phases dans la méthode de classement par spaghetti : le pré-traitement, la phase analogique et le post-traitement. Le pré- et le post-traitement sont linéaires en la longueur n de la liste de nombres à trier. La phase analogique utilise le comportement d'un système physique qui atteint très rapidement sa position d'équilibre et a un coût constant. La complexité du tri par spaghettis est donc linéaire ($2n+1$ à comparer à la complexité en $n\text{Log}n$ de l'algorithme de tri "rapide" classique).

En effet, l'objet défini par la variation d'une quantité dans le temps et dans l'espace, objet qui est le fondement des simulations de phénomènes physiques, est un objet "universel". Une analogie nous permettra de faire comprendre ce que nous entendons par là. Il est classique en informatique, de fonder la programmation sur un concept suffisamment puissant pour permettre d'émuler toute autre notion qui serait nécessaire à la réalisation d'un calcul. Par exemple on peut fonder le calcul sur les lambda-expressions. On peut aussi fonder le calcul sur des algèbres d'arbres avec les types abstrait algébriques, ou bien sur la logique avec les clauses de Horn.

Nous défendons l'idée que la structure qui représente la variation d'une quantité dans le temps et dans l'espace, est une structure aussi fondamentale que les lambda-expressions, les arbres ou les clauses de Horn.

I.3. Un langage pour la simulation et pour la PPSN

Notre objectif est donc de développer un langage parallèle de haut-niveau pour la simulation des phénomènes physiques, et qui serait un véhicule possible pour le développement d'un nouveau style de programmation (nommément : « la résolution parallèle de problèmes par simulation de phénomènes naturels », ou PPSN acronyme de « Parallel Problem Solving from Nature »).

I.3.1. Un langage déclaratif

I.3.1.a. Programmer avec des équations

Dans le paradigme de programmation « Programmer = Simuler » nous avons suggéré qu'un programme correspondait à la description d'un fragment de monde (qui peut être abstrait ou réel). Un langage de programmation dans lequel un programme correspond à la description d'un certain objet, est un *langage déclaratif*. Dans un tel langage, un objet est décrit par des équations et l'exécution d'un programme, qui est un système d'équations, correspond à la construction des objets qui sont solutions des équations.

I.3.1.b. Exemples de langages de programmation déclaratifs

Il existe plusieurs exemples de langages qui répondent à cette approche. Par exemple :

- En FP [BAC 78], on construit des fonctions : un programme est une fonction définie par des équations et l'exécution d'un programme est l'application de cette fonction à des arguments. Il peut y avoir plusieurs fonctions qui sont solutions des équations fonctionnelles, mais la fonction qui est effectivement appliquée, est *la plus petite* au sens d'une certaine relation d'ordre entre les fonctions.
- En PROLOG, on construit l'ensemble des valeurs de vérité d'un prédicat : un programme est un système logique et une exécution du programme énumère les solutions de ce système logique. Contrairement à FP, le but d'un programme PROLOG est de calculer toutes les solutions.
- En **Γ** [BAN 88] [BER 89], ont construit des "réactions chimiques" qui agissent sur des "milieux chimiques". Un milieu chimique est un multi-ensemble (i.e. un ensemble admettant des répétitions d'éléments) dont les éléments sont les « molécules ». Une réaction chimique indique quelles molécules réagissent ensembles et quelles est la molécule résultant de la réaction.
- Les langages data-flow, comme par exemple LUCID [WAD 85], manipulent des suites finies ou infinies de valeurs. Un programme LUCID est un système d'équations qui définit des relations entre des suites de valeurs en entrée du programme et les sorties du programme. L'exécution d'un programme conduit à énumérer les valeurs des suites en sortie. Il existe toujours une solution à un programme LUCID, cette solution étant éventuellement la suite vide. Il peut exister plusieurs suites

solutions : dans ce cas, l'évaluation du programme conduit à énumérer les valeurs de la plus petite suite, au sens d'une certaine relation d'ordre entre les suites.

– Dans la théorie des type abstraits algébriques on construit des algèbres et des relations d'équivalence entre les termes de ces algèbres. Grossièrement, un terme d'une algèbre correspond à un arbre. Si la spécification est exécutable, l'évaluation d'un terme correspond à la réduction d'un arbre à une forme normale équivalente. Il est courant d'associer à la spécification (i.e. au "programme") une algèbre particulière parmi celles qui sont solutions de la spécification, par exemple l'algèbre initiale [Handbook xxx].

Ces objets (fonction, prédicat, multi-ensemble, suite, algèbre) sont utiles à l'informaticien : ils sont suffisamment riches pour être universels, c'est-à-dire pour être le fondement d'un langage de programmation. Par ailleurs, ils correspondent à des concepts importants et donc constituent un sujet d'étude en soi. Cependant, fonction prédicat ou algèbre ne sont pas des objets qui interviennent directement dans la description des modèles des phénomènes naturels.

En effet, quand on pense à la simulation, on pense d'abord à la simulation des phénomènes physiques ou naturels, et par suite à des phénomènes qui prennent place dans le *temps* et dans l'*espace* : simuler c'est simuler le comportement, l'évolution, le changement d'une ou plusieurs entités, toutes notions qui renvoient à celle de *mouvement* et donc à celles de temps et d'espace.

I.3.2. Systèmes dynamiques : les variations d'une valeur dans l'espace et dans le temps

La description des phénomènes dynamiques fait appel à la notion de système et d'état d'un système. Un *système* est 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 des grandeurs appelées *variables* dont la valeur évolue au cours du temps. L'ensemble des variables qui décrivent le système forme l'*état* du système. Les variations dans le temps de cet état composent la *trajectoire* du système et constituent le phénomène que l'on veut modéliser. L'espace dans lequel se déroule le phénomène correspond à l'ensemble des variables et à la structure de leurs valeurs[1].

Nous donnerons le nom général de *système dynamique* aux modèles qui décrivent un état qui évolue dans le temps. L'objet que nous voulons construire à travers un programme est donc un système dynamique, et comme nous le construisons sur un calculateur digital, ce sera un *système dynamique discret*.

I.3.2.a. Les systèmes dynamiques discrets

Un système dynamique discret (ou **SDD**) est un système où les variations des valeurs des variables dépendent d'événements discrets et où les valeurs sont discrètes et quantifiées. 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[2] de valeurs d'un type scalaire quantifié (représentation informatique classique des booléens, des entiers, des réels).

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 :

---

[1] Il est toujours possible de "coder" deux valeurs simples en une seule valeur plus complexe par exemple par une paire. C'est pourquoi la notion d'espace est rendue à la fois par l'ensemble des variables qui décrivent un système et par le type des valeurs prises.

[2] Par exemple une image de cinéma ne constitue pas un SDD car si c'est bien une variable dont la valeur évolue de manière discrète 24 fois par seconde, 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.

. *systèmes de production industrielle* (chaîne d'assemblage, processus de fabrication),
. *modélisation des mécanismes économiques*,
. *réseaux d'ordinateurs*,
. *modélisation de trafics* (automobile, aérien, etc.),
. *systèmes de files d'attentes*,
. *systèmes de contrôle et de commande* (pilotage d'une centrale électrique),
. *réseaux d'automates* (commande d'un ascenseur, d'une machine-outil),
. *modèles connexionistes* (réseaux de neurones formels),
. *circuits VLSI synchrones*,
. *modélisation event-driven* (modèles de la physique qualitative),
. *modélisation des systèmes à états logiques ou symboliques* (comme par exemple l'exécution d'un programme parallèle),
. etc.

La modélisation des « systèmes naturels » donne souvent lieu à des modèles continus, mais les modèles continus sont presque toujours réductibles à des SDD. En effet, le traitement mathématique des systèmes continus passe par le formalisme 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 initiales et 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. [SMI 85, §1]). 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 domaine du Traitement Numérique Intensif actuellement en forte expansion.

Il est indubitable que les SDD prennent une importance toujours croissante, en particulier en Recherche Opérationnelle [IEEE 89], en Théorie des Systèmes [PAG 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 [MEY 90] ou [SCH 90].

I.3.3. Un langage parallèle

Revenons à présent sur la notion de langage déclaratif. Un tel langage a la propriété d'être naturellement parallèle.

I.3.3.a. Parallélisme de contrôle et parallélisme de flux

Si un programme se fonde sur la résolution d'équations, alors
– la notion d'instruction n'existe pas,
– le programmeur ne fait pas intervenir une notion d'ordre des calculs.

Cette situation n'est pas habituelle dans les langages de programmation impératifs. Elle a cependant l'avantage d'autoriser implicitement l'évaluation parallèle des calculs à effectuer, du moment que l'on respecte les dépendances entre les expressions. Autrement dit, il est possible d'extraire automatiquement le parallélisme de contrôle et le parallélisme de flux [SAN 90] contenus dans le processus de résolution. Cette approche du parallélisme fait l'objet de nombreuses recherches et elle est mise en œuvre soit à la compilation (algorithmes systoliques [QUI 89], conception de circuits VLSI [JOH 83]) soit à l'exécution (architectures data-flow [GAU 91], architectures fonctionnelles [THA 87]).

I.3.3.b. Parallélisme de données

De plus, la simulation des modèles de la physique fait souvent usage de structure de données de type *tableau* : un tableau correspond par exemple à la discrétisation d'un milieu continu (comme un fluide). Comme les lois de la physique doivent être les même partout (les lois de la mécanique des fluides s'appliquent en tout point du fluide), ces tableaux sont manipulés comme *des touts homogènes*, ce qui correspond au parallélisme de données. Le parallélisme de données permet d'exprimer que des activités identiques ont lieu à des endroits différents. Un langage tel que nous sommes en train de l'imaginer, est donc, par nature, un langage permettant d'exprimer du parallélisme de données.

I.3.3.c. Nécessité d'exploiter le parallélisme dans les grandes simulations

L'exploitation du parallélisme dans les programmes de simulation est par ailleurs largement motivée. En effet, l'étude systématique de différentes classes de modèles et la prise en compte d'un nombre toujours plus grand de paramètres dans des modèles de plus en plus sophistiqués nécessitent de plus en plus de puissance de calcul, que seuls les grands ordinateurs parallèles sont capables de fournir. La mise en œuvre d'un langage pour faire de grandes simulations *doit* donc permettre d'*exploiter* tout le parallélisme qu'il est capable d'*exprimer*.

I.4. La construction déclarative des systèmes dynamiques

La section précédente montre qu'un langage adapté à la simulation et à la PPSN :

- doit se rapprocher le plus possible de la *description par des équations* des modèles mathématiques ;
- doit permette de manipuler *des variations de valeurs dans le temps et dans l'espace* ;
- est naturellement parallèle car les objets manipulés sont par essence parallèles.

Nous voyons donc s'esquisser peu à peu un langage répondant à nos objectifs. C'est un langage déclaratif parallèle, permettant de définir des variations de valeurs dans l'espace et au cours du temps.

Nous appelerons *trajectoire* les variations d'une valeur au cours du temps. Nous appelerons *champ* les variations d'une valeur dans l'espace. Dans les sections suivantes, nous allons préciser quelle est la structure d'une trajectoire et la structure d'un champ, et qu'elle est la forme des équations qui permettent de les définir.

La notion de trajectoire sera représentée par le concept informatique de *stream* qui sera étudié dans le chapitre II. La notion de champ sera représentée par le concept informatique de *collection* qui sera étudié dans le chapitre III. La variation d'une valeur dans le temps **et** dans l'espace sera représentée par le concept informatique de *tissu* qui sera présenté au chapitre IV. L'implémentation des streams par des automates et des collections par des environnements et des tableaux, sera esquissée dans la seconde partie du chapitre V (Cf. la figure page ix).

I.4.1. Les structures du temps

Une trajectoire correspond aux variations d'un état au cours du temps ; c'est donc une fonction du temps dans l'ensemble des états. Mais quel est la structure du temps ? Il existe plusieurs modèles du temps : par exemple le temps de la mécanique newtonienne est différent du temps de la mécanique relativiste, et le temps des phénomènes biologiques est différent du temps psychologique.

Le reste de ce paragraphe a pour but de situer, parmi différentes possibilités, le modèle du temps que nous allons choisir : un temps discret et synchrone.

I.4.1.a. Succession, simultanéité et durée

Pour comparer les différentes représentations du temps, nous allons nous appuyer sur une analyse fort ancienne. Traditionnellement le temps est constitué d'*instants* qui sont les objets élémentaires du temps, ses "atomes". Les instants du temps sont structurés par trois relations :

- la *simultanéité* (en même temps),
- la *succession* (avant ou après),
- la *durée* (depuis/jusqu'à).

Ces relations sont figurées sur la figure I.1.

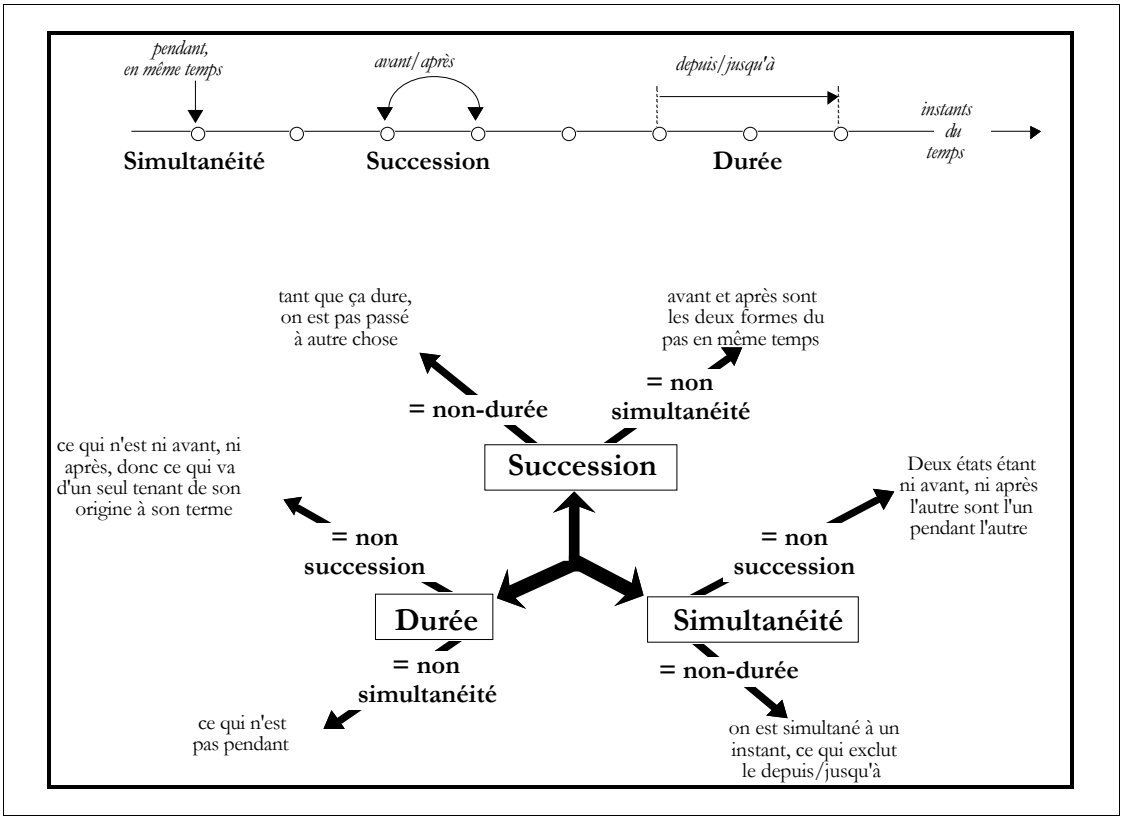


Figure I.1 : *Les trois concepts qui traditionnellement définissent le temps, et leurs oppositions.*

Chacune de ces relations s'opposent aux deux autres. Ainsi, ce qui survient ni avant et ni après, et qui ne dure pas, doit se produire en même temps : la simultanéité est donc la négation à la fois de la succession et de

la durée. Par ailleurs, la succession s'oppose à la durée, puisque « tant que ça dure, on est pas passé à autre chose », etc. La figure I.1 résume ces relations et leurs oppositions.

Cette présentation du temps va nous permettre de classer les différents modèles du temps. Un modèle du temps se définit par la donnée d'un ensemble d'instants et des trois relations précédentes. Puisque la négation de deux de ces relations permet de définir la troisième, il suffit en fait de définir la simultanéité et la succession des instants ; la durée peut alors se reconstruire comme étant tous les instants qui se succèdent ou qui sont simultanés à deux instants données qui marquent le début et la fin de la durée.

I.4.1.b. Les instants du temps : les modèles discrets et continus du temps

L'ensemble des instants du temps peut être un ensemble continu ou discret. Dans le cas d'un ensemble continu, il est toujours possible de glisser un instant dans la succession de deux instants choisis de façon arbitraire. De tels modèles de calcul existent dans certains langages de simulation dite continue (par exemple CSMP [SPE 76] ou DYNAMO [PUG 70]) : ces systèmes permettent d'étudier le comportement de systèmes décrits à l'aide d'équations différentielles ; le calcul correspond à la résolution numérique de ces équations en recourant cependant à la discrétisation et en utilisant une routine d'intégration.

Des modèles de programmation fondés sur un temps continu, étaient utilisés par les calculateurs analogiques répandus dans les années 50. Ces calculateurs tiraient parti des correspondances entre des équations mathématiques et des phénomènes électriques : par exemple la valeur d'une variable était représentée par une tension électrique. Les machines analogiques n'ont pas résisté à la concurrence des ordinateurs digitaux à cause de leur précision limitée, de la gamme restreinte de problèmes qu'elles pouvaient résoudre et des manipulations que nécessitait leur programmation (le câblage manuel des équations à résoudre).

Les calculateurs digitaux sont pilotés par des horloges digitales et sont à présent universellement répandus en informatique. Ils mettent en œuvre, au niveau du programmeur, un modèle de temps fondé sur un ensemble discret d'instants. Dans un modèle de temps discret, les instants correspondent à des *événements atomiques*.

I.4.1.c. Succession et simultanéité : les modèles synchrone et asynchrone du temps

Une fois que l'on dispose des instants, que leur ensemble soit discret ou pas, il faut les ordonner pour réaliser le concept de succession. « Avant » et « après » définissent une succession *linéaire* des instants. C'est par exemple la succession des points de `I R` dans le cas d'un temps continu, ou bien la succession des entiers de `I N` dans le cas d'un temps discret.

L'idée de succession linéaire est rendue par la notion de *relation d'ordre*. Nous pouvons donc distinguer, comme on le fait pour les relations mathématiques, les ordres temporels *partiels* des ordres temporels *totaux*. Dans le cas des ordres partiels, certains instants sont incomparables : ils ne sont ni avant, ni après, ni « l'un pendant l'autre ». Nous pouvons donner deux exemples : en physique relativiste, un observateur ne peut décider si un événement de l'espace-temps en dehors de son propre cône de lumière a lieu avant, pendant ou après l'instant d'observation ; dans cet exemple, le temps est continu. L'informatique distribuée offre elle de nombreux exemples de temps discrets partiels : entre deux synchronisations globales, les événements de calculs locaux à deux ordinateurs ne sont pas comparables. On peut parler de modèle *asynchrone* du temps quand la relation de succession des instants est partielle.

Un ordre de succession total entre les instants du temps définit non seulement la succession mais aussi la simultanéité : deux instants t_1 et t_2 sont simultanés, si t_1 ne se produit pas avant t_2 et si t_2 ne se produit pas

avant t_1. On peut alors parler de modèle *synchrone* du temps. Le modèle synchrone et continu du temps est par excellence la droite des réels[1] `I R`. Le modèle synchrone et discret du temps est par excellence `I N` muni de la relation d'ordre naturel sur les entiers. Cependant l'indexation des instants par un entier représente seulement la succession des événements et rien n'impose que la « quantité de temps[2] » qui s'écoule entre deux instants repérés par des entiers successifs soit la même quelques soient ces entiers.

I.4.1.d. Choix d'un modèle synchrone et discret du temps

Pour notre langage, nous faisons le choix d'un temps synchrone et discret. Il y a plusieurs raisons pour choisir un tel modèle du temps :

- Il est possible d'émuler les modèles continus par discrétisation. Cela ne va pas sans poser des problèmes, mais de toute façon c'est indispensable si on veut pouvoir faire des calculs numériques sur un ordinateur digital.
- Suivant le principe de "qui peut le plus, peut le moins", un modèle discret du temps permet d'émuler sans problème un temps asynchrone ou bien un temps où les instants sont partiellement ordonnés.

Ce choix a pour conséquence que :

- un instant est repéré (nommé) par une *date* qui peut être un entier,
- deux *événements* peuvent avoir la même date.

Un événement correspond au repérage par une date d'un phénomène : une tâche de calcul, le passage du système par un état donné, le franchissement d'un seuil par une valeur, etc.

I.4.1.e. Modèles du temps et langages de programmation

On peut espérer trouver ou à défaut développer un langage de programmation dont le modèle du temps correspond au modèle que nous avons adopté. Les langages de programmation impliquent tous un certain modèle du temps, car il faut bien préciser quand les calculs à réaliser doivent être effectués les uns par rapport aux autres. Les modèles du temps que l'on peut observer en examinant les langages de programmations apparaissent comme différentes versions de temps discret :

- Dans les langages impératifs séquentiels comme PASCAL, la succession est exprimée par des structures de contrôle : par exemple le point-virgule dans « `a ; b` » permet d'indiquer que le calcul `a` prendra place *avant* le calcul `b`. Les structures de contrôle des langages impératifs séquentiels construisent un ordre *total*. De plus cet ordre est dit *strict* : les instructions s'effectuent les unes après les autres, jamais en même temps ; il n'y a pas de notion de simultanéité des calculs. C'est pourquoi on dit aussi que ces modèles de calcul sont *séquentiels* : une seule action (de calcul) peut avoir lieu à un instant donné.
- Les langages impératifs parallèles comme OCCAM permettent de construire des ordres dits *partiels* en spécifiant que des instants de calcul sont incomparables. Par exemple, la structure de contrôle `PAR` indique que des calculs *peuvent* se dérouler en parallèle et donc que la succession ou la simultanéité

---

[1] On peut construire `I R` à partir de l'idée de corps *ordonné*, i.e. un corps sur lequel il existe une relation d'ordre totale compatible avec cette structure algébrique. L'opération d'addition correspond à la possibilité de se translater dans le temps, la multiplication permet d'insérer un instant entre deux instants arbitraires. Si on demande une propriété de maximalité (on ne peut pas étendre algébriquement le corps ordonné dans un ensemble ordonnable d'une façon qui prolonge l'ordre de départ) et si on demande une propriété de borne supérieure (i.e. tout ensemble de nombres non vide et majoré possède un plus petit majorant), on obtient `I R` à un isomorphisme près. L'intérêt de cette construction est qu'elle fonde la construction de `I R` à partir d'une relation d'ordre totale, donc de la succession synchrone, indépendamment de notions topologiques de continuité.

[2] L'expression « quantité de temps » est quelque peu paradoxale : en effet, pour mesurer une quantité de temps, il faudrait un *autre* temps pour mesurer l'écoulement du premier. Il est donc impossible de vérifier dans l'absolu "l'isotropie" du temps. Cela est vrai aussi pour un temps continu. Nous l'employons donc ici dans un sens intuitif pour l'informaticien qui l'associe à une certaine quantité de travail exécutée par une tâche entre deux instants qui sont alors qualifiés de *logiques* et qui ne correspondent pas à deux *dates consécutives*.

effective de ces calculs n'est pas connaissable. Dans les langages de type Acteurs [CLI 81] [AGH 85], qui ne sont plus impératifs, l'asynchronisme va encore plus loin, puisqu'il ne reste plus qu'une seule structure de contrôle permettant d'indiquer l'idée de succession : l'envoi de message ; la seule contrainte de succession est qu'un message sera reçu après son envoi.

Il faut remarquer que le parallélisme n'est pas la conséquence de la partialité de l'ordre de succession des calculs. Associer le parallélisme avec la partialité d'un ordre de succession est une approche spécifique d'une vision en terme de *concurrence*. A l'opposé de cette approche, le parallélisme de données est un exemple de modèle de programmation avec un temps synchrone et les programmes data-parallèles SIMD sont des modèles de calculs séquentiels. Le data-parallélisme s'introduit par la porte de la simultanéité : des calculs ont lieu en même temps.

Un autre exemple de modèle de calcul *non-séquentiel*, i.e. dans lequel plusieurs actions peuvent avoir lieu au même instant, et qui est fondé sur un modèle totalement ordonné du temps, est illustré par les langages réactifs temps-réels synchrones (comme par exemple ESTEREL [BER 87], LUSTRE [HAL 91] ou SIGNAL [GAU 87]). Dans ces langages, on peut exprimer grâce à des structures de contrôle spéciales, la simultanéité de deux actions et non plus seulement la succession comme dans les langages impératifs classiques. On peut exprimer par exemple : « à midi, faire telle action », expression absolument impossible dans le langage OCCAM[1], puisqu'il est impossible d'assurer le synchronisme de deux calculs. On peut noter que pour assurer la simultanéité des calculs, il faut faire l'hypothèse que les calculs ont lieu avec une durée logique nulle (hypothèse de synchronisation forte). Dans la pratique, il suffit qu'un calcul commencé se termine avant l'occurrence de tout autre événement.

Nous poursuivrons dans le chapitre II l'examen des structures informatiques permettant de représenter, dans un langage de programmation, le modèle du temps que nous avons choisi.

I.4.2. Les structures de l'espace

Nous allons à présent examiner la notion d'espace. Un espace est un ensemble de *points*. Un point est à l'espace ce qu'un instant est au temps. De la même façon que les instants du temps sont munis d'une « structure temporelle », une relation d'ordre, les points de l'espace sont munits d'une « structure spatiale » : c'est la notion de voisinage. Enfin, la notion de champ joue pour l'espace un rôle analogue à la notion de trajectoire pour le temps. Ainsi, les variations d'une valeur dans l'espace, un champ, constituent une fonction de l'espace dans un ensemble de valeurs.

Temps et espace sont des structures indissolublement liées. Par exemple, l'espace est ce qui rend possible la simultanéité : si deux événements se produisent en même temps, alors ils doivent se produire de quelque manière « en des endroits différents ». Inversement, c'est le temps qui permet la coïncidence spatiale (au sens de faire coïncider deux figures géométriques).

Dans ce document, pour la version 1.0 du langage 81/2, nous allons cependant considérablement simplifier les relations entre l'espace et le temps : on supposera que le temps et l'espace sont des notions indépendantes et qui n'interagissent pas. En particulier, on supposera que la structure de l'espace est invariante dans le temps et que la structure du temps est indépendante des points de l'espace. Donnons deux exemples qui montrent que ces hypothèses sont restrictives :

– Dans la modélisation d'un phénomène de croissance, par exemple le développement d'une plante ou d'un animal, la description de la structure d'un état varie au cours du temps : il faut plus de données

[1] Dans le langage OCCAM, de type concurrent, les calculs sont asynchrones, même si on parle de communication synchrone à propos de ce langage.

pour décrire l'organisme à maturité qu'au début de son développement. Par exemple, on peut décider de décrire un animal par l'état de ses cellules. Une cellule correspond à un point de l'espace qui structure l'animal. Le nombre des cellules varie au cours du temps : une cellule naît, entretient des relations de voisinage précises avec d'autres cellules, et meure. Une modélisation de ce type d'un phénomène de croissance nécessite donc un espace dont la structure est dynamique.

– En physique relativiste, la succession des événements dépend du point de l'espace où l'on se trouve : si un événement e_1 ne se trouve pas dans le cône de lumière d'un événement e_2 alors on pourra trouver un point d'observation où e_1 apparaît comme précédent e_2 et on trouvera un autre point où e_2 apparaît comme précédent e_1. Pour modéliser ce type de phénomène il faut que la succession des événements, donc ce que nous avons appelé la structure du temps, dépende des points de l'espace.

Avec un temps et un espace indépendants, la valeur qui décrit le système dynamique à un instant donné du temps est prise dans un ensemble fixé à l'avance : c'est l'ensemble des états. L'ensemble des états représente toutes les descriptions possibles à un instant donné du système dynamique. Un élément de cet ensemble possède une certaine structure, toujours la même par hypothèse, qui est l'espace dont nous parlons. L'ensemble des états est aussi parfois appelé "espace des états" dans la littérature et il faut se garder de confondre cet "espace" avec celui dont nous traitons.

I.4.2.a. Deux exemples de systèmes dynamiques qui ont une structure spatiale

Prenons l'exemple d'un film de cinéma ; la projection d'un film est un système dynamique. La trajectoire du système est la suite des images du film tous les $1/24^{ème}$ de seconde. Une image constitue un état. C'est un état qui peut être décrit par la couleur et l'intensité lumineuse de chaque point de l'écran de projection. Un état est donc une fonction $\varphi(x, y)$, où x et y représentent les coordonnées d'un point de l'écran et appartiennent à un segment de $\mathbb{R}$. L'image par φ d'un point (x, y) est elle-même une valeur composée par exemple par l'intensité lumineuse et quatre nombres qui indiquent la couleur dans le repère Rouge-Vert-Bleu. Un état est donc un objet qui peut être compliqué et qui possède une structure, dans notre exemple, celle de la surface de l'image.

Dans l'exemple de la projection d'un film de cinéma, le système dynamique est décrit par une seule variable dont la valeur à un instant donné est une fonction. Prenons à présent l'exemple d'un film de télévision. L'écran d'un téléviseur est discrétisé par la grille des pixels. Pour simplifier, prenons un téléviseur noir et blanc qui ne possède que quatre pixels disposés en un tableau 2×2. Cette fois-ci, l'état du système dynamique est décrit par *une seule* variable qui prend sa valeur dans $\{\text{blanc, noir}\}^4$: cet ensemble dénote l'ensemble des quadruplets de valeurs « blanc » ou « noir ».

De manière équivalente, il est possible de décrire l'état de ce système par *quatre* variables *distinctes* qui prennent alors leur valeur dans l'ensemble {blanc, noir}. Les deux descriptions du système sont équivalentes : on a seulement échangé de la complexité dans la structure d'une valeur contre de la complexité dans la description d'un état. Cet échange correspond à un *codage*. La différence entre les deux codages réside dans les parties de l'espace désignées par les noms des variables : quand la description utilise quatre variables, une variable désigne un point ; quand la description utilise une variable, cette variable désigne quatre points.

I.4.2.b. La trajectoire des points de l'espace

Un codage étant décidé (par l'observateur), le système dynamique est décrit, à un instant donné, par un ensemble de valeurs attachées aux points de l'espace. La valeur d'un point est une valeur *scalaire*. Elle varie dans le temps, formant la trajectoire du point. L'ensemble des trajectoires scalaires des points de l'espace constituent une trajectoire composée : la trajectoire de l'ensemble de points (Cf. figure I.2).

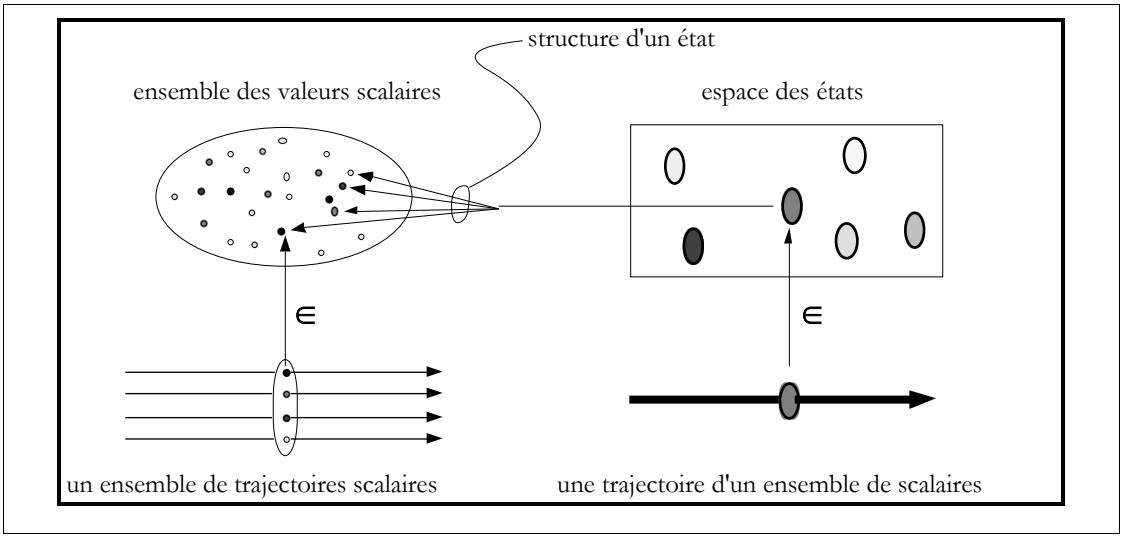


Figure I.2 : *Trajectoire scalaire et trajectoire composée.*

I.4.2.c. Notion de voisinage

De même que le temps est un ensemble d'instants muni d'une relation d'ordre, la succession, les points de l'espace sont munis d'une relation de voisinage. La structure de voisinage définit : « ici », « à coté », etc. Il existe plusieurs structures de voisinages différentes. La figure I.3 illlustre trois sortes de voisinages correspondant à la discrétisation d'une surface par trois maillages différents.

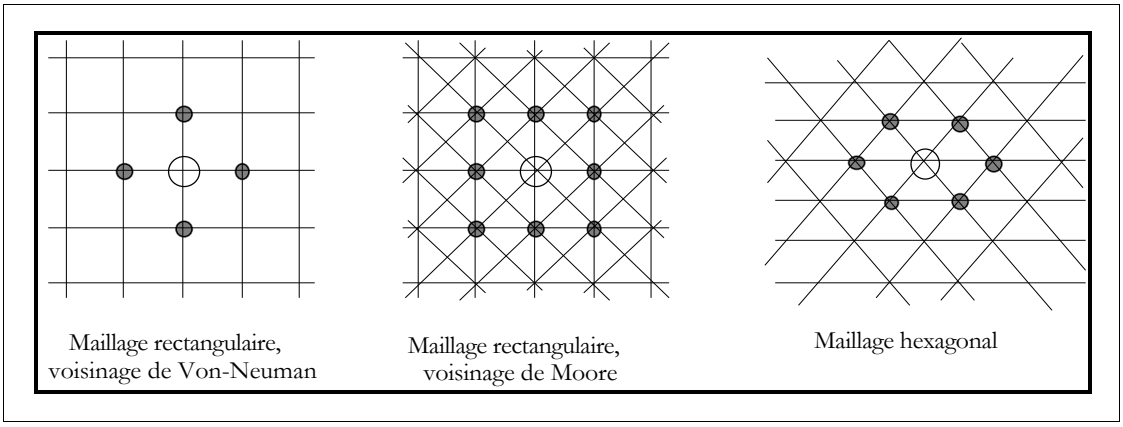


Figure I.3 : *Trois sortes de voisinages correspondant à la discrétisation d'une surface par un maillage. Un point de l'espace est désigné par ses coordonnées dans la grille. Les coordonnées d'un point voisin d'un point P donné sont calculables simplement à partir des coordonnées de P.*

I.4.2.d. Le choix d'un modèle de l'espace

Nous choisissons un modèle d'espace discret et fini où tous les points sont voisins :

– La contrainte d'un espace discret et fini est plus forte que la contrainte correspondante d'un temps simplement discret. Cela tient au fait que les valeurs d'un champ doivent être toutes *simultanément* accessibles alors que les valeurs d'une trajectoire sont égrenées dans le temps. Par suite, un champ ne

peut être défini sur un espace arbitrairement garnd à cause de la mémoire bornée des ordinateurs, alors qu'une trajectoire peut être définie sur un intervalle de temps arbitrairement long.

– Dans la version 1.0 de notre langage, nous ne nous préoccuperons pas de la notion de voisinage (et a fortiori de l'idée d'évolution des voisinages au cours du temps). Cela veut dire en particulier, que la valeur d'un champs en un point peut dépendre de la valeur du champ en n'importe quel autre point : on dira alors que tous les points sont voisins et la notion d'espace se réduit alors à celle d'ensemble de points.

Les points de l'espace étant en nombre fini, on peut les désigner, ou désigner un ensemble de points, par un entier de $\mathtt{I\,N}$. Par exemple, un point d'une image de télévision de 2×2 pixels (Cf. §I.4.2.a) peut être désigné par un entier appartenant à $\mathtt{\{0, 1, 2, 3\}}$. Cette désignation est parfois mal commode, par exemple quand on veut désigner les variables d'états. On utilise alors un identificateur alphanumérique. On parlera donc de la variable « $\mathtt{x}$ » au lieu de dire « le point 0 », etc.

Par ailleurs, il est agréable (Cf. l'exemple des maillages de la figure I.3) de pouvoir hiérarchiser la désignation des points, et d'utiliser comme ensemble d'identificateurs, un sous-ensemble fini $\mathtt{I\,E}$ de $\mathtt{I\,N}^m$.

I.4.3. La définition des trajectoires par une fonction d'évolution

Modéliser un système dynamique consiste à définir sa trajectoire. Une trajectoire correspondant à une suite d'états au cours du temps, on peut décrire une trajectoire en définissant une fonction de l'ensemble des instants dans l'espace des états. Nous avons choisi un temps discret et synchrone, par suite une trajectoire est une fonction $\mathtt{F}$ de $\mathtt{I\,N}$ dans l'espace des états. Un état est lui-même une fonction d'un ensemble $\mathtt{I\,E} \subset \mathtt{I\,N}^m$ dans un ensemble de valeurs.

Mais qu'en est-il de la fonction $\mathtt{F}$? Nous voulons définir cette fonction de manière déclarative par des équations, afin de nous rapprocher des formalismes mathématiques habituels. La définition d'une fonction demande précisément la donnée de deux choses :

– le domaine de définition de la fonction : l'état d'un système n'est pas nécessairement défini pour tous les instants ; autrement dit, une trajectoire est une fonction partielle de $\mathtt{I\,N}$ dans l'espace des états.

– un mécanisme de calcul permettant d'associer un état à chaque instant pris dans le domaine de définition.

La spécification du domaine de définition de la trajectoire consiste à définir « *quand* il se passe quelque chose ». C'est l'objet du "calcul d'horloge" et nous reviendrons sur cet aspect dans le prochain chapitre. Pour simplifier la suite de l'exposé, nous allons supposer dans ce chapitre que les trajectoires sont définies pour tout $\mathtt{t} \in \mathtt{I\,N}$ et nous allons examiner le problème du mécanisme de calcul de la trajectoire, afin de préciser les mécanismes de calcul permis et pertinents. Les mécanismes de calcul qui sont permis sont ceux qui sont *effectifs*, c'est-à-dire ceux qui permettent le calcul des valeurs de la trajectoire par un algorithme sur un ordinateur digital. Les mécanismes de calcul pertinents sont ceux qui correspondent à une *évolution causale* des états du système, c'est-à-dire ceux qui respectent la structure du temps.

I.4.3.a. Un exemple de trajectoire définie de manière non effective par des équations

Une trajectoire étant définie par des équations, il faut que ces équations fournissent un moyen *effectif* de calculer les valeurs de cette trajectoire. Donnons un exemple d'équation définissant une trajectoire $\mathtt{T}$ qui n'est pas effective :

$$\mathtt{\forall\ t \in I\,N,\ T(t+1) + T(t-1) = 2\,T(t)} \qquad (1)$$

Cette équation n'est pas effective car il existe une infinité de solutions et aucune n'est plus canonique qu'une autre. Par exemple, toutes les fonctions $\mathtt{F}_{\mathtt{f_0,f_1}}$ de la forme :

$$\begin{cases} F_{f_0,f_1}(0) = f_0 \\ F_{f_0,f_1}(1) = f_1 \\ F_{f_0,f_1}(t+2) = (t+2)\,f_0 - (t+1)\,f_1, \ \forall\ t \quad \mathbb{N} \end{cases}$$

où f_0 et f_1 sont des quantités arbitraires, sont des solutions de l'équation (1). Pour éviter qu'il y ait une infinité de solutions, il faut imposer des contraintes supplémentaires. Par exemple :

$$\begin{cases} T(0) = T_0 \\ T(100) = T_{100} \\ T(t+2) + T(t) = 2\,T(t+1), \ \forall\ t \quad \mathbb{N} \end{cases} \qquad (2)$$

On peut montrer que ce système de trois équations définit T de manière unique. Cependant pour calculer les états aux instants $1, \ldots, 99$, il faut résoudre un système de 99 équations linéaires. Cela est encore possible, car il existe un algorithme pour résoudre les systèmes linéaires. Cependant, si on permet n'importe quel type d'équation pour définir une trajectoire, nous n'aurons plus de méthode générale pour *calculer automatiquement* les états, même si un argument de nature mathématique nous permet d'assurer que ces équations définissent bien la trajectoire de manière unique.

I.4.3.b. Des équations définissant une fonction d'évolution

Afin de permettre le calcul effectif des états d'un système, nous devons donc restreindre les types d'équations permis. Cette restriction a deux buts :

- assurer que la modélisation du système est effective, i.e. que l'on puisse simuler le modèle ;
- assurer qu'il y a une solution unique, ou au moins canonique, au système d'équations qui définit la trajectoire.

Cette dernière condition indique que la modélisation est dans un certain sens complète : elle décrit le système réel de manière unique (on dit que la description est *catégorique*).

Mais nous voulons plus. En effet, à un instant donné, l'état dans lequel se trouve le système est une *fonction* des états passés : c'est la *causalité* des modèles qui décrivent les systèmes physiques. Plus précisément, on dira qu'un système T est *causal*[1] s'il existe une fonction f telle que pour tout instant t et toute durée $\tau > 0$,

$$T(t_0 + \tau) = f(\tau, T(t_0)) \qquad (3)$$

L'équation (3) affirme que l'état d'un système à un instant $(t_0+\tau)$ ne dépend que de l'état initial $T(t_0)$ du système (indépendamment de l'instant t_0 qui est choisi) et de la durée τ pendant laquelle il a évolué. Remarquons que la causalité est complètement liée à la relation de succession qui structure les instant du temps : l'expression $t_0+\tau$ n'est que la traduction par des nombres de la relation de succession de deux instants t_0 et $(t_0+\tau)$ séparés d'une durée τ.

Généralement, on ne sait pas exprimer explicitement la fonction f pour toute durée τ. Par contre, il est souvent possible de déterminer la fonction g exprimant le passage d'un état à l'état suivant. Cette fonction, qui est la fonction d'évolution élémentaire du système, est reliée à f. Ici le temps est discret et donc : $g(x) = f(1, x)$. A partir de g et d'un état initial, il suffit d'itérer l'application de cette fonction pour obtenir la suite des états du système à tous les instants.

Notre problème est donc maintenant de ne permettre que des équations qui définissent des systèmes causaux. Ce problème sera abordé dans le chapitre suivant. Le rôle du compilateur d'un langage fondé sur des équations définissant des systèmes causaux, sera d'extraire de ces équations deux choses : un état initial $T(0)$

[1] Cf. [AUL 89] pour la notion de système causal. Remarquons que la propriété de causalité s'exprime que le système soit discret ou continu.

et la fonction d'évolution. La synthèse de cette fonction par le compilateur du langage offre le moyen de résoudre numériquement les équations du système dynamique et d'énumérer, dans l'ordre des instants croissants, la suite des états du système.

I.4.4. La définition des champs comme des touts

Un champ correspond à une fonction d'une partie finie de $\mathtt{I\ N}^m$ dans l'ensemble des valeurs. Comment définir une telle fonction ?

La causalité restreint les trajectoires possibles d'un système aux systèmes dont l'état à un instant donné peut s'exprimer à partir de l'état aux instants passés. De manière analogue, il nous faudrait développer une notion de "causalité spatiale" qui restreindrait les champs possibles à ceux dont la valeur d'un point peut s'exprimer comme dépendant uniquement des points voisins (interdiction des "actions à distance"). Cependant, nous avons décidé de ne pas nous préoccuper de la notion de voisinage pour la version 1.0 décrite dans ce document : tout point étant potentiellement voisin de tout autre, la causalité spatiale est toujours vérifiée.

Une autre contrainte est qu'il faut absolument manipuler les champs globalement et non pas à travers les images de chacun des points. Cela revient à vouloir manipuler des ensembles *comme des touts*. Cette approche est *nécessaire* pour plusieurs raisons :

– **Invariance des lois d'évolution :** les lois d'évolution d'un système dynamique sont les mêmes en tout point de l'espace (et à tout instant du temps).

  Les lois physiques ne dépendent généralement pas du lieu (ni du moment) où elles s'appliquent. Il est donc naturel de vouloir traiter globalement tous les points de l'espace puisque le traitement sera le même partout[1].

– **Approche uniforme et globale :** nous voulons traiter uniformément l'espace et le temps.

  Le concept de trajectoire permet de traiter implicitement le temps à travers des opérateurs temporels : quantification, échantillonnage et retard. Ces opérateurs permettent de définir la succession et la simultanéité des événements et opérent sur les trajectoires en leur entier. Nous verrons ces opérateurs dans le chapitre suivant. De manière analogue, nous voulons définir des opérateurs spatiaux qui permettraient d'articuler la notion de points et de voisinages et qui manipulent des champs comme des touts.

– **Taille du modèle :** il n'est pas possible d'expliciter les relations entre chaque point de l'espace.

  Les champs résultent souvent de la discrétisation d'une variable continue. Ils comportent donc généralement un très grand nombre d'éléments (plusieurs milliers voire plusieurs millions en traitement numérique intensif) et il n'est pas possible d'expliciter les relations entre chacun de ces éléments, i.e. de les écrire à la main.

– **Efficacité :** le traitement déclaratif des ensembles doit se faire globalement afin d'être efficace.

  Cette raison sera développée dans le chapitre V concernant le parallélisme.

---

[1] … partout ou presque, puisqu'il y a des conditions aux limites (et des conditions initiales), qui sont les lieux et les instants où l'état du système dynamique modélisé échappe à ses lois d'évolution et doit être donné de manière explicite et ad-hoc par le programmeur. L'exemple de la diffusion de la chaleur dans un fil d'acier qui est traité au chapitre IV en est une bonne illustration.

I.4.4.a. Manipulation globale d'un ensemble de valeurs et langages de programmation

La plupart des langages de programmation classiques offrent des structures de données correspondant à une réunion de données plus élémentaires mais n'offrent comme opération primitive que l'accès à un élément ; par exemple les vecteurs en C, les tableaux en PASCAL, FORTRAN77, les listes en LISP, ML… Les opérations qui opèrent globalement sur les ensembles doivent alors être reconstruites (séquentiellement) à partir des opérations sur les données élémentaires. Ce n'est pas le cas des tableaux en APL, des ensembles en SETL ou des relations en SQL : on peut, par exemple, calculer le maximum des éléments d'un tableau APL par une expression élémentaire, qui n'indique en rien l'accès itératif aux éléments du tableau ; ou bien, on peut faire le produit de deux relations en SQL, sans indiquer de référence aux tuples les constituant.

Pour bien marquer la différence entre des ensembles de valeurs manipulés à travers l'accès à un élément et les ensembles que l'on manipule comme des touts, nous parlerons dans ce dernier cas de *collection*. Le problème de la définition déclarative des champs se résume donc à construire un langage où l'on peut définir des collections par des équations.

I.5. Le langage 8$_{1/2}$

Nous voyons se préciser peu à peu un langage répondant à nos desiderata et la suite de ce document va s'attacher à le définir plus précisément. Le chapitre II va définir la représentation informatique d'une trajectoire causale : c'est le concept de *stream*. Le chapitre III se consacre aux variations d'une valeur dans l'espace : cela sera représenté informatiquement par des *collections*. Le chapitre IV "recollera" les morceaux pour étudier les variations d'une valeur dans le temps **et** dans l'espace : c'est la notion de *tissu*. Un tissu représente l'évolution au cours du temps de l'état d'un système dynamique discret. Le chapitre V présentera quelque éléments d'implémentation parallèle des tissus.

Le langage expérimental **81/2** est un langage parallèle déclaratif, permettant de définir ces objets : des trajectoires causales dont les valeurs, à un instant donné, pris dans un ensemble discret et totalement ordonné, sont des champs discrets et finis. Cet objet, un tissu, peut se représenter dans un trièdre Temps-Espace-Valeurs (Cf. figure I.4).

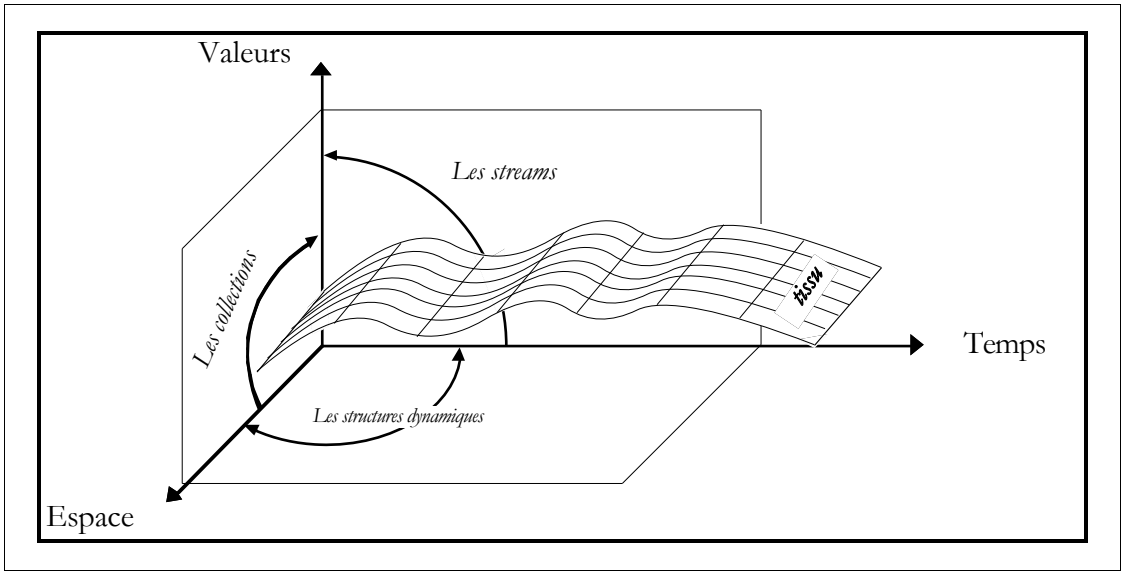


Figure I.4 : *Un objet définit par un programme* 8$_{1/2}$ *est une surface dans le repère Temps-Espace-Valeurs.*

Un programme 81/2 correspond à la définition d'un petit monde parfaitement défini par ses lois d'évolution. L'évaluation d'un programme 81/2 consiste à simuler l'évolution de ce monde, et donc à dérouler les conséquences des lois qui le régissent.

The 81/2 engine weaves flowers and leaves just as the Analytical Engine weaves algebraic patterns.

II. Les streams 8½

II.1. Introduction

L'objectif de ce chapitre est double. Il s'agit de :

- construire *une structure de données* qui permette de représenter et de manipuler comme un tout la variation des valeurs d'une variable d'état dans le temps ;
- de définir de tels objets par des équations.

Nous obtiendrons alors un langage de programmation déclaratif, dans lequel un programme correspond à un système d'équations et l'exécution de ce programme à la résolution de ce système. Ce langage est nommé 81/2.

Rappelons que, à la suite de l'analyse du chapitre I, nous avons choisi un ensemble discret totalement ordonné d'instants comme modèle du temps. Les variations d'un état dans ce temps correspondent donc à un ensemble discret totalement ordonné de valeurs.

Nous allons commencer par donner plusieurs exemples d'ensembles discrets totalement ordonnés dans les langages de programmation et discuter de leurs utilisations. Dans un deuxième temps, nous allons élaborer, en nous appuyant sur les exemples précédents et sur un petit problème de modélisation, notre propre structure de données. Enfin, nous examinerons comment définir une telle structure de données par des équations.

En fait, 81/2 manipule des structures plus complexes que celles que nous présentons dans ce chapitre. Il faudra attendre le chapitre suivant pour compléter notre construction. Cependant, la structure de données que nous construisons dans ce chapitre est un objet 81/2 valide et les exemples que nous donnons sont des programmes 81/2 à part entière.

II.2. La notion de stream temporel

II.2.1. Suites, listes, séries et streams

La notion d'*ensemble discret totalement ordonné de valeurs*, qu'on va appeler *suite* pour simplifier, se retrouve un peu partout en mathématique et en informatique, avec des noms différents et des usages variés. Nous aurons besoin d'une notation pour désigner un tel ensemble :

```
A = <1; 2; 3; 4; 5<
```

désigne une suite nommée `A` de cinq valeurs. On utilise le caractère `<` pour délimiter l'énumération des valeurs de cette suite, plutôt que `{` et `}` afin de bien marquer que l'ordre des éléments a une importance : les éléments sont ordonnés linéairement de gauche à droite, la valeur `1` précède la valeur `2` qui précède la valeur `3`, etc. On insiste encore sur la *succession* des éléments en utilisant le point-virgule pour séparer les éléments, de manière analogue à PASCAL où le point-virgule indique le séquencement des instructions : ici on ne séquence pas des instructions mais des valeurs. Dans le cas où les éléments de l'ensemble `A` ne sont pas en nombre fini, on utilisera les trois petits points pour indiquer que l'énumération se poursuit :

```
A = <1; 2; 3; 4; 5; 6; … <
```

Passons à présent en revue quelques exemples, en les confrontant avec nos deux objectifs.

II.2.1.a. Les listes en Lisp

Les listes "à la Lisp" sont une structure de données classique en informatique. Leur implémentation utilise habituellement un chaînage entre les valeurs de la liste, grâce à la notion de doublet. Tous les éléments de la liste sont présents dans la mémoire d'un programme « en même temps » mais on ne peut accéder à un élément « qu'en passant par les éléments précédents ». C'est donc bien une structure linéairement ordonnée d'éléments, seulement ces éléments ne sont pas ordonnés linéairement dans le temps mais dans l'espace : par exemple, il est possible de renverser l'ordre d'une liste, par l'opération `reverse`. Si on interprète cette opération en terme de succession temporelle, cela veut dire qu'il est possible de renverser l'ordre du temps. Une autre conséquence du fait que tous les éléments d'une liste sont présents simultanément en mémoire, est qu'on ne peut pas avoir de liste infinie d'éléments car la mémoire des ordinateurs est bornée.

Remarquons que d'autres structures de données sont totalement ordonnées dans les langages de programmation : les vecteurs en APL, les vecteurs en PASCAL ou en C, les tuples en SETL, les chaînes de caractères en C, etc. Cependant, dans ces structures de données, il est possible d'accéder directement à un élément, sans "passer" par les autres éléments, c'est-à-dire sans respecter la succession des éléments.

Dans tous ces exemples, les éléments de la structure de données sont énumérés dans la mémoire de l'ordinateur et pas dans le temps.

II.2.1.b. Les boucles dans un langage impératif

Dans les langages de programmation classiques, il existe une notion qui permet d'énumérer *dans le temps* une suite de valeurs : il s'agit de la notion de boucle. Par exemple, la boucle suivante écrite en `C`

```
for(i = 10; i >= 5; --i)
     a = (100 - i*i)
```

va énumérer séquentiellement deux suites de valeurs : la suite `I` des valeurs de l'index `i` de la boucle : `I = <10; 9; 8; 7; 6; 5<` et aussi la suite `A` des valeurs de la variable `a` calculées dans le corps de la boucle : `A = <0; 9; 36; 51; 64<`. Contrairement aux listes en Lisp, ces valeurs ne sont pas

présentes simultanément dans la mémoire d'un ordinateur. On ne sait pas faire d'opération `reverse` dessus. On peut aussi envisager de construire des suites infinies, énumérées par exemple par une boucle `forever` (une boucle « tant que » dont la condition de sortie n'est jamais valide).

Le problème avec les boucles est qu'il s'agit d'une structure de contrôle. C'est un fragment de code, et qu'en tant que tel dans un langage impératif, on ne peut pas le manipuler comme une structure de données : on ne sait pas donner un nom à une boucle, faire l'addition de deux boucles, etc.

II.2.1.c. Les pipes en UNIX et les séries en CommonLisp

Afin de répondre au problème précédent, on peut essayer de transformer les boucles en une structure de données. C'est la notion de *série*. Une série est, pour citer [WAD 85], une structure « mathématiquement respectable » et [WAT 91] : « (…) series expressions are to loops as structured control constructs are to gotos ».

Les `pipe`s de communication en UNIX (et leurs descendants, les `socket`s BSD ou les `stream`s SystemV) sont des exemples de séries : on peut créer un `pipe`, lui donner un nom, le passer en argument à une fonction, etc. Mais il y a peu d'opérations qui touchent aux valeurs du `pipe` et qui les manipulent comme des touts ; on peut seulement lire une valeur dans un `pipe` : les opérations sur les valeurs d'un `pipe` sont locales, elles ne portent pas sur l'ensemble des valeurs du `pipe` mais sur un seul élément du `pipe` à la fois.

La notion de série a été aussi introduite dans le langage CommonLisp (sous la forme d'une bibliothèque de macros) grâce à la capacité qu'à Lisp de manipuler uniformément le code des programmes comme des données. Une série en CommonLisp est une suite de valeurs construite à partir d'autres suites de valeurs. Les opérations de construction sont les suivantes :

- *Transduction* : on combine au moyen d'une fonction, élément par élément, les valeurs de plusieurs suites afin d'en former une nouvelle. Par exemple, on peut faire l'addition, élément par élément, de deux suites.
- *Filtrage* : à partir d'une suite, on calcule la sous-suite de valeurs qui vérifient un prédicat donné.
- *Énumération* : on peut transformer une liste Lisp classique en série, ou bien transformer une fonction `f` et un argument initial `a` en la série : `<a; f(a); f(f(a)); f(f(f(a))); … <`.
- *Décalage :* on insère en tête d'une série une valeur grâce à l'opérateur *shift*. Le résultat est un décalage de la série de départ, avec introduction d'une nouvelle valeur en tête. Par exemple, le `shift(0, <1; 2; 3<)` a pour résultat la série : `<0; 1; 2; 3<`.
- etc.

Ces opérations, qui sont des macros Lisp, se transforment en des boucles qui énumèrent les éléments des séries. Par exemple, la série `A` des valeurs de `a` qui est calculée par la boucle écrite en `C` du paragraphe précédent, peut s'écrire avec le langage des séries comme :

```
(setq I (list-to-serie '(10 9 8 7 6)))
(setq A (transduction (lambda (i) (- 100 (* i i))) I))
```

On peut remarquer que les séries sont affectées à des variables CommonLisp classiques. En fait, la valeur affectée correspond à une fonction qui à la n^{ème} invocation fournit la n^{ème} valeur de la liste. Ainsi, le résultat de `(list-to-serie '(10 9 8 7 6))` est la fonction anonyme[1] :

---

[1] Cette fonction est plus exactement une *clôture* : la variable `value` est une variable locale à la fonction anonyme définie dans le corps du `let`. Sa valeur est rémanente. A chaque appel, la tête (`car`) de la liste `value` va être retournée et `value` va être modifiée afin de pointer sur le reste de la liste (`cdr`). La fonction `prog1` permet d'évaluer en séquence une liste d'expressions et de retourner la valeur de la première expression évaluée.

```
(let ((value '(10 9 8 7 6))
      (lambda () (prog1 (car value) (setq value (cdr value)))))
```

Il faut faire plusieurs remarques :

- Ici, la manipulation des séries est de nature impérative : une même variable Lisp peut ainsi dénoter des série différentes au cours de l'exécution du programme.
- Afin de permettre la construction du code correspondant à une série, certaines expressions sont interdites (Cf. [WAT 91]). En particulier, il est interdit de définir récursivement une série comme on définit récursivement une fonction : par exemple, on ne peut pas définir en Lisp :

  ```
  (setq UN (shift 1 (transduction (lambda (u) (1+ u)) UN)))
  ```

  Cette série débuterait par la valeur `1` puis serait égale à elle-même plus `1`, ce qui définit la suite des entiers naturels.

II.2.1.d. Suites récurrentes en mathématiques

Comme le montre l'exemple précédent, les définitions récursives sont un moyen de définition très puissant. Les *suites récurrentes* en mathématiques offrent un exemple de suite qu'on peut définir récursivement. En mathématiques, une suite est une fonction, ayant pour ensemble de définition une partie de $\mathbb{N}$ représentant les indices de la suite, dans un certain ensemble de valeurs. L'indice d'un élément représente le rang de cet élément dans la suite. La suite :

$(U_n)_{\mathbb{N}}$

dénote une suite, nommée `U` et indexée par des entiers. Pour définir cette suite, on peut donner une expression générique qui donne la valeur d'un élément d'indice `n`. Par exemple :

$(U_n)_{\mathbb{N}} : U_n = U_{n-1} + 1$

Cette définition est une équation à résoudre sur l'ensemble des suites. Dans ce cas particulier, cette équation admet une infinité de solutions, toutes les suites de la forme :

$\langle u_0; u_0+1; u_0+2; u_0+3; \ldots \rangle$

où u_0 est une valeur arbitraire. La définition

$$(V_n)_{\mathbb{N}} : \begin{cases} V_0 = 1 \\ V_n = V_{n-1} + 1, \text{ pour } n > 1 \end{cases}$$

admet une solution unique, la suite $V = \langle 1; 2; 3; \ldots \rangle$ des entiers naturels.

Cette manière de définir les suites se réfère explicitement aux éléments de la suite. En conséquence, cette notation apparaît comme une notation commode pour fournir le système des équations qui doivent être vérifiées par *les éléments* de la suite. C'est donc une abréviation pour la donnée du système comportant une infinité d'équations de la forme :

$$\begin{cases} V_0 = 1 \\ V_1 = V_0 + 1 \\ V_2 = V_1 + 1 \\ V_3 = V_2 + 1 \\ \ldots \end{cases}$$

Si on veut vraiment manipuler les suites comme des objets en-soi et non à travers les éléments qui la composent, il faut introduire des opérations *globales* sur les suites, comme par exemple l'opération de différence Δ qui, a une suite donnée, fait correspondre la suite des différences des éléments consécutifs $(U_{n+1} - U_n)$. Si on applique Δ à la suite $\langle 1; 2; 3; \ldots \rangle$, on obtient la suite : $\langle 1; 1; \ldots \rangle$. Avec de telles opérations globales, on peut écrire de "vraies" équations entre suites. Par exemple :

$$\begin{cases} U_0 = 1 \\ \Delta U = U \end{cases}$$

où U est une suite inconnue à trouver. On pourra arguer que la condition $U_0 = 1$ porte encore sur un élément et non sur la suite globale : ce n'est donc pas une équation sur des suites au sens défini ci-dessus. Mais en introduisant la fonction `first` qui, à une suite, associe la valeur de son premier élément, on peut réécrire le système précédent en un système d'équations uniquement sur les suites :

$$\begin{cases} \text{first } U = 1 \\ \Delta U = U \end{cases}$$

Il existe une solution unique à ce système, $U_n = 2^n$.

II.2.1.e. Les streams en LUCID

Le langage LUCID [ASH 76] [WAD 85] permet d'écrire des équations entre des suites[1]. Un programme LUCID *est un système d'équations* dont les solutions sont des suites. Une équation LUCID prend toujours la forme :

variable **=** *expression*

L'*exécution* du programme correspond à *résoudre* ces équations, c'est-à-dire à calculer les valeurs qui composent les suites définies par le programme. Les opérateurs du langage agissent *tous* sur des suites. Les suites en LUCID s'appellent des *streams*. Par exemple, si `A` et `B` correspondent aux streams :

```
A = < 1;  2;  3;  4;  5 <
B = < 2;  7;  4;  8; 33 <
```

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

```
A + B ⇒ < 3;  9;  7;  12;  38 <
```

Le signe ⇒ se lit : « a pour valeur », ou bien « s'évalue en ».

De même que l'opérateur `+`, toutes les opérations classiques sur les scalaires s'étendent naturellement en des opérations sur les streams, en opérant élément par élément (transduction). Mais LUCID offre aussi des opérateurs supplémentaires, qui ne correspondent pas à l'extension d'un opérateur scalaire. Par exemple, l'opérateur `previous` permet de créer le stream U_{n-1} à partir du stream U_n :

```
          A = < 1;  2;  3;  4;  5; … <
previous A ⇒ < ⊥;  1;  2;  3;  4;  5; … <
```

La première valeur de `previous A` est indéfinie, ce qui se note par le symbole ⊥. On remarque que la deuxième valeur de `previous A` correspond à la première valeur de `A`, et plus généralement, la `n`ème valeur de `previous A` correspond à la `(n-1)`ème de `A`. Pour cette raison l'opérateur `previous` est appelé *opérateur de délai*. La combinaison d'une valeur indéfinie avec une autre valeur est encore une valeur indéfinie. Par exemple :

```
A + previous A ⇒  < ⊥; 3;  5;  7;  9; … <
```

Pour se débarrasser de cette valeur indéfinie, on peut utiliser l'opérateur `fby` (abréviation de « followed by ») qui permet de d'implémenter l'opérateur `shift` vu précédemment :

$$U = < U_0;\ U_1;\ U_2;\ \dots <$$
$$V = < V_0;\ V_1;\ V_2;\ \dots <$$

[1] On peut citer comme autre langage, le langage HASKELL [HUD 92] qui grâce à l'évaluation paresseuse, permet de définir des listes infinies par des équations récursives.

`U fby V` $\Rightarrow$ `< U0; V0; V1; V2; … <`

le résultat est un stream dont la première valeur est celle de l'argument gauche de `fby` et qui se poursuit comme l'argument droit. Donc, par exemple

`<1< fby (A + previous A)` $\Rightarrow$ `< 1; 3; 5; 7; 9; … <`

L'opérateur `previous` et l'opérateur `fby` permettent d'écrire l'équation récursive d'un compteur, ce que l'on ne pouvait pas faire avec le langage des séries :

`cpt = 0 fby (1 + previous cpt)` (1)

Attention, les constantes `0` et `1` désignent en fait de manière abrégée les streams : `<0; 0; …<` et `<1; 1; …<`. L'équation (1) se lit de la manière suivante : « `cpt` est un stream qui est égal à 0 suivi de `cpt` retardé et augmenté de 1 ». Si on regarde au niveau de chaque élément de `cpt`, on peut se persuader que l'équation précédente est équivalente à

$$\begin{cases} cpt_0 = 0 \\ cpt_{n+1} = cpt_n + 1 \end{cases}$$

Sous cette forme, il apparaît peut-être plus clairement que la solution de l'équation (1) est le stream `<0; 1; 2; 3; …<`.

La notion de stream développée en LUCID ne correspond cependant pas à ce que nous voulons, elle est "trop puissante" : en effet, elle permet d'écrire des streams dont le $n^{ème}$ élément est calculé en fonction d'un élément de rang `m`, avec `m>n`. Par exemple, de manière symétrique à l'opérateur `previous`, LUCID introduit l'opérateur `next` tel que

`U = < U0; U1; U2 <`
`next U` $\Rightarrow$ `< U1; U2 <`

Si nous interprétons les streams temporellement, cela veut dire qu'une valeur peut dépendre du futur ! et cela montre qu'on ne peut interpréter la succession des éléments d'un stream LUCID comme le passage du temps : les valeurs d'un stream LUCID *doivent toutes être stockées en mémoire*[1] et sont calculées "à la demande", dans un ordre qui n'est pas forcément l'ordre naturel "de gauche à droite" d'une suite temporelle.

II.2.2. La notion de stream en 81/2

Résumons les enseignements que nous avons tirés de nos exemples :

- Les listes sont des ensembles de valeurs totalement ordonnés. Dans les langages impératifs, les listes sont des structures de données dont les valeurs sont stockées en mémoire toutes en même temps. La notion de succession est donc artificiellement imposée par des opérateurs qui ne permettent qu'un accès séquentiel aux éléments des listes. On ne peut pas non plus avoir de liste infinie.
- Le concept de boucle dans un langage de programmation impératif correspond mieux à la notion de suite telle que nous la désirons : les valeurs sont énumérées ou calculées au cours du temps. Le problème est qu'on ne peut pas manipuler des boucles aussi facilement que des données : on ne peut pas les définir récursivement, on ne dispose pas d'opérateurs permettant les composer globalement, etc.

[1] On verra au chapitre V que 81/2 ne stocke en mémoire que la valeur des streams à un instant donné : cela est cohérent avec le concept de temps causal. Il nous semble important de distinguer les variations d'une valeur dans le temps (les streams en 81/2) des variations d'une valeur dans l'espace (les collections en 81/2) : en effet, les opérations utilisées pour définir ces structures de données sont différentes et présentent des propriétés spécifiques. En LUCID, une telle distinction n'est pas faite, ce qui conduit a une notion principalement spatiale des suites.

– La notion mathématique de suite permet de définir des suites par des équations. Le langage LUCID, avec la notion de stream, adopte cette approche, mais introduit des opérations qui ne permettent plus d'interpréter la progression dans le stream comme le passage du temps.

Nous sommes donc tentés d'*associer la notion de stream à celle de trajectoire* (Cf. le chapitre précédent) : une succession de valeurs dans le temps semble correspondre naturellement à la notion stream, *si* on prend soin de restreindre les opérations sur les streams afin de ne conserver qu'un sous-ensemble d'opérateurs compatibles avec une interprétation temporelle (en interdisant l'opérateur `next` par exemple).

Mais quels opérateurs doit-on permettre et est-ce que les streams LUCID offrent toutes les opérations utiles ? Nous allons voir que dès que l'on s'intéresse à l'interprétation temporelle *de deux streams considérés en même temps*, plus aucune opération LUCID ne permet une interprétation temporelle cohérente[1]. Cela nous amènera à reconstruire une nouvelle notion de stream comme étant une *suite séquentielle temporisée à mémoire bornée*.

Dans ce qui suit, une suite désignera informellement un ensemble ordonné de valeurs et un stream correspondra à une suite qui s'interprète comme une succession temporelle de valeurs.

II.2.2.a. Suites ***séquentielles***

Traditionnellement, une suite est une *fonction* d'une partie de `I N` (représentant les indices de la suite) dans un certain ensemble de valeurs. Cette formalisation convient aussi bien aux listes qu'aux vecteurs ou aux tableaux. Cette formalisation est très générale, car elle permet de créer à peu près n'importe comment des suites à partir d'autres suites.

On appellera *fonction séquentielle* de suites une fonction `f` sur les suites, tels que le $n^{ème}$ élément du résultat est calculé uniquement en fonction des éléments de rang `m` avec `m≤n` des arguments. Par exemple :

$$U = < U_0;\ U_1;\ U_2;\ \ldots <$$
$$f(U) = < U_0;\ U_0+U_1;\ U_0+U_1+U_2;\ \ldots <$$

`f` est une fonction séquentielle. Par contre, la fonction `g`

$$U = < U_0;\ U_1;\ U_2;\ \ldots <$$
$$g(U) = < U_0+U_1;\ U_1+U_2;\ U_2+U_3;\ \ldots <$$

n'est pas une fonction séquentielle. LUCID permet d'écrire des fonctions sur les suites qui ne sont pas séquentielles.

L'idée de fonction séquentielle est importante car elle correspond au respect de l'ordre temporel : on ne peut composer à un instant donné que des valeurs qui proviennent du passé, les valeurs du futur sont inaccessibles. Nous appellerons *suites séquentielles* les suites qui s'expriment comme fonction séquentielle d'autres suites. Le langage que nous recherchons ne doit permettre de construire que des suites séquentielles.

II.2.2.b. Suites séquentielles ***à mémoire bornée***

En fait, il nous faut une propriété plus forte que la séquentialité : il ne suffit pas que les combinaisons entre streams respectent l'ordre temporel, il faut aussi qu'à n'importe quel instant, la valeur d'une suite ne dépende que des valeurs se trouvant dans une *fenêtre temporelle de taille fixée*.

---

[1] Dans un langage comme HASKELL, qui offre une notion de stream fondée sur l'évaluation paresseuse, on ne dispose pas d'opérateur comme `next` et on a donc une interprétation temporelle de la succession des éléments du stream. Cependant, comme en LUCID, cette interprétation n'est plus cohérente quand on considère deux streams à la fois.

En effet, si ce n'est pas le cas, au fur et à mesure qu'on avance dans le temps et dans la suite des valeurs du stream, il faut garder en mémoire de plus en plus de valeurs passées afin de pouvoir calculer les valeurs futures. Nous désirons donc imposer une propriété de *mémoire bornée*.

Par exemple la fonction `f` donnée ci-dessus, est une fonction qui respecte la propriété de mémoire bornée : pour calculer le `n`ème élément du résultat, il suffit de connaître la valeur d'indice (`n-1`) et de lui ajouter le `n`ème élément de l'argument. Nous discuterons au §II.2.4 un exemple de suite séquentielle à mémoire non bornée.

II.2.2.c. Suites ***temporisées***

Si on observe les "cases mémoires" d'un ordinateur pendant l'exécution d'un programme, on peut enregistrer des suites qui sont *séquentielles* et à *mémoire bornée* : elles sont séquentielles car la valeur d'une case mémoire est calculée par le programme en fonction des valeurs présentes dans la mémoire (un programme ne peut pas utiliser pour faire un calcul, des valeurs qu'il ne calculera que plus tard) ; et ces suites sont à mémoire bornée car le calcul de la valeur d'une case mémoire ne peut dépendre que des valeurs, en nombre fini, d'autres cases mémoires de l'ordinateur.

Mais le concept de suite séquentielle à mémoire bornée ne rend pas très bien compte du fait que lors de l'exécution d'un programme, les cases mémoires "voient défiler" des valeurs à des *rythmes différents*. La notion de stream LUCID ne permet pas non plus de distinguer deux suites de valeurs identiques mais produites à des rythmes différents. Prenons par exemple le programme `C` suivant :

```
int i, j, k;
for (i = 0; i < 3; i++)
     for (j = 0; j < 2; j++)
          k = i+j;
```

et notons les valeurs successives des variables `i`, `j` et `k` : on observe les suites (Cf. figure II.1) :

```
        i ⇒  < 0;       1;       2 <
        j ⇒  < 0;  1;  0;  1;  0;  1 <
k = i+j ⇒  < 0;  1;  1;  2;  2;  3 <
```

Ces suites n'ont pas le même nombre d'éléments bien qu'elles aient été observées pendant la même durée de temps, car les indices "ne vont pas à la même vitesse". Dès lors, faire l'addition des suites `i` et `j`, en respectant une propriété de séquentialité, ne peut se faire en additionnant naïvement les suites élément par élément : le 2ème élément de la suite `i` est dans le "futur" du 2ème élément de la suite `j`. L'addition naïve de `i` et de `j`, celle qui est faite en LUCID, donne pour résultat : `<0; 2; 2<` ce qui est bien différent des valeurs observées pour `k`.

Le problème vient du fait qu'un stream en LUCID ne tient compte que de la succession des valeurs pour *une seule* variable. Or on ne peut pas associer un écoulement absolu du temps au passage des valeurs dans une suite : s'il est vrai que du temps passe chaque fois qu'on passe d'un élément à un autre dans une suite, la durée écoulée dépend de la suite. Cette durée est figurée typographiquement par un espacement plus ou moins grand dans l'exemple ci-dessus, où nous avons essayé de mettre sur une même verticale les éléments des suites qui occurrent dans la même tranche de temps.

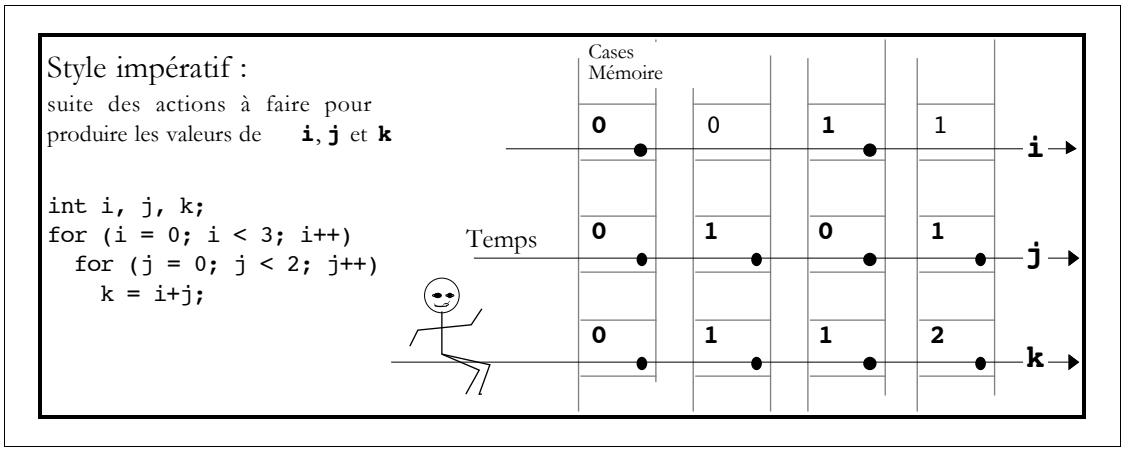


Figure II.1 : *Suites des valeurs observées pour les variables* `i`, `j` *et* `k` *dans un programme* `C`.

Il faut donc enrichir notre concept de suite séquentielle à mémoire bornée pour tenir compte d'une façon ou d'une autre de la durée qui s'écoule quand on passe d'un élément à un autre.

Nous appellerons *suite temporisée*, une suite qui permet de distinguer des *successions* identiques de valeurs se produisant à des instants différents. La situation est analogue a celle de la structure de données « arbre » : deux arbres peuvent avoir les *mêmes feuilles dans le même ordre*, mais posséder des arrangements différents pour les nœuds intérieurs (Cf. figure II.2).

Les suites temporisées sont indispensables quand on veut décrire le *couplage* de plusieurs systèmes dynamiques discrets. Chacun de ces systèmes a son propre système de datation : il évolue à un *rythme* qui lui est propre et l'indiçage de la suite des états du système correspond à des événements qui lui sont propres. Si on veut décrire le système résultant de la composition de ces deux (sous-) systèmes, il faut plonger les instants `t` propres à chaque système dans un temps global. Ce plongement introduit des "trous", des "vitesses de progression", des "instants initiaux" différents, …, suivant que les deux systèmes partagent ou ne partagent pas tel ou tel événement. La composition de systèmes dynamiques ayant leur temps propre, impose donc de construire une notion de trajectoire un peu plus compliquée que celle de suite simple[1].

[1] Une trajectoire doit être une fonction *partielle* de `I N`, et non une fonction totale. Le domaine de définition de la trajectoire correspond alors aux instants où il se passe quelque chose.

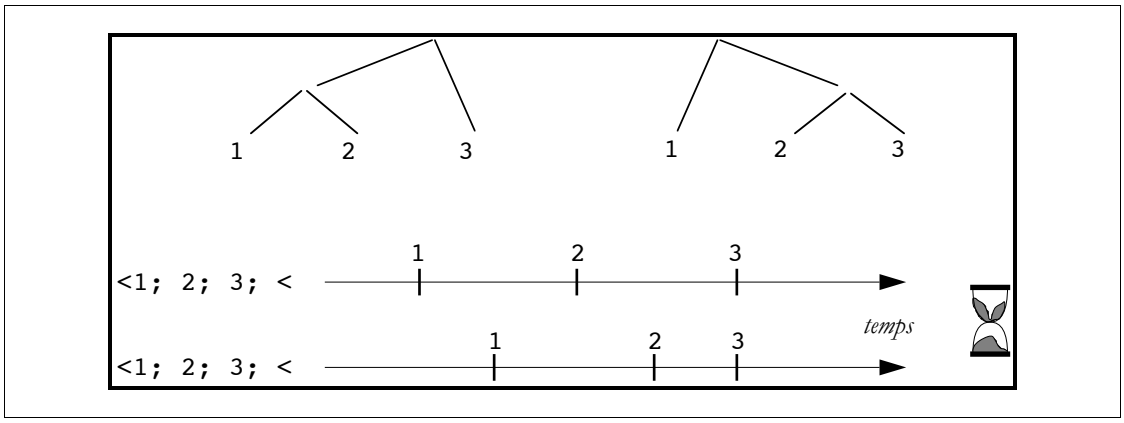


Figure II.2 : *En haut de la figure : deux arbres différents ayant les même feuilles mais des structures différentes . En bas de la figure : deux successions identiques de valeurs qui correspondent à des suites temporisées différentes.*

II.2.2.d. Deux exemples incorrects de suite temporisée

La manière la plus évidente de réaliser des suites temporisées est d'*associer explicitement une date absolue* à chaque élément de la suite. Nous rejetons cette approche car elle présente de nombreux inconvénients :

- Trois systèmes de datation absolue coexistent dans une simulation : le temps du modèle réel, le temps logique exprimé par le programme de simulation, le temps physique du système de calcul qui évalue le programme de simulation.
- Pour ce qui est du temps du modèle, on ne s'intéresse souvent qu'à la succession de ces valeurs, à l'articulation de leur ordonnancement et pas à la durée réelle des processus. Par exemple, il peut très bien ne pas exister dans le modèle d'horloge globale permettant de dater chaque événement.
- Le temps du système de calcul n'est évidement pas pertinent pour la datation des événements du modèle.
- On demande bien évidement au temps logique de la simulation de respecter les relations de succession qui sont vraies dans le temps du modèle[1]. On ne lui demande pas de respecter les durées du modèle réel (sinon on ne pourrait plus par exemple faire de *prévision* météorologique).

Une autre solution possible, pour représenter des suites temporisées, consiste à se dire que les difficultés soulevées par l'addition naïve de `i` et de `j` disparaissent si, pour chaque valeur de `j`, il y a une valeur *correspondante* de `i`. Cette approche consiste à "normaliser" les suites de manière à ce que la *durée* qui s'écoule entre éléments successifs d'une suite soit toujours *la même*, et cela pour toutes les suites. De manière imagée sur l'exemple précédent de la boucle de calcul, cette solution consiste à observer toutes les cases mémoires au même instant.

Cependant, cette méthode ne conserve pas assez d'information : si on n'observe pas assez fréquemment, des changements de valeurs vont passer inaperçus. Si on observe trop souvent, on va noter comme ayant

[1] Cependant, le programme de simulation exprime des calculs qui correspondent à l'écoulement d'un temps physique. L'interprétation suivante permet de conciler ces trois systèmes de datation : le compilateur s'arrange pour ordonnancer les calculs de la simulation afin que tout se passe comme si on disposait d'un calculateur idéal où les calculs s'effectuent instantanément. Ainsi, on néglige en 81/2 la durée nécessaire à l'élaboration physique des valeurs. Dans l'exemple des boucles imbriquées, cela revient à considérer que la 1ère valeur de `i` intervient dans le même instant que la 1ère valeur de `j`. Autrement dit, la durée physique des calculs (comme par exemple le déroulement des instructions nécessaires au contrôle de la boucle `j`) est considérée comme nulle. Une telle hypothèse est faite par exemple dans les langages temps-réel synchrones comme ESTEREL, LUSTRE ou SIGNAL : c'est l'hypothèse dite de fort synchronisme.

changé, des cases mémoires qui n'ont pas changé de valeur. La situation se complique encore, car une case mémoire peut très bien changer de valeur pour reprendre la même valeur. Avec cette approche, il n'est donc pas possible de modéliser des quantités comme « le nombre de fois où une case mémoire change de valeur ».

II.2.2.e. Comment représenter la notion de suite temporisée

Nous ne pouvons donc pas nous passer de la notion de suite temporisée. Plusieurs solutions équivalentes sont possibles pour représenter une suite temporisée à l'aide des suites habituelles. L'idée est de noter la valeur de chaque stream chaque fois "qu'il se passe quelque chose", i.e. chaque fois qu'un des streams du programme progresse, et de noter en même temps par un booléen si chaque stream a progressé ou non.

Une suite temporisée `X` sera donc représentée par *deux* suites habituelles v`X` et h`X`. La suite v`X` représente l'observation des valeurs de `X` et la suite h`X` indique si l'élément correspond à un changement de valeur de `X`. On appelle v`X` la *suite des valeurs* de `X` et h`X` s'appelle l'*horloge* de `X`.

En reprenant l'exemple des boucles imbriquées du §II.2.2.c :

```
i =  < 0;    1;    2 <
hi = < V; F; V; F; V; F <                avec V = vrai, F = faux
vi = < 0; 0; 1; 1; 2; 2 <

j =  < 0; 1; 0; 1; 0; 1 <
hj = < V; V; V; V; V; V <
vj = < 0; 1; 0; 1; 0; 1 <

k =  < 0; 1; 1; 2; 2; 3 <
hk = < V; V; V; V; V; V <
vk = < 0; 1; 1; 2; 2; 3 <
```

Cet exemple appelle plusieurs remarques :

- Il est facile de reconstruire `i` à partir de v`i` et h`i` : ce sont les éléments de v`i` tels que l'élément correspondant de h`i` a la valeur booléenne `vrai`. Il en va de même pour `j`, `k`, etc.
- La valeur de v`i` quand l'élément correspondant de h`i` a pour valeur `faux`, est la même que la valeur précédente de v`i`. Cela correspond à la constatation qu'une valeur observée reste la même s'il n'y a pas eu de changement.
- Le *rang* d'un élément de v`i` ou de h`i` correspond à un *instant d'observation* de la mémoire. Le passage d'un élément à un autre dans les suites v`X` et h`X` correspond au passage du temps. Les suites v`i` et v`j` (ou h`i` et h`j`, ou v`i` et h`k`, etc.) sont *synchrones :* le $\mathtt{n}^{\text{ème}}$ élément de ces suites correspond au même instant d'observation de `i`, `j` ou `k`.
- En conséquence, le calcul de v`k` est facile : il s'agit de l'addition élément par élément des suites v`i` et v`j`. On remarque que la valeur de `i` est disponible pour le calcul, même entre deux instants où `i` change (i.e. même quand h`i` a pour valeur le booléen `faux`). Cela correspond au fait que nous voulons pouvoir observer la valeur d'une variable, même entre deux changements de cette valeur.
- De même, le calcul de h`k` est simple : il s'agit du `ou logique`, élément par élément, des suites h`i` et h`j`. En effet, la valeur de `k` change chaque fois que la valeur de `i` *ou* de `j` change, puisque par définition, `k = i+j`.
- Nous avons introduit l'horloge du stream `X` en disant que c'était une suite de booléens indiquant un changement dans la valeur de `X`. Cette définition est à prendre *au sens large* : il est possible que `X` change de valeur pour reprendre éventuellement la même valeur. Par exemple, si nous remplaçons la définition de `k` dans les boucles imbriquées par la définition suivante : « `k = j – j` », la valeur de `k` va rester identique à `0` pendant le parcours des boucles. Mais l'horloge de `k` est celle de `j` car chaque fois

que `j` change, il faut refaire une soustraction et écrire le résultat dans la case mémoire correspondant à `k` (et il se trouve seulement que ce résultat est identique au résultat de la soustraction précédente).

II.2.3. Stream 8½ = suites séquentielles temporisées à mémoire bornée

Nous pouvons à présent définir l'objet qui nous servira à modéliser une suite de valeurs dans le temps :
un **stream 8½** est une suite temporisée qui est une
fonction séquentielle à mémoire bornée d'autres streams 8½.

En fait, un stream 8½ est une structure un peu plus complexe car elle intègre aussi un aspect spatial. Cet enrichissement sera abordé dans le chapitre suivant. En attendant, dans la suite de ce chapitre, les valeurs d'un stream 8½ seront appelées des scalaires et elles correspondent à des entiers, des booléens, des flottants, etc. Dans la suite, et sauf mention explicite, le terme de stream réfère à la notion 8½ de stream et le terme de suite réfère à la notion usuelle de suite.

II.2.3.a. Définitions des notions de tic, top, instant et événement

Une suite temporisée `X` est un couple (v`X`, h`X`), où v`X` et h`X` sont des suites. Le rang d'un élément de v`X` ou de h`X` s'appelle un **tic**. Un tic sur lequel h`X` a pour valeur le booléen `vrai` est un **top** de `X`.

Tic et top s'interprètent de la manière suivante : soit S un système décrit par les variables d'état `X` et `Y`

- Les tics de `X`, qui sont aussi ceux de `Y`, correspondent aux instants où le système S change d'état. Ce sont les instants du temps concernant le système. Les autres instants du temps ne sont pas pertinents pour la description de l'évolution du système.
- Les tops de `X` correspondent aux instants où la variable `X` est *susceptible de changer* de valeur (et donc S est aussi susceptible de changer d'état). Un tic qui n'est pas un top pour le stream `X` correspond à la survenue d'un événement qui n'a pas de conséquence sur le stream `X` ; c'est un événement qui à une influence sur un autre stream du système modélisé.

La figure II.3 représente les relations entre les instants du temps, les tics du système S et les tops des streams `X` et `Y`.

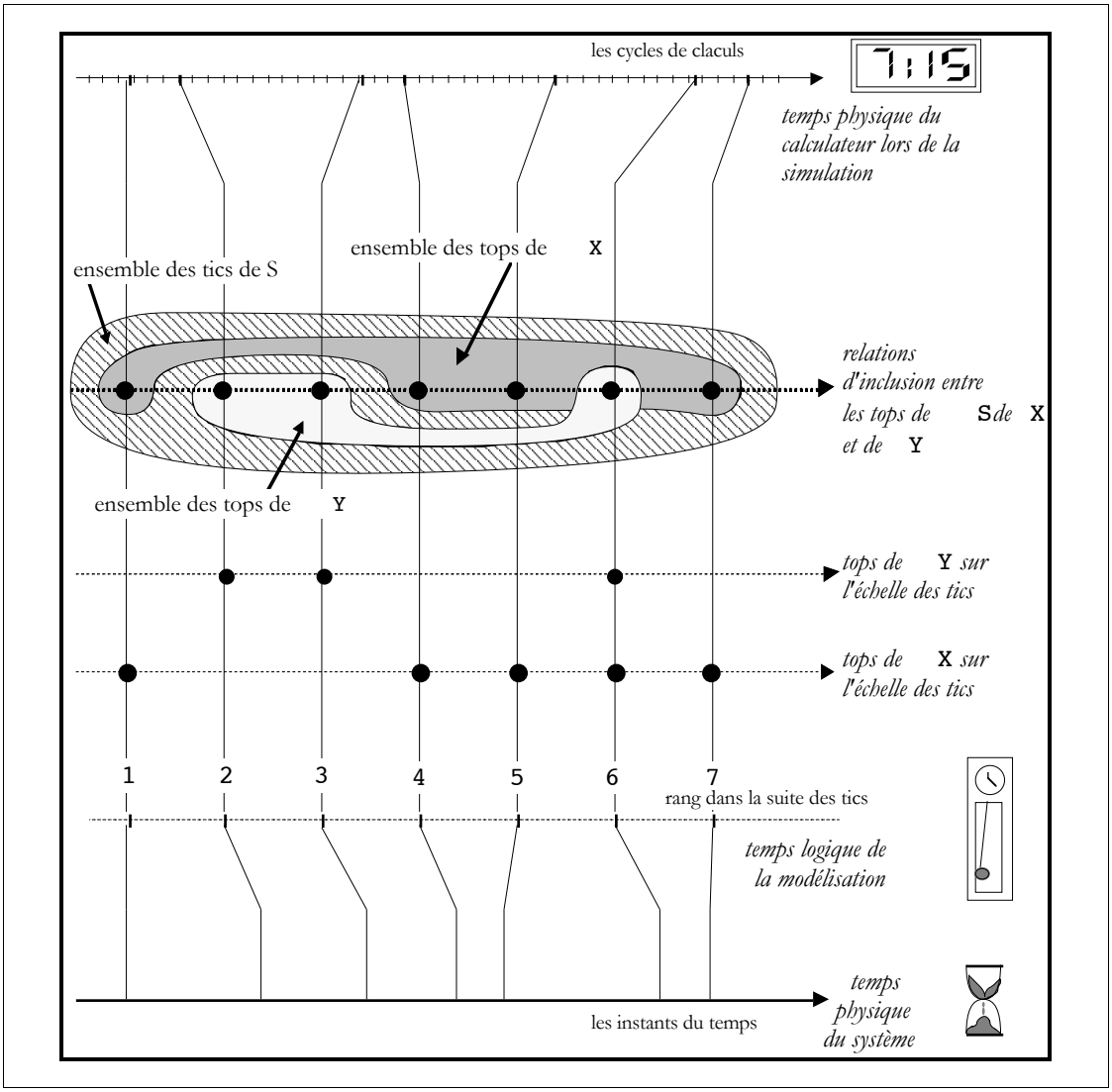


Figure II.3 : *Relations entre les instants du temps du phénomène (temps réel), le temps du calculateur pendant la simulation et les tics (temps logique) et les tops d'un système S décrit par deux streams correspondant à l'observation des deux variables d'état* ***X*** *et* ***Y****.*

On peut numéroter les tics puisqu'ils correspondent à un rang dans une suite. En 81/2, la numérotation commence par convention à 0. La numérotation des tics est une datation logique qui rend compte de la succession des phénomènes et des calculs, mais elle est indépendante de la durée qui s'écoule effectivement dans le temps réel ou le temps du calcul (Cf. la figure II.3).. Par contre cette numérotation est temporellement cohérente et le passage d'un tic au tic suivant (ou d'un top au top suivant) correspond à du temps qui passe car :

- la succession des tics correspond à l'ordonnancement linéaire des instants du temps ;
- la valeur d'un stream 81/2 en un tic donné ne peut dépendre que des valeurs des streams 81/2 en des tics précédents (propriété de séquentialité des streams qui correspond à la *causalité des trajectoires*, Cf. le chapitre précédent) ;
- la comparaison de deux streams sur le même tic correspond à la comparaison de deux valeurs au même instant et est donc temporellement cohérente.

Par conséquent, la succession des tics correspond à une notion *discrète* du temps. Par ailleurs, un tic correspond à la modification de la valeur d'un des streams qui composent le système modélisé. La

modification d'un stream correspond à un *événement* (par exemple le changement d'une valeur dans la mémoire d'un ordinateur, une observation d'un phénomène, etc.). Le passage d'un tic au tic suivant ne correspond pas à une durée fixée mais à l'*occurrence d'un événement*. Le modèle du temps qui est en jeu ici est appelé *événementiel* : le temps s'écoule uniquement quand il se passe quelque chose. Ce modèle du temps est bien connu des informaticiens ; par exemple, il est mis en œuvre dans la *simulation par événements discrets*.

II.2.3.b. Streams correspondant à un nombre fini d'événements

Une variable d'état peut changer de valeur un nombre fini de fois. Cependant, l'observation de la valeur de cette valeur se déroulera "du début à la fin des temps". Par convention, la fin des temps ne se produit jamais, et donc les suites horloge et valeur qui constituent les streams sont des suites infinies. La manipulation des streams correspond alors à la manipulation uniforme de suites qui sont toujours infinies, alors que l'utilisation de suites finies auraient conduit à combiner à la fois des suites finies et des suites infinies.

Une trajectoire qui a un nombre fini d'éléments correspond à une variable qui ne change plus de valeur au bout d'un certain temps, et donc à un stream dont l'horloge a toujours la valeur `faux` à partir d'un certain rang. Par exemple, si on reprend la variable `i` du programme `C` utilisé au §II.2.2.c, le stream 81/2 associé à `i` sera :

```
i =  < 0;    1;    2 <
hi = < V; F; V; F; V; F; F; F; F; F; F; F … <
vi = < 0; 0; 1; 1; 2; 2; 2; 2; 2; 2; 2; 2 … <
```

Le stream vide, noté `<<`, est un cas particulier de trajectoire à nombre fini d'éléments, à savoir aucun. Par suite, l'horloge h`<<` du stream vide est une suite qui a toujours pour valeur booléenne faux car à aucun instant, on n'a pu faire une observation de la valeur de la suite. Par convention, sa suite de valeurs v`<<` est une suite de valeurs indéfinies ⊥ :

```
h<<  = <F;  F;  F;  F;  F; … <
v<<  = <⊥;  ⊥;  ⊥;  ⊥;  ⊥; … <
```

Nous appellerons *domaine de définition* d'un stream, l'ensemble des tics sur lesquels la valeur du stream est définie (i.e. n'est pas ⊥). Un stream est défini à partir de l'instant de son 1er top. On peut donc remarquer que cet ensemble est soit l'ensemble vide ∅, soit un intervalle de la forme `[n, +∞[`.

II.2.3.c. Une définition des streams en terme de streams

La définition d'un stream fait appel à des suites usuelles pour représenter v`X` et h`X`. Cependant, une suite de valeurs v`X` et une horloge h`X` peuvent aussi être vues comme des streams. Leurs horloges hv`X` et hh`X` sont alors égales et correspondent à la suite qui comporte une infinité d'éléments tous à `vrai` ; leurs valeurs vv`X` et vh`X` sont identiques à v`X` et h`X` :

```
hi = < V; F; V; F; V; F; F … <    vi = < 0; 0; 1; 1; 2; 2; 2 … <
vhi= < V; F; V; F; V; F; F … <    vvi= < 0; 0; 1; 1; 2; 2; 2 … <
hhi= < V; V; V; V; V; V; V … <    hvi= < V; V; V; V; V; V; V … <
```

Un stream dont l'horloge est une suite de valeurs booléennes `vrai` est dit *normalisé*.

On peut donc définir les streams de la manière suivante : l'ensemble des streams, nommé `STREAM`, est un ensemble sur lequel on a défini deux opérations, h et v, et qui contient un élément distingué `Clock` (qui correspondrait en terme de suites, à une suite de booléens tous à vrai). On demande en plus que si `X`

appartient à STREAM, alors hhX = Clock, hvX = Clock et vvX = vX (techniquement, v est idempotente et h est Clock-potente).

II.2.4. Les streams 8½ versus les suites à mémoire non-bornée et versus les suites instantanées

II.2.4.a. Stream 8½ et suite à mémoire non-bornée

On peut se poser la question de savoir s'il y a des systèmes que l'on ne peut pas modéliser avec un formalisme satisfaisant à la propriété de mémoire bornée. Imaginons par exemple un système dynamique (X) dont la valeur à l'instant t+1 dépend des états du système aux temps t, t−1, t−2, …, 0 :

$$X(t+1) = F\ \{X(t), X(t-1), \ldots, X(0)\}$$

La fonction F dépend de l'*ensemble* des états passés de X. La fonction F est donnée a priori et ne peut pas changer au cours du temps (c'est le déterminisme du système). Quel peut être la forme de la fonction F ? Prenons par exemple :

$$F\ \{X(i) : i \in I\!N,\ i \leq t\} = \sum_{i=0}^{t} X(i)$$

Mais avec une fonction de cette forme, X(t+1) ne dépend pas vraiment des t états précédents : il suffit de rajouter un seul stream S pour obtenir un système équivalent et qui est, lui, à mémoire bornée :

$$\begin{cases} X(0) = \ldots \\ X(t+1) = \sum_{i=0}^{t} X(t) \end{cases} \quad \equiv \quad \begin{cases} S(0) = X(0) \\ S(t+1) = X(t) + S(t) \\ X(0) = \ldots \\ X(t+1) = S(t) \end{cases}$$

Le nouveau système (S, X) est à mémoire bornée puisque le calcul de son état à l'instant t ne fait intervenir que son état à l'instant t−1. Cela veut dire que le système initial X n'était pas en essence non-borné.

Pour empêcher la réduction du système (X) à un système à mémoire bornée, on peut modifier la fonction F afin qu'elle dépende de chaque état du passé de manière différente à chaque instant :

$$F\ \{X(i) : i \in I\!N,\ i \leq t\} = \sum_{i=0}^{t} a_i(t)\, X(i)$$

à chaque instant t, la fonction F fait la somme pondérée par $a_i(t)$ des états passés X(i). Les coefficients de pondération $a_i(t)$ dépendent de l'instant t. Ce type de système dynamique ne peut pas être représenté par des suites à mémoire bornée. Remarquons cependant que la définition du système demande la donnée d'une infinité de valeurs : les $(a_i(t))_{i<t}$.

Existe-t'il des phénomènes naturels ou artificiels dont la description demande la spécification de manière extensive d'une infinité de valeurs ? Une fonction d'une variable continue correspond à une infinité de valeurs (par exemple, l'image d'un film, Cf. §I..4.2.a). Cependant si on discrétise ce modèle continu, on retombe sur un nombre fini de valeurs (la lumière aux points de discrétisation de l'image).

Par ailleurs l'invariance des lois d'évolution de la physique, i.e. ce sont les même lois qui s'appliquent à tout instant du temps (et en tout point de l'espace), justifie notre intérêt pour des systèmes de description à mémoire bornée. En effet, essayons d'imaginer la description d'un système dont la loi d'évolution dépend du temps. Si la loi d'évolution dépend *fonctionnellement* du temps, on peut faire entrer le temps dans les variables de description du système et on se ramène ainsi à une évolution indépendante du temps (mais avec une fonction d'évolution plus compliquée et un espace d'états un peu plus riche, Cf. l'exemple ci-dessus). Si la loi

d'évolution du système ne peut s'exprimer fonctionnellement en fonction du temps, alors la description du système doit comprendre la donnée explicite de toutes les lois spécifiques, et il y en aura autant que d'instants du temps (ou de points de l'espace). On peut douter du caractère effectif d'une telle description : dans le cas d'un système discret, un modèle qui ferait intervenir une infinité de valeurs différentes serait caduque. Autant décrire le phénomène par la suite littérale de ce qui se passe[1].

C'est pourquoi nous restreignons les streams 81/2 aux streams qui sont des combinaisons à mémoire bornée d'autres streams.

II.2.4.b. Comparaison des streams 81/2 et des streams LUSTRE et SIGNAL

Les langages LUSTRE [CAS 87] et SIGNAL [LEG 86] sont des langages fondés sur un modèle data-flow synchrone dérivé de LUCID, qui introduisent une notion de stream tout à fait semblable à celle que nous venons de présenter, afin de pouvoir spécifier des système réactifs temps-réel. C'est LUSTRE et SIGNAL qui ont introduit, par rapport à LUCID, la restriction de combinaison séquentielle et de mémoire bornée.

Cependant, en raison de leur vocation temps-réel, ces langages s'intéressent principalement à la simultanéité des événements : leur but est de pouvoir spécifier des comportements du type : « à midi faire telle action » ou bien « sur telle occurrence d'un événement, exécuter telle action ». Par suite, ils ne s'intéressent principalement qu'à la combinaison de streams ayant la même horloge. Des techniques ont été développées pour interdire les programmes combinant des streams d'horloges différentes. Par exemple, la valeur d'un stream LUSTRE ou SIGNAL ailleurs qu'en un top est indéfinie.

L'approche de 81/2 est différente : en 81/2 on peut faire l'addition `Z = X+Y` de deux streams d'horloges différentes, l'horloge résultante étant l'union des horloges des deux streams (Cf. l'exemple des boucles imbriquées `i` et `j` donné au §II.2.2.c). Cette opération entre streams 81/2 suppose que :

- la valeur courantes des streams `X` et `Y` est disponible, s'il le faut, pendant les instants séparant deux changement de valeurs ;
- chaque fois que l'un des deux streams `X` ou `Y` (ou les deux) est susceptible de changer de valeur, le stream `Z` est susceptible de changer de valeur (l'horloge h`Z` est construite à partir des horloge h`X` et h`Y` par le compilateur 81/2 pour respecter cette contrainte).

Ce type de comportement est relié en 81/2 aux opérateurs qui permettent de combiner les streams. Cependant, rien n'empêche d'introduire d'autres opérateurs, ayant un autre comportement. Par exemple, on pourrait introduire une autre définition de `+`, notée ⊕, et qui correspondrait à l'addition synchrone des streams comme en LUSTRE/SIGNAL. On peut définir l'opération ⊕ par l'horloge et la valeur du résultat :

```
hX = < hx0; hx1; … <      vX = < vx0; vx1; … <
hY = < hy0; hy1; … <      vY = < vy0; vy1; … <

h(X⊕Y) ⇒ < hx0 & hy0;
           hx1 & hy1;
           … <
```

---

[1] Cependant on peut considérer que les $a_i(t)$ correspondent à des entrées du système `X` et qu'ils sont produits par un autre système non modélisé par nécessité ou par commodité. Il peut exister ainsi une manière finie de décrire les a_i et le système formé par la réunion du système `(X)` plus le système de production des $a_i(t)$ peut être un système à mémoire bornée. La non-finitude du système `(X)` provient de sa description partielle. Il n'est alors pas possible de calculer les `X(t)`, on ne peut plus faire de simulation, mais on peut quand même vouloir étudier théoriquement le système `(X)` partiellement spécifié. Par exemple, la loi d'évolution donnée plus haut est linéaire pour les `X(i)`, si on considère la variation des $a_i(t)$ très lente devant celle des `X(i)`, ce qui permet de déduire certaines propriétés sur la trajectoire du système, alors que si on considère explicitement le système total, on ne fait plus cette approximation et cette propriété n'est plus immédiatement perceptible.

```
v(X⊕Y) ⇒ < si hx0 & hy0 alors vx0 + vy0 sinon ⊥;
           si hx1 & hy1 alors vx1 + vy1 sinon ⊥;
           … <
```

Les opérations sur les streams 81/2 sont définies par leurs effets sur les horloges et les valeurs, quelques soient ces effets. Les techniques de typage, d'évaluation et de compilation qui sont développées pour 81/2 sont générales et indépendantes d'une définition particulière d'opérateur.

Un jeu donné d'opérateurs définit les relations de succession et de simultanéité qui sont exprimables entre les éléments d'un stream (en restant dans un modèle du temps discret séquentiel totalement ordonné). Les techniques de compilation en 81/2 ne supportent pas un jeu de relations plutôt qu'un autre, du moment que les opérateurs s'expriment à travers leurs effets sur v et h. Cependant, les opérateurs décrits dans le compilateur 81/2 actuel, correspondent au modèle du temps présenté plus haut à travers l'exemple des boucles imbriquées, où une valeur persiste et reste disponible jusqu'au prochain changement. Ce modèle du temps n'est pas celui de LUSTRE ni de SIGNAL qui développent une conception instantanée des valeurs.

II.3. Opérations sur les streams

II.3.1. Streams constants

II.3.1.a. Streams correspondants aux constantes scalaires

Un type scalaire est un type de base : booléen, entier, flottant, etc. Toutes les constantes scalaires ***s*** peuvent être étendues en un stream constant **s** possédant un seul top, au tic 0, et dont la valeur est le scalaire ***s***. On utilise la même écriture pour dénoter le stream constant et le scalaire correspondant. Ainsi, « **17** » dénote le stream constant

```
h17 = < V;  F;  F;  F; … <
v17 = <17; 17; 17; 17; … <
```

De même, « **true** » dénote le stream :

```
htrue =   < V; F; F; F; … <
vtrue =   < V; V; V; V; … <
```

Il n'y a pas d'ambiguïté dans un programme 81/2 entre la dénotation des scalaires et des streams constants car on n'y utilise jamais de scalaires en tant que tels.

II.3.1.b. Streams constants correspondant à des horloges prédéfinies

Il existe aussi des streams prédéfinis de type booléen dont la valeur est toujours **vraie** mais qui ont une infinité de tops. Ces constantes sont utilisées dans les programmes non pas pour leur valeur (qui ne varie jamais), mais pour leur horloge (qui servira à créer d'autres horloges intéressantes). Ces constantes se dénotent à l'aide du mot-clé **Clock** et sont de trois sortes :

- **Clock** (le mot-clé sans argument)

  qui correspond à une horloge comportant une infinité de tops :

```
hClock =  < V; V; V; V; … <
vClock =  < V; V; V; V; … <
```

Cette constante est différente du stream constant `true` (i.e. le stream obtenu à partir de la constante scalaire booléenne ***`true`***) car `Clock` a une infinité de tops, alors que `true` a un seul top à l'instant 0.

- `Clock n` (le mot-clé avec un argument)

  où n est un nombre positif, est un stream de booléen, ayant toujours la valeur `vrai`, qui a un top tous les n tops du stream `Clock` :

```
h(Clock 3) =   < V; F; F; V; F; F; V; F; … <
v(Clock 3) =   < V; V; V; V; V; V; V; V; … <
```

- `Clock -n`

  où n est un nombre positif, est un stream booléen, dont la valeur est toujours `vrai`, et qui, *en moyenne*, a un top tous les n tops du stream `Clock`, la distribution des tops étant uniforme. Un exemple d'exécution possible est :

```
h(Clock 2) =   < V; F; F; V; V; F; V; F; V; F; … <
v(Clock 2) =   < V; V; V; V; V; V; V; V; V; V; … <
```

On peut se demander pourquoi ne pas avoir choisi pour les constantes scalaires étendues en streams, une horloge comportant une infinité de tops (par exemple la même que celle du stream `Clock`, ce qui aurait évité d'introduire cette constante). Cela ne nous semble pas approprié pour deux raisons :

– L'élaboration d'une valeur scalaire constante ne nécessite des calculs qu'une seule fois, "au début des temps". Il est donc logique que cela soit rendu par un stream ayant un seul top.

– L'horloge d'une expression comme « `S + 1` » *doit être* naturellement la même que l'horloge de `S`. Cela n'est pas possible si le stream constant `1` a ses propres tops. En 81/2, lcette expression ne sera définie que quand `S` sera définie. Cette définition ne pouvant survenir que pour un tic supérieur ou égal à `0` (rappelons que le temps commence avec le tic `0`), et l'unique top de `1` survenant en `0`, l'horloge de « `S + 1` » est exactement celle de `S`.

II.3.2. Extension des opérateurs scalaires

Toutes les opérations sur des valeurs de type scalaire peuvent s'étendre naturellement pour devenir des opérations entre streams. Le stream résultant est le stream des combinaisons par l'opération scalaire des valeurs instantanées des streams arguments. C'est l'exemple de l'addition entre deux streams donné plus haut. Un autre exemple est la conditionnelle "à la Lisp" qui s'étend aussi sur les streams et qui est une fonction à trois arguments.

Cependant, il faut noter que :

– Les extensions des opérations scalaires sont *strictes* : cela veut dire que le stream résultat n'a une valeur définie que quand tous les streams arguments ont leur valeur définie.

  Cela est valable même pour la conditionnelle, alors qu'usuellement « `if SI then ALORS else SINON fi` » est défini suivant la valeur de l'expression booléenne `SI` : si cette expression prend la valeur booléenne `vrai` alors la conditionnelle est définie quand l'expression `ALORS` est définie, même si l'expression `SINON` n'est pas définie (et inversement quand l'expression `SI` prend la valeur `faux`[1]).

[1] La conditionnelle est dite stricte en son premier argument car il faut que la valeur de la condition soit définie pour que la conditionnelle soit définie, mais elle n'est pas stricte en ses autres arguments.

– Sur son domaine de définition, l'horloge d'une extension d'expression scalaire est donnée par le *ou* logique des horloges de ses arguments : le stream résultat "progresse" chaque fois qu'un des arguments au moins progresse.
– Quand on a besoin de la valeur d'un stream à un tic donné, et que cet instant n'est pas un top pour le stream, la valeur utilisée est la valeur calculée au top précédent. Si le top précédent n'existe pas, le stream résultant n'est pas défini et sa valeur est ⊥.

Détaillons un exemple pour la conditionnelle. Soient trois streams définis par :

```
hSI  = <V; V; V; V; V; … <
vSI  = <V; F; V; F; V; … <

hALORS    = <V; F; F; F; F; … <
vALORS    = <1; 1; 1; 1; 1; … <

hSINON    = <F;  V;  F;  V;  F; … <
vSINON    = <⊥; 10; 10; 20; 20; … <
```

Le domaine de définition de « `if SI then ALORS else SINON fi` » est donné par l'intersection des domaines de définitions : [0, +∞[∩ [0, +∞[∩ [1, +∞[. Sur le domaine de définition, un tic est un top si c'est un top pour l'un des trois arguments `SI`, `ALORS` ou `SINON`. Donc :

```
h(if SI then ALORS else SINON fi) = <F;  V;  V;  V;  V; … <
v(if SI then ALORS else SINON fi) = <⊥; 10;  1; 20;  1; … <
```

II.3.3. Retard

La valeur en un top d'un stream retardé `$T` est la valeur de `T` au top précédent. En terme de suite, l'opération de retard peut se voir comme un décalage "vers la droite". Le stream retardé n'est défini qu'à partir du 2ème top de `T` : en effet, la valeur précédant le premier top de `T` n'existe pas et la valeur correspondante du stream retardé est donc ⊥. Par exemple :

```
   hT  = < F;   V;   F;   V;   F;   V … <
   vT  = < ⊥;   1;   1;   2;   2;   3 … <
h($T)  = < F;   F;   F;   V;   F;   V … <
v($T)  = < ⊥;   ⊥;   ⊥;   1;   1;   2 … <
```

les tops ont été figuré en gras, le tic correspondant au premier top de `T` a été souligné et le tic correspondant au deuxième top de `T` est mis en italique

Remarquons que si le stream `T` possède un seul top, alors le stream retardé `$T` ne possède aucun top.

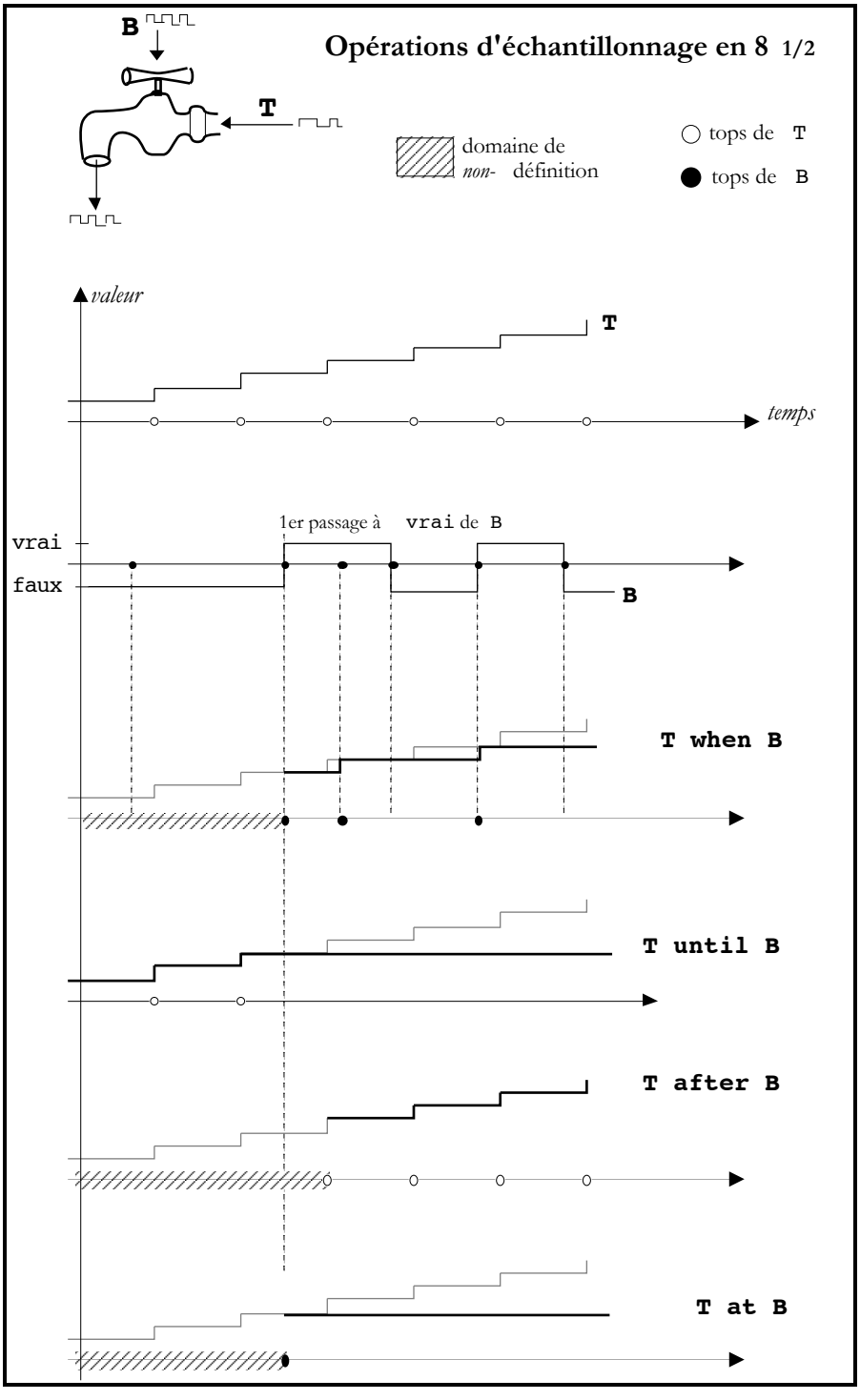


Figure II.4 : *Les opérations d'échantillonnage en 81/2. Nous avons représenté le résultat des opérations d'échantillonnage à l'aide d'un chronogramme. Il ne faut pas se laisser abuser par cette représentation utilisant des segments : les streams sont des entités discrètes et les tops sont portés sur les axes figurant les streams.*

II.3.4. Échantillonnage

L'échantillonnage `T when B` consiste à sélectionner un sous-stream `E` dans un stream `T` grâce à un stream booléen `B`. Le stream résultant `E` est le stream des valeurs de `T` observées sur les tops de `B` qui ont la valeur booléenne `vrai` (Cf. figure II.4).

```
hT = <F; V; F; V; F; V …<     vT = <⊥; 1; 1; 2; 2; 3 …<
hB = <V; V; F; V; F; V …<     vB = <F; V; V; V; V; F …<
hE = <F; V; F; V; F; F …<     vE = <⊥; 1; 1; 2; 2; 2 …<
```

Il faut remarquer que l'extension aux streams de la conditionnelle `if … then … else … fi` et l'échantillonnage `when` sont des opérations essentiellement différentes l'une de l'autre : on ne peut écrire l'une uniquement à l'aide de l'autre. En effet, l'horloge d'une conditionnelle correspond à la réunion des horloges des arguments, alors que l'horloge du `when` est une sous-horloge de son 2ème argument. L'opérateur `when` est le seul opérateur qui permette de construire un stream dont l'horloge dépend d'une valeur calculée.

À partir de l'opérateur `when`, qui est l'opération d'échantillonnage primitive en 81/2, on peut construire d'autres opérations d'échantillonnage plus spécialisées, `until`, `after` et `at` :

- `until` correspond à un "filtre passe-bas" : la valeur et l'horloge du stream `(T until B)` sont celles de `T`, jusqu'à l'occurrence de la première valeur `vrai` de la *condition* `B`. Ensuite, le stream résultat n'a plus aucun top.
- `after` correspond à un "filtre passe-haut" : la valeur et l'horloge du stream `(T after B)` sont celles de `T` à partir de la première occurrence à `vrai` de `B`. Le stream résultat est indéfini avant.
- `at` correspond à un "dirac" : le stream `(T at B)` n'a qu'un seul top et sa valeur est la valeur observée sur `T` lors de l'occurrence de la première valeur à `vrai` de `B`.

La figure II.4 représente sous forme de chronogramme les opérations d'échantillonnage.

II.3.5. Horloges et domaines de définition des expressions entre streams

Notons d`X` le domaine de définition du stream `X :` c'est l'ensemble des tics pour lesquels la suite v`X` n'est pas ⊥. Notons t`X` l'ensemble des tops de `X` : c'est l'ensemble des tics tels que h`X` est à `vrai`. Attention, d`X` et t`X` sont des ensembles, et pas des suites comme v`X` ou h`X`.

Notons par ⊕ une opération binaire scalaire quelconque étendue aux streams (par exemple l'addition de deux streams). La notation `S(n)` où `S` est une suite, dénote la n^{ème} valeur de la suite `S`. Avec ces notations, on a les relations suivantes (Cf. figure II.5) :

$$d(cste) = \mathbb{I}\mathbb{N} \qquad d(Clock) = \mathbb{I}\mathbb{N}$$
$$t(cste) = \{ 0 \} \qquad t(Clock) = \mathbb{I}\mathbb{N}$$

$$d(A \oplus B) = d(A) \cap d(B)$$
$$t(A \oplus B) = (t(A) \cup t(B)) \cap d(A \oplus B)$$

$$d(\$A) = \{ n \mid n \in d(A) \text{ et } \exists m, m<n, hA(m) = vrai \}$$
$$t(\$A) = t(A) \cap d(\$A)$$

$$d(A \text{ when } B) = d(A) \cap d(B)$$
$$t(A \text{ when } B) = \{ n \mid vB(n) = vrai \text{ et } n \in t(B) \} \cap d(A \text{ when } B)$$

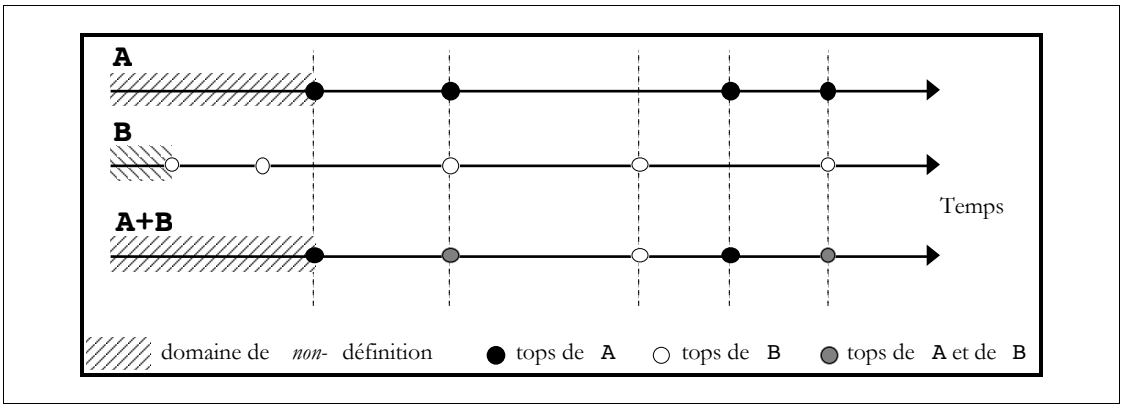


Figure II.5 : *Domaine de définition et horloge d'une expression arithmétique entre deux streams, en fonction des domaines de définition et des horloges des streams arguments.*

D'autres exemples des quatres classes d'opérations, qui illustrent les règles précédentes, sont donnés dans la table ci-dessous.

tic	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
5	5										
v5	5	5	5	5	5	5	5	5	5	5	5
h5	V	F	F	F	F	F	F	F	F	F	F
S		*2*	*3*		*2*		*6*		*7*		*8*
T	*8*		*7*		*5*	*6*	*1*	*1*	*4*		
S + T		10	10		7	8	7	7	11		12
v(S+T)	⊥	10	10	10	7	8	7	7	11	11	12
h(S+T)	F	V	V	F	V	V	V	V	V	F	V
$S			2		3		2		6		7
v$S	⊥	⊥	2	2	3	3	2	2	6	6	7
h$S	F	F	V	F	V	F	V	F	V	F	V
$5											
v$5	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
h$5	F	F	F	F	F	F	F	F	F	F	F
B	*V*		*F*	*F*	*V*	*V*	*F*		*V*		*F*
if B then S else T		2	7	7	2	2	1	1	7		4
v(if … fi)	⊥	2	7	7	2	2	1	1	7	7	4
h(if … fi)	F	V	V	V	V	V	V	V	V	F	V
S when B					2	2			7		
v(S when B)	⊥	⊥	⊥	⊥	2	2	2	2	7	7	7
h(S when B)	F	F	F	F	V	V	F	F	V	F	F

Dans la table précédente, nous avons représenté les streams de la manière suivante : le nom du stream est écrit en **gras**. Les horloges et les valeurs des streams sont données en "`maigre`". Les caractères en *italique* indiquent des streams que nous donnons arbitrairement afin de construire l'exemple ; en caractère droit sont figurées les expressions qui doivent être calculées. Un espace laissé blanc à un instant donné dans la description d'un stream signifie soit que le stream n'est pas défini, soit que cet instant n'est pas un top. Par exemple, à l'instant 0, la somme **`S + T`** n'est pas définie : la valeur de **`T`** est définie, mais pas celle de **`S`**, il n'est donc pas possible de calculer la valeur de la somme. Par contre, à l'instant 3, la somme est définie, mais il ne s'agit pas d'un top : ni **`S`** ni **`T`** ne changent de valeur à cet instant.

II.3.6. Expression implicite versus expression explicite des instants du temps

Les opérations sur les streams introduites en 81/2, manipulent les streams globalement sans faire intervenir explicitement le rang d'un élément dans une suite, comme c'est le cas dans la notation :

$$U_{n+1} = f(U_n)$$

Dans la formule précédente, le rang `n` d'un élément apparaît explicitement. Si on attache une interprétation temporelle à une telle suite, le rang d'un élément correspond à un instant du temps. L'expression des instants du temps doit-elle être explicite ou implicite ? Laisser implicite les tics et les tops dans la manipulation des streams 81/2, présente de grands avantages :

- le stream est décrit globalement, sans faire référence à un instant. On décrit donc le stream « en soi » et non pas via une de ses instantiations à un instant `t` même si cet instant `t` est un instant générique.
- Puisque le seul opérateur de décalage dans le temps qui est permis est le retard, il n'est pas possible de faire référence à un élément futur dans le stream. Si au contraire il est possible d'accéder explicitement au rang d'un élément, rien n'empêche alors de construire des expressions non cohérentes avec une interprétation temporelle des streams.
- La dépendance du présent par rapport au passé est prise en charge par un opérateur banalisé `$` qui a le même statut par exemple qu'un opérateur arithmétique. Cette "internalisation" du passage du temps dans les opérateur 81/2 permet une plus grande puissance d'expression. Ainsi, c'est le compilateur qui synthétise les tics nécessaires à la description d'un système, à partir des tics des sous-systèmes (i.e. des instants pertinents dans l'observation de ces sous-systèmes). Par ailleurs, il est possible de retarder toute une expression complexe `S` en considérant l'expression `$S`, alors que si on avait proposé une notation faisant explicitement intervenir un instant courant `t`, il aurait fallu manipuler la définition de `S` (par exemple, en substituant toutes les occurrences de l'indice `t` par `(t-1)` dans la définition de `S`).

Contrairement à 81/2, Le langage ALPHA [MAU 89] permet de définir des équations récurrentes où des ensembles d'indices apparaissent explicitement. Des opérateurs permettent de manipuler ces ensembles d'indices comme des touts. Cela permet de considérer un programme comme la description d'un ensemble de points (ou indices), une valeur étant attachée à chaque point. Ce langage, utilisé pour le développement d'algorithmes systoliques, ne différencie pas les indices qui représentent l'écoulement du temps des indices qui représentent des points de l'espace. Le programmeur ne raisonne donc pas en terme d'écoulement temporel, mais uniquement en terme d'espace d'indexation abstrait. Les indices qui, à l'implémentation, correspondront à une exécution séquentielle (et donc au passage du temps) sont calculés arbitrairement à partir des espaces d'indexation, avec pour but, par exemple, de minimiser le temps d'exécution. Comme le langage ALPHA est destiné au développement d'algorithmes qui seront implémentés matériellement (par exemple sur un circuit VLSI), les ensembles d'indices que l'on considère sont restreints aux polyèdres convexes et les manipulations de ces ensembles sont restreintes aux transformations affines : avec ces restrictions, on dispose d'outils permettant l'implémentation complètement statique de ces systèmes mais on ne peut pas, par exemple, échantillonner un stream (ce qui reviendrait à considérer des "trous" dans les polyèdres).

II.4. Définir des streams par des équations

II.4.1. Système d'équations 81/2

Les opérateurs que nous venons de définir nous permettent de construire des expressions de streams qui sont des fonctions séquentielles et à mémoire bornée d'autres streams. Il nous reste maintenant à nommer ces expressions et à les combiner de manière déclarative. Une *équation* de la forme

```
identificateur = expression-de-stream
```

définit un stream dont le nom est ***identificateur*** et dont l'horloge et la valeur sont celles de ***expression-de-stream***. Un programme 81/2 est constitué par un ensemble d'équations, par exemple :

```
(2)  { A = 5
     { B = 6 when Clock
     { C = A + $B
```

Ces trois équations constituent un *système d'équations* qui définit trois streams **A**, **B** et **C**. Ce système d'équations est aussi un programme déclaratif 81/2 qui spécifie des relations entre les streams d'entrée et les streams de sortie.

Nous appelons *streams d'entrée* les streams qui n'ont pas de définition et qui correspondent à des données du programme ; dans l'exemple (2) il n'y en a aucun. Nous appelons *streams de sortie* tous les streams qui sont définis par le programme ; ici **A**, **B** et **C**. Si on le désire, les valeurs successives des streams de sorties peuvent être imprimées au cours de la simulation.

L'exécution du programme (2) correspond à résoudre numériquement le système d'équations, c'est-à-dire à énumérer les valeurs des streams **A**, **B** et **C** pour tous leurs tops successivement et dans l'ordre croissant.

Le programme (2) est un programme 81/2 valide, qui peut-être compilé et évalué. Son évaluation imprimera :

```
A ⇒ < 5 <
B ⇒ < 6;  6;  6;  6 … <
C ⇒ < ⊥; 11; 11; 11 … <
```

En fait, les streams constituent un cas particulier d'une structure plus générale, les *tissus* qui combinent dans un cadre déclaratif, le concept de stream et celui de collection. Pour simplifier, dans la suite de ce chapitre, nous considérons seulement des streams scalaires, i.e. des streams dont la valeur est de type entier, flottant ou booléen.

Remarquons bien que 81/2 ne permet pas d'écrire n'importe quelle équation. Par exemple, le système suivant :

```
{ 1 = T + U*U
{ 2*T = 3*U
```

n'est pas un programme 81/2 valide car les équations ne sont pas données sous la forme ***identificateur = expression*** (**1** et **2*T** ne sont pas des identificateurs de variable).

II.4.2. La définition par cas des streams

II.4.2.a. Les équations quantifiées

Il y a des cas où une seule équation ne suffit pas à définir un stream. Considérons par exemple le stream `C` du programme (2) : ce stream n'est pas défini au tic 0 car `$B` n'est pas défini au tic 0. Or on aimerait bien pouvoir spécifier une valeur initiale pour `C`. Pour cela, on utilise deux équations : une équation *valable pour le premier top* de `C` et une équation *valable pour tous les autres* tops de `C` :

```
{ A = 5
  B = 6 when Clock
  C@0 = -27
  C = A + $B
```

Le premier top est numéroté 0 et l'équation définissant `C` pour son premier top est « `C@0 = -27` ». Le symbole `@0` se lit « au top 0 ». Attention à ne pas confondre le numéro d'un top avec son rang en tant que tic. Le premier top d'un stream peut très bien survenir au tic 3673.

On dit que l'équation « `C@0 = -27` » est *quantifiée* pour le top 0 par analogie avec la *définition par cas* d'une fonction f en mathématique :

$$\begin{cases} f(0) = 1 \\ f(n) = n\ f(n-1), \forall n>0 \end{cases}$$

La quantification permet de préciser le domaine de validité d'une équation.

Attention, *la quantification ne porte pas sur les instants, mais sur les tops* : une quantification correspond à un *aiguillage*, un choix entre la valeur de plusieurs streams (pour le stream `C` le choix se fait entre le stream « `-27` » et le stream « `A + $B` »), la commande de cet aiguillage étant le numéro du prochain top attendu pour produire le stream résultat (Cf. figure II.6).

II.4.2.b. Pourquoi utiliser des équations quantifiées plutôt qu'un opérateur

Le lecteur qui se souvient de l'opérateur `fby` (« followed by ») du langage LUCID, se demandera pourquoi nous introduisons une notion de quantification des équations, plutôt que d'utiliser un opérateur supplémentaire permettant de combiner deux streams. L'approche utilisée en LUCID est aussi celle suivi par LUSTRE et SIGNAL. Avec cette approche, l'exemple précédent devient : `C = (-27) fby (A + $B)`. Cette écriture est très parlante, mais il devient difficile de faire des quantifications multiples comme par exemple :

```
{ C@0 = -27
  C@30 = 33
  C = A + $B
```

En effet, en LUCID, il faudrait écrire une expression qui comporterait 30 fois `fby`.

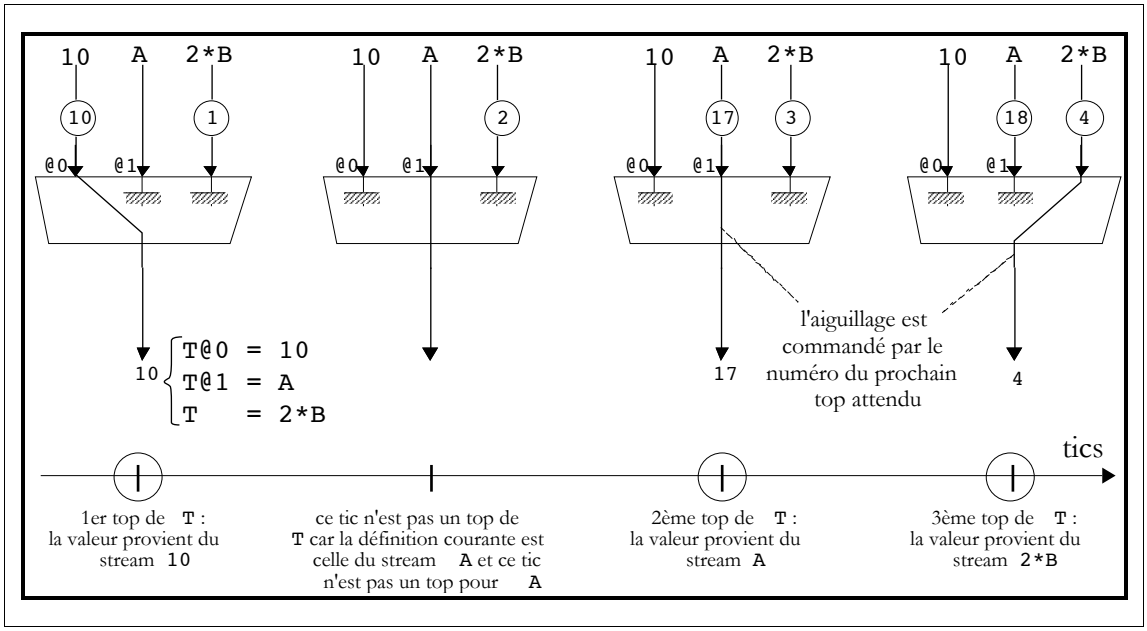


Figure II.6 : *La définition quantifiée des streams représentée comme un aiguillage entre différents streams. Les valeurs d'un stream en un top sont représentées comme des jetons circulant sur des arcs. Une définition quantifiée permet de choisir de quelle équation (i.e. de quel arc) proviendra lavaleur au prochain top du stream. Ce choix dépend du numéro du prochain top attendu. On ne passe d'une équation de définition à une autre, que quand un top s'est produit pour le stream défini par l'équation courante. Par exemple, au deuxième tic, l'équation courante est celle qui définit le stream* **T** *comme étant égal au stream* **A**. *Ce tic n'est pas un top de* **A** *et par suite, cette équation reste l'équation de définition courante de* **T** *jusqu'à ce qu'un top proviennent de* **A** *et déclenche le changement de définition.*

Définition par des équations gardées

Cependant, l'inconvénient présenté par `fby` en LUCID n'est pas fondamental. Il existe une raison plus profonde qui nous amène à préférer la définition par cas introduite par les équations quantifiées, à un opérateur `fby` sur les streams.

La raison est que la quantification des équations correspond à une *définition gardée* et n'importe quel prédicat[1] pourrait être utilisé à la place du nombre qui suit le symbole `@`. Quand un nombre `k` suit le symbole `@`, ce nombre correspond à un *prédicat temporel* qui est vrai quand le prochain top attendu a pour numéro `k` et qui est a la valeur faux sinon. Le système suivant utilise les équations gardées de manière plus complexe :

(3)
```
X @ 0       = 10
X @ ($X>5)  = $X - 1 when Clock
X @ ($X≤5)  = $X + 1 when Clock
```

`X ⇒ < 10; 9; 8; 7; 6; 5; 6; 5; 6; 5 … <.`

Le système (3) définit un stream `X` dont le comportement est le suivant : sa valeur initiale est 10, chaque fois que sa valeur est supérieure à 5 sa valeur est décrémentée de 1 (au rythme de `Clock`) et chaque fois que sa valeur est inférieure ou égale à 5, sa valeur est incrémentée de 1 (au rythme de `Clock`).

[1] Le langage des prédicats autorisés et la stratégie de gestion des conflits (si deux prédicats valident deux équations simultanément) sont en cours de recherche dans 81/2. Le programmeur doit actuellement se restreindre à un seul prédicat possible : « `@n` » qui a la valeur vrai au `n`ème top

On ne peut pas traduire une définition gardée par un ensemble de définitions 81/2 non gardées. Par exemple, la définition :

```
{ X@g = a
{ X   = b
```

où `g` est un prédicat quelconque, qui se traduit par un stream booléen `g'`, ne se traduit pas par :

```
X = if g' then a else b
```

car les horloges des deux expressions sont différentes : l'expression conditionnelle a plus de tops que l'expression gardée (Cf. II.3.2). Ainsi, il faut au moins rajouter un opérateur du type `fby` afin de pouvoir réécrire l'expression correspondant à un prédicat `@n` en se passant des gardes.

II.4.3. Programmer avec des équations

Quand on programme avec des équations, l'exécution du programme correspond au calcul de la solution du système d'équations. La question qui se pose donc à présent est de savoir si un système d'équations entre streams admet une solution et si cette solution est unique :

- Si un système d'équations entre streams 81/2 n'admet pas de solution, alors cela veut dire que le programme 81/2 n'est pas valide : on ne peut rien calculer car il n'y a rien à calculer.
- Si un système d'équations entre streams 81/2 admet plusieurs solutions, alors il faudra choisir entre calculer une solution particulière (mais il faudra que nous la caractérisions d'une manière ou d'une autre dans l'ensemble des solutions), et calculer toutes les solutions (ce qui est fait par exemple en PROLOG).

La réponse à la première question est : « oui, il existe toujours une solution à un système d'équations de streams 81/2 ». Ainsi un programme 81/2 calcule toujours quelque chose de correctement déterminé. La réponse à la deuxième question est : « si le système est récursif, la solution n'est pas unique ». Dans ce cas, on peut *comparer* les solutions entre elles, à l'aide d'une relation d'ordre, et s'arranger pour qu'un programme calcule toujours *la plus petite* de ces solutions.

En conclusion, l'évaluation d'un programme 81/2 calcule la plus petite solution du système d'équations. Nous allons voir plus précisément ce que cela signifie.

II.4.3.a. Système récursif et système non-récursif

Un système d'équations est *récursif* si une variable apparaît dans sa propre définition, soit directement (c'est le cas de la variable `x` dans le système (4) ci-dessous) soit indirectement (cas de la variable `u` dans le système (5)) :

(4)
```
{ x = $x + 1
```

(5)
```
{ u = v
{ v = w
{ w = $v
```

Dans les deux cas, la récursion se traduit par un *cycle* dans le *graphe des dépendances entre les variables* : une variable `x` dépend d'une variable `y` si `y` apparaît dans la définition de `x` (Cf. figure II.7).

Si un système n'est pas récursif, alors il n'y a pas de cycle dans le graphe des dépendances entre les variables. On dit que le graphe des dépendances est *acyclique*. On peut alors calculer très simplement les streams du programme : on commence par calculer les streams qui ne dépendent d'aucun autre stream, puis on propage les calculs par un parcours en largeur dans le graphe des dépendances (Cf. figure II.8. La situation est analogue à celle d'un système d'équations linéaires *triangulaires* : dans un système linéaire triangulaire, on

peut ordonner les variables de telles sorte que le calcul de la valeur de la variable **x** ne dépende que des valeurs des variables **y** qui précèdent **x**.

Par conséquent, si le système d'équations n'est pas récursif, l'algorithme précédent nous montre qu'il existe une solution unique au système d'équations.

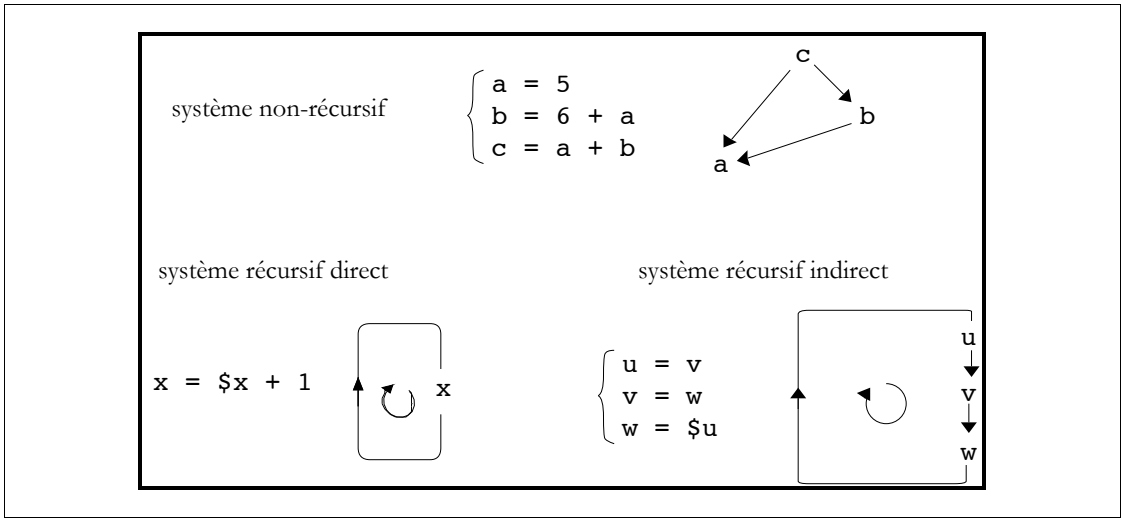


Figure II.7 : *Systèmes récursifs et non-récursifs visualisés grâce à leur graphe de dépendances entre les variables.*

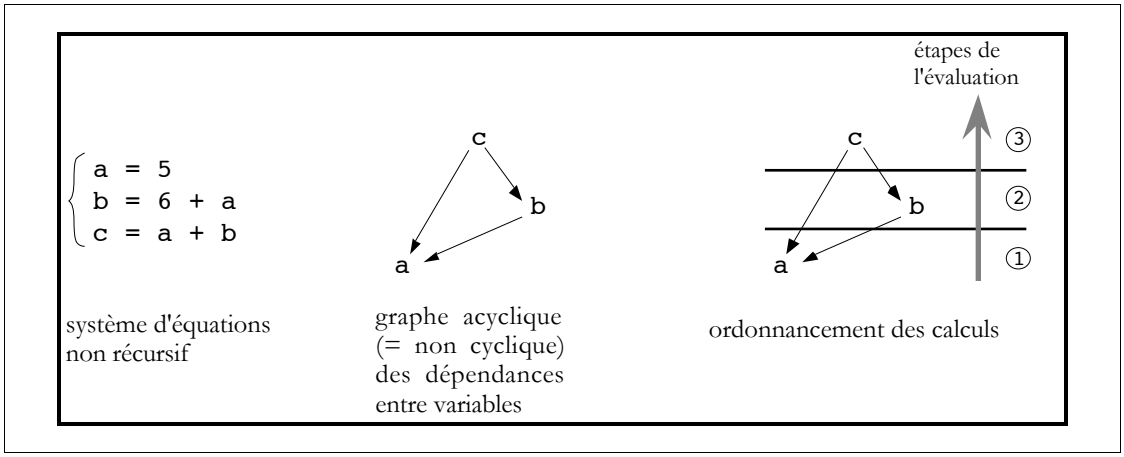


Figure II.8 : *Ordonnancement des calculs dans le cas d'un système non-récursif.*

II.4.3.b. La plus petite solution d'un système récursif

Cette situation n'est plus vraie pour un système récursif. Pour illustrer l'existence et la multiplicité des solutions d'un système d'équations récursives, examinons le programme récursif suivant :

(6) { `P = P + 1`

Rappelons que l'on cherche la solution de ces équations dans l'ensemble des suites temporisées. Si le système (6) n'admet pas de solution dans `I R`, il admet une solution dans l'ensemble des suites : c'est la suite vide **<<**. En effet, nous avons défini la suite vide **<<** par :

```
v<< = < ⊥; ⊥; ⊥;  … <
h<< = < F; F; F;  … <
```

On vérifie bien avec les règles de calcul 81/2, que

```
<< + 1 = <<
```

pour s'en convaincre, il suffit de calculer le domaine de définition de `<< + 1` et de s'apercevoir qu'il est vide. Par suite, `P = <<` constitue bien une solution de (6). On peut montrer, en raisonnant par l'absurde, que c'est la seule solution.

Cet exemple n'est pas une démonstration mais illustre le fait qu'on peut toujours trouver au moins une solution à un système d'équations entre streams 81/2. Regardons à présent un exemple dans lequel il y a plusieurs solutions possibles :

(7)
```
{ R@0 = 3
{ R = $R + 1
```

Intuitivement, ce programme correspond à un "compteur" qui compte à partir de `3`. Ce système admet pourtant une infinité de solutions : tout stream qui a chaque top augmente de `1`. Le fait qu'il y ait une infinité de solutions possibles, provient de ce que l'on n'a pas spécifié *quand* le stream progresse : il peut progresser tous les tics pairs, ou tous les tics correspondant à un nombre premier, etc. Par exemple, les streams suivants (un espace laissé blanc représente un tic) sont solutions de (7) :

```
R1 = <3; 4; 5; 6; 7; 8; 9; 10; 11; 12; … <
R2 = <3;    4;    5;    6;  … <
R3 = <3<
R4 = <3;                4; 5<
```
etc.

Quand un système d'équations admet plusieurs solutions, nous voulons n'en retenir qu'une *et* nous voulons que les programmes 81/2 soient déterministes : ils doivent calculer toujours la même solution, indépendamment du compilateur 81/2, des conditions de l'exécution, etc. Le choix de la solution calculée ne doit donc pas dépendre d'un choix d'implémentation : il doit provenir d'un critère interne. Ce critère interne, on le matérialise par une relation d'ordre, qui classe les streams solutions entre eux, et on choisit la plus petite des solutions. Pour cela, il faut que la relation d'ordre choisie permette de comparer les différentes solutions entre elles. C'est le cas si cette relation d'ordre est un ordre total. Mais cette relation d'ordre peut être un ordre partiel si on montre que les différentes solutions d'un système d'équations sont toujours comparables. C'est le cas avec l'ordre choisi sur les streams.

La relation d'ordre que nous avons choisie pour comparer les streams est la relation *préfixe* : $\mathtt{s}_1 \leq \mathtt{s}_2$ si $\mathtt{s}_1$ est un préfixe de $\mathtt{s}_2$. Le stream $\mathtt{s}_1$ est un préfixe de $\mathtt{s}_2$ s'il existe un indice `n` tel que les `n` premières valeurs de h$\mathtt{s}_1$ et de h$\mathtt{s}_2$ coïncident, de même que les `n` premières valeurs de v$\mathtt{s}_1$ et v$\mathtt{s}_2$, et si l'horloge de $\mathtt{s}_1$ a toujours pour valeur le booléen `faux` pour un élément de rang supérieur ou égal à `n`. Par exemple

```
true ≤ Clock
```

car `true` et `Clock` coïncident sur l'instant 0, puis `true` n'a plus de tops (Cf. §II.3.1).

Pour le système (7), le plus petit stream solution `R`i sera le stream `R3`. Tous les autres streams solutions commenceront par la valeur 3 à cause de l'équation quantifiée. Le stream `R3` n'ayant plus d'autres tops, validera de facto l'équation non quantifiée (puisque l'ensemble des tops qui doivent la vérifier est vide). Le stream `R3` est donc une solution de (7) et il sera préfixe de toutes les autres solutions. C'est donc ce stream qui sera calculé par 81/2.

En conclusion, le programme (7) ne définit pas un compteur mais une constante. Que le lecteur se rassure, il est possible de définir des compteurs en 81/2 ! (voir l'exemple (8) plus bas).

II.4.3.c. Vérifier que des streams sont solutions d'un système d'équations

L'objet du chapitre V est de développer les techniques nécessaires pour résoudre un système d'équations récursives entre streams. Pour le moment, nous nous contenterons de vérifier qu'un stream donné est solution ou non d'un système d'équations entre streams. Cette vérification s'appuie sur le fait qu'on peut associer une fonction à un système d'équations. En effet, une équation de la forme :

```
x = expression
```

peut se lire comme

```
x = f(x, y, z, …)
```

où `f` est une fonction et `x`, `y`, `z` … sont les variables qui apparaissent dans `expression`. Plus généralement, les programmes 81/2 prennent tous la forme suivante :

$$\begin{cases} x_1 = f_1(x_1, x_2, \ldots, x_n) \\ x_2 = f_2(x_1, x_2, \ldots, x_n) \\ \ldots \\ x_n = f_n(x_1, x_2, \ldots, x_n) \end{cases}$$

ce qui peut s'abréger par

```
X = F(X)
```

avec $F = (f_1, f_2, \ldots, f_n)$ un vecteur de fonctions, et $X = (x_1, x_2, \ldots, x_n)$ un vecteur de streams. Par exemple le programme

```
A = 5
B = 6 when Clock
C@0 = -27
C = A + $B
```

se traduit par `X = F(X)` ou `X = F'(X)` suivant que l'on calcule le premier top de `C` ou les autres tops, avec `X`, `F` et `F'` qui se définissent comme suit

$$X = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \qquad F\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \text{ when Clock} \\ -27 \end{bmatrix} \qquad F'\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \text{ when Clock} \\ x + \$y \end{bmatrix}$$

La solution d'un système d'équations `X = F(X)` est le vecteur de streams `S` qui vérifie `S = F(S)`. Une telle valeur `S` est un *point fixe* de la fonction `F`. Pour vérifier qu'un vecteur de streams donné `S'` est solution du système d'équations, il suffit donc de calculer `F(S')` et de vérifier s'il est égal à `S'`.

Comme exemple, vérifions que pour le programme suivant :

(8)
```
X@0 = 3
X = $X + 1 when Clock
```

le stream `T` défini par

```
T  = <3; 4; 5; …<
hT = <V; V; V; … <
vT = <3; 4; 5; …<
```

est solution du programme alors que le stream `U`

```
U  = <3; 4; 5; …<
hU = <V; F; V; F; V; F; V; V; F; V; … <
vU = <3; 3; 4; 4; 5; 5; 6; 7; 7; 8;  …<
```

n'est pas solution du programme (8). Le tableau ci-dessous calcule pas à pas **`F(T)`** et **`F(U)`**, où **`F`** est extraite du programme (8). Dans ce tableau, une expression en gras comme par exemple **`$T + 1`** doit être comprise comme le résultat de l'application de la fonction **`λx.$x+1`** au stream **`T`**.

tics	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	…
T	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	…
vT	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	…
hT	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	*V*	…
U	*3*		*4*		*5*		*6*	*7*		*8*	…
vU	*3*	*3*	*4*	*4*	*5*	*5*	*6*	*7*	*7*	*8*	…
hU	*V*	*F*	*V*	*F*	*V*	*F*	*V*	*F*	*V*	*F*	…
Clock	V	V	V	V	V	V	V	V	V	V	…
$T		3	4	5	6	7	8	9	10	11	…
$U			3		4		5	6		7	…
$T + 1		4	5	6	7	8	9	10	11	12	…
$U + 1			4		5		6	7		8	…
$T+1 when Clock		4	5	6	7	8	9	10	11	12	…
$U+1 when Clock			4	4	5	5	6	7	7	8	…
(T@0 = 3 $T+1 when Clock)	3	4	5	6	7	8	9	10	11	12	…
(U@0 = 3 $U+1 when Clock)	3		4	4	5	5	6	7	7	8	…

Nous n'avons pas fait explicitement figurer les suites de valeurs et les horloges de chaque sous-expression, mais nous avons utilisé la convention typographique de mettre des espaces sur un tic qui n'est pas un top. On remarque, en comparant la ligne ***T*** et l'avant-dernière ligne du tableau que :

`T = F(T)` avec `F` fonction définie par `F(X) = (X@0 = 3; X = $X+1 when Clock)`

et en comparant la ligne *V* et la dernière ligne du tableau, on peut voir que

`U ≠ F(U)`

On vérifie donc que `T` est solution ; en fait, le système (8) admet une solution unique. Remarquons la différence entre le système (7) et le système (8) : dans la définition (8) de `T`, la progression du stream solution est imposée par l'horloge de `Clock` grâce à l'opération d'échantillonnage.

II.5. Exemples de programmation avec les streams 8½

II.5.1. Différences finies et sommes d'une suite

Soit un stream `U` donné. On peut calculer le stream `dU` des différences entre deux de ses éléments consécutifs :

```
dU = U - $U
```

Le stream des différences secondes, s'écrirait :

```
ddu = dU - $dU
```

On peut calculer aussi la somme `iU` courante de ses valeurs successives

```
iU = U + $iU
```

Mais cette définition ne calcule pas ce que l'on veut : aucune valeur initiale n'étant assignée à `iU`, l'expression `$iU` n'est jamais définie. Autrement dit, la solution de l'équation précédente est le stream vide. L'expression correcte de la somme courante du stream `U` est :

```
{ iU@0 = U
{ iU   = U + $iU
```

avec cette définition, le premier top de `iU` a lieu en même temps que le premier top de `U` et la valeur de `iU` est à cet instant celle de `U`. Ensuite, chaque fois que `U` change de valeur, i.e. a un top, `iU` change aussi de valeur. A un top donné différent du premier top, la valeur de `iU` est celle de `iU` au top précédent plus la valeur de `U`.

Le tableau suivant illustre ces définitions pour un stream `U` donné (on figure les tics par des blancs) :

```
U   = < 1;  2;  3;  4;  5;  6;  7; … <
dU  = <     1;  1;  1;  1;  1;  1; … <
ddU = <         0;  0;  0;  0;  0; … <
iU  = < 1;  3;  6; 10; 15; 21; 28; … <
```

II.5.2. Équations récurrentes

Les exemples précédents sont des exemples simples d'équations récurrentes en mathématiques. Plus précisément, nous nous intéressons aux suites mathématiques définies par des équations du type :

$$\begin{cases} X_0 = x_0 & \text{une condition initiale} \\ X_{n+1} = F(X_n) & \text{l'état courant dépend de l'état passé par une fonction d'évolution } F \end{cases}$$

Ce type de définition se traduit immédiatement en 8½ par un système d'équations qui peut, par exemple, prendre la forme suivante :

$$\begin{cases} X@0 = x_0 \\ X = F(\$X) \text{ when Clock} \end{cases}$$

On voit que le programme 8½ a une forme tout a fait comparable, exceptée la présence de l'échantillonnage « `when Clock` » qui est là pour fixer le rythme d'évolution de la suite : les tops de `Clock` correspondent à l'énumération des indices de la suite X_n. Nous avons déjà vu dans l'exemple (7) que l'absence de « `when`

`Clock` » résultait en la définition d'un stream ne possédant qu'un seul top (provenant de la définition quantifiée, i.e. ici « `X@0 =` x_0 »).

II.5.2.a. Multiples conditions initiales

Les équations récurrentes possèdent souvent des conditions initiales qui portent sur les `k` premières valeurs de la suite :

$$\begin{cases} X_0 = x_0 \\ X_1 = x_1 \\ \dots \\ X_k = x_k \\ X_{n+k+1} = F(X_{n+k}, X_{n+k-1}, \dots, X_n) \end{cases}$$

Ce type de système se traduit en 81/2 par :

$$\begin{cases} X@0 = x_0 \\ X@1 = x_1 \text{ when Clock} \\ \dots \\ X@k = x_k \text{ when Clock} \\ X = F(\$X, \$\$X, \dots, \$\dots_{\text{k fois}}\dots\$X) \text{ when Clock} \end{cases}$$

La présence de « `when Clock` » dans les équations quantifiées `@1`, … `@k`, peut surprendre. Afin d'éclaircir le pourquoi de cette définition, examinons la définition suivante :

$$\begin{cases} X@0 = x_0 \\ X@1 = x_1 \\ \dots \\ X@k = x_k \\ X = F(\$X, \$\$X, \dots, \$\dots_{\text{k fois}}\dots\$X) \text{ when Clock} \end{cases}$$

qui diffère de la précédente par l'absence des « `when Clock` » dans les définitions quantifiées. Regardons comment se comporte ce dernier système au cours du temps :

- **au tic 0**

  Les expressions x_0, …, x_k sont supposées être des streams constants correspondant à des constantes scalaires. Pour de tels streams, le tic 0 correspond à un top. Les expressions `$X`, …, `$…$X` ont une valeur indéfinie. Par suite, l'expression `F($X, …, $…$X)` a une valeur indéfinie, ainsi que `F($X, …, $…$X) when Clock`.

  La définition quantifiée de X sélectione le stream de l'équation quantifiée `@0` pour calculer le premier top de `X`. Il se trouve que le stream x_0 qui correspond à cette équation a justement un top au tic `0`. La valeur de x_0 devient donc aussi la valeur de `X`.

- **au tic 1**

  Le tic 1 ne correspond pas à un top pour les streams constants x_0, …, x_k. En fait, ces streams n'auront plus jamais de top.

  La valeur de `$X` n'est définie qu'à partir du deuxième top de `X` (Cf. §II.3.3). Le tic 1 est donc un top de `$X` que si ce tic est aussi un top de `X`. Dans ce cas, la valeur de `$X` sera la valeur de `X` au top précédent, i.e. au tic 0. Si le tic 1 n'est pas un top pour `X`, la valeur de `$X` n'est pas définie.

  La définition quantifiée de `X` agit comme un aiguillage et la valeur du stream `X` doit provenir à présent de l'équations quantifiée `@1`, puisqu'on veut élaborer la valeur du deuxième top de `X`. Par conséquent, le tic 1 est un top pour `X` si c'est un top pour x_1 (auquel cas la valeur de x_1 devient la valeur de `X`). Mais, le tic 1 n'est pas un top de x_1.

  En conclusion, le tic 1 est un tic pour `X`, la valeur de `$X` est indéfinie et on peut passer au calcul du tic suivant.

- **au tic 2**

  En fait, la même situation se reproduit : l'équation qui définit `X` est l'équation quantifiée `@1` et cela sera vrai, jusqu'à ce qu'un top provienne de cette équation, auquel cas, l'équation de définition de `X` deviendra l'équation quantifiée `@2`.

  Mais, le stream x_1 n'aura plus jamais de top. Aucun des tics à venir n'amènera de top qui débloquera la définition de `X`. Aucun des tics à venir n'est donc un top pour `X`.

La définition précédente ne réalise donc pas le calcul d'une suite récurente : il faut forcer le passage de `X` d'une définition à une autre. L'utilisation de `when Clock` permet d'imposer des tops aux constantes x_i et donc de faire progresser le système. En effet, une expression comme « `c when Clock` » a pour horloge et valeur :

```
h(c when Clock) = <V; V; V; V; V; V; … <
v(c when Clock) = <c; c; c; c; c; c; … <
```

Quand donc on attend, à partir du tic `1`, le prochain top de `X` sur la définition quantifiée `@1`, cette définition produit un top (au même instant) et fournit la valeur x_1 comme nouvelle valeur de `X` (au tic 1). Et ainsi de suite : le top suivant de `X` se produit au tic 2 et a pour valeur x_3, etc.

Pour obtenir une écriture "plus symétrique" du système, on aurait pu écrire « $X@0 = x_0$ `when Clock` », ce qui n'aurait rien changé au résultat : les tops supplémentaires de « x_0 `when Clock` » par rapport à « x_0 » correspondent à tous les tics strictement supérieurs à 0 et ceux-ci ne sont pas utilisés par la définition de `X` : seul le premier top de l'expression quantifiée `@0` est utilisé.

II.5.2.b. Systèmes d'équations récurrentes

Les streams peuvent se définir récursivement les uns en fonction des autres. Par exemple :

$$\begin{cases} U_0 = 0 \\ U_{n+1} = U_n + 1 \\ V_0 = 1 \\ V_{n+1} = V_n * U_{n+1} \end{cases}$$

Ce système définit les suites $U_n = n$ et $V_n = n!$ et il se traduit en 81/2 en utilisant le schéma précédent :

```
U@0 = 1
U   = $U + 1 when Clock
V@0 = 1
V   = $V * U when Clock
```

En fait ici, un seul « `when Clock` » serait suffisant, par exemple sur la définition de `U`. En effet, l'horloge de « `$V * U` » est, sur le domaine de définition correct, l'union des horloges de `$V` et de `U`. Cette horloge contient donc au moins les tops de `U`, qui sont ceux de `Clock`. L'adjonction de « `when Clock` » sur la définition de `V` est donc inutile, mais garde une forme symétrique aux définitions de `U` et `V`.

II.5.2.c. Systèmes d'équations récurrentes partielles

Il arrive parfois que l'on définisse des suites dont l'ensemble des indices n'est pas `I N`, mais un sous-ensemble de `I N`. Par exemple, dans le cas des méthodes numériques de relaxation dites « rouge et noire », on est amené à calculer une suite U_{2n} (l'itération « rouge ») et une suite V_{2n+1} (l'itération « noire »). On aboutit ainsi à une définition qui prend par exemple la forme suivante :

$$\begin{cases} U_{2n+2} = F(V_{2n+1}) \\ V_{2n+1} = G(U_{2n}) \end{cases}$$

En 81/2, on peut générer une horloge pour les tics pairs et une horloge pour les tics impairs de `Clock`, par exemple à partir d'un compteur, et utiliser ensuite ces horloges pour définir `U` et `V` de la manière suivante :

```
cpt@0 = 0
cpt   = $cpt + 1 when Clock

Hpair   = 0 == (cpt % 2)
Himpair = not Hpair

U@0 = u0 when Hpair
U   = F(V) when Hpair

V@0 = v1 when Himpair
V   = G(U) when Himpair
```

Le compteur `cpt` compte les tops de `Clock` ; le stream `Hpair` est un stream de type booléen qui a pour valeur `vrai` quand `cpt` est un multiple de 2 (l'opérateur `%` est l'opérateur modulo et `==` est l'opérateur d'égalité, comme dans le langage `C`). Contrairement à `Clock` ou à `true`, la valeur de `Hpair` n'est pas constante : elle vaut `faux` quand `cpt` prend une valeur impaire. Les tops de `Hpair` sont ceux de `cpt` qui sont ceux de `Clock`. Les tops d'une expression « `X when Hpair` » sont, sur l'intersection des domaines de définition de `X` et `Hpair`, les tops de `Hpair` sur lesquels `Hpair` prend la valeur `vrai`. Ces tops sont donc ceux pour lesquels `cpt` prend une valeur paire. On peut faire le même raisonnement pour `Himpair`.

On remarque qu'on n'utilise pas de délai dans les expressions générales de `U` et de `V` : les tops de `U` et de `V` étant entrelacés, la valeur de `V` nécessaire au calcul de `U` est celle disponible à l'instant courant et réciproquement.

A titre d'illustration, si on prend `F(x) = G(x) = x+1`, et $u_0 = 0$, $v_1 = 1$, on obtient les streams `U` et `V` suivants :

```
hU  =  <V;  F;  V;  F;  V;  F;  V;  F;  …  <
vU  =  <0;  0;  2;  2;  4;  4;  6;  6;  …  <

hV  =  <F;  V;  F;  V;  F;  V;  F;  V;  …  <
vV  =  <⊥;  1;  1;  3;  3;  5;  5;  7;  …  <
```

II.5.2.d. Compter les tops d'un stream

La suite des entiers naturels, énumérés au rythme de `Clock`, est définie en 81/2 par exemple par :

```
n@0 = 1 when Clock
n   = 1 + $n when Clock
```

Si on veut compter les tops d'un stream `X`, il suffit de construire un stream booléen qui vaut `vrai` pour chaque top de `X` et de remplacer `Clock` par ce stream dans les équations précédentes. Un stream qui vaut `vrai` et a pour tops exactement les tops de `X`, est donné tout simplement par « `X == X` ». Par suite, un compteur de tops du stream `X` est donné par :

```
tops_de_X@0 = 1 when (X == X)
tops_de_X   = 1 + $tops_de_X when (X == X)
```

Comme l'expression « when `(X == X)` » est fréquemment utilisé, il existe en 81/2 un opérateur d'échantillonnage analogue à `when` mais qui n'est commandé que par l'horloge de son deuxième argument ; c'est `synchro` :

```
tops_de_X@0 = 1 synchro X
tops_de_X   = 1 + $tops_de_X synchro X
```

Remarquons que le premier top de **X** ne se produit pas nécessairement en 0, il est donc indispensable de conditionner la constante 1 par « synchro X » sinon on introduirait un top parasite au tic 0. L'expression « c synchro X » a pour tops exactement les tops de **X** et a pour valeur **c**. Par exemple (les tics qui sont des tops de **X** sont en **gras**) :

```
                hX =  < F; F; F;  V; F; V; V; F; F; V; … <
h(c synchro X) =  < F; F; F;  V; F; V; V; F; F; V; … <
v(c synchro X) =  < ⊥; ⊥; ⊥;  c; c; c; c; c; c; c; … <
```

II.5.2.e. Deux suites récurrentes bien connues

La suite de Fibonacci peut être construite au fil des tops de Clock par :

```
⎧ fib@0 = 1 when Clock
⎨ fib@1 = 1 when Clock
⎩ fib   = $fib + $$fib when Clock
```

La suite des factorielles, se programmera par exemple par :

```
⎧ n@0 = 0
⎪ n = $n + 1 when Clock
⎪
⎪ fact@0 = 1
⎩ fact   = n*$fact
```

dans cet exemple, le stream n impose ses tops au stream fact.

II.5.3. Modélisation d'une file d'attente

Une file d'attente est un processus qui sert de *tampon* entre des *clients* et un *serveur*. Nous voulons observer la longueur d'une file d'attente.

Nous supposons que la probabilité λ de recevoir une demande de service est constante. Le nombre λ correspond à la probabilité de recevoir une demande de service entre les instants t et t+1. Nous supposerons aussi que les demandes de service sont séquentielles : entre deux instants consécutifs, il y a au plus une seule demande de service. Le serveur est défini par une probabilité μ de fin de service : cette probabilité est constante et correspond à la probabilité que le service en cours de traitement s'achève entre les instants t et t+1.

La file d'attente, décrite par un nombre représentant le nombre de clients en attente, peut évoluer de quatre manières différentes :

- s'il y a une arrivée de client et une fin de service, la longueur de la file d'attente reste la même ;
- s'il y a une arrivée et pas de fin de service, la longueur de la file d'attente augmente de 1 ;
- s'il y a une fin de service et pas d'arrivée de client, le nombre de clients en attente diminue de 1, ou reste à 0 ;
- s'il n'y a ni arrivée, ni fin de service, le nombre de clients en attente reste constant.

Par ailleurs, la file d'attente est initialement vide, et il n'y a un service que s'il y avait précédemment un client dans la file d'attente.

Le système 81/2 correspondant à la simulation de la file d'attente peut donc s'écrire de la manière suivante :

```
arrivée = Clock -(1/λ)
service = $file ≥ 1
fin_de_service = service & Clock -(1/μ)
file@0 = 0
file = if arrivée & fin_de_service then $file
       else if arrivée then $file + 1
       else if fin_de_service then $file -1
       else $file
```

On peut remarquer que `file` a un top chaque fois que `arrivée` ou que `fin_de_service` a un top, ce qui est bien en accord avec le fait que la longueur de la file d'attente est modifiée soit par l'arrivée d'un client, soit par une fin de service.

III. Les collections 8½

III.1. Introduction

Avec la notion de stream nous avons modélisé l'aspect *temporel* d'un système dynamique. Mais les systèmes dynamiques présentent aussi un aspect *spatial* à travers la structure d'un état. Un état est une fonction d'un ensemble de points, les points de l'espace, dans un ensemble de valeurs. Les points de l'espace peuvent avoir un nom explicite : c'est le cas d'une variable d'état à valeur scalaire. Mais ils peuvent aussi avoir un nom implicite : si on s'intéresse à une image de télévision (Cf. l'exemple développé au §I.4.2.a) une variable d'état correspond alors à un ensemble de points (les pixels) qui eux n'ont pas de nom explicite, mais qui peuvent être désignés implicitement ; par exemple, le premier pixel en haut et à gauche de l'image. En accord avec cette analyse, on peut désigner dans le langage 81/2 les points d'un espace de deux manières différentes :

- par un identificateur alphabétique (un nom de variable),
- par un index entier pris dans l'intervalle `[0, n]`.

Dans ce chapitre, nous étudierons essentiellement la désignation par un index, le nommage par un identificateur alphabétique constituant une commodité qui peut toujours s'y ramener. Nous verrons dans le prochain chapitre comment donner un nom à un point. Soit `S` un ensemble de valeurs appelées *scalaires* : booléens, flottants, entiers, etc. Un ensemble de valeurs prises dans `S` et indexées par un entier pris dans l'intervalle `[0, n]` correspond à un vecteur de `n+1` élément de `S`. L'ensemble S^n des vecteurs de `n` éléments pris dans `S` est défini par le produit cartésien $S \times \ldots{}_{\text{n fois}} \ldots \times S$.

L'objet correspondant aux variations d'une valeur dans l'espace, un champ, est donc un élément de S^n. Une trajectoire dans S^n est un *tissu* : c'est un stream de vecteurs. Mais de manière duale, on peut voir un tissu comme un vecteur de streams. Le tissu est l'objet défini par une équation 81/2. Les équations que nous avons vues dans le chapitre précédent définissent des streams de scalaires, qui sont un cas particulier de tissus (en prenant `n = 1`). Nous pouvons aussi voir un champ c comme un cas particulier de tissu en considérant un stream ayant un seul top et dont la valeur en ce top, est le champ c. Ainsi, dans ce chapitre, "tout se passe en un seul top".

III.2. Manipuler des ensembles de valeurs "comme des touts"

Rappelons (Cf. §I.4.a), que nous voulons manipuler les champs *comme des touts*. Cette caractéristique est *nécessaire* pour plusieurs raisons :

- *Taille du modèle* : il n'est pas possible d'expliciter les relations entre chaque point de l'espace.
- *Invariance des lois d'évolution* : les lois d'évolution d'un système dynamique sont les mêmes en tout point de l'espace (et à tout instant du temps).
- *Approche uniforme et globale* : nous voulons traiter uniformément l'espace et le temps.
- *Efficacité* : le traitement déclaratif des ensembles doit se faire globalement afin d'être efficace.

Nous devons donc développer un langage permettant de manipuler des vecteurs comme des touts. Pour marquer qu'on les manipule globalement, nous utiliserons le terme de *collection*. Une collection présente les caractéristiques suivantes :

- une collection a un nombre borné d'éléments. Cette condition est plus forte que la condition d'un ensemble discret d'instants, car les éléments d'une collection devant être disponibles tous simultanément, il faut pouvoir les loger dans la mémoire bornée d'un ordinateur.
- Les points de l'espace sont indépendants du temps (Cf. §I.4.2). Par suite, le nombre d'éléments de la collection ne dépend pas de l'instant considéré ou de la trajectoire particulière suivie par le système.

III.2.1. Collections et représentations des champs associés à une étendue physique

Souvent, un champ correspond à la variation d'une valeur sur une étendue physique (par exemple une image photographique). Une collection correspond alors à une *discrétisation simple* de ce champ : la structure même de cette discrétisation est posée à priori par le programmeur et elle n'évolue pas au cours de la simulation.

Dans ce document, nous avons décidé de simplifier le traitement de l'espace et par conséquent nous n'envisageons pas la notion de voisinage. Cela se traduit pour les collections par le fait que l'indexation des éléments d'une collection est arbitraire et ne reflète pas la structure de l'espace sous-jacent à un champ. La figure III.1 indique à titre d'exemple plusieurs façons, utilisées dans la littérature, pour représenter les points d'une surface par une collection [MIL 89].

Bien que l'on puisse représenter tout champ discret par une collection, en numérotant de manière arbitraire les points du champ par un entier et en considérant le vecteur des valeurs indexées par ces entiers, il est bien commode de pouvoir désigner des portions de l'espace et non plus seulement des points. En fait, c'est indispensable si l'on veut désigner des régions de l'espace et des portions de champ comme des touts. Cela conduit à hiérarchiser les points de l'espace en considérant qu'une région de l'espace peut se voir comme un point dans une structure d'espace d'ordre supérieure ; nous avons déjà fait cela quand nous avons donné un nom φ à une image de cinéma (Cf. §I.4.2.a), l'image désignait alors l'ensemble de ses points `(x, y)`.

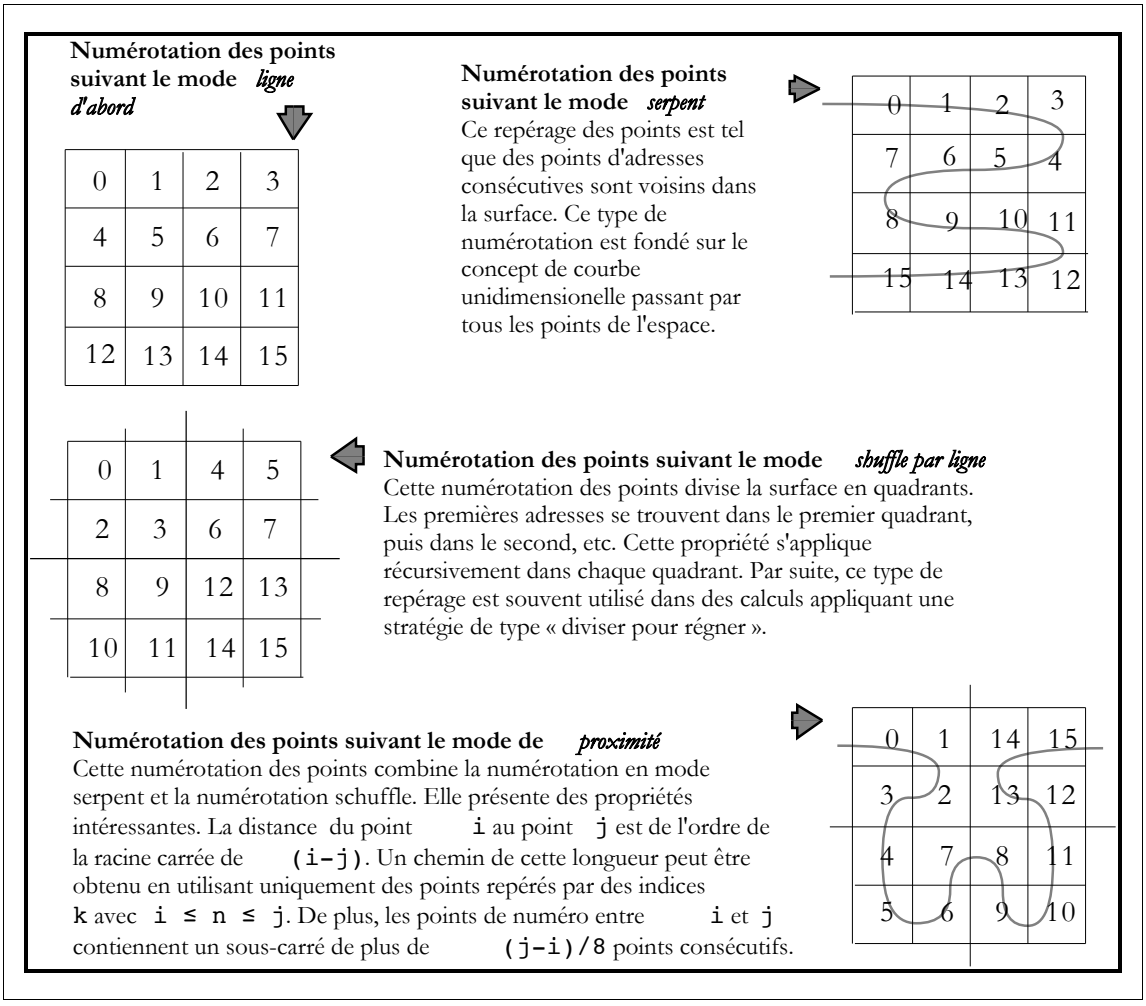


Figure III.1 : *Plusieurs discrétisations possibles d'un champ à deux dimensions par une collection. Le problème est d'associer un élément de la collection à un point de la surface discrétisée. On a représenté la surface par un grand rectangle et les points discrétisés de la surface par les petits rectangles. Une collection étant un vecteur, il suffit de savoir numéroter les points d'une surface par des entiers pris dans l'ensemble* `{0, …, n}` *pour obtenir la collection associée. On peut attribuer un numéro arbitraire à chaque point mais il y a des numérotations qui sont plus usitées que d'autres. Cette figure présente quatre numérotations courantes : les numéros de chaque petit rectangle correspondent à l'indice du point dans la collection associée.*

Cela conduit donc naturellement à *imbriquer les collections*. Une collection imbriquée peut se voir comme un *arbre* : les feuilles de l'arbre correspondent aux points atomiques de l'espace et les nœuds intérieurs correspondent à des régions de l'espace. Le nombre d'imbrications s'appelle la *dimension* de la collection (Cf. figure III.2).

Dans ce chapitre, nous ne considérerons que des imbrications *homogènes*. Une imbrication de collections est homogène si toutes les régions de l'espace définies par les imbrications sont semblables. Autrement dit, les éléments d'une collection homogène ont tous la même dimension et si on considère la collection homogène comme un arbre, le nombre de fils d'un nœud de l'arbre est le même pour tous les frères du nœud. La figure III.2 donne un exemple de collection homogène et de collection non homogène. Nous verrons les collections non homogènes dans le chapitre IV.

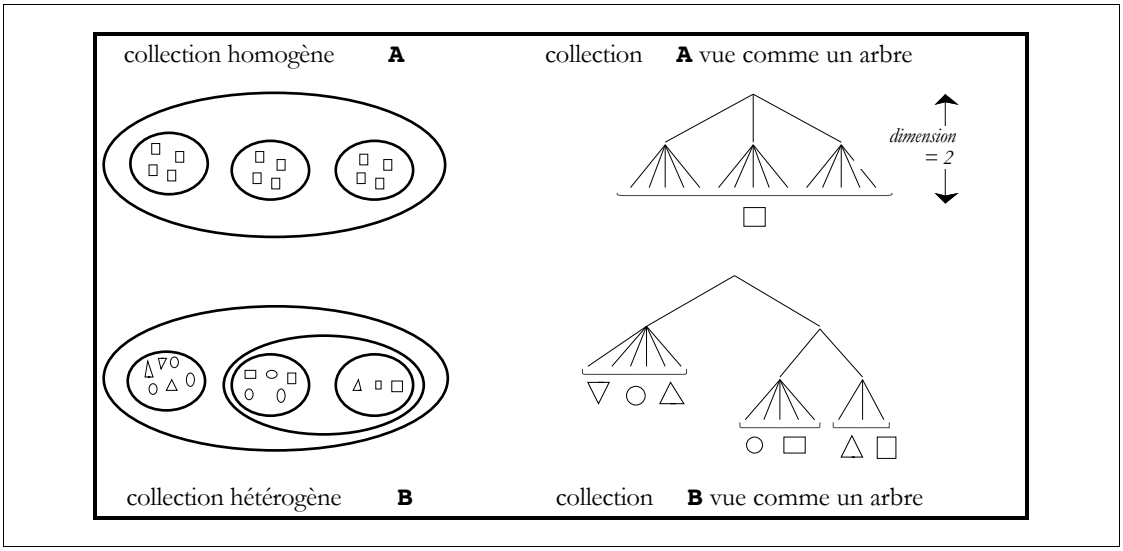


Figure III.2 : *Imbrication homogène et non-homogène de collections.* **A** *est homogène car chaque élément de* **A** *a la même dimension et chaque élément de* **A** *a le même nombre de frères dans la structure. Ce n'est pas le cas de* **B** *: les éléments de* **B** *n'ont pas tous la même dimension (on ne peut donc parler de* la *dimension de* **B***) et le nombre de frères d'un élément, à un niveau donné, n'est pas constant.*

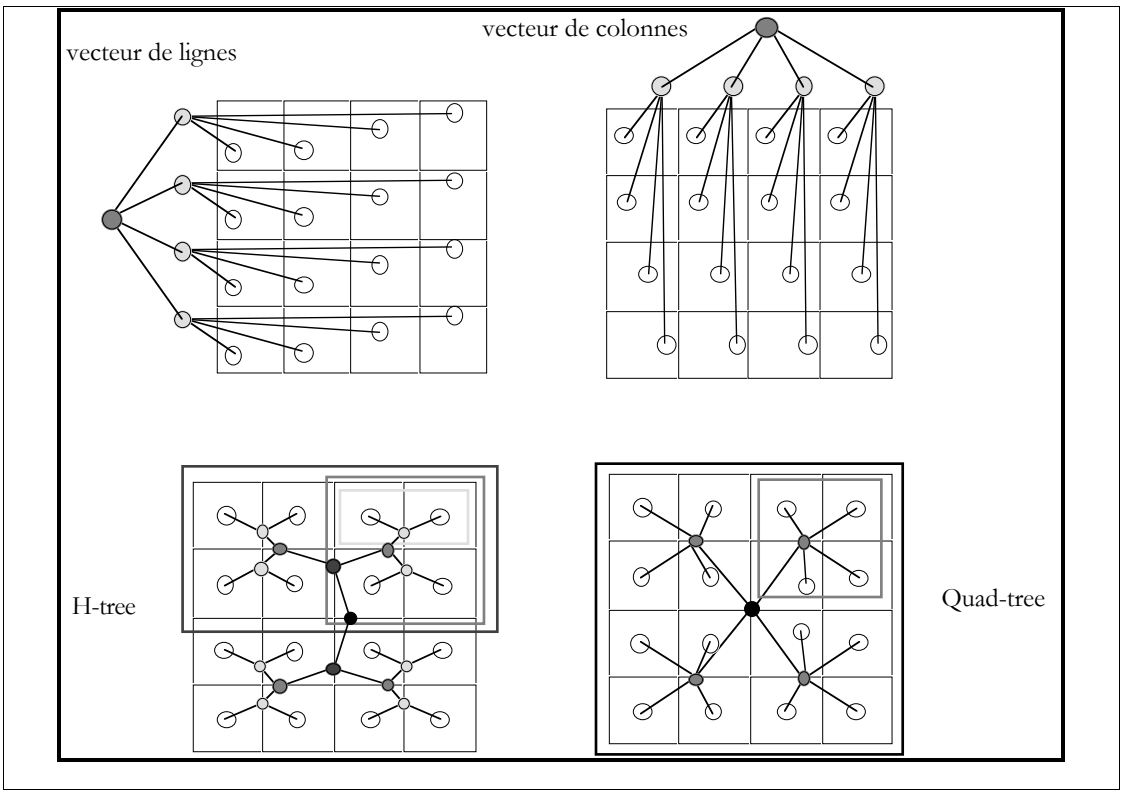


Figure III.3 : *Quatre manières classiques (parmi d'autres) de représenter une surface par des imbrications de collections. Les ronds représentent les éléments de la collection. Les ronds blancs sont des scalaires et correspondent aux points de la surface. Les ronds gris et noirs correspondent à des collections et sont des éléments dans une collection d'ordre supérieur. On a délimité sur le H-tree et le Quad-tree par une bordure de même grisé, les régions de l'espace qui correspondaient à un élément de l'imbrication des collections.*

La figure III.3 donne plusieurs exemples classiques de représentation d'un champ sur une étendue physique à deux dimensions (par exemple une image) par des collections imbriquées. Ces représentations ont des propriétés différentes qui font que dans un cas donné, on utilise une représentation plutôt que telle autre. Par exemple, dans le cas d'un H-tree ou d'un Quad-tree, le nombre d'imbrications est dépendant du nombre de points de la surface alors que dans le cas d'une représentation par hyperplan (vecteur de lignes ou vecteur de colonnes) le nombre d'imbrications dépend de la dimension de l'espace physique à représenter.

Les exemples de la figure III.1 et III.3 nous montrent que la représentation de l'espace par une collection, ou par une imbrication de collections, est largement arbitraire et qu'il n'y a pas une représentation "naturelle" qu'une autre. En particulier, la dimension d'une collection ne correspond pas nécessairement à la dimension de l'espace à représenter (Cf. figure III.1 et le H-tree et le Quad-Tree dans la figure III.3).

III.2.2. Géométrie d'une collection

Dans tous le reste du document, le terme de collection désignera aussi bien une collection d'éléments scalaires qu'une collection imbriquée. Dans ce chapitre, les collections sont toutes des collections homogènes. Puisque nous permettons les imbrications homogènes de collections, nous allons devoir parler de la structure de ces collections. Nous nommerons cette structure *géométrie*.

D'un point de vue purement informatique, en termes de structure de données, la géométrie d'une collection correspond au type d'un vecteur. On peut décrire plus ou moins précisément le type d'une imbrication de vecteurs. Dans le langage `C` par exemple, le type d'un vecteur fait intervenir le type des éléments et leur nombre[1], mais dans le langage ML la description du type d'un vecteur se réduit au seul type de ses éléments.

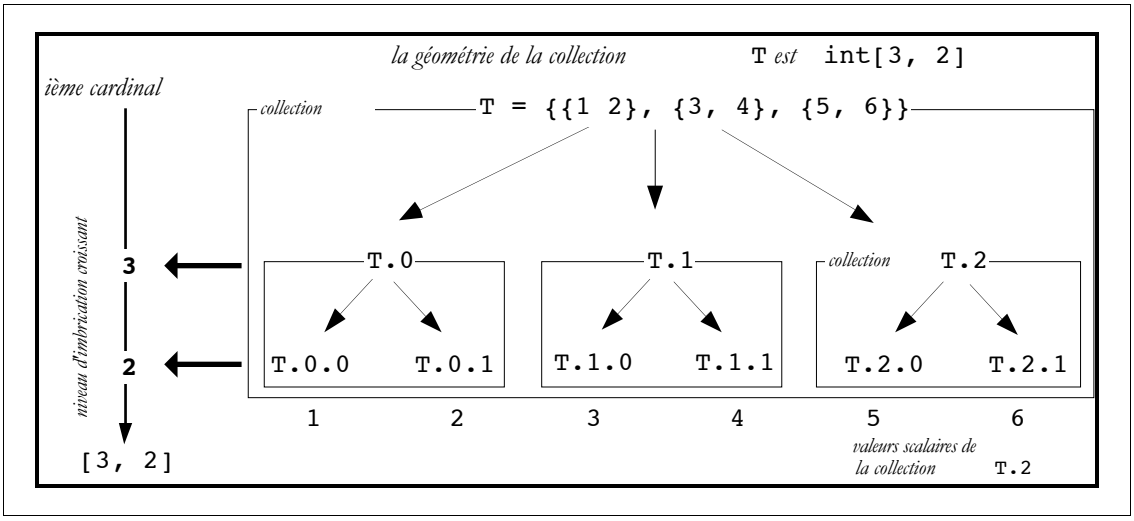


Figure III.4 : *La géométrie des collections imbriquées homogènes de 8½.*

En 8½, on décrira la géométrie des collections homogènes comme le type des éléments des collections feuilles, suivi par une liste de nombres entre crochets, le $i^{ème}$ nombre représentant le nombre d'éléments

[1] Sauf éventuellement pour la première dimension : `int A[][5][2]` est une déclaration valide de vecteur en `C`. Elle correspond à un vecteur dont le nombre d'éléments est inconnu, ces éléments étant des vecteurs de 5 éléments de 2 entiers.

contenus dans le i^{ème} niveau d'imbrication (Cf. figure III.4). Ce nombre sera désigné sous le nom de i^{ème} *cardinal.*

III.2.3. Notations

III.2.3.a. Énumération et désignation des éléments d'une collection

Nous allons noter les éléments d'une collection entre `{` et `}`. Les éléments d'une collection 81/2 sont indexés à partir de 0. Par exemple la collection de 4 éléments dont le i^{ème} élément vaut i s'écrit :

```
{0, 1, 2, 3}
```

Si on suppose que la collection correspond à la discrétisation usuelle d'un espace unidimensionnel, alors les virgules correspondent à l'adjacence spatiale, de la même manière que le point-virgule correspond à la succession temporelle des éléments d'un stream. La collection vide est celle qui ne comporte aucun élément. Elle se note

```
{}
```

L'élément `j` de la collection `v` se note

```
v.j
```

Par exemple `{0 1 2 3}.2` ⇒ `2`. L'imbrication des collections se note à travers l'imbrication des accolades (Cf. figure III.4).

III.2.3.b. Géométrie d'une collection homogène

La géométrie d'une collection de `n` éléments de type α s'écrit :

```
α[n]
```

Par suite, si nous avons besoin de dénoter la géométrie des « collections de `n` collections de `m` entiers », nous écrirons

`(int[m])[n]` ou sans les parenthèses : `int[m][n]`

Cette écriture se simplifie en

`int[n, m]` (attention, remarquez l'inversion[1] des cardinaux !)

Si la valeur de `n` et de `m` est laissée indéterminée, alors cette géométrie désigne toutes les collections qui correspondent à l'imbrication homogène à deux niveaux d'entiers. On peut aussi spécifier *partiellement* une géométrie, en omettant le type scalaire :

```
[n, m]
```

dénote la géométrie d'une collection de `n` collections de `m` *scalaires*, le type de ces scalaires (entier, booléen, …) étant indéterminé. Le contexte doit permettre au compilateur de rétablir le type scalaire manquant.

III.2.3.c. Fonction

Nous aurons besoin dans la suite de pouvoir spécifier des fonctions. La définition d'une fonction correspond simplement à une expression paramétrée qui débute par le mot-clé `function` :

[1] La liste décrit ainsi naturellement les ième cardinaux dans l'ordre des imbrications.

```
function incr(x) = x+1
function f(x, y) = x+y
```

On ne peut pas définir de type pour une fonction : on n'indique ni le type des arguments, ni le type du résultat. Les fonctions sont des expressions *polymorphes* qui peuvent s'appliquer à toutes sortes de collections. Par exemple, la fonction `incr` précédente peut s'appliquer à des vecteurs de tailles différentes. Ainsi, avec les définitions précédentes, on peut écrire :

```
float a[2] = 33.0
int b[4] = 2

incr(a) ⇒  {34.0, 34.0}
incr(b) ⇒  {3, 3, 3, 3}
```

L'application d'une fonction se note comme en `C`, ou bien, sans parenthèses ni virgules pour la liste des arguments :

`f(1, 2) ⇒ 3` ou bien `f 1 2 ⇒ 3`

Mais pour les fonctions prédéfinies usuelles (fonctions arithmétiques, …, prédicats `≤`, …), nous utiliserons les conventions infixes habituelles et nous écrirons « `2≥3` » plutôt que « `≥(2, 3)` ».

III.3. Les opérations sur les collections

Jusqu'à présent, les collections ne se distinguent pas des vecteurs `C` ou des tableaux PASCAL. Ces langages n'offrent comme opération primitive sur les vecteurs, que l'accès à un élément. Les opérations qui opèrent globalement sur les ensembles doivent alors être reconstruites (séquentiellement) à partir des opérations sur les données élémentaires. Dans cette section, nous allons examiner les opérations qui permettent de manipuler des vecteurs comme des touts, ce qui justifie leur renommage en « collections ».

Au passage, et bien que cela ne soit pas la préoccupation de ce chapitre, il faut remarquer que les opérations qui manipulent des ensembles comme des touts sont au fondement de la programmation data-parallèle. En effet, les opérations sur les collections encapsulent un ensemble d'opérations élémentaires concurrentes et coordonnées sur les éléments de la collection, ensemble d'opérations qui pourront être implémentées de manière parallèle sur une machine cible. Les éléments de la collection seront répartis dans la (les) mémoire(s) de la machine parallèle et les opérations sur les collections se traduiront par des calculs répartis entre les processeurs de la machine parallèle. Cette approche du parallélisme de données sera développée dans le chapitre V.

III.3.1. Les trois extensions d'une fonction à une fonction sur les collections

Si `f` est une fonction qui prend des arguments de type `t`, il existe traditionnellement trois façons d'étendre `f` pour la faire agir sur des vecteurs d'éléments de type `t` et de taille quelconque : l'alpha-extension, la bêta-réduction et le balayage.

III.3.1.a. Alpha-extension

L'alpha-extension[1], notée `^` en 81/2, est un opérateur qui permet de transformer une opération sur des éléments de type `S` en une opération sur des collections de α, en appliquant la fonction élément par élément (Cf. figure III.5). La forme la plus simple est sans doute l'alpha-extension d'une fonction unaire :

```
function f(x) = -x

f^ {0, 1, 2, 3}
⇒  {f(0), f((1), f(2), f(3)}
⇒  {0, -1, -2, -3}
```

L'alpha-extension d'une fonction unaire correspond à l'opérateur `map` en Lisp et consiste à appliquer la fonction à chaque élément de la collection.

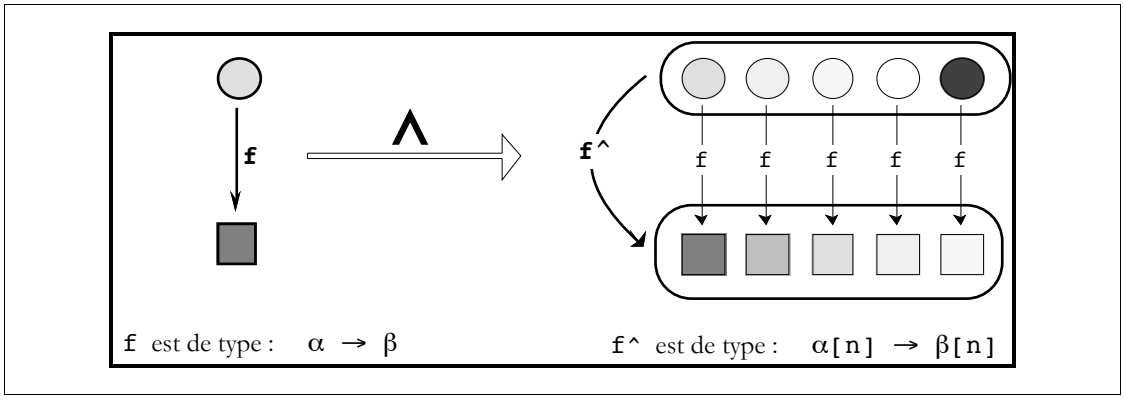


Figure III.5 : *Schéma de principe d'une alpha-extension :* `^` *est un opérateur qui prend une fonction* `f` *et qui renvoie une fonction* `f^` *qui agit sur des collections.*

Cette définition se généralise pour des fonctions à plusieurs arguments :

```
+^ {1, 2, 3} {4, 5, 6}
⇒  {1+4, 2+5, 3+6}
⇒  {4, 7, 9}
```

Un autre exemple avec une fonction à trois arguments :

```
function g(x, y, z) = if (x) then y else z fi;

g^ {true, false, true} {1, 2, 3} {100, 200, 300}  ⇒  {1, 200, 3}
```

III.3.1.b. Alpha-extension explicite et implicite : ambiguïté

L'alpha-extension peut être *explicite*, comme dans les exemples précédents, avec l'opérateur `^`, ou bien *implicite* pour certains opérateurs courants, ce qui permet d'alléger les écritures. En effet, pour étendre une fonction scalaire afin d'agir sur une imbrication à `n` niveaux, il faut utiliser `n` fois le signe `^`, par exemple :

```
+^^  {{1, 2}, {3, 4}}  {{10, 20}, {30, 40}}
```

[1] le nom provient de la communauté APL et n'a rien à voir avec le lambda-calcul, et il en va de même pour la bêta-réduction.

ce qui est une écriture inutilement lourde, l'alpha-extension correcte pouvant être déterminée à partir des arguments. C'est pourquoi, en 81/2, l'apha-extension est implicite pour tous les opérateurs arithmétiques (`+`, `*`, …), les opérateurs relationnels, (`<`, `≤`, …), les opérateurs logiques (`&&`, `||`, …) et la conditionnelle `if…then…else…fi`. On utilise aussi la notation infixe usuelle pour les opérations standards.

L'alpha-extension de l'addition des collections précédentes s'écrit alors simplement :

```
{{1, 2}, {3, 4}} + {{10, 20}, {30, 40}}
```

l'opérateur « + » étant surchargé afin d'opérer aussi bien sur les scalaires que sur des collections de géométrie quelconque.

Cependant en 81/2, l'alpha-extension n'est implicite que pour les fonctions définies initialement sur les scalaires. Pour les autres fonctions, qui agissent dès le départ sur des collections, l'alpha-extension doit être explicite. En effet, l'alpha-extension implicite des fonctions peut être ambiguë lorsqu'on applique à des collections imbriquées une fonction qui agit déjà sur des collections.

Pour illustrer ce problème, prenons l'exemple de la fonction `reverse` qui inverse l'ordre des éléments d'une collection :

```
reverse  { {1, 2},  {3, 4} } ⇒ { {3, 4},  {1, 2} }
```

correspond à l'application usuelle de la fonction `reverse` sur une collection, alors que

```
reverse^ { {1, 2},  {3, 4} } ⇒ { reverse{1, 2},  reverse{3, 4} }
                               ⇒ { {2, 1},  {4, 3} }
```

correspond à l'alpha-extension de la fonction `reverse`. On note bien que le résultat n'est pas du tout le même. Visuellement, en représentant les collections de collections de scalaires par des tableaux :

`reverse` $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix}$ alors que `reverse^` $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$

Autrement dit, suivant qu'on applique `reverse` ou l'alpha-extension de `reverse`, on inverse les lignes ou les colonnes du tableau. Si l'alpha-notation reste implicite, on ne peut plus exprimer les deux fonctions. Une autre illustration de ce problème est donnée par la fonction `card1` qui retourne le nombre d'éléments au "top-level" d'une collection :

```
card1  { {a, b, c},  {d, e, f} }  ⇒  2
card1^ { {a, b, c},  {d, e, f} }  ⇒  {3, 3}
```

III.3.1.c. Alpha-extension des constantes scalaires

Par ailleurs, pour alléger les écritures, les constantes scalaires sont *implicitement* surchargées afin de dénoter une collection de dimension quelconque dont tous les éléments ont pour valeur ce scalaire. Par exemple :

```
1 + {2, 3}  ⇒  {1, 1} + {2, 3}$ ⇒  {3, 4}

2 + {{10, 11}, {22, 33}}
⇒ {{2, 2},  {2, 2}} + {{10, 11}, {22, 33}}
⇒ {{12, 13}, {24, 35}}
```

Suivant le contexte, le compilateur (et le programmeur) est capable de transformer les constantes scalaires dont le type ne correspond pas aux contraintes imposées par le contexte, en une collection de géométrie correcte.

III.3.1.d. Bêta-réduction

La bêta-réduction, notée `\` en 81/2, est un opérateur qui transforme une fonction `f` de deux variables scalaires en une fonction sur les collections : la fonction `f` est utilisée pour combiner tous les éléments d'une collection (Cf. Fig III.6) :

```
+\ {1, 2, 3}  ⇒  1 + (2 + 3)  ⇒  6

max\ {1, 2, 3}  ⇒  max 1 (max 2 3)  ⇒  3
```

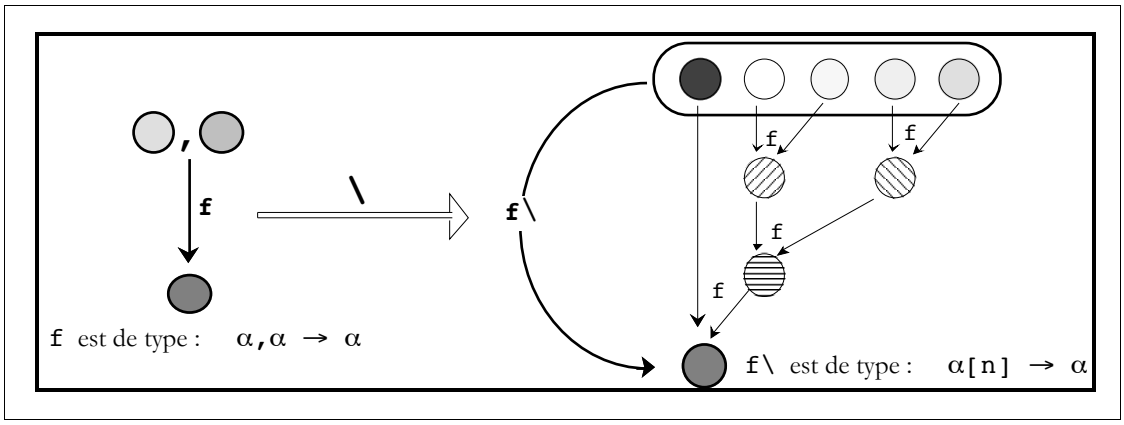


Figure III.6 : *Schéma de principe d'une bêta-réduction.*

L'ordre de réduction n'étant pas précisé, la fonction `f` doit être associative si l'on veut obtenir un résultat bien déterminé. En 81/2, cet ordre est dépendant de l'implémentation et il est de la responsabilité du programmeur de n'utiliser que des fonctions associatives[1]. Par exemple, en utilisant la division (notée `/`), on obtiendra :

$$/\backslash\ \{1, 2, 3, 4\} \quad \Rightarrow (1/2) / (3/4) \Rightarrow \frac{\frac{1}{2}}{\frac{3}{4}} \Rightarrow \frac{2}{3}$$

$$\text{ou bien} \quad \Rightarrow 1 / ((2/3) / 4) \Rightarrow \frac{1}{\frac{\frac{2}{3}}{4}} \Rightarrow 6$$

etc.

Si la collection est imbriquée, la bêta-réduction s'applique aux éléments (scalaire ou collection) du top-level de la collection. Ainsi (en figurant la première dimension suivant une colonne) :

```
max\ | 7  12  14  8  |
     | 16 10  11  10 |
```

[1] Si on fixe un ordre d'évaluation, comme en APL, il ne sera plus possible d'implémenter cette opération de manière parallèle. Si la fonction de réduction est associative, on peut organiser les calculs en un arbre binaire balancé de profondeur $\log_2 n$, arbre que l'on réduira en parallèle, ce qui est à comparer avec les `n` étapes du calcul séquentiel. Un compilateur ne peut pas déterminer si une fonction quelconque est associative.

⇒ `max(7,16)  max(12,10)  max(14,11)  max(8,10)`

⇒ `16 12 14 10`

Si on veut bêta-réduire chaque ligne, il faut bien sûr utiliser une alpha-extension. Par exemple :

`(max\) ^` `7 12 14 8` / `16 10 11 10` ⇒ **Erreur !**⇒ **Erreur !**

Pour obtenir la somme de tous les éléments de la collection, il suffit de bêta-réduire deux fois, indifféremment suivant les colonnes : (`max\)\` ou bien suivant les lignes : (`((max\)^)\)^`.

Comme la réduction suivant une dimension autre que la première est une opération courante, on introduit l'abréviation : `f n\` qui est définie récursivement par

```
f 1\  =  f \
f n\  =  (f (n-1)\) ^
```

Par exemple, avec une collection de géométrie `int[1 3 2]` :

```
+ 3\ { {{111, 112}}, {{121, 122}}, {{131, 132}} }
⇒  (+ 2\) ^  {{{111, 112}}, {{121, 122}}, {{131, 132}}}
⇒  (+\)   ^^ {{{111, 112}}, {{121, 122}}, {{131, 132}}}
⇒ {(+\)^{{111, 112}},  (+\)^{{121, 122}}, (+\)^{{131, 132}}}
⇒ {{+\{111, 112}}, {+\{121, 122}}, {+\{131, 132}}}
⇒ {{223}, {243}, {263}}
```

III.3.1.e. Opération de balayage

La bêta-réduction se généralise grâce à l'opérateur de *balayage*, nommé aussi *préfixe* ou *scan* et noté `\\` en 81/2 : cette opérateur retourne une collection dont l'élément `i` a pour valeur la réduction des `i` premiers éléments de la collection de départ :

```
+ \\ {1, 1, 1, 1} ⇒ {1, 2, 3, 4}
+ \\ {1, 2, 3, 4} ⇒ {1, 3, 6, 10}

* \\ {1, 2, 3, 4} ⇒ {1, 2, 6, 24}

max \\ {1, 3, 2, 4} ⇒ {1, 3, 3, 4}
```

On peut bien sûr appliquer le balayage à des collections imbriquées, et le comportement est le même que pour la bêta-réduction :

`max\\`

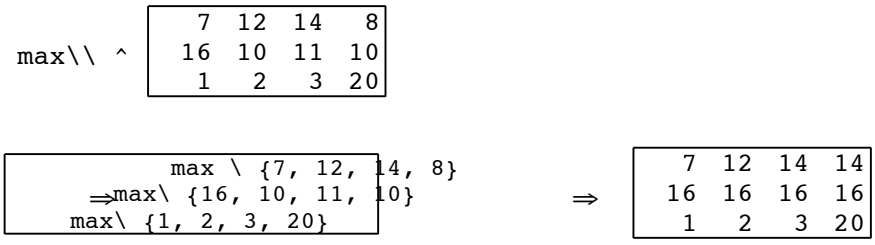
7	12	14	8
16	10	11	10
1	2	3	20

⇒

7	12	14	8
max(7,16)	max(12,10)	max(14,11)	max(8,10)
max(16,1)	max(12,2)	max(14,3)	max(10,20)

⇒

7	12	14	8
16	12	14	10
16	12	14	20



III.3.2. Opérations géométriques

Les opérations géométriques sont celles qui touchent à l'organisation d'une collection, indépendamment des éléments qui la composent. Nous avons déjà rencontré de telles opérations quand nous avons mentionné la fonction `reverse` dans les exemples précédents. Nous considérerons ici principalement : l'imbrication, la projection, la concaténation et les sélections.

III.3.2.a. Imbrication

L'imbrication permet de créer une collection en donnant la liste de ses éléments. C'est une *fonction* n-aire qui correspond à la notation que nous avons déjà adoptée pour noter les collections. Par exemple

```
{ 1,  (2+3),  6-2 }
```

est une *expression* qui s'évalue en la collection

```
{1, 5, 4}
```

L'opérateur n-aire **{**…**,**…**}** correspond à la fonction `list` en Lisp. Notons que si l'opérateur **{**…**,**…**}** existe en 81/2 et permet de créer des collections par l'énumération des éléments, l'opérateur équivalent n'existe pas sur les streams et on ne peut pas créer un stream par l'énumération de ses éléments[1].

III.3.2.b. Projection

L'imbrication permet de construire des collections, les projections correspondent à l'opération inverse et permettent d'accéder à un élément d'indice `i` dans une collection :

```
{1, 2, 3}.1 ⇒ 2
```

L'opérateur de projection utilise la notation déjà introduite pour désigner l'élément d'une collection. Rappelons que le premier élément d'une collection est indicé par 0.

III.3.2.c. La composition des collections

Une collection correspond à la représentation d'un champ, c'est-à-dire à la variation d'une valeur sur un espace de points. La considération simultanée de deux espaces E_1 et E_2 représente un nouvel espace $(E_1 \cup E_2)$ contenant l'union des points de E_1 et de E_2. Sur ce nouvel espace, on peut construire un champ `C` à partir de la donnée d'un champ C_1 sur E_1 et d'un champ C_2 sur E_2 . Si `A` est la collection correspondant à C_1 et `B` la collection correspondant à C_2, la collection correspondant à `C` sera notée `A#B` (Cf. figure III.7) et appelée la *composition* de `A` et de `B`.

---

[1] Rappelons que la notation **<1; 2; …<** du chapitre II ne correspond pas à un opérateur 81/2 sur les streams. Il s'agit d'une notation utilisée dans ce document pour figurer ce qui "se passe dans le temps" lors de la production des valeurs d'un stream.

Il y a de multiples façons possibles de composer deux collections sans toucher à la valeur de leurs éléments (on ne fait que considérer les deux collections simultanément). Par exemple, à partir de

`A = {x, y, z}` et `B = {a, b, c}`

on peut construire

`{x, y, z, a, b, c}` *ou bien* `{x, a, y, b, z, c}`
ou bien `{{x, y, z}, {a, b, c}}` *ou bien* `{a, b, c, x, y, z}`
ou bien `{{x, a}, {y, b}, {z, c}}` *ou bien* `{x, y, z, {a, b, c}}`
etc.

La construction pertinente est celle qui permet de conserver la structure de l'espace de départ quand on restreint `A#B` au points d'une des collections de départ. Autrement dit, l'opérateur `#` doit conserver les voisinages. Cependant, nous avons décidé dans la version 1.0 de 81/2 de simplifier le traitement de l'espace et nous ne tenons pas compte de la notion de voisinage. Cela se traduit sur les collections par le fait que l'indexation des éléments d'une collection est arbitraire et ne reflète pas la structure de l'espace sous-jaccent. Par conséquent, l'indexation des éléments de `A#B` est aussi arbitraire.

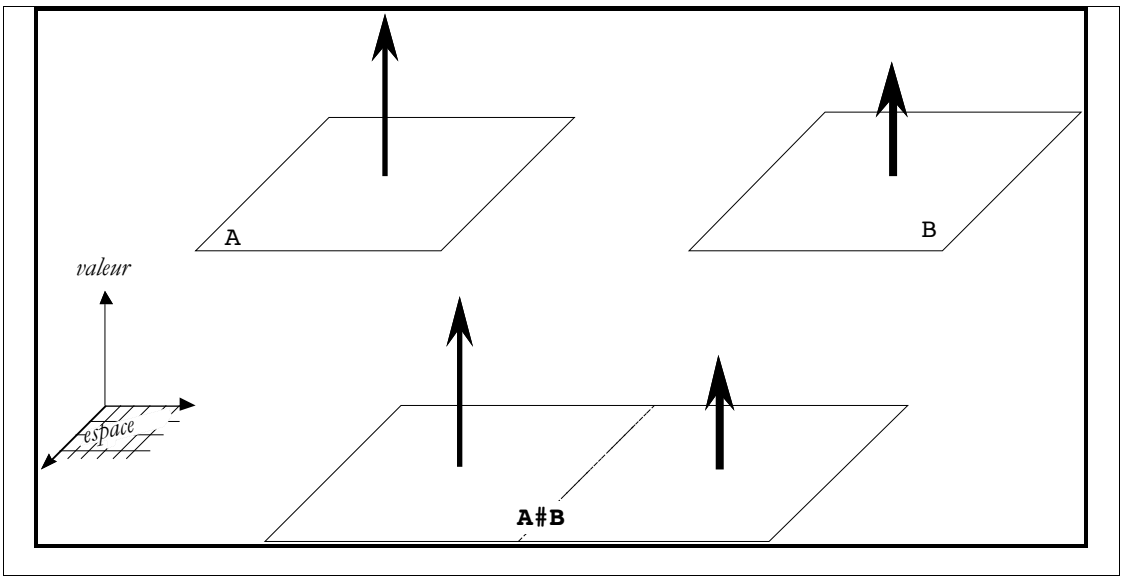


Figure III.7 : *Représentation de l'union de deux champs. Les flèches en gras symbolisent les valeurs du champ en chaque points ede l'espace.*

Pour le moment, nous choisissons donc arbitrairement l'interprétation suivante pour l'opération de composition (cette interprétation sera complétée dans le chapitre IV). L'opérateur `#` correspond, pour ce qui est de l'indexation des éléments des collections homogènes, à la *concaténation* des collections. Concaténer deux collections consiste simplement à les mettre "bout à bout" :

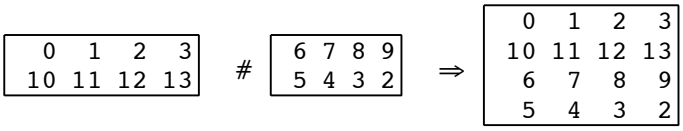
`{1, 2, 3} # {4, 5, 6, 7}` ⇒ `{1, 2, 3, 4, 5, 6, 7}`

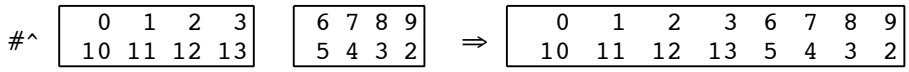
C'est donc l'opération analogue à la fonction `append` en Lisp pour les listes. Mais nous verrons dans le prochain chapitre, que la composition ne se réduit pas à la seule concaténation des éléments des collections homogènes.

La composition utilisée conjointement avec l'alpha-extension permet d'ordonner les éléments suivant d'autres dimensions que la première dimension :

```
#^ {{0, 1, 2, 3}, {10, 11, 12, 13}} {{6, 7, 8, 9},  {5, 4, 3, 2}}
     ⇒    { {0,   1,  2,  3} # {6, 7, 8, 9},
            {10, 11, 12, 13} # {5, 4, 3, 2} }
     ⇒    { {0,   1,  2,  3, 6, 7, 8, 9},
            {10, 11, 12, 13, 5, 4, 3, 2} }
```

Visuellement, en représentant arbitrairement la première dimension suivant une colonne :





III.3.2.d. Relations entre imbrication, projection et composition

Composition et projection sont liées : si `n` est le 1-cardinal (i.e. le nombre d'éléments eu top-level) du vecteur `A` alors

`si i<n  (A#B).i ≡ A.i`
si `i≥n  (A#B).i ≡ B.(i-n)`

Composition et imbrication sont liées par l'égalité suivante :

```
{a, b, …, d} # {x, y, …, z} ≡ {a, b, …, d, x, y, …, z}
```

III.3.2.e. La sélection simple

En 81/2 on dispose de deux opérations de sélection, la *sélection simple* et la *sélection composée*. Elles permettent de choisir parmi les éléments d'une collection pour composer un nouvelle collection. Pour cela, on donne la collection de départ et la collection des indices des éléments à sélectionner :

```
A = {1, 2, 3, 4, 5, 6}                          /* collections de départ */
I = {0, 0, 2, 1}          /* collection des indices des éléments à prendre dans A */

A(I) ⇒ {1, 1, 3, 2}
```

La sélection simple se note avec des parenthèses `()`. On remarque que le résultat est une collection qui a autant d'éléments que la collection des indices `I`. Chacun des éléments de la collection-résultat est obtenu en sélectionnant sa valeur dans la collection de départ grâce à l'indice donné par l'élément correspondant de `I`. Autrement dit,

`A(I).i  = A.(I.i)`

Voici un autre exemple, où les éléments de `I` ne sélectionnent pas des scalaires, mais des collections :

```
B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
B(I) ⇒ {{1, 2, 3}, {1, 2, 3}, {7, 8, 9}, {4, 5, 6}}
```

L'opération de sélection est donc équivalente à ce qu'on appelle une « opération gather » en FORTRAN parallèle[1].

III.3.2.f. Géométrie de la sélection simple

En fait, l'opération de sélection simple en 81/2 est un peu plus générale : la collection des indices peut être une collection imbriquée : la géométrie de cette collection devient alors la géométrie aux premiers niveaux du résultat. Reprenons les mêmes collections `A` et `B` et soit `J` une collection imbriquée d'indices :

```
J = {{0, 1}, {2, 0}}

A(J) ⇒ {{1, 2}, {3, 1}}
B(J) ⇒ {{{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {1, 2, 3}}}
```

Les accolades en gras sont les imbrications qui proviennent de la hiérarchie de `J`, alors que les accolades en maigre proviennent de la structure des éléments de `A` ou de `B`. En termes de géométrie, on peut dire que si `X` a pour géométrie $[x_1, \ldots, x_p]$ et `I` a pour géométrie $[i_1, \ldots, i_q]$ alors `X(I)` a pour géométrie $[i_1, \ldots, i_q, x_2, \ldots, x_p]$. Cette généralisation se comprend facilement en termes de substitution d'arbre : on remplace les *feuilles* de la collection des indices par les éléments sélectionnés (Cf. figure III.8).

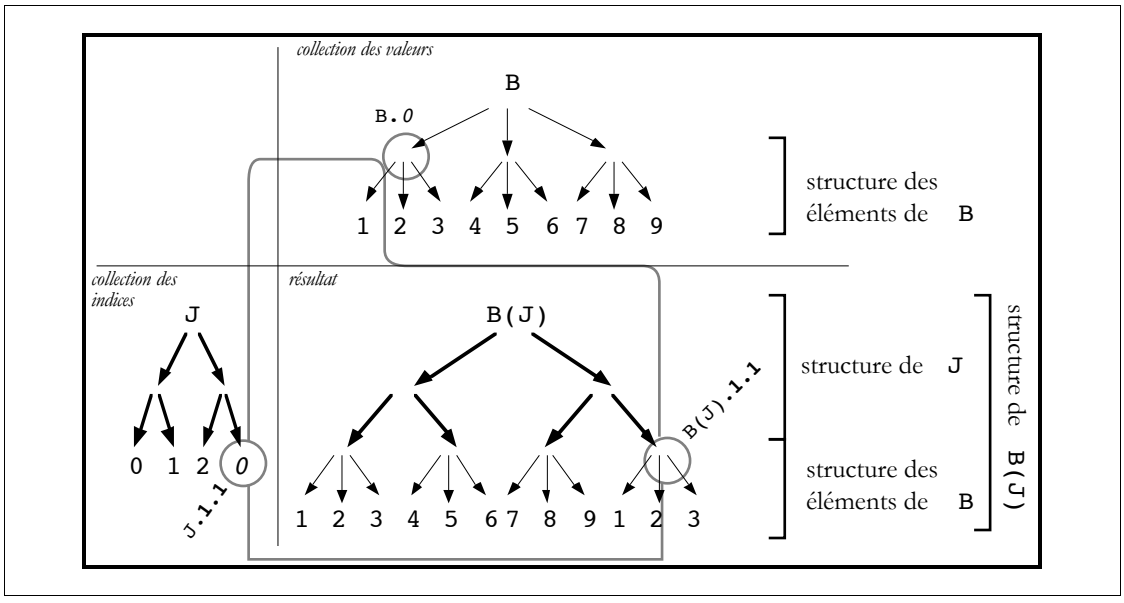


Figure III.8 : *Illustration de la sélection simple à partir d'une représentation en arbre des collections.*

III.3.2.g. La sélection composée

L'opération de sélection simple est une opération qui réalise un ensemble de projections. Chaque projection correspond à une des valeurs stockées dans la collection des indices. La structure de l'ensemble des projections est donnée par la structure des indices. Si la collections des indices se réduit à un seul élément, la sélection simple redevient une projection :

---

[1] Dans un langage impératif comme C, où on ne peut pas manipuler les vecteurs comme des touts, on aurait dû écrire : `for(i=0; i<max; i++) Resultat[i] = A[ I[i] ];`

si I = { p } alors A(I) ≡ A.p

Cela est vrai quelque soit la structure de **A**. Si **A** est une collection a plusieurs dimension, on peut appliquer des projections successives, par exemple :

(A.p).q ou plus simplement A.p.q

L'expression équivalente en utilisant des opérations de sélection simple est obtenue comme pour la projection, par application successive, i.e. :

si I = { p } et J = { q } alors (A(I))(J) ≡ A.p.q

Ces applications successives sont malcommodes car elles obligent à séparer les indices qui désignent un point ou une sous-collection de **A** en plusieurs collections (autant de collections que de projections successives).

C'est pourquoi on introduit la *sélection composée* « **.()** » qui utilise la collection des indices successifs pour atteindre un élément :

si IJ = {p, q} alors A.(IJ) ≡ A.p.q

Cette opération, qui coïncide avec une succession de projections quand la collection des indices est un vecteur de dimension 1 (une projection dans chaque dimension), se généralise, afin de représenter des « ensembles de projections successives ».

Plus précisément, une « collection feuille » de la collection des adresses (et non plus cette fois un « scalaire feuille » comme dans le cas de la sélection simple), correspond à la liste des indices nécessaires pour sélectionner l'élément du résultat dans le vecteur de départ. Par exemple :

```
B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}          /* collection de départ */

K = {{0, 0}, {1, 1}, {2, 2}}                   /* une collection d'indices */
L = {{{0, 0}, {0, 1}}, {{0, 1}, {1, 1}}}       /* une collection d'indices */

B.(K)  ⇒ {B.0.0,  B.1.1,  B.2.2}
       ⇒ {1, 5, 9}

B.(L) ⇒ {{B.0.0,  B.0.1},  {B.0.1,  B.1.1}}
      ⇒ {{1, 2}, {2, 5}}
```

Nous avons figuré en gras la structure héritée de la collection des indices. Il s'agit de toute la structure des indices moins la dernière dimension. Les éléments de la dernière dimension correspondent aux collections qui représentent la suite des indices successifs à appliquer à la collection argument. En termes de géométrie, si **X** a pour géométrie $[x_1, \ldots, x_p]$ et **I** a pour géométrie $[i_1, \ldots, i_q, r]$ avec $r \leq p$, alors **X.(I)** a pour géométrie $[i_1, \ldots, i_q, x_{r+1}, \ldots, x_p]$.

L'opération de sélection composée s'interprète facilement en termes de substitution d'arbres : les vecteurs-feuilles de la collection des indices représentent *les chemin d'accès* aux données stockées dans la collection source (Cf. figure III.9).

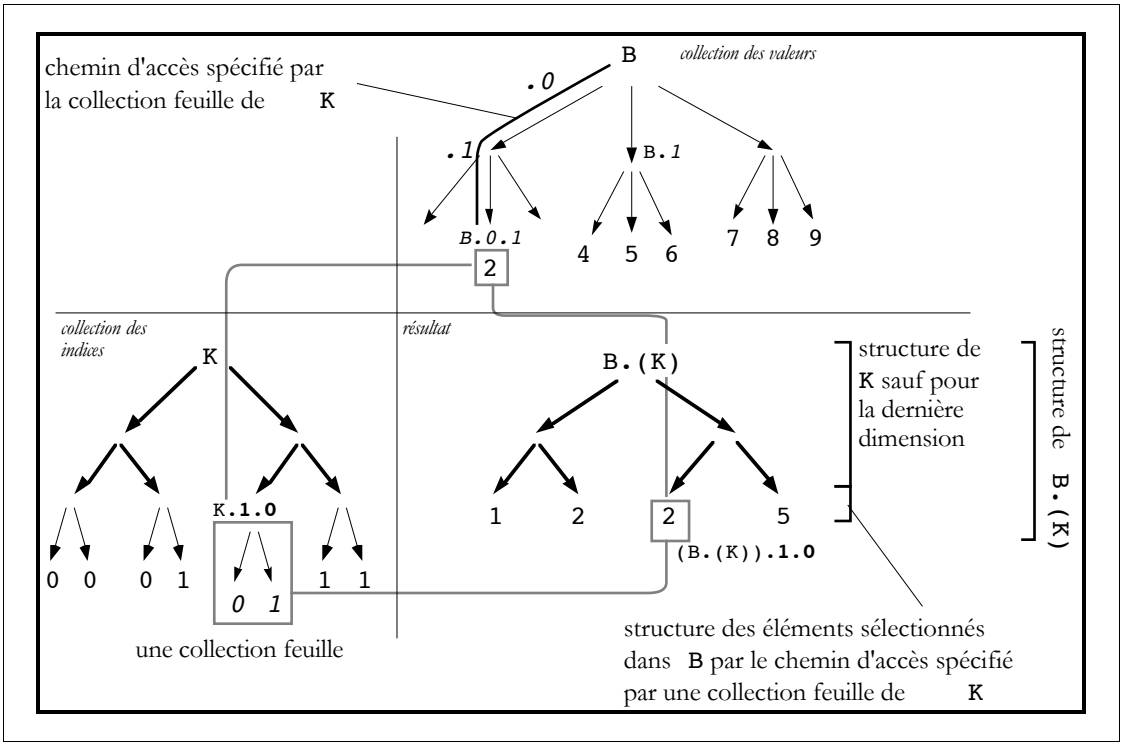


Figure III.9 : *Illustration de la sélection composée en termes de substitution d'arbres. La collection des indices correspond à un ensemble structuré de chemins d'accès à une valeur ou à une sous-collection de la collection source ; un chemin d'accès est représenté par une collection d'indices qui correspondent aux projections successives dans des dimensions successives.*

III.3.2.h. Construction d'une collection d'indices

Afin de faciliter la construction des collections d'indices pour les opérations de sélection, (Cf. les exemples d'utilisation des sélections dans la section suivante), nous avons introduit en 81/2 un opérateur semblable à l'opérateur *iota* en APL : l'opérateur noté « ' » en 81/2 prend un scalaire `n` et construit la collection des `n` premiers entiers :

```
'n ⇒ {0, 1, 2, …, (n-1)}
```

La valeur du i[ème] élément de cette collection est égal à la valeur de son indice. Sélection simple et iota sont reliés par l'identité :

`A('n) ≡ A` avec `n` le premier cardinal de `A`.

Afin de faciliter l'utilisation de la sélection composée, on étend l'opérateur iota afin qu'il accepte une collection en argument. L'opérateur iota généralisé prend une collection et retourne la collection des indices correspondant à la géométrie de l'argument (si on parle en terme de coordonnées d'un point pour désigner la suite des indices permettant d'accéder à un point, alors l'opérateur iota généralisé permet de construire la collection des coordonnées). Par exemple, si `B` est une collection de géométrie `[2, 3]` :

```
'B ⇒ {{{0, 0}, {0, 1}, {0, 2}}
      {{1, 0}, {1, 1}, {1, 2}}}
```

La géométrie de `'B` est `[2, 3, 2]`. Visuellement :

```
si  B =  ·  ·  ·        alors  'B ⇒  {0, 0}  {0, 1}  {0, 2}
         ·  ·  ·                     {1, 0}  {1, 1}  {1, 2}
```

La sélection généralisée et le iota généralisé sont reliés par la relation :

```
X.('X) ≡ X
```

Enfin, l'opérateur *cardinal* appliqué à une collection homogène `T` de dimension `n`, renvoie une collection de `n` entiers, dont le `i`ème élément a pour valeur le `i`ème cardinal de `T`. L'opérateur cardinal se note en 81/2 de manière infixe par `|·|` ; par exemple, appliqué à la collection `B` définie ci-dessus :

```
|B| ⇒ {2, 3}
```

L'opérateur cardinal en 81/2 est l'équivalent de l'opérateur ρ en APL donnant le "rank" d'un tableau. La dimension d'une collection homogène peut s'obtenir en appliquant deux fois `||` :

```
||B|| ⇒ 2
```

III.3.2.i. Coercion d'une collection par troncature et duplication

Sélection simple et sélection composée permettent d'exprimer n'importe quelle "découpe" dans une collection. On introduit cependant une syntaxe spéciale pour la troncature/duplication, car cette opération intervient souvent. Cette opération, appliquée à une collection `A`, se note `·:[·]` en 81/2. Soit `A` une collection "à découper",

```
A:[c1, c2, …, cn]
```

produit une collection dont la géométrie résultante est `[c1, c2, …, cn]`. Cette opération permet de modifier le nombre d'éléments dans une dimension, mais ne permet pas de modifier le nombre de dimensions. En termes de géométrie, si

```
|A| ⇒ {a1, a2, …, an}
```

alors

```
| A:[c1, c2, … cn] | ⇒ {c1, c2, …, cn}
```

Si $c_i \leq a_i$, on ne retient dans la `i`ème dimension de `A` que les c_i premiers éléments. Si $c_i > a_i$, on répète cycliquement les éléments de `A` dans la `i`ème dimension jusqu'à en compter c_i. Par exemple :

```
A = {1, 2, 3, 4}

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

B = {{1, 2}, {3, 4}, {5, 6}}

B:[2, 3] ⇒ {{1, 2, 1}, {3, 4, 3}}
```

Pour la collection `B`, il y a troncature suivant la première dimension et duplication suivant la deuxième dimension.

III.3.3. Quelques exemples d'expressions de collections

III.3.3.a. Somme et produit sur un ensemble de valeurs

Les expressions `+\` et `*\` correspondent aux symboles ∑ et ∏ utilisés en mathématiques. Par exemple :

$\sum_{j=0}^{n-1} f(j)$ devient `+\ (f^ 'n)` et $\prod_{j=0}^{n-1} f(j)$ devient `*\ (f^ 'n)`

Si nous voulons calculer $\prod_{i=1}^{n} i$, la factorielle de `n`, il suffit donc d'appliquer une bêta-réduction sur la collection `{1, …, n}` :

```
function fact(n) = *\ ('n + 1)
```

Supposons que nous n'ayons pas l'opérateur iota à notre disposition. On peut alors obtenir la collection des valeurs de `1` à `n` comme un balayage sur une collection de `n` éléments ayant tous la valeur `1` :

```
1:[n] ⇒ {1, … n fois …, 1}   /* on construit une collection de 1 */
+\\ {1, … n fois …, 1} ⇒ {1, 2, …, n}        /* le balayage génère la collection des entiers */
*\ {1, 2, …, n} ⇒ 1*2* … *n                  /* le balayage permet de faire le produit des entiers */
```

et donc, on peut aussi écrire `fact` sans opérateur iota :

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

III.3.3.b. Quantification sur les éléments d'une collection

L'utilisation de la bêta-réduction conjointement avec les opérateurs logiques permet d'implémenter une quantification existentielle ou universelle sur les éléments d'une collection :

```
∀x∈X P(x)    devient    && \ (P^ X)
∃x∈X P(x)    devient    || \ (P^ X)
```

où `P` est un prédicat booléen, `&&` dénote comme dans le langage `C` le « ET logique », `||` dénote le « OU logique » et `X` est la collection dans laquelle `x` prend sa valeur.

III.3.3.c. Aplatissement

La composition se réduit sur les collections homogènes à la concaténation. La concaténation est une opération associative. On peut donc parfaitement l'utiliser comme opérateur dans une bêta-réduction. L'utilisation de la composition dans une opération de bêta-réduction permet alors "d'aplatir" le premier niveau d'une collection homogène (en anglais *flattening*). Par exemple :

```
# \ {{1, 2, 3}, {4, 5, 6}, {7, 8, 9} {10, 11, 12}}
⇒ {1, 2, 3} # {4, 5, 6} # {7, 8, 9} # {10, 11, 12}
⇒ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

III.3.3.d. Produit scalaire, produit matriciel

La bêta-réduction permet d'écrire simplement le produit scalaire de deux collections de dimension 1 :

```
function scal(A, B) = + \ (A*B)
```

La multiplication de matrices, demande que l'on choisisse d'abord une représentation des matrices en termes de collections. Par exemple, on peut décider de représenter une matrice *X* de dimension (n, m) par la collection `X` de ses colonnes :

$$X = \begin{pmatrix} x_{00} & \cdots & x_{0n} \\ \cdots & \cdots & \cdots \\ x_{m0} & \cdots & x_{mn} \end{pmatrix} \Leftrightarrow \texttt{X} = \{\{x_{00}, \ldots, x_{m0}\}, \{x_{01}, \ldots, x_{m1}\}, \ldots, \{x_{0n}, \ldots, x_{mn}\}\}$$

La géométrie de la collection `X` est `[n, m]`. La multiplication XY de deux matrices peut alors s'écrire en terme de produit scalaire des vecteurs ligne et colonne de la matrice. Il faut transposer la matrice Y, puis dupliquer chaque élément de cette matrice autant de fois qu'il y a de colonnes, puis faire le produit scalaire.

En 81/2, il existe une opération de *produit externe* et une opération de *produit interne* semblables au opérations APL de même nom, et permettant de réaliser simplement le produit matriciel et plus généralement des opérations de calculs tensoriel (Cf. le manuel de référence).

III.3.3.e. Les décalages

Donnons quelques exemples de réarrangement des éléments d'une collection : les décalages. Un *décalage circulaire*[1] vers la droite des `n` éléments d'une collection `A` peut s'écrire de la manière suivante :

```
function RightCshift(A) = A(('n - 1) % n)
```

l'opérateur `%` représente le *modulo*. L'expression `('n - 1) % n` génère la collection :

```
{n-1, 0, 1, …, n-2}
```

qui permet de faire la sélection correspondant à un décalage circulaire vers la droite. Un décalage circulaire vers la gauche s'écrirait :

```
function LeftCshift(A) = A(('n + 1) % n)
```

Le décalage vers la droite avec l'introduction d'une valeur `c` en tête peut s'écrire sans sélection en utilisant la composition et la troncature :

```
function RightEoshift(A, c) = (c#A):[n]
```

ou bien encore : `function RightEoshift(A, c) = c # (A:[n-1])` en tronquant avant de faire la concaténation.

Les décalages circulaires sur des collections imbriquées le long d'une dimension spécifique utilisent la sélection généralisée. Par exemple, les décalages circulaires `Nord`, `Sud`, `Ouest` et `Est` d'une collection de géométrie `[n, m]`, s'écrivent :

```
function  Nord(A)  =  A.( ('A + {{{-1, 0}}}:[m, n, 2]) % n)
function  South(A) =  A.( ('A + {{{+1, 0}}}:[m, n, 2]) % n)
function  West(A)  =  A.( ('A + {{{0, -1}}}:[m, n, 2]) % m)
function  East(A)  =  A.( ('A + {{{0, +1}}}:[m, n, 2]) % m)
```

Le principe de ces fonctions est simple : il s'agit de générer la bonne permutation des indices d'un point. Celle-ci s'obtient en ajoutant un déplacement aux coordonnées d'un point. Par exemple, le déplacement vers le `Nord` est pris ici par convention égal à un déplacement de `-1` sur la première coordonnée.

---

[1] En FORTRAN90, l'opération de décalage circulaire se nomme `CSHIFT` et le décalage avec introduction d'une valeur se nomme `EOSHIFT`.

III.3.4. Les algèbres de Bird-Meertens

Les opérations que nous avons introduites définissent un calcul sur les collections. Ce calcul a été étudié[1] par R. S. Bird et L. G. L. T. Meertens du point de vue des identités algébriques, i.e. des expressions qui sont équivalentes.

Nous allons nous restreindre aux opérations d'alpha-extension explicite `^`, de réduction `\` et de concaténation `#` sur les collections 81/2 et donner quelques résultats. Nous nous restreignons à ces trois seules opérations, car, l'alpha-extension et la réduction comptent parmi les principales opérations qui permettent de manipuler des collections et l'opération `#`, qui coïncide ici avec la concaténation, permet de construire des collections à partir de scalaires et d'autres collections[2]. Ces trois opérations constituent donc le noyau d'une algèbre sur les collections bâtie à partir de l'algèbre des scalaires.

Cette présentation rapide a pour but :

– de caractériser les collections que l'on peut construire à partir de ces trois opérations de base : c'est la notion d'homomorphisme de concaténation ;

– d'illustrer les manipulations algébriques permises par la manipulation d'une collection comme un tout : à travers l'exemple de la construction de la fonction `reverse`.

Plus généralement, une algèbre de collections permet de *simplifier* des expressions entre collections et de les implémenter en termes d'*opérations* plus *élémentaires*. Pour les homomorphismes de concaténation (Cf. ci-dessous), qui contiennent en particulier toutes les fonctions injectives sur les collections, le noyau minimal des opérateurs nécessaires à leur calcul est constitué seulement par la bêta-réduction et l'alpha-extension.

Remarquons que sur une machine parallèle, l'alpha-extension a un coût unitaire alors que la réduction a un coût logarithmique en la taille de la collection à réduire, si l'on suppose que le calculateur a autant de processeurs que les collections ont d'éléments (Cf. le chapitre V). Les algèbres de Bird-Meertens sont une voie que l'on peut suivre pour la formalisation d'un calcul data-parallèle fondé sur le concept de collection.

III.3.4.a. Identités remarquables

Des identités remarquables relient entre eux les trois opérateurs `^`, `\` et `#`. On note par `f` et `g` des fonctions quelconques et par, `A` et `B` des collections quelconques. Alors,

- `l'alpha-extension se distribue par rapport à la concaténation`
  `f^(A#B) = f^A # f^B`

- la composition ∘ des fonctions est distributive par rapport à l'alpha-extension
  `(f∘g)^A = (f^ ∘ g^)A`

- l'alpha-extension et l'opérateur inverse d'une fonction, noté « $^{-1}$ », commutent
  $(f^)^{-1} = (f^{-1})^$

---

[1] Cf. par exemple [BIR 87] dont les exemples de ce paragraphe sont tirés. Pour une introduction aux relations entre les algèbres Bird-Meertens et le parallélisme, Cf. les travaux de Skillicorn [SKI 90].

[2] On suppose ici que si s_1 et s_2 sont des scalaires, et `C` une collection `{c, …}`, alors $s_1 \# s_2 \Rightarrow \{s_1, s_2\}$ et $s_1 \# C \Rightarrow \{s_1, c, \ldots\}$. En 81/2 ce problème ne se pose pas car les scalaires sont identifiés avec les collections de géométrie `[1]`. Dans les algèbres de Bird-Mertens, les scalaires sont soigneusement distingués des collections, ce qui amène à étendre naturellement la définition de la concaténation.

• la bêta-réduction se distribue par rapport à la concaténation

```
f \ (A#B) = f(f\A, f\B)
```

sous la condition que `A` et `B` soient des collections non vides et que `f` soit une fonction associative.

III.3.4.b. Collections ayant zéro ou un élément

La bêta-réduction d'une collection `{a}` à un seul élément est définie par `f\{a} = a`. Si on veut étendre la définition de la bêta-réduction à la collection "vide" `{}` qui a `0` élément, il faut alors poser :

```
f\{} = e
```

où `e` est l'élément neutre de `f`. Par exemple pour l'addition, `+\{} = 0`. Si `f` n'a pas d'élément neutre, la bêta-réduction n'est pas définie sur le vecteur vide. La raison de cette extension est de préserver la distributivité de `\` par rapport à `#` dans le cas où un des arguments est la collection vide. Cette collection est en effet l'élément neutre de `#`, et par suite :

```
f \ ({a}#{}) = f(f\{a}, f\{}) = f({a}, f\{})
```

par distributivité. Mais

```
f \ ({a}#{}) = f\{a} = {a}
```

car `{}` est élément neutre de `#`. Les deux expressions ont une valeur identique pour tout `a` si on prend pour valeur de `f\{}`, l'élément neutre de `f`.

Cependant, dans 81/2, nous éviterons la manipulation de la collection vide et un programme 81/2 qui a la collection vide pour solution est un programme détecté comme incorrect à la compilation (Cf. le manuel de référence du langage).

III.3.4.c. Homomorphisme, réduction et alpha-extension

Par définition, une fonction `h` sur les collections est un homomorphisme de concaténation (ou encore un #-homomorphisme, ou tout simplement un homomorphisme), s'il existe une fonction dyadique ⊕ tel que :

```
h(x#y) = h(x) ⊕ h(y)
```

Si `h` est un homomorphisme, `h` est défini de manière unique par ⊕ et par sa valeur sur les collections ayant un seul élément (les singletons). Autrement dit, si on définit `f` par

```
function f (a) = h {a}
```

la fonction `h` est définie par la *seule* donnée de ⊕ et de `f`.

Le qualificatif d'« homomorphisme » appliqué à une fonction `f` signifie que `f` conserve une certaine structure entre son domaine de définition et son ensemble d'arrivée. La structure qui est conservée ici est celle de la concaténation : une collection peut s'écrire comme la concaténation de deux sous-collections et on peut donc décomposer une collection quelconque en la concaténation canonique des singletons construits à partir de ses éléments. La définition de ce qu'est un homomorphisme de concaténation indique que si `f` est un homomorphisme, alors l'évaluation de `f` est compatible avec cette décomposition canonique.

Le théorème suivant exprime que chaque #-homomorphisme est la composition d'une bêta-réduction et d'une alpha-extension et inversement :

théorème : Une fonction h est un #-homomorphisme si et seulement si :

```
h = (⊕\) ∘ (f^)
```

pour un certain opérateur ⊕ et une certaine fonction `f`.

Beaucoup de fonctions que nous avons rencontrées sont des #-homomorphismes. Par exemple, une bêta-réduction ou bien une alpha-extension sont des #-homomorphismes : si `g` est la fonction à alpha-étendre, et ⊕ la fonction dyadique à bêta-réduire, posons

```
function identité(x) = x
function gg(a) = { g(a) }
```

alors

```
⊕\ = (⊕\)∘(identité^)
g^ = (#\)∘(gg^)
```

pour tout `g` et tout ⊕, ce qui montre que `^` et `\` sont des #-homomorphismes. `La` fonction `card1` qui retourne le nombre d'éléments d'une collection, et qui peut se définir par

```
function card1(x) = |x|.0
```

est aussi un #-homomorphisme puisqu'on a l'égalité :

```
card1 = (+\)∘(un^)    avec  function un(x) = 1
```

En effet, la fonction `un` alpha-étendue va donner une collection de même taille que l'argument et dont tous les éléments sont égaux à `1`. La bêta-réduction avec l'addition permet ensuite de compter le nombre des éléments.

III.3.4.d. Homomorphisme et fonction injective

Quelles sont les fonctions qui sont des #-homomorphismes ? Toutes les fonctions sur les collections ne sont pas des homomorphismes. Nous allons montrer que les fonctions `h` injectives, i.e. telles que : `h(x) = h(y) ⇔ x = y`, sont des homomorphismes. Les homomorphismes ne se réduisent pas aux fonctions injectives, comme le montre l'exemple de la fonction `card1`.

Si `h` est injective, la fonction h^{-1} est définie sur le domaine image de la fonction `h`. Posons alors

$$u \oplus v = h(h^{-1}u \ \#\ h^{-1}v) \qquad (1)$$

Il s'en suit que :

$$h(x\#y) = h(h^{-1}(hx) \ \#\ h^{-1}(hy)) = hx \oplus hy$$

ce qui montre que `h` est un homomorphisme et que la fonction ⊕ le caractérisant est donnée par (1).

III.3.4.e. Un exemple d'application : la programmation de la fonction reverse

Donnons une application du résultat précédent : la fonction `reverse` est clairement injective et est son propre inverse. Par suite, `reverse` est un #-homomorphisme et on peut utiliser (1) pour construire la fonction ⊕ en 81/2. La fonction `reverse` satisfait :

```
reverse = (⊕\)∘(f^)
```

avec

```
x ⊕ y = reverse(reverse x # reverse y)
f a = reverse {a} = {a}
```

Il est facile de voir que

```
x ⊕ y = y # x
f a = {a}
```

et par suite, en 81/2 on peut définir `reverse` par

```
function  rcat(x, y)   = y # x
function  singleton(a) = {a}
function  reverse(x)   = rcat \ (singleton^x)
```

III.4. Collections définies par des équations

Nous disposons à présent d'opérations qui nous permettent de manipuler les collections comme des touts. Nous allons utiliser ces opérations pour écrire des équations. Une équation entre collections prend une des deux formes suivantes :

identificateur = expression-collection

ou

identificateur spécification-géométrie = expression-collection

Dans le premier cas, la géométrie de la collection est implicitement définie. Dans le second cas, la géométrie déclarée par le programmeur à droite[1] de *identificateur* et avant le signe d'égalité, spécifie explicitement la géométrie de la collection définie.

En fait, les équations précédentes sont des équations qui définissent des streams 81/2 dont la valeur a un instant donné est une collection. Mais les streams correspondants ont un seul top. Lors de l'exécution du programme 81/2, la construction effective des valeurs des éléments de la collection met en œuvre des calculs dont l'organisation est prévue par le compilateur, et qui prendront un certain temps. Mais pour le programmeur, *tout se passe comme si la construction d'une collection est instantanée* et prend place en un seul top.

Dans le paragraphe suivant nous allons décrire comment le compilateur calcule la géométrie de chaque collection. Puis dans le reste de cette section, nous verrons comment on peut calculer les valeurs des éléments d'une collection récursive dont la géométrie est connue.

III.4.1. Inférence des géométries par le compilateur

L'inférence des géométries est une caractéristique très importante du langage 81/2. La surcharge systématique des constantes simplifie considérablement l'écriture des expressions et offre beaucoup de confort au programmeur. Elle permet aussi et surtout la définition de *fonctions génériques*, c'est-à-dire de fonctions qui s'appliquent à des arguments indépendamment de leur géométrie (c'est le cas de la définition de la fonction `reverse` donnée en exemple au §III.3.4.e). En contrepartie, le compilateur doit être capable d'inférer la géométrie exacte de chaque collection (Cf. le chapitre V et [GIA 92]).

III.4.1.a. Le problème posé par les géométries implicites

En 81/2, le compilateur est capable d'inférer la géométrie d'une expression dans la plupart des cas. Cela permet de libérer le programmeur de l'obligation de spécifier la géométrie de chaque collection. Par exemple :

```
A = '5                                                    (1)
```

[1] En pratique, en 81/2, la spécification de la géométrie complète, incluant le type scalaire des éléments, utilise une notation infixe à la `C` : on écrira par exemple `float A [2 3]` pour indiquer une collection `A` de deux collections de trois flottants. L'indication du type scalaire aussi bien que de la structure de la géométrie est optionnelle.

définit une collection `A` dont la géométrie, calculée par le compilateur, est `[5]`. L'équation (1) est donc équivalente à la définition suivante

```
A[5] = '5
```

qui spécifie explicitement la géométrie de `A`.

Cependant, beaucoup d'équations sur les collections admettent une infinité de solutions qui ne diffèrent que par la géométrie :

```
B = 3                                                    (2)
```

admet pour solution toute collection dont tous les éléments feuilles ont pour valeur `3`. La situation est similaire à celle rencontrée lors du chapitre précédent pour les horloges des streams. Cette infinité de solutions provient ici de la surcharge des constantes scalaires en constantes collections (notons cependant que l'expression `{3}` dénote sans ambiguïté une collection de un seul scalaire de valeur `3`). Quand une solution admet plusieurs géométries, le compilateur 81/2 rapporte ce fait au programmeur et choisit une géométrie par défaut. Dans le cas de l'équation (2), la géométrie choisie par le compilateur sera `[1]`. L'équation (1) est donc équivalente à l'équation avec géométrie explicite :

```
B[1] = 3
```

III.4.1.b. Inférence par le compilateur des géométries d'un système d'équations

Par contre, si la collection `B` est définie dans un système d'équations et apparaît dans une autre équation :

```
⎧ B = 3
⎩ C = '5 + B
```

le compilateur aurait inféré sans aucune ambiguïté que la géométrie de `B` est `[5]`, cela étant imposé par le contexte d'utilisation de `B`. On voit donc qu'une équation considérée isolément peut être ambiguë (i.e. admettre plusieurs solutions), alors qu'elle n'est plus ambiguë quand elle est considérée à l'intérieur d'un système d'équations. Un système d'équations 81/2 (i.e. un programme) définit un "contexte d'utilisation" des collections qui permet au compilateur de lever certaines ambiguïtés.

Trois mécanismes permettent au compilateur de calculer les géométries de toutes les collections qui apparaissent dans un programme 81/2 :

- La géométrie de la collection peut être déclarée explicitement par le programmeur, lors de la définition de la collection. C'est le cas par exemple dans « `B[5] = 3` ». La spécification de la géométrie suit la notation exposée au paragraphe III.2.3.b.
- La collection intervient dans un contexte qui impose une géométrie. C'est le cas de la définition (1) qui impose la géométrie de `[5]` pour `A`. Toute expression impose des contraintes sur la géométrie des arguments et du résultat. Par exemple, l'expression « `A+B` » impose que `A` et que `B` et que le résultat aient la même géométrie ; l'expression `1:[n]` dénote sans ambiguïté une collection de géométrie `[n]`, etc. Le compilateur 81/2 utilise ces contraintes pour inférer la bonne géométrie de chaque collection. Nous n'exposerons pas ces règles, celles-ci étant évidentes.
- La synthèse des géométries des collections d'un système d'équations par le compilateur peut néanmoins laisser subsister une infinité de solutions, exactement comme au chapitre précédent dans lequel nous avions des équations entre streams qui admettaient une infinité de solutions. Par exemple, le programme suivant :

```
⎧ A = 1
⎨ B = 2                                                  (3)
⎩ C = A + B
```

admet comme solutions tous les triplets de collections `(X, Y, Z)` de même géométrie et dont la valeur des feuilles est respectivement 1, 2 et 3. Dans ce cas, c'est « la plus petite solution » qui sera calculée par le programme. On retiendra que « la plus petite solution » est celle qui compte le moins de dimensions (mais au moins une) et dans chaque dimension, celle qui compte le moins d'éléments. Dans l'exemple (3), le programme calculera donc comme résultat : `A = {1}`, `B = {2}`, `C = {3}`. Cette règle revient à choisir la collection qui a la structure la plus simple possible (et non réduite à la collection vide).

Il faut noter que :

– il est toujours possible pour le programmeur de lever l'ambiguïté sur les géométries introduites par la surcharge des constantes scalaires, en spécifiant explicitement la géométrie de chaque collection ;

– la spécification complète de la géométrie inclut le type scalaire des éléments feuilles de la collection. L'inférence du type scalaire par le compilateur ne présente pas de difficultés[1].

III.4.2. La récursion spatiale

Mise à part l'indication optionnelle de la géométrie, la définition des collections par des équations est tout à fait analogue à la construction des streams par des équations (Cf. le chapitre II). Nous passons donc les détails et nous examinons directement quelques exemples d'équations récursives sur les collections.

III.4.2.a. La définition de 'n par une récursion spatiale

Nous voulons écrire la définition d'un vecteur dont le `i`ème élément vaut `i`. Bien sûr, on peut utiliser l'opérateur iota mais nous voulons l'écrire ici en termes d'opérations élémentaires. Une définition simple et efficace fait appel au balayage :

```
iota[10] = (+ \\ 1) - 1 (4)

iota ⇒ {0, 1, …, n-1}
```

Le nombre d'éléments de `iota` est spécifié en partie gauche de l'équation, par la géométrie `[10]`. Sans cette indication, il y aurait une infinité de solutions à cette équation et le compilateur choisirait la géométrie `[1]` :

```
(+ \\ 1) - 1 ⇒ {0}
```

La définition (4) de `iota` n'est pas une équation récursive. Si les équations récursives sur les streams correspondent à une « récursion temporelle » où la valeur d'un élément d'un stream fait intervenir la valeur des éléments précédents du stream, les équations récursives sur les collections correspondent à une « récursion spatiale » où la valeur d'un élément de la collection dépend de la valeur d'autres éléments de la collection. La définition suivante de `iota` est une définition récursive spatiale :

```
iota = 0 # (iota:[9] + 1)                                    (5)
```

Cette équation définit la collection `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`. En effet :

– On a pas besoin de définir explicitement la géométrie de `iota` car la géométrie de `iota:[9]` est forcément `[9]`. La constante `1` a donc aussi obligatoirement une géométrie de `[9]` et le compilateur affecte par défaut la géométrie de `[1]` à la constante `0`. Le résultat a donc une géométrie de `[10]`.

---

[1] On peut utiliser un mécanisme de typage scalaire comme celui du langage ML par exemple. Cependant, le compilateur 81/2 admet des coercions implicites (par exemples d'entiers en flottants ou en booléens).

– Intuitivement, on a déjà vu qu'une expression comme `0#X:[n]` représente un décalage vers la droite de la collection `X` ; or, une propriété de `iota` est que la valeur d'un élément est égale à la valeur de l'élément précédent augmentée de `1`. D'où la définition (5).

Montrons plus formellement que l'équation (5) définit bien `'10`. La collection `iota` a 10 éléments scalaires pour les raisons précédentes. La géométrie étant fixée, regardons ce qu'il en est des valeurs. Le premier des 10 éléments de `iota` a pour valeur `0`. Si on note $\texttt{iota}_j$ le $j^{ème}$ élément de `iota` alors l'équation peut se réécrire en :

$$\texttt{iota} = \{0\} \# \{\texttt{iota}_0, \ldots, \texttt{iota}_9\}:[9] + \{1_0, \ldots, 1_9\}$$
$$\texttt{iota} = \{0\} \# \{\texttt{iota}_0 + 1, \ldots, \texttt{iota}_8 + 1\}$$
$$\texttt{iota} = \{0, \texttt{iota}_0 + 1, \ldots, \texttt{iota}_8 + 1\}$$

or, $\texttt{iota}_0 = 0$ et donc : $\texttt{iota} = \{0, 1, \texttt{iota}_1 + 1, \ldots, \texttt{iota}_8 + 1\}$ ce qui montre que $\texttt{iota}_1 = 1$, et donc : $\texttt{iota}_2 = 2$, etc. Finalement[1] :

$$\texttt{iota} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

III.4.2.b. Récursion spatiale et dépendance entre points d'une collection

Pour le programmeur, tout se passe comme si la définition récursive de `iota` faisait "surgir instantanément" la collection solution de l'équation dans sa totalité : il n'a pas besoin de spécifier quels sont les calculs à faire pour chaque élément de la collection, ni dans quel ordre il faut les faire.

On pourrait donc croire qu'il est possible de calculer indépendamment la valeur de chaque élément de la collection, d'autant plus que chaque opérateur de la définition de (5) admet une évaluation de la valeur d'un élément indépendamment de la valeur des autres éléments (c'est le cas pour l'addition, la concaténation et la troncature). Or ce n'est clairement pas le cas : la figure III.10 montre bien que pour calculer $\texttt{iota}_n$ il faut disposer de $\texttt{iota}_{n-1}$. Cette dépendance s'interprète facilement à partir du décalage introduit par l'opérateur `#` (tous les autres opérateur n'introduisent aucun décalage).

L'exemple de `iota` est donc important : il montre que contrairement à notre intuition, le calcul de la valeur d'un élément d'une collection définie par une équation ne peut pas toujours se dérouler en parallèle avec le calcul de la valeur des autres éléments. Ce phénomène se produit uniquement si la collection dépend d'elle-même, donc quand la définition est récursive. C'est le compilateur qui décide tout seul comment organiser les calculs.

[1] Au passage, on remarquera que cet exemple illustre bien les manipulations formelles sur les collections permises par un langage déclaratif où on peut remplacer une variable par sa définition.

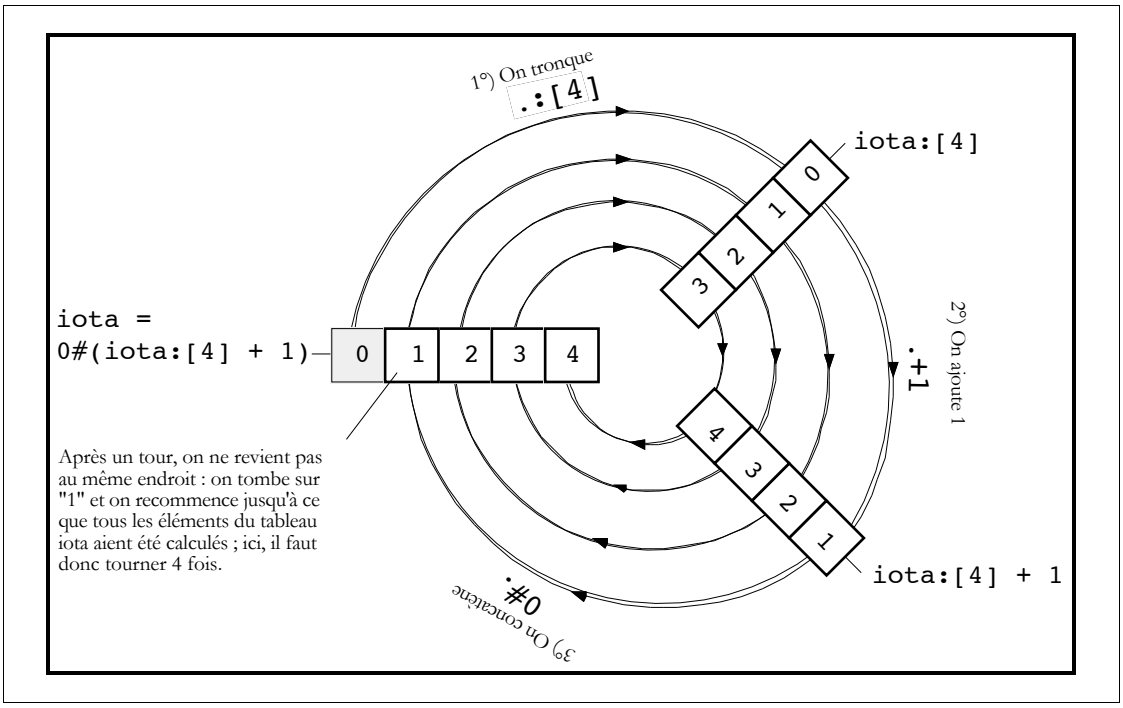


Figure III.10 : *Calcul de* `iota` *(ici on a réduit le nombre d'éléments de iota à 5) La figure illustre les dépendances entre les valeurs des points des différentes sous-expressions de la définition de* `iota`*. Le programmeur ne se préoccupe pas d'organiser les calculs. C'est le compilateur qui met en place le code correct permettant de construire la collection récursive.*

III.4.3. Le calcul d'une fonction récursive primitive par une collection définie par une récursion spatiale

Sur le même schéma de principe que `iota`, on peut définir facilement des collections `F` dont le `i`ème élément vaut `f(i)`, la fonction `f` étant une fonction récursive dont l'évaluation pour l'argument `i` ne fait intervenir que des valeurs `f(j)` avec `j<i`. Une telle fonction est dite *récursive primitive*. La définition de `F` se fait par une récursion spatiale. Par exemple :

```
{ Iota[n] = 0 # (Iota:[n-1] + 1)
  Fact[n] = 1 # ((Iota+1) * Fact:[n-1])
```

Notez que ce ne sont pas des fonctions que l'on définit, mais des collections : pour calculer la factorielle de `5`, il faut explicitement remplacer `n` par `5` dans les deux lignes de l'exemple précédent, le résultat cherché étant la valeur du 5ème élément de la collection `Fact` (on peut y accéder par `Fact.4`). L'intéret d'une collection par rapport à une fonction, est que a valeur de toutes les factorielles pour `x<n` est disponible dans la collection `Fact`[1].

Les fonctions qui correspondent à `Iota` et `Fact` sont des fonctions récursives *du premier ordre* : elle ne font qu'un appel récursif ; ainsi, dans la traduction en termes de collection, la collection n'apparaît qu'une

[1] Si on veut définir la *fonction* factorielle à travers une récursion spatiale, on peut utiliser la définition suivante :

```
function factorielle(n) = {fact[n] = 1 # ((iota+1) * fact:[n-1])}.fact.n
```

L'application de la fonction `factorielle` crée un système d'équations dont on va chercher le `n`ème élément de l'équation de `fact`. Cette construction utilise la notion de système qui sera exposée au chapitre suivant.

seule fois dans le membre droit. Notons que souvent une récursion du premier ordre correspond à un balayage. Par exemple, pour factorielle, on a la définition équivalente

```
Fact[n] = * \\ ('n + 1)        /*définition sans récursion spatiale */
```

Regardons à présent ce qui se passe pour la fonction de Fibonacci dont la récurrence est d'ordre 2 :

```
function fib(x) = if x == 0
                  then 1
                  else if x == 1
                       then 1
                       else fib(x-1) + fib(x-2)
                       fi
                  fi
```

La collection `Fib` associée à cette fonction `fib` peut s'écrire

```
Fib[n] = 0#Fib:[n-1] + {0, 1}#Fib:[n-2]
```

L'expression de `Fib` admet, comme pour `Iota`, une interprétation du calcul en termes de décalages (Cf. figure III.11).

Attention, les collections concaténées en tête des appels récursifs de `Fib`, ne sont pas forcément constituées des valeurs initiales de la suite. En effet, comme le montre la figure III.11, la règle de récurrence s'applique dès le 1er élément du tableau. De manière générale, si on a une relation de récurrence de la forme :

$s_0, s_1, s_{n+1} = g(s_n, s_{n-1})$

on pourra définir une collection `G` par :

$G = g(s_0 \# S, \{e_0, e_1\} \# G)$

avec

$g(s_0, e_0) = s_0$ et $g(s_0, e_1) = s_1$

i.e. e_0 est élément neutre à droite de `g` et $e_1 = (g(s_0, \cdot))^{-1}(s_1)$. Dans le cas de Fibonacci, `g` est la fonction d'addition et donc $e_0 = 0$ et $e_1 = s_1 - s_0$

Plutôt que de trouver les valeurs initiales correctes, on aurait pu plus simplement utiliser une expression conditionnelle pour forcer la valeur correcte des éléments d'indices 0 et 1 :

```
Fib[n] =  if Iota == 0
          then 1
          else if Iota == 1
               then 1
               else v#Fib:[n-1] + {v, v}#Fib:[n-2]
               fi
          fi
```

Les valeurs `v` n'ont pas d'influence sur le résultat final, elles servent uniquement à décaler à droite la collection `Fib`. La collection `Iota` permet de conditionner le calcul en repérant le ième élément par i. Remarquons que cette dernière équation est syntaxiquement très proche de la définition de la fonction `fib`.

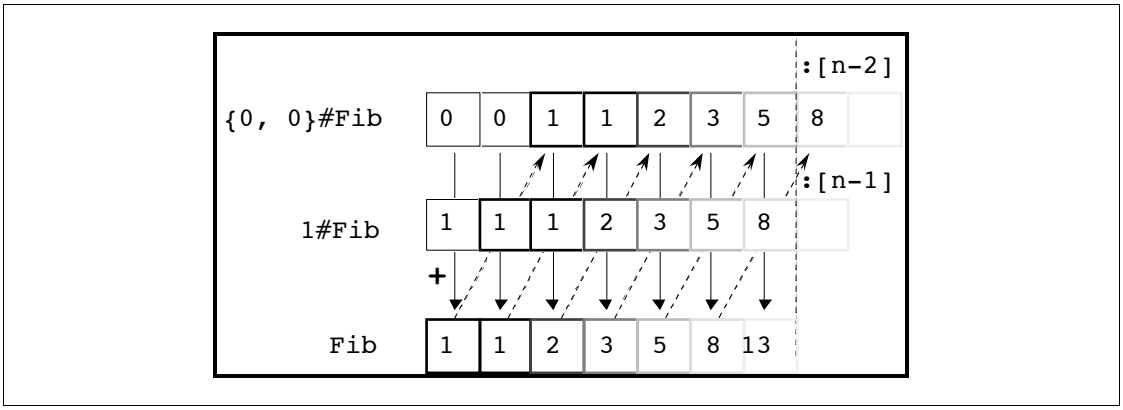


Figure III.11 : *Expression récursive de la collection* `Fib` *(son ième élément a pour valeur la valeur du ième élément de la suite de Fibonacci commençant par* `0` *et* `1`*). Chaque ligne représente une sous-expression de la définition de* `Fib.` *Les flèches représentent des dépendances entre les éléments des collections. Une flèche verticale correspond à une dépendance entre deux éléments de même indice. La concaténation introduit un décalage vers la droite et cette translation des valeurs est figurée par les flèches obliques. La barre verticale indique la troncature des collections. Les différents niveaux de gris indiquent, du plus sombre au plus clair, l'ordre d'évaluation des éléments.*

III.4.3.a. Encore Fibonacci : récursion et décalage à gauche

Une fonction récursive primitive `f` n'utilise pour élaborer la valeur de `f(i)` que des valeurs `f(j)` avec `j<i`. Traduit en termes de récursion spatiale, la valeur d'un élément de la collection ne dépend que des valeurs des éléments à gauche, ce qui conduit à n'utiliser, pour l'expression des récurrences spatiales correspondant à des récursions primitives, que des décalages vers la droite.

Nous allons donner une troisième version de la collection de Fibonacci. L'intérêt de cette version est qu'elle fait apparemment intervenir la valeur du point `i+1` pour le calcul de la valeur du point `i`. La définition va donc utiliser un décalage vers la gauche, contrairement aux exemples que nous avons vu jusqu'à présent et qui ne font appel qu'à des décalages vers la droite (à travers l'utilisation de #).

Pour établir la nouvelle définition de Fibonacci, nous allons dans un premier temps raisonner sur des collections sans tenir compte de leur géométrie. Nous ajusterons le nombre des éléments après. Soit `F` la collection de Fibonacci, par définition on a pour tout `i≥0`, l'égalité :

```
F.(i+2) = F.(i+1) + F.i
```

On peut donc étendre cette relation entre des nombres à une relation entre des collections en utilisant un décalage à gauche :

```
Left (Left F) = (Left F) + F
```

Left représente un décalage à gauche de 1 élément et donc « `(Left(X)).i) = X.(i+1)` » pour tout `i≥0`. Par conséquence, en concaténant la même collection à chaque membre de l'égalité

```
0#1#(Left (Left F)) = 0#1#((Left F) + F)
```

Or vu les valeurs de `F.0` et `F.1`, on a : `f = 0#1#(Left (Left F))`, et donc :

```
F = 0#1#((Left F) + F)
```

Nous avons une expression de `F` en terme de décalage à gauche.

Un décalage à gauche peut s'exprimer par une sélection, mais peut aussi s'exprimer comme la troncature de la collection à ses n-1 derniers éléments, n étant le cardinal de la collection. Si une collection T a n

éléments, la collection `T` tronquée aux (n-1) derniers éléments s'écrit « `T:[1-n]` » : en 8½, un argument négatif dans une troncature sélectionne les éléments de l'argument à partir de la fin et dans le sens des index décroissants (Cf. le manuel de référence).

Il reste enfin à accorder la géométrie de chaque sous-expression grâce à des troncatures. En effet, telle quelle, cette équation est erronée puisqu'elle additionne `Left(F)` et `F` qui n'ont pas le même nombre d'éléments.

On obtient donc une troisième définition possible pour une collection de Fibonacci à 10 éléments :

```
F[10] = {0, 1} # (F:[1-10]:[10-2] + F:[10-2])
      = {0, 1} # (F:[-9]:[8] + F:[8])
```

(le facteur `2` qui apparaît dans la troncature `:[10 - 2]` correspond simplement à la taille de l'entête `{0, 1}`).

La troncature `:[-9]` appliquée à une collection de `10` éléments correspond à un décalage vers la gauche de *1* élément. En effet, le premier élément de `F:[-9]` correspond au deuxième élément de `F`, etc. Cependant, dans la définition récursive de `F`, ce décalage à gauche de *1* "cran" est appliqué à une expression qui se trouve être décalée vers la droite de *2* crans (de par `{0,1}#…`) et par suite, le décalage global qui intervient dans cette définition de `F` est un décalage de `2 - 1 = 1` cran vers la droite. Ce n'est donc qu'en apparence que le calcul du point `i` fait intervenir la valeur du point `i+1` : à l'exécution, c'est bien la valeur du point `i-1` qui sera requise pour calculer la valeur de `i`.

III.4.4. Calcul des expressions récursives de tableaux et définitions récursives de fonctions

Les exemples précédents montrent qu'il y a une relation profonde entre les définitions récursives de collections et les fonctions récursives. En effet, on peut associer à chaque collection une fonction d'un entier dont la valeur pour l'entier `j` est la valeur du `j`ème élément de la collection (c'est même la définition abstraite d'un champ). La question qui se pose alors est : peut-on "représenter" par une définition récursive de collection n'importe quelle fonction récursive ?

La question demande à être précisée : en effet en 8½, les collections ont un nombre d'éléments fixé à l'avance, alors que l'on peut demander la valeur d'une fonction pour une valeur d'argument quelconque. Il faut donc convenir d'un domaine fini {1, …, n} sur lequel on veut calculer les valeurs correspondantes de la fonction. Mais cela ne suffit pas.

Dans les exemples précédents, il était facile d'associer une définition récursive de collection à une fonction : les fonctions `f` prises en exemple ne nécessitaient que le calcul de `f(j)` avec `j<i` pour le calcul de `f(i)`. L'expression récursive de la collection correspondante ne faisait donc intervenir effectivement que des décalages « vers la droite », même si des décalages « vers la gauche » apparaissaient. Il est donc possible de calculer les valeurs de la collection, et donc de résoudre l'équation récursive qui la définit, en commençant par calculer le 1er élément du collection, puis le second, etc. À chaque pas du calcul, il y a dans la collection toutes les valeurs nécessaires. Cette façon de procéder est illustrée dans les figures III.10 et III.11 où l'on voit le report des valeurs calculées à chaque pas du calcul dans les différentes sous-expressions de la définition.

On peut bien sûr imaginer des expressions de collections récursives plus générales, par exemple en faisant intervenir une sélection quelconque plutôt qu'un décalage. Partons de la fonction

```
function g(x, y) = if x == y then 1 else g(x, y+1)*(y+1) fi
```

qui peut se traduire par la définition récursive de la collection

```
G[3,2] =  if diagonale(G) then 1
          else Nord(G) * (1+'card1(G)):|G|
          fi

function diagonale(T) = same ^^ 'T
function same(x) = x.0 == x.1
```

`diagonale` est une collection 2D de booléens qui ont la valeur `vrai` sur la diagonale, l'expression de la fonction `Nord` et de la fonction `card1` ont été données plus haut et l'expression `(1+'card1(G)):|G|` génère la valeur correcte de `y+1`. La fonction `g` a un graphe d'appels représenté en figure III.12. Cette fonction est telle que

$$g(x, y) = \frac{\text{factorielle}(x)}{\text{factorielle}(y)}$$

si $x \geq y$ et elle est indéfinie si $x < y$. En particulier,

```
g(x, 0) = factorielle(x)
```

Cette fonction calcule les produits de la factorielle dans le sens croissant, alors que la définition classique les calcule en allant de `n` vers `1` (or il est utile de faire les produits dans le bon sens, par exemple si on veut calculer une factorielle dans un corps non-commutatif, mais cet exemple est artificiel et a surtout pour but d'illustrer notre propos).

On voit que la collection `G` qui représente la fonction `g` ne peut jamais être bien définie, et cela quelque soit la taille choisie pour cette collection (Cf. figure III.12) :

- soit la collection ne contient pas tous les éléments nécessaires au calcul d'un élément quelconque de la collection,
- soit la collection contient des éléments dont la valeur n'est pas définie.

En fait, dans le cas général, la définition par récurrence d'une fonction fait appel aux valeurs de la fonction pour d'autres éléments quelconques et non pas uniquement aux valeurs pour des éléments "plus petits" (sinon on serait assuré que toutes les fonctions récursives sont totales, i.e. sont définies pour tout point de `I N`, alors que l'on sait bien que ce n'est pas le cas).

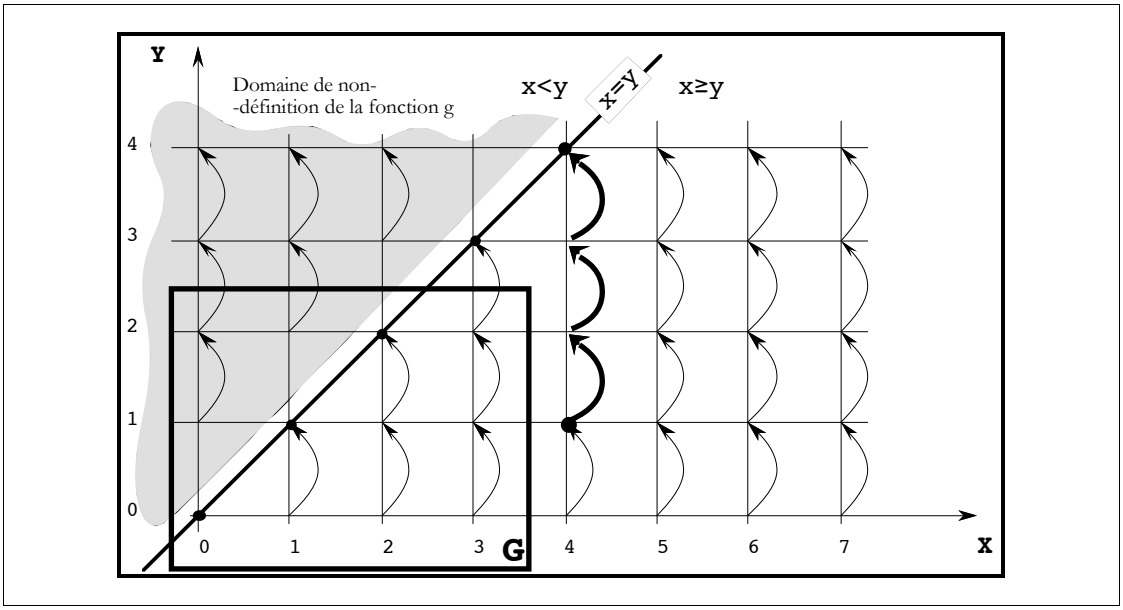


Figure III.12 : *Graphe des appels de la fonction* `g(x, y)` *:*

```
g(x, y) = if x == y then 1 else g(x, y+1)*(y+1) fi.
```

Cette fonction n'est pas définie pour `x` *plus petit que* `y`. *La valeur d'un élément n'est pas définie si son graphe de dépendances n'est pas fini. Cela correspond à un calcul qui ne termine jamais. Le graphe de dépendances d'un élément est l'ensemble des éléments qui sont atteignables à partir de lui. On a représenté en plus gras le graphe de dépendances de l'élément* `(4,1)`. *La collection* `G` *de géométrie* `[3, 2]` *est figurée : on remarque qu'elle comporte des éléments dont la valeur n'est pas définie et des éléments dont les dépendances ne sont pas toutes entières dans* `G`, *par exemple l'élément* `(3,0)`.

Cet exemple montre que deux options s'offrent à nous :

– soit on permet n'importe quelle expression récursive, et on peut alors exprimer des collections dont le calcul va boucler,

– soit on restreint les expressions récursives permises à celles dont on sait, à l'avance et facilement, qu'elles permettent le calcul de la valeur de chaque élément.

Nous choisissons cette dernière approche : en effet, nous voulons définir des collections, avec leur caractéristique, et non des fonctions dans toutes leur généralité. Il est donc naturel d'exiger que tous les éléments de la collection aient une valeur bien définie (en tout cas pour les collections homogènes dont il est question dans ce chapitre). De plus, nous restreignons les expressions récursives à celles dont on peut décider de manière certaine, qu'une évaluation par ordre croissant des éléments de la collection aboutira a un résultat correct.

En contrepartie de ces restrictions, nous sommes assurés de la bonne définition des éléments d'une collection et de l'efficacité du calcul. Par exemple le calcul des collections Fibonacci se fait en temps linéaire avec la taille de la collection, alors que le temps de calcul de l'expression récursive de la fonction Fibonacci est, lui, exponentiel. Cela s'explique parce qu'au moment du calcul d'un élément `i`, on dispose de la valeur de tous les éléments `j≤i`.

III.4.5. Équations récursives admissibles

Une équation récursive sur les collections est *admissible* si tous les éléments de la collection sont bien définis et si l'on peut décider *de manière certaine* que le calcul des éléments d'une collection dans *l'ordre des index croissants* est compatible avec le graphe des dépendances de l'équation récursive[1].

Le terme « de manière certaine » qui apparaît dans la définition de l'admissibilité, indique que nous nous baserons sur un critère *syntaxique* pour éliminer les expressions non-admissibles. Un critère syntaxique est un critère fondé sur la forme du programme, sans tenir compte de ce que le programme calcule réellement. Cela nous amène à interdire des expressions qui sont bien définies, mais que le compilateur aurait dû analyser sémantiquement pour décider si oui ou non elles étaient acceptables. Par exemple le programme

```
C[5] = …
B[5] = …
A[10] = C # A(B)
```

correspond à une collection `A` que l'on peut calculer, si tous les éléments de `B` ont une valeur qui est inférieure à `5` (car à ce moment là, `A(B)` ne fait référence qu'à des éléments de `C`). Ce n'est pas le cas pour une collection `B` quelconque. Pour accepter ce programme, il faudrait que le compilateur soit capable de faire de la preuve de propriétés sur les programmes (et on sait que le problème est indécidable en général).

Le problème de la détermination de l'admissibilité revient à calculer, à la compilation, une *approximation* du graphe des dépendances. Une approximation du graphe des dépendances est un graphe qui inclut le graphe des dépendances réelles. Ainsi, si on respecte les dépendances de l'approximation, on respecte les dépendances du programme. Mais la construction d'une approximation est plus facile que la construction du vrai graphe des dépendances. Ce problème est traité dans le chapitre V. Les techniques que nous développons, permettent en plus de déterminer si le calcul des éléments d'une collection peuvent se faire en parallèle ou non.

III.4.6. Exemples d'équations récursives non-admissibles

Nous allons donner ici quelques exemples d'expressions non admissibles :

```
A = A(B)
```

n'est pas admissible. Il suffit par exemple de prendre `B=0` pour que le calcul de `A.i` dépende de `A.i`. Il existe une infinité de solutions qui répondent à l'équation précédente (pour une valeur de `B` donnée). Par exemple, pour `B = {0, 1}`, toutes les collections à deux éléments sont solutions ; pour `B = {1, 0}`, tous les collections `A` de la forme `{v, v}` sont des solutions. En effet,

```
A = {v, v}({1, 0}) = {{v, v}.1, {v, v}.0} = {v, v}.
```

L'exemple suivant correspond à une équation qui n'admet pas de solution :

```
A = A.i
```

n'est pas admissible, quelque soit `i`, puisqu'indépendamment des valeurs de `i`, il est impossible de trouver une géométrie de collection qui vérifie cette équation[2]. Pour la même raison, nous interdisons des expressions de la forme

---

[1] L'admissibilité qui est utilisée ici porte sur les définitions récursives de collections. On trouvera dans [WAD 81] un critère d'admissibilité pour les équations récursives de streams et dans [SIJ 89] une notion d'admissibilité pour les listes paresseuses à la Haskell.

[2] La construction d'un objet qui réponde à cette équation est néanmoins possible et correspond à un arbre infini. Nous étudions, dans les extensions de 81/2, le calcul et la manipulation de telles expressions.

```
A = { A }
```

Les exemples précédents correspondaient à des équations ayant zéro ou plus d'une solution. Mais nous pouvons définir une collection par une équation qui accepte une et une seule solution, *sans que* cette définition soit admissible.

L'exemple que nous allons donner a un rapport avec la définition des fonctions harmoniques qui interviennent comme solutions de certaines équations différentielles. Les fonctions harmoniques sont des fonctions à valeurs sur un espace telles qu'en chaque point, leur valeur corresponde à la moyenne des valeurs des points voisins. Pour simplifier, plaçons nous dans un espace 1D discrétisé. Ainsi, en chaque point **x** de cet espace :

$$f(x) = \frac{1}{2}(f(x-1) + \frac{1}{2}f(x+1) \quad (1)$$

Les fonctions harmoniques sont généralement définies en donnant la relation (1) et les conditions aux bords, par exemple : $f(0) = f_0$ et $f(4) = f_4$. Cela nous amène à définir la collection **F** suivante :

```
left = {0, 1, 2}
right = {2, 3, 4}
F[5] = f0 # (0.5*F(left) + 0.5*F(right)) # f4
```

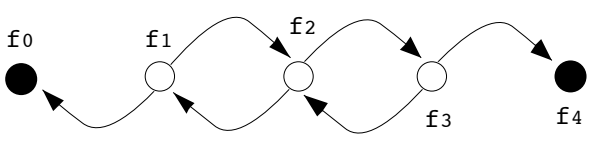
On peut se convaincre que :

- L'équation (1) plus les conditions frontières $f(0)=f_0$ et $f(4)=f_4$ suffisent à définir la fonction **f** de manière unique[1].
- La définition de **F** n'est pas une équation admissible car le calcul de **F.i** fait appel au calcul de **F.(i-1)** et **F.(i+1)**.

Pourquoi rejeter ces équations qui ont une solution unique et bien définie ? C'est parce qu'il n'est pas possible de résoudre (1) uniquement par une série de **substitutions** : chaque fois qu'on veut calculer un élément d'un collection de Fibonacci, et que cet élément n'a pas de valeur définie, il suffit de le remplacer par sa définition et d'itérer le procédé. Cette série de substitutions finit toujours par atteindre les valeurs **Fib(0)** et **Fib(1)** qui sont connues. Le compilateur 81/2 "renverse" alors cette suite de substitutions et ordonne les calculs de manière à ce que la valeur d'un élément soit toujours calculée avant d'être requise.

Dans la définition (1) ce n'est pas le cas, bien que le schéma récursif soit *apparemment le même* que pour Fibonacci (la seule chose importante qui change, c'est la géométrie des arguments des concaténations). Mais, les règles de substitution seules ne suffisent pas à calculer la valeur d'un élément car les éléments terminaux de la récursion ne sont pas les mêmes que pour Fibonacci. Pour résoudre (1), il faut savoir résoudre un système linéaire, car (1) se "déplie" en :

$$\begin{cases} f_0 = \ldots \\ f_1 = \frac{1}{2}f_0 + \frac{1}{2}f_2 \\ f_2 = \frac{1}{2}f_1 + \frac{1}{2}f_3 \\ f_3 = \frac{1}{2}f_2 + \frac{1}{2}f_4 \\ f_4 = \ldots \end{cases}$$



[1] En fait, on peut réécrire la relation (1) sous la forme : « **f(x+1) = 2 f(x) - f(x - 1)** »,forme sous laquelle on reconnaît une équation linéaire aux différences finies. On sait alors que l'espace vectoriel des solutions est de dimension 2 et que la donnée de deux valeurs suffit à définir **f** de manière unique. Ces deux valeurs correspondent aux conditions aux bords f_0 et f_4.

Si on regarde le graphe des dépendances de ces équations, on se rend compte que c'est un graphe cyclique. On ne peut "briser" ces cycles qu'en utilisant des propriétés algébriques[1].

□

[1] On peut vouloir résoudre le système précédent par point fixe (à ne pas confondre avec la résolution d'un système d'équations linéaires par une méthode de relaxation). Si on se place sur le domaine des fonctions continues (au sens des chaines) de {0, 1, 2, 3, 4} dans `I N` muni d'un ordre de Scott [VAN 92], on trouvera comme solution la fonction `f` définie par : `f(0) =` f_0, `f(4) =` f_4 et `f(1) = f(2) =` f`(3) =` ⊥. Cette solution ne correspond pas à la solution désirée. En termes d'éléments du treillis, on aurait voulu obtenir une solution *maximale*, c'est-à-dire ayant une valeur différente de ⊥ en tout point. L'exemple que nous avons donné admet une seule solution maximale, obtenue en résolvant le système linéaire, mais plusieurs solutions non maximales (par exemple, celle que nous venons de donner). La différence entre les collections et les streams, est que nous admettons les streams non maximaux : ils correspondent au choix de la solution ayant le plus petit nombre de tops. Mais nous ne voulons pas, dans la version 1.0 du langage 81/2, définir de collections *homogènes* non maximales.

IV. Programmer avec des tissus 8½

Dans les deux chapitres précédents nous avons vu comment utiliser des équations pour définir des streams et des collections. Nous allons maintenant unir ces deux structures de données dans une structure unique, le ***tissu***. Un tissu est une collection de streams, ou bien, point de vue équivalent, un stream de collections. Un programme 8½ est un système d'équations qui définit des tissus. L'exécution de ce programme énumère les valeurs qui constituent le tissu ; cette énumération se fait suivant l'axe temporel.

Dans ce chapitre, nous introduirons la notion de *système* qui permet de hiérarchiser la définition des tissus d'un programme. Dans la seconde partie du chapitre, nous donnerons des exemples complets de programmes 8½. Il faut noter que les exemples donnés dans les chapitres précédents à propos des streams et des collections sont des programmes 8½ valides mais restreints : un programme-stream correspond à un tissu dont l'aspect collection est réduit à un scalaire ; un programme-collection définit des tissus dont l'aspect stream correspond à des streams n'ayant qu'un seul top.

Nous évoquerons brièvement le fonctionnement de l'environnement de programmation 8½, ses limitations actuelles et le pourquoi de celles-ci, en annexe. Le lecteur intéressé trouvera d'autres exemples, une description plus détaillée du langage et les commandes du compilateur dans le manuel de référence [MIC 94].

IV.1. La notion de tissu 8½

Le langage 8½ introduit une nouvelle structure de données, le *tissu* qui correspond à un stream de collections ou, point de vue dual, à une collection de streams (Cf. figure IV.1). Nous parlerons du *stream support* du tissu pour désigner l'aspect temporel d'un tissu, et nous parlerons de *collection support* du tissu pour désigner l'aspect spatial d'un tissu. Un *point* d'un tissu est un des élément de la collection support du tissu.

Un tissu 8½ est une structure de données qui associe une valeur à chaque point d'une collection et à chaque instant d'un stream. On peut donc représenter le tissu comme un "volume" : une des dimensions du

volume est associée à l'écoulement du temps, une autre à l'aspect spatial du tissu et la troisième dimension représente l'ensemble dans lequel le tissu prend sa valeur. La figure IV.2 illustre ce point de vue pour deux tissus `T` et `Q`. Dans le repère Temps-Espace-Valeur un programme 81/2 définit un "volume" ou une "surface" qui représente l'ensemble des valeurs d'un tissu au cours du temps.

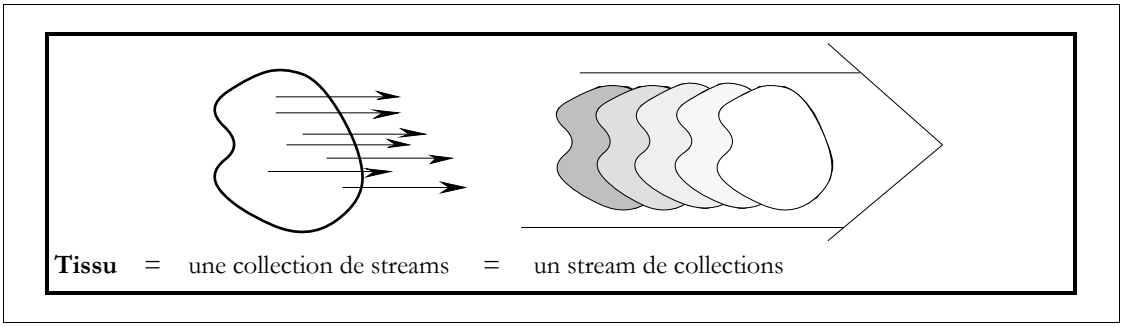


Figure IV.1 : *Deux points de vues sur un tissu 81/2.*

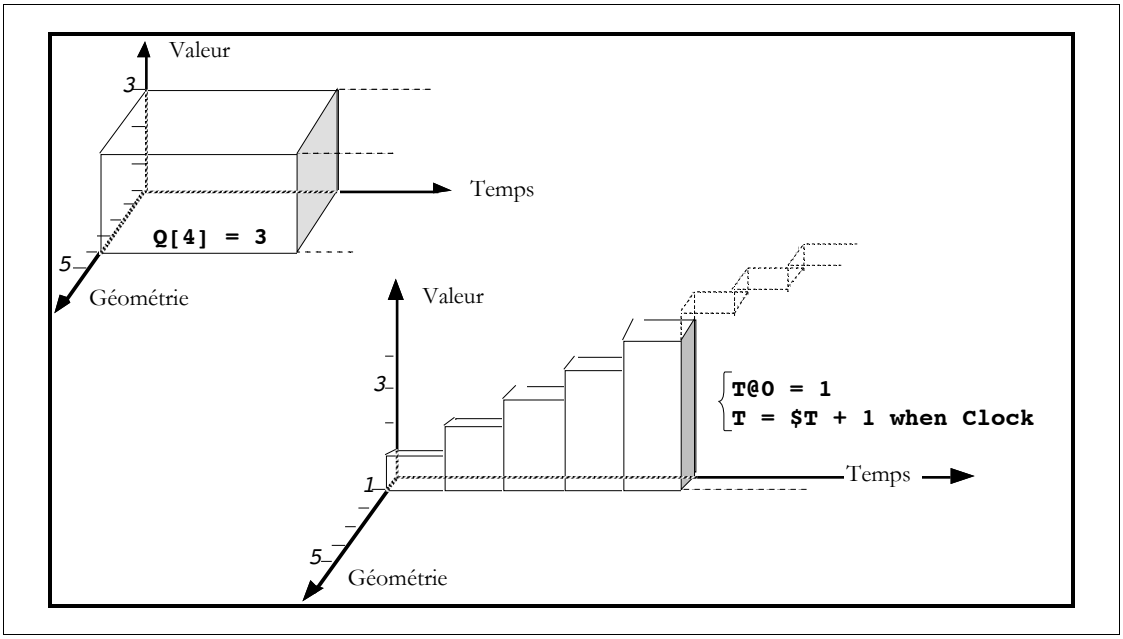


Figure IV.2 : *Exemple de deux tissus figurés dans un repère* Temps-Espace-Valeur.

La combinaison des concepts de stream et de collection en un tissu est *orthogonale*. Cela veut dire que l'aspect spatial et l'aspect temporel n'interfèrent pas dans un tissu pour les opérations que nous avons présentées, et que celles-ci s'étendent donc naturellement sur les tissus de la manière suivante :

- une opération sur les streams est étendue sur les tissus en s'appliquant sur tous les streams correspondant aux différents points de la collection support du tissu ;
- une opération sur les collections est étendue sur les tissus en s'appliquant sur toutes les collections correspondant aux différents instants du stream support du tissu.

Pour des raisons de commodité, quand dans ce chapitre nous parlons de collection, nous voulons parler de la collection support d'un tissu, et quand nous parlons de stream, nous voulons parler du stream support du tissu.

IV.1.1. Exemple de la résolution d'une équation aux dérivées partielles par une méthode de différences finies

Donnons tout de suite un exemple de programme 81/2 qui utilise à la fois l'aspect temporel et l'aspect spatial des tissus.

On considère une barre de métal très fine, que l'on identifie à un segment de `I R`. Nous nous intéressons à l'évolution de la température dans cette barre (elle a, par exemple, été chauffée de manière non homogène et elle se refroidit) : la température dépend du point `x` de la barre et dépend du temps `t`. Notons `u(x, t)` la température en un point `x` à un instant `t` : les physiciens nous disent que la fonction `u` est la solution d'une équation parabolique aux dérivées partielles[1]

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

Une méthode de résolution explicite utilise un schéma aux différences finies de cette équation sur une grille $(X_i = ih,\ T_j = jk)$ où h et k sont les *pas* de discrétisation des variables `x` et `t`. On obtient en discrétisant l'équation précédente par exemple [SMI 85] :

$$\frac{U_{i,j+1} - U_{i,j}}{k} = \frac{U_{i+1,j} - 2Ui,j + U_{i-1,j}}{h^2}$$

ce qui peut se réécrire de la manière suivante

$$U_{i,j+1} = rU_{i-1,j} + (1 - 2r)U_{i,j} + rU_{i+1,j} \qquad \text{avec } r = k/h^2$$

ce qui permet de calculer la température inconnue $U_{i,j+1}$ en un point `(i, j+1)` de la grille en fonction des températures connues le long de la $j^{ème}$ ligne (Cf. Figure IV.3). Les températures à l'instant initial `(j = 0)` sont données par la distribution initiale $U_{i,0}$ de la température dans la barre (c'est une donnée du problème), et des températures aux bords : $U_{0,j}$ et $U_{X/h,t}$ (ce sont les conditions imposées aux frontières du système, appelées aussi conditions aux limites dans ce type de problème).

Le programme 81/2 correspondant est facile à dériver de ces définitions : la suite temporelle des valeurs d'un tissu est utilisée pour représenter l'axe temporel de la grille de discrétisation et le concept de collection est utilisé pour représenter la dimension spatiale du problème (Cf. figure IV.3) :

```
Begin[1] = condition à un bout de la barre                (6)
End[1]   = condition à l'autre bout de la barre           (7)
U@0      = distribution de la température à t = 0         (8)
U        = (Begin # V # End) when Clock                   (9)
```

Les équations (6–9) décrivent les températures de la barre en agrégeant les conditions initiales et les conditions aux limites. Le tissu `V` représente l'intérieur de la barre (la barre moins les deux bords). Le tissu `Clock` correspond au stream `Clock` du chapitre II et il nous sert à discrétiser l'écoulement du temps.

[1] On suppose que la chaleur se diffuse le long de la barre, sans échange avec l'extérieur autre qu'aux deux extrémités.

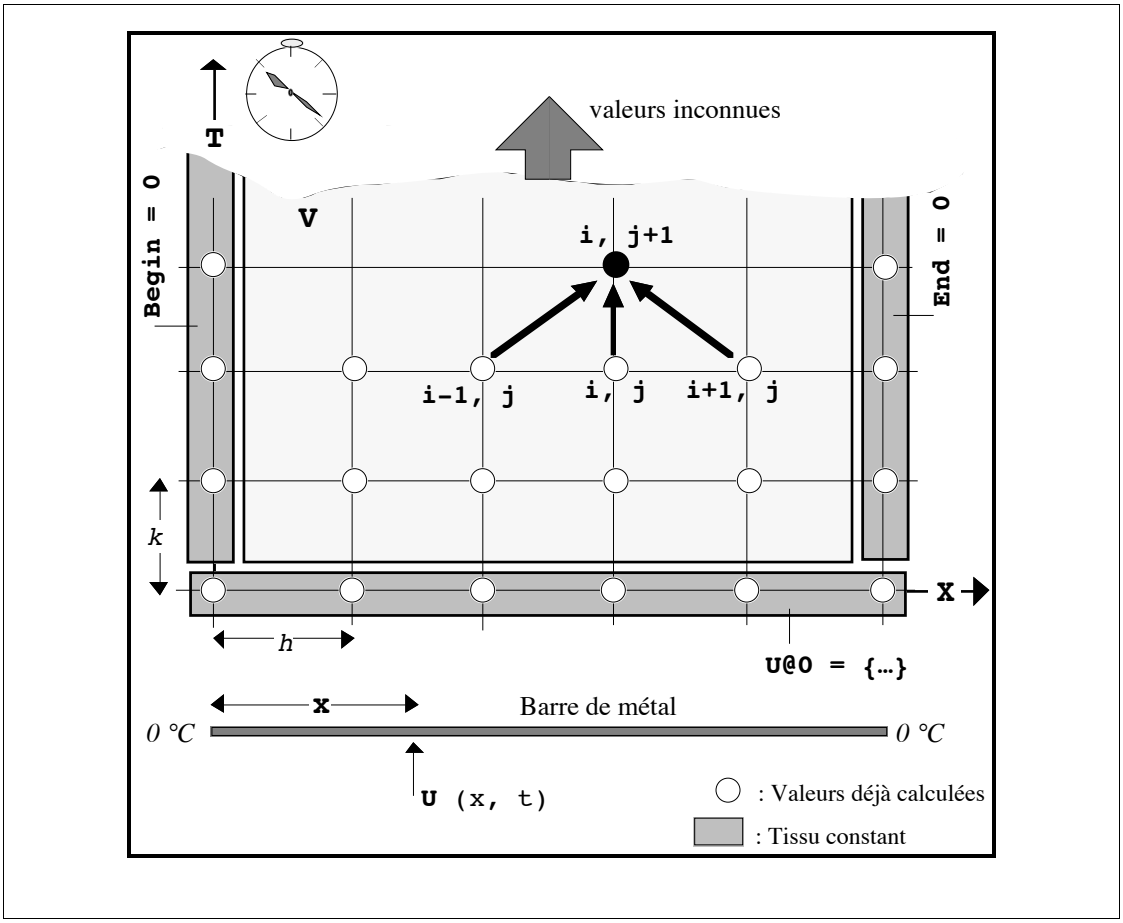


Figure IV.3 . *Résolution par une méthode aux différences finies d'une équation parabolique. Les équations 81/2 correspondant aux conditions initiales et aux conditions aux limites sont données en gras.*

Le tissu `V` est défini par les équations 81/2 suivantes :

```
n = 100              /* nombre de points de discrétisation de la barre */
m = 100-2      /* nombre de points de l'intérieur de la barre */
h = X/n              /* pas de discrétisation de la barre */
r = k/(h*h)          /* k est le pas de discrétisation temporelle */

leftU   = ($U)('m) /* décalage gauche de la barre + troncature */
middleU = ($U)('m+1)    /* sélection de l'intérieur V dans la barre U
*/
rightU  = ($U)('m+2)    /* décalage droit de la barre + troncature */

V[m] = r*leftU + (1 - 2*r)*middleU + r*rightU
```

V, l'intérieur de la barre, est un tissu comportant `n-2` points. `leftU`, `middleU` et `rightU` correspondent aux décalages spatiaux de la valeur passée de la barre. Ces décalages sont obtenus à partir d'une sélection en générant les indices des éléments que l'on veut sélectionner (Cf. §III.3.2.e).

La figure IV.4 montre le résultat de l'exécution de ce programme 81/2.

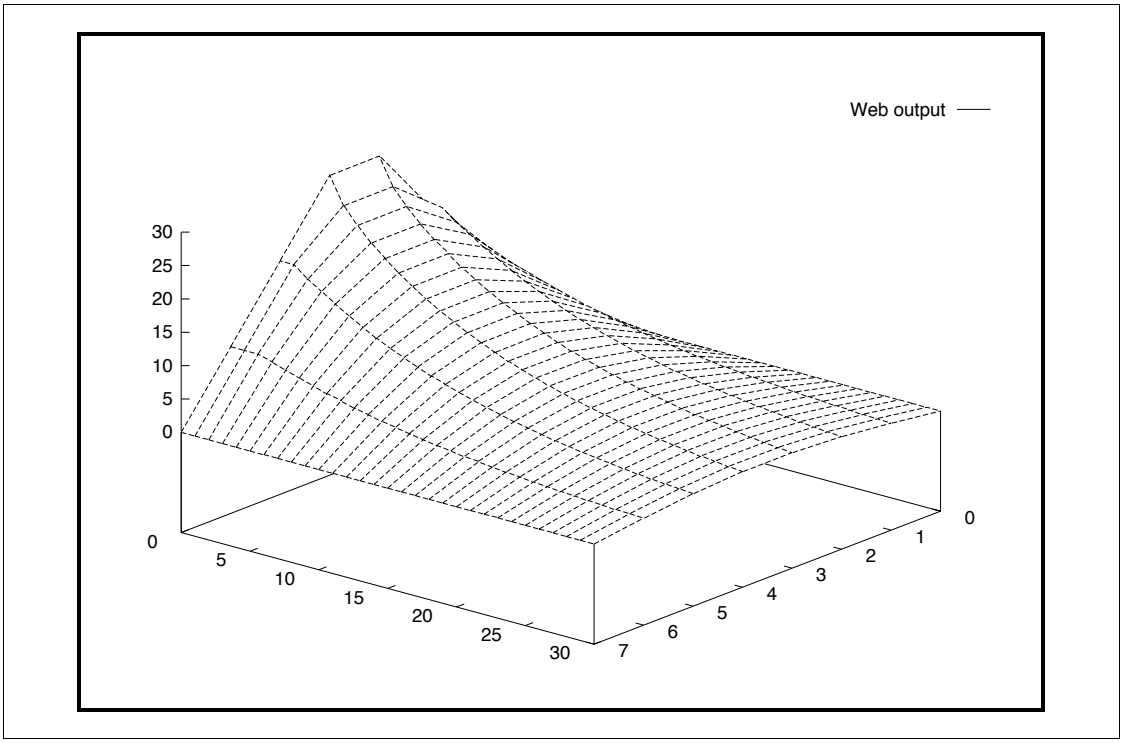


Figure IV.4 : *Résultat graphique de l'évaluation du système d'équations définissant la diffusion de la chaleur dans une barre de métal. Ce résultat est obtenu par la commande* `!plot 80 U` *tapée sous le compilateur interactif* 8,5 *(Cf. annexe), après avoir entré le programme précédent. Cette commande permet de visualiser les valeurs du tissu* `U` *pour les 80 premiers tics.*

IV.2. La notion de système

Dans le chapitre précédent, nous nous sommes volontairement restreints aux collections homogènes dont les éléments sont indexés par des entiers pour représenter la notion de champ. Nous allons à présent lever ces deux restrictions, afin de pouvoir représenter :

- des tissus dont la collection support est à valeur non homogène ;
- des tissus dont les points sont explicitement nommés.

Bien plus qu'une simple extension de convenance, les tissus sont ainsi non seulement des « streams de collections » mais aussi le support de la notion de *système d'équations*.

Un *système* 81/2 est défini comme un tissu dont les points sont explicitement nommés. L'unification des notions de « stream de collections » et de « système d'équations » présente de grands avantages :

- Elle est naturelle : nous verrons que le rôle des équations 81/2 se réduit à la dénotation d'expressions ; si les tissus permettent de dénoter des expressions, alors ils jouent de facto le rôle des équations.
- Elle est uniforme : ainsi, il n'y a point besoin d'introduire de nouvelles entités dans le langage pour représenter un système d'équations. Un programme 81/2 étant un système d'équations, c'est aussi un tissu. Par conséquent, les programmes et les objets créés par un programme sont représentés de manière uniforme.
- Elle ouvre des perspectives intéressantes : les programmes étant de même nature que les données, un tissu présente à la fois une nature numérique et symbolique. Par exemple, nous donnerons une

interprétation des tissus en termes de modules réutilisables, la composition des tissus (opérateur #) jouant le rôle de composition des systèmes d'équations.

Pour unifier à travers le concept de tissu les notions de « stream de collections » et de « système d'équations », nous devrons définir successivement dans cette section :

- la nomination explicite des éléments d'une imbrication,
- les collections non homogènes et leur traitement,
- la notion de système,
- les systèmes complets et incomplets,
- la valeur d'un système incomplet,
- la composition des systèmes.

IV.2.1. Nommer explicitement les éléments d'un tissu

Nous avons vu dans le chapitre I que l'ensemble des variables d'état qui décrivent un système dynamique correspondent à un champ d'un ensemble de noms (les noms des variables) dans un ensemble de valeurs, alors que d'autres champs correspondent à une fonction d'une partie de `I N` dans un ensemble de valeurs.

Les collections du type de celles définies dans le chapitre précédent correspondent à ce dernier cas. Nous appellerons de telles collections, des *collections implicites*, comme raccourci de « collections dont les éléments sont implicitement dans un intervalle de `I N` ».

Nous voulons à présent définir des *collections explicites* i.e. « des collections dont les éléments sont explicitement nommés ». Et plus exactement, nous voulons définir des *tissus systèmes*, c'est-dire des tissus dont la collection support est explicite.

IV.2.1.a. Nommer les éléments d'une imbrication de tissus

Pour nommer un élément `e` dans une imbrication (Cf. §III.3.2.a), il suffit d'utiliser l'équation « `nom = e` » à la place de cet élément dans l'imbrication. Par exemple :

```
A = {  x = 0,
       1,
       y = 1+1}
```

La collection `A` est une collection homogène dont certains éléments, le premier et le troisième, sont explicitement nommés[1]. La valeur du premier élément de `A` est le scalaire `0`, la valeur du deuxième élément de `A` est le scalaire `1`, la valeur du troisième élément de `A` est le scalaire `2`. Cependant, la collection `A` n'est pas équivalente à la collection `B` définie par :

```
B = {0, 1, 2}
```

car une collection est une fonction d'un ensemble de noms dans un ensemble de valeurs, et les ensembles de noms de `A` et de `B` *ne sont pas les mêmes*.

Nommer les éléments d'une collection est une commodité qui permet d'y accéder par un nom plutôt que par un index. Par exemple :

```
A.x
```

[1] Nous devrions écrire en toute rigueur : « Le tissu A est un tissu dont la collection support est homogène et dont certains éléments, le premier et le troisième suivant la collection support, sont des tissus qui sont explicitement nommés ». On pardonnera, afin d'éviter cette lourdeur, l'imprécision du vocabulaire de cette section qui mélange un peu cavalièrement collection et tissu.

permet d'accéder au premier élément de `A`. On peut cependant aussi accéder aux éléments de `A` par leur index[1]. Ainsi,

```
A.x   et   A.0
```

désignent le même élément de `A`, et il en va de même pour

```
A.y   et   A.2
```

mais donner un nom à un élément d'une collection permet de s'affranchir de la connaissance de l'ordre de cet élément dans l'énumération des éléments de l'imbrication[2].

IV.2.1.b. Dénotation d'une expression par une équation 81/2

L'utilisation d'une équation pour nommer les éléments d'un tissu n'est pas due au hasard : une équation de la forme

```
variable = expression
```

est fondamentalement la construction qui permet d'associer une variable à une valeur (la valeur de `expression` qui est un tissu 81/2). Une équation *de cette forme* ne sert à rien d'autre qu'à *donner des noms*, comme nous allons l'illustrer.

Pour illustrer le fait qu'une équation de cette forme ne sert qu'à donner un nom à des sous-tissus d'un tissu, il nous suffit de convertir tout programme 81/2 en une expression sans équations. Cela est possible, et nous allons l'illustrer sur un exemple, mais il faut d'abord introduire la notion de *tissu racine*.

Jusqu'ici, pour définir les programmes 81/2, nous avons simplement donné les équations qui définissaient les tissus du programme. À présent, nous pouvons voir les équations de définition des tissus comme les éléments d'un tissu, appelée tissu racine et qui est « le » programme 81/2. Nous avons uniquement omis jusqu'à présent d'utiliser les virgules entre deux équations. Ces virgules sont nécessaires. Ainsi, quand on écrit le programme 81/2 correct :

```
A = 0,
B = 1
```

cela définit le tissu racine comme étant : « `{A = 0, B = 1}` », les accolades étant implicitement rajoutées par le compilateur. Le tissu racine est donc toujours défini par une imbrication.

Illustrons à présent comment on peut "extirper" toutes les équations d'un programme 81/2. Partons du programme suivant comportant une seule équation :

```
T = {a = 1,  b = a + 2}
```

Nous pouvons traduire l'expression qui définit `T` par :

```
T = {1,  T.0 + 2}
```

---

[1] Le domaine de définition d'une collection conçue comme une fonction d'un ensemble de noms vers un ensemble de valeurs, est donc un ensemble de la forme `D∪E`, où `D` est l'ensemble `{0, …, n-1}` des index et `E` est un ensemble d'identificateurs alphabétiques, éventuellement vide (si aucun élément de la collection n'est explicitement nommé). L'association d'une image à un nom se fait explicitement à travers le signe `=` et l'association d'une image à un index se fait implicitement par le rang de l'image dans la donnée des images de la collection.

[2] Le même nom peut être utilisé plusieurs fois dans une collection : `C = {x = 0, x = 1}`. L'accès via un nom réfère toujours la dernière définition attachée à un nom : `C.x` a pour valeur le tissu `1`. En cas de liaisons multiples, on voit que l'ordre de présentation des définitions a une importance.Pour récupérer les autres définitions attachées à `x`, on peut utiliser un index, ou bien la notation `A.<n>x` où `n` indique que l'on s'intéresse à la $n^{ème}$ liaison de `x`. (Cf. le manuel de référence)

ce qui permet de se débarrasser des noms « `a` » et « `b` » mais nous n'avons pas encore éliminé l'équation portant sur `T`. Si nous voulons extirper toute trace de nomination explicite, il faut pouvoir référer implicitement `T`. `T` étant un élément du tissu racine, il est possible d'y référer par son index. Pour éviter une régression infinie, nous ne donnerons pas de nom au tissu racine. Cependant, lors de l'énumération des éléments du tissu racine, il est possible de référer à l'élément d'index `n` par l'expression « `.n` », notation qui est en accord avec le fait que le tissu racine n'a pas de nom. Ainsi, le programme consistant en la seule définition de `T`, peut se traduire par un programme équivalent[1] sans aucun nom, consistant en la seule expression

```
{1, .0.0 + 2}
```

L'expression « `.0.0` » dénote la première expression de la première expression du tissu racine. On peut généraliser cet exemple, afin de prendre en compte les équations quantifiées, par exemple en utilisant un opérateur permettant de séquencer les streams grâce à des prédicats temporels, à la manière de l'opérateur `fby` vu au §II.4.2.b.

On peut ainsi transformer tout programme et extirper les équations ; cette transformation constitue d'ailleurs la première phase de la compilation. Cela nous montre que les équations en 81/2 ne sont qu'un moyen commode pour dénoter des expressions.

IV.2.2. Les collections hétérogènes en 81/2

Nous voulons représenter des systèmes d'équations par des tissus 81/2. Nous venons de voir que l'aspect "équation" est pris en charge par la nommination des éléments d'une imbrication. Mais cela ne suffit pas pour représenter des systèmes d'équations par des tissus 81/2, car un système d'équations définit des tissus qui en général n'ont pas le même type (i.e. dont la collection support est d'un type différent). Nous avons donc besoin du concept d'imbrication non homogène (Cf. §III.2.1).

Une collection non homogène, ou encore *hétérogène*, est une collection dont les éléments non pas tous la même géométrie. Par exemple :

```
A = {1, {2, 3}}
```

est une collection non homogène : le premier élément de `A` a une géométrie de `[1]` alors que le deuxième élément de `A` a pour géométrie `[2]`. Autre exemple, la collection

```
B = {{true, false}, {2.718, 3.141}}
```

n'est pas une collection homogène car le type scalaire du premier élément (`bool`) est différent du type scalaire du deuxième élément (`float`).

IV.2.2.a. Géométrie d'une collection hétérogène

Nous avons vu dans le chapitre précédent que la géométrie peut être indiquée optionnellement lors de la définition d'une collection. Il n'est plus possible d'utiliser une notation de type « liste des cardinaux » pour indiquer la géométrie d'une collection non-homogène. On emploie à la place le mot-clef `system` avant l'identificateur de l'équation (le choix du vocable « system » pour qualifier une géométrie hétérogène sera justifié par la section suivante) :

---

[1] Le nouveau programme est équivalent à l'ancien du point de vue de l'observation de l'exécution du programme. En effet, nous avons donné plus haut l'exemple de deux tissus qui définissaient des collections dont les valeurs étaient identiques, mais qui n'étaient pas définies sur le même domaine. La collection support du nouveau programme n'est pas la même que celle de l'ancien programme, puisqu'il n'y a plus aucun nom de défini. Cependant, les noms ne sont pas observables de l'extérieur du programme. Lors de l'exécution, les seules données auxquels on a accès, sont la suite ordonnée dans le temps des valeurs des collections valeurs du stream support du tissu racine.

```
system A = {1, {2, 3}}
```

indique que la collection **A** est une collection dont la géométrie n'est pas homogène, mais cette notation ne permet pas de préciser la géométrie de chaque élément de **A**.

Nous n'avons pas besoin d'une notation permettant d'indiquer la géométrie de chaque élément d'une collection hétérogène. En effet, le compilateur est toujours capable d'inférer sans ambiguïté la géométrie des collections hétérogènes. Par contre, le compilateur ne peut pas toujours inférer sans ambiguïté si une collection est homogène ou hétérogène, ce qui rend indispensable l'emploi du mot-clef **system** pour forcer éventuellement le compilateur à choisir un type donné. L'ambiguïté provient des coercions implicites entre types scalaires. Un exemple typique de cette situation est illustré par la définition

```
A = {1, 2.0}
```

doit-elle être interprétée comme une collection à deux éléments scalaires flottants (la constante entière **1** étant convertie en un flottant), ou bien, cette définition doit-elle être vue comme la définition d'un système hétérogène contenant un entier et un flottant ? L'ambiguïté vient du fait que le compilateur est capable d'effectuer automatiquement des conversions entre scalaires (Cf. le manuel de référence).

Dans le premier cas, le compilateur doit utiliser implicitement une coercion, ce qui n'est pas le cas de la seconde interprétation. Cependant, le compilateur 81/2 ne choisit pas l'interprétation conduisant au minimum de conversions : si c'est possible, le compilateur effectue les conversions nécessaires pour construire des collections homogènes. Dans l'exemple précédent, et en l'absence d'indication supplémentaire, le compilateur infère que le tissu **A** est une collection de deux flottants. Dans tous les cas, la spécification explicite de la géométrie forcera le compilateur à construire le tissu désiré :

```
system A = {1,  2.0}
```

construira un système contenant un nombre entier et un nombre flottant, l'équation

```
float A[2] = {1,  2.0}
```

construira une collection homogène de deux nombres flottants, de même d'ailleurs que « **float A = {1, 2.0}** », alors que

```
int A = {1,  2.0}
```

construit une collection homogène de deux entiers. Par ailleurs, le compilateur ne s'oppose pas à ce que le programmeur déclare explicitement une géométrie hétérogène pour une collection qui aurait pu être homogène :

```
system B = {true, false}
```

est une équation valide qui définit une collection hétérogène, bien qu'elle eusse pu être considérée comme homogène.

IV.2.2.b. Opérations sur les collections hétérogènes

Les éléments d'une collection hétérogène peuvent avoir des types scalaires différents. Il n'est donc pas possible de définir par exemple l'addition de deux collections hétérogènes. On ne peut pas appliquer l'opérateur cardinal car leur géométrie n'est pas décrite par une collection d'entiers, etc.

Les opérations permises sur les collections hétérogènes sont donc réduites. Actuellement, trois opérations seulement sont permises[1] :

[1] Dans les extensions de 81/2 nous travaillons à étendre les opérations permises sur les collections hétérogènes.

– l'accès à un élément de la collection grâce à l'opérateur de projection (le point) ;
– l'imbrication des collections grâce aux accolades ;
– la composition.

La composition des collections hétérogènes correspond à la concaténation des éléments des arguments, comme pour les collections homogènes.

IV.2.3. La notion de système 8½

Un *système* 8½ est un tissu dont la collection support est hétérogène.

Puisqu'on peut nommer explicitement les éléments d'une collection, un système 8½ correspond donc à un système d'équations. Un système 8½ est donc une unité d'encapsulation. Cette unité d'encapsulation peut être utilisée pour structurer une modélisation complexe, par exemple en associant un système 8½ à chaque sous-système identifié dans le modèle. On peut aussi utiliser les systèmes 8½ pour limiter la portée des variables dans un grand programme. On peut encore utiliser l'encapsulation pour développer un style de programmation "objet" (voir l'exemple traité au §IV.3.7).

IV.2.3.a. Un exemple d'utilisation de système

L'exemple suivant utilise les tissus systèmes pour structurer la simulation du comportement d'un animal informatique [MEY 90] appelé `wlumf`. Les systèmes sont utilisés ici pour encapsuler les définitions qui se rapportent à chaque entité de la simulation.

Dans ce petit exemple, il y a deux sous-systèmes : l'`environnement` qui présente de la nourriture au `wlumf` de manière périodique, et le `wlumf` lui-même. Le `wlumf` a faim quand son taux de `glucose` descend en dessous d'un certain seuil. Le taux de `glucose` remonte quand le `wlumf` mange. Le `wlumf` mange uniquement quand il a faim et à la condition que l'`environnement` lui présente de la nourriture. Un `wlumf` est donc un *système réactif* qui répond aux sollicitations de l'environnement (`nourriture`) en fonction d'une condition interne (`faim`).

Cela se traduit par le programme 8½ suivant :

```
frequence_nourriture = Clock -4,
glucose_initialement = 6,
max_glucose          = 10,
niveau_de_faim       = 4,
metabolisme          = Clock,

system environnement = {
     temps@0 = 0,
     temps   = $temps + 1 when frequence_nourriture,
     nourriture = (temps % 4) == 0,
},

system wlumf = {
     glucose@0 = glucose_initialement,
     glucose  = if mange
                  then max_glucose
                   else max(0, $glucose-1 when metabolisme)
                fi,
```

```
        faim = $glucose < niveau_de_faim,
        mange = faim & environnement.nourriture
    }
```

Ce programme demande les deux explications suivantes :

- Le tissu `nourriture` est un tissu booléen qui est `vrai` chaque fois que le compteur `temps` est un multiple de 4 (l'opérateur d'égalité se note `==` et l'opération de modulo se note `%`). Ce compteur est incrémenté à une fréquence de un top en moyenne tous les `fréquence_nourriture` tops de l'horloge de `Clock`.
- La fonction `max` retourne le maximum des deux arguments : le taux de glucose ne peut pas être négatif. Quand le `wlumf` vient de manger, le taux de glucose remonte à `max_glucose`, sinon, le taux de glucose décroît avec le temps, au rythme de `metabolisme`.

Une exécution du programme est illustrée par la figure IV.5.

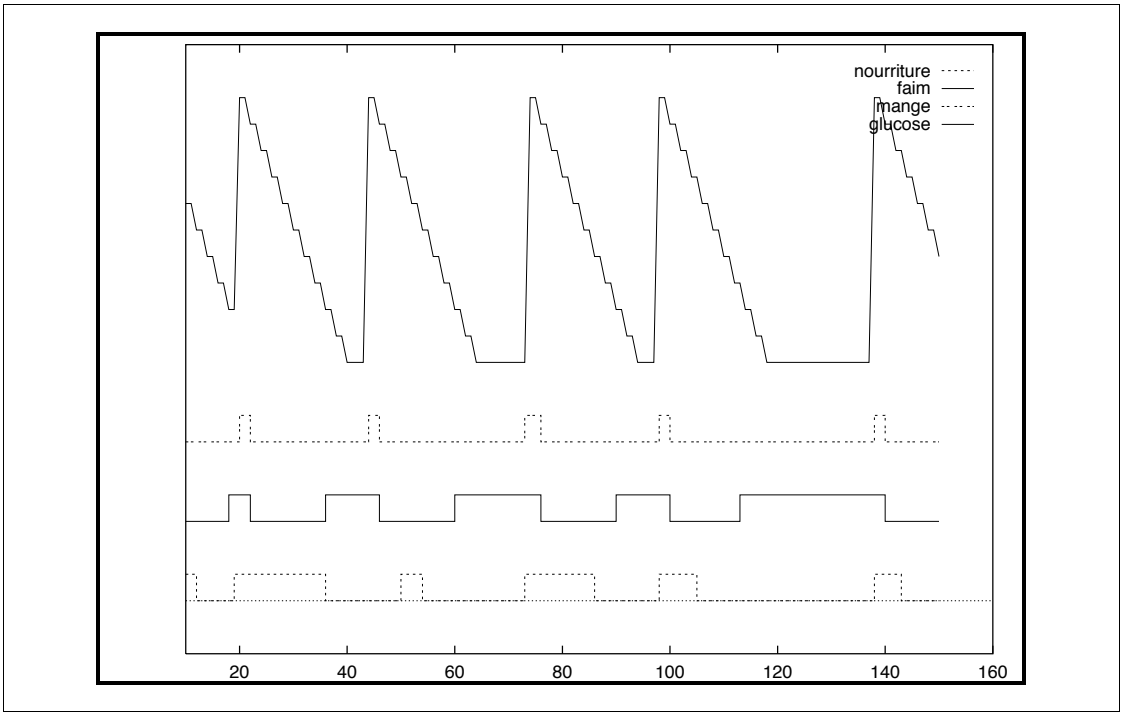


Figure IV.5 : *Résultat graphique de l'évaluation du système d'équations définissant le comportement d'un* `wlumf`.

IV.2.3.b. La hiérarchie des systèmes

On peut imbriquer les tissus systèmes comme on imbrique les collections. On obtient ainsi une hiérarchie de systèmes. Chaque équation d'un programme 81/2 appartient ainsi à un système. Les équations que l'on écrit "au top-level" et qui n'ont pas l'air d'apparaître entre des accolades englobantes, appartiennent au tissu racine. Un programme 81/2 est constitué de ce système racine et le programmeur définit les points de ce système.

L'identificateur `nourriture` n'est pas directement visible en dehors de `environnement` (comme les labels des champs d'une structure `C` ou PASCAL). Par ailleurs, la définition de `mange` utilise le tissu `environnement` qui n'est pas défini dans le système `wlumf`. Le tissu correspondant sera cherché dans le système englobant : les règles de visibilité de 81/2 sont identiques à celles des blocs `C`.

IV.2.3.c. Opérations sur les systèmes

Que deviennent les systèmes quand on les combine dans des expressions ?

Il n'est pas possible de définir dans la version 1.0 de 81/2 des systèmes dont la géométrie est considérée comme homogène par le compilateur. Trois opérations seulement sont donc disponibles pour combiner des systèmes : la projection, qui permet d'accéder à un élément d'un système par un nom, l'imbrication qui permet de hiérarchiser les systèmes, et la composition.

Nous savons composer des collections dont la géométrie est homogène : la composition correspond à la concaténation. Nous pouvons étendre cette définition sans difficulté aux collections hétérogènes implicites. Mais pour décrire l'effet d'une composition sur un système, il nous faut préciser ce que devient le nommage lors d'une composition de systèmes. Pour cela, nous devons introduire un peu de vocabulaire.

IV.2.4. La composition des systèmes 81/2

IV.2.4.a. Variable libre et variable liée, système incomplet et système complet

Le nommage des points d'un système définit un *environnement* : un environnement est un ensemble de liaisons « variable ↔ valeur ».

Une variable *liée dans un système* (ou bien *par* un système) est une variable qui apparaît en membre gauche d'une équation *du système*. La notion de liaison est donc relative à un système. On définit de même la notion de variable libre dans un système : elle n'apparaît pas en partie gauche d'une des équations du système. Un système *autonome* est un système où toutes les occurrences de variables correspondent à des variables liées par ce système. Un système *complet* est un système où toutes les occurrences de variables correspondent à des variables liées soit dans le système, soit dans un des systèmes englobant.

Une variable libre dans un système, peut être liée si on considère le système englobant. Une variable libre dans un système réfère au tissu de même nom défini dans l'environnement englobant "le plus proche". Dans l'exemple suivant, nous distinguons par un indice les occurrences du nom `c` afin de pouvoir les désigner dans notre explication :

```
S = {
        c1 = 1,
        a1 = 2,
        A  = { a2 = c2,  b = 3 },
        B  = { c3 = d,  d = c4 + a3 },
        C  = A#B,
    }
```

Si on regarde le système `A`, la variable c_2 est libre. Si on regarde le système englobant `S`, la variable c_2 est liée. Dans cet exemple, l'occurrence c_2 de la variable `c` désigne le tissu c_1 et l'occurrence c_4 réfère le tissu c_3. Le système `A` est donc complet mais il n'est pas autonome. La variable a_3 réfère à la définition a_1, le système `S` est donc autonome.

IV.2.4.b. Valeur d'un système incomplet

Pour calculer le résultat d'un système d'équations, il faut que chaque variable du système ait une définition : un programme doit donc être autonome. Mais dans l'exemple du `wlumf` donné plus haut, le `wlumf` n'est pas autonome (le tissu `environnement` n'est pas lié dans le système `wlumf`). Par contre, c'est un système complet. Peut-on considérer le système `wlumf` en lui-même comme un programme 81/2

valide ? Et plus généralement, peut-on considérer les systèmes incomplets comme des programmes 81/2 valide ?

- *Si la réponse est non*, alors nous devons distinguer deux types de systèmes : les systèmes qui peuvent constituer des programmes valides et ceux qui ne le peuvent pas. Un programme valide est alors clairement un système autonome. Or si on veut développer des "sous-systèmes réutilisables", ces systèmes seront incomplets, et donc non-autonomes.
- *Si la réponse est oui*, alors nous devons définir la valeur d'un système incomplet. Dans ce cas, tout système 81/2 est un programme 81/2 valide, on peut donc définir simplement la composition des systèmes, etc.

Nous choisissons la deuxième solution.

On peut voir un système qui possède des variables libres comme une boîte possédant des entrées nommées : précisément les variables libres du système. Le système attend alors les valeurs de ces variables, sur ces entrées, afin de calculer les tissus dont la définition dépend de ces variables (Cf. Figure IV.6). Prenons par exemple le système incomplet :

```
f = {
          cpt@0 = 0,
          cpt = $cpt + 1 when Clock,
          a = b + cpt
    }
```

Ce système attend un tissu `b` afin de produire un tissu `a`. On peut reformuler cette interprétation en adoptant le point de vue : « tissu = stream de collections ». Avec ce point de vue, le système attend sur chaque top de `b` les valeurs de `b` afin de produire les valeurs de `a` pour le top en question.

Or, un objet qui, si on lui passe une valeur, calcule une autre valeur, correspond à une *fonction*. Nous devons donc voir les *tissus incomplets* comme des « *streams de collections de fonctions (sur les scalaires)* ».

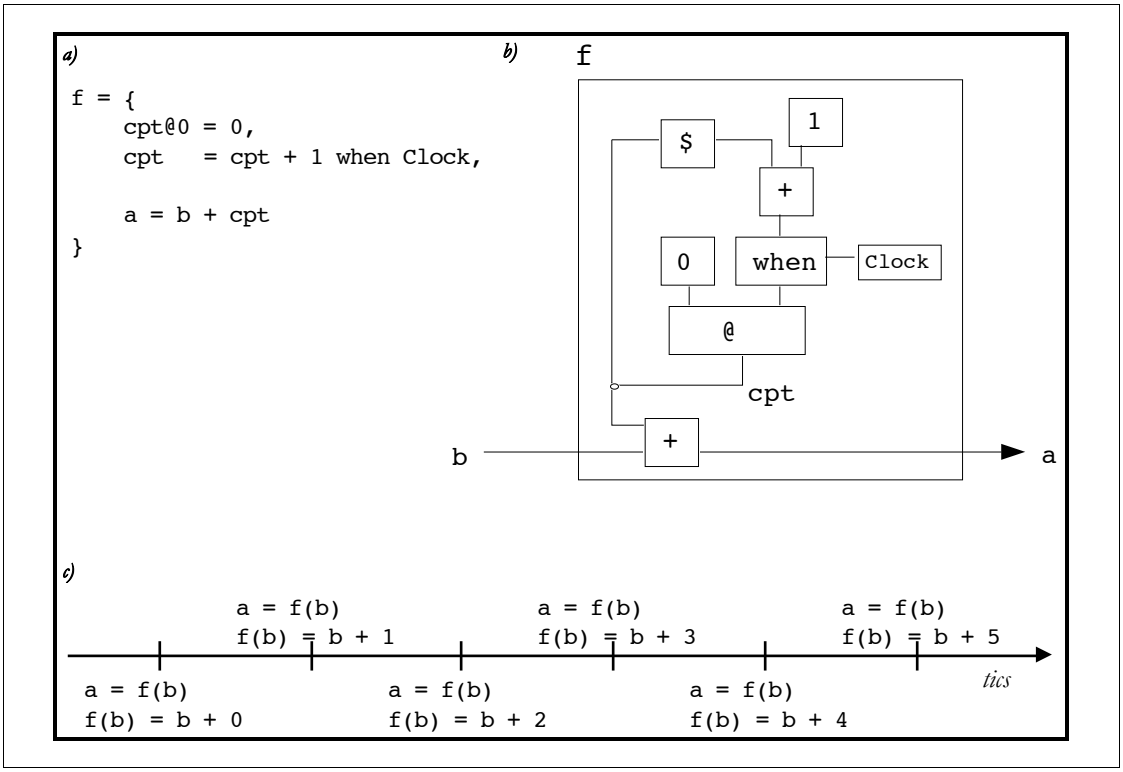


Figure IV.6 : *Interprétation d'un tissu incomplet en terme de stream de collections de fonctions.*
a) *donne la définition 81/2 du système incomplet* `f`
b) *représente ce système d'équations en termes de boîtes attendant des entrées pour produire des sorties. Une boîte correspond à un opérateur élémentaire (par exemple* `+`*) ou à un système (cas de la boîte* `f`*). Regardons maintenant les tissus comme des streams de collections : sur les arcs de cette représentation transitent les valeurs des streams ; la variable* `b` *étant libre, une valeur de* `a` *constitue une fonction qui attend une valeur de* `b` *pour produire une valeur scalaire. La suite des fonctions qui sont produites en* `a` *au cours du temps est illustrée sur le schéma c)*
c) *figure la suite des valeurs de if. Au premier tic, il s'agit de l'identité, au deuxième tic, c'est la fonction qui incrémente* `b` *de 1, au troisième tic c'est la fonction qui incrémente* `b` *de 2, etc.*

IV.2.4.c. Liaison des variables dans une composition

Puisqu'un système incomplet correspond à un tissu de fonctions (i.e. un stream de collections de fonctions), il serait agréable de pouvoir compléter ce système, ce qui correspondrait à l'application du tissu de fonctions à un tissu de valeurs.

Or nous avons déjà un moyen pour compléter les tissus incomplets : il s'agit de l'imbrication. En effet, si `A` est un système incomplet, notons `A'` l'ensemble des équations de `A`. Il est possible de construire un système `S` complet

```
S = { A', B' }
```

où `B'` correspond à l'ensemble des équations nécessaires pour compléter `A`. Appelons `B` le système formé de ces équations, alors nous savons que (Cf. §III.3.2.d) que

```
S = A # B
```

L'opération de composition des tissus correspond donc à la concaténation des valeurs *et* à l'*union des environnements*.

Regardons précisément ce qui se passe sur un exemple. Examinons dans le système **S** ci-dessous, la composition de **A** et de **B** qui définit le tissu **C**

```
S = {
        c  = 1,
        A  = { a1 = c1,  b = 3 },
        B  = { c2 = d1,  d2 = c3 + a2 },
        C  = A#B,
    }
```

La composition doit contenir l'ensemble des valeurs de **A** et de **B** (Cf. §III.3.2.c). Même en faisant abstraction de l'ordre des éléments, la composition doit contenir les équations qui proviennent de **A** et de **B**, ce qui correspond au système

```
A#B = {   a1 = c1,
          b = 3,
          c2 = d1,
          d2 = c3 + a2  }
```

Au niveau des liaisons entre les variables et les tissus définis, il se passe quelque chose dans cet exemple pour les variables libres de **A** et de **B** : elles ne sont plus libres dans **A#B**. Par exemple, les règles de liaison nous disent que c_1 et c_3 réfèrent à la définition c_2 et que d_1 réfère à l'équation de d_2. Le mécanisme de liaison[1] est illustré à la figure IV.7.

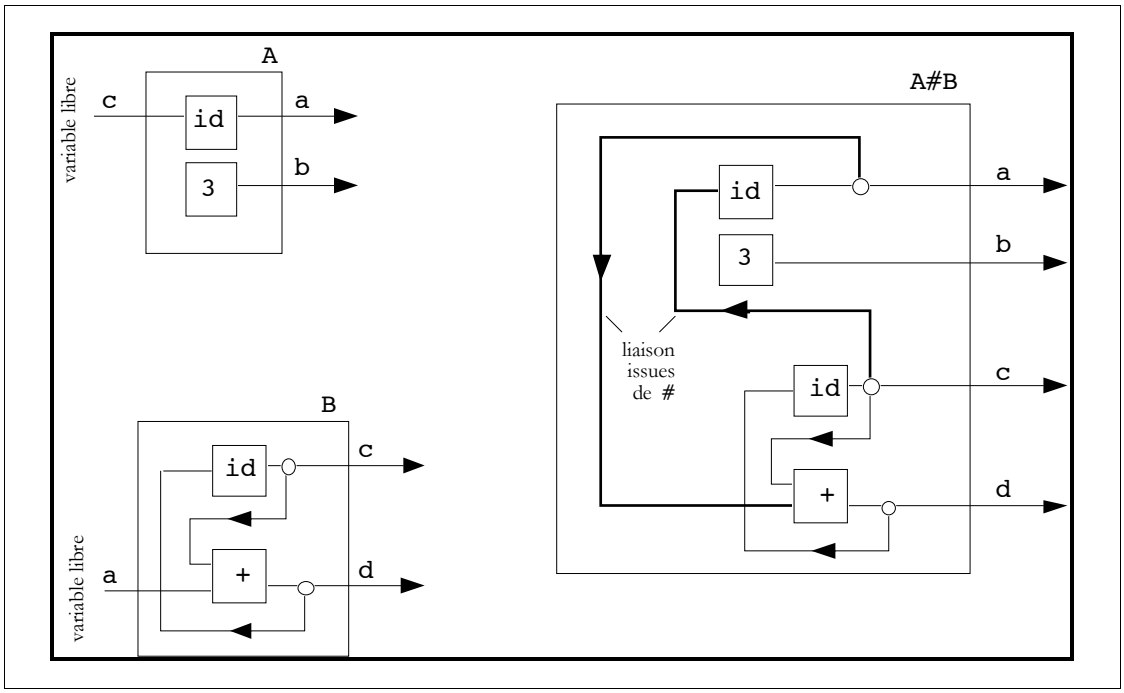


Figure IV.7 : *Mécanisme de création de liaisons lors de la composition de deux systèmes. La figure illustre pour les définitions de* **A** *et* **B** *précédentes, ce mécanisme de liaison en termes de graphe data-flow hiérarchique. Le système englobant* **S** *n'a pas été représenté.*

[1] Plus précisément, le mécanisme de liaison est *lexical* et obéit aux règles données aux §IV.2.2.a et §IV.2.3.b.

Nous venons de voir que la composition permet de lier des variables libres. Un système incomplet peut donc être vu comme une fonction (des variables libres) et la composition comme l'application d'une fonction. Le résultat de l'application d'une telle fonction est soit un système complet, soit un système qui possède encore des variables libres.

On peut ré-utiliser un système autant de fois que l'on veut dans une composition en le référant à l'aide de son nom. On peut donc voir un système incomplet comme un élément paramétrable et réutilisable, les paramètres étant les variables libres du système. La composition permet de fournir ces paramètres et "d'instantier" le système. On peut encore voir un système comme une classe, la concaténation jouant alors le rôle de constructeur. Nous verrons plus loin qu'elle permet aussi d'émuler la notion d'héritage.

La composition des environnement se fonde sur les noms des variables, et donc, en termes d'application des fonctions, nous nous sommes affranchis de l'ordre des arguments. En termes de combinateur, la concaténation unifie les différentes opérations de composition traditionnellement utilisées dans les langages fonctionnels comme FP : mise en série, mise en parallèle, feed-back, Cf. par exemple [DEL 86].

IV.2.5. Fonction et système

Les systèmes incomplets jouant le rôle de fonctions sur les tissus, nous n'avons pas réellement besoin de la construction `function` du §III.2.3.c. En effet, la déclaration de la fonction `f`

```
function f(arg1, arg2, …, argn) = expression
```

peut se traduire par un système `F` incomplet :

```
F = { expression }
```

L'application de la fonction `f`

```
f(a1, a2, …, an)
```

se traduit alors par la composition des systèmes :

```
(F # {arg1 = a1, arg2 = a2, …, argn = an}).0
```

La construction `function` peut être vue comme une simple abréviation commode pour construire et compléter des systèmes, la traduction se faisant de la manière donnée ci-dessus (cela n'est cependant pas implémenté de cette manière dans le langage).

Les concepts de fonction et de système se complètent en 81/2 pour permettre la description simple de « module réutilisable ». Par exemple, il est courant de vouloir décrire un compteur qui compte les tops d'un tissu `X` donné. Cette fonction correspond à la définition suivante :

```
function  compteur(X) =
          {
               cpt@0 = 0 synchro X,
               cpt = $cpt + 1 synchro X
          }.cpt
```

L'argument `X` correspond au tissu dont on veut compter les tops. Le corps de la fonction `compteur` est une expression de la forme « ***système*.cpt** ». La valeur de cette expression est celle de `cpt`, le tissu `cpt` étant défini dans l'expression ***système***. Cette expression est une imbrication dont la valeur est un système mais qui n'a pas reçu de nom particulier. À chaque application de `compteur`, un système est créé, qui encapsule une définition de `cpt`. Ces différentes définitions ne "rentrent pas en collision" et ne sont d'ailleurs même pas visibles ou accessibles dans le programme, car elles sont encapsulées chacune dans un système *anonyme*. On peut aussi définir une fonction qui fait la somme des valeurs successives d'un tissu :

```
function somme(X) = { sigma@0 = X, sigma = $sigma + X }.sigma
```

et utiliser la fonction `somme` et `compteur` pour définir la fonction `moyenne` d'un stream :

```
function moyenne(X) = somme(X) / (1 + compteur(X))
```

IV.3. Exemples d'applications 81/2

Dans la dernière partie de ce chapitre, nous donnons des exemples de programmes 81/2 empruntés à divers domaines d'applications :

- les systèmes dynamiques discrets,
- les systèmes dynamiques continus,
- les réseaux d'automates,
- le traitement d'image,
- les transformations de temps en espace,
- la simulation par événements discrets,
- la programmation objet en 81/2,
- la programmation fonctionnelle en 81/2,
- la traduction en 81/2 des structures de contrôle répétitives et impératives
- les collections à géométrie dynamique.

Le lecteur trouvera d'autres exemples et une description complète de l'environnement de programmation du langage version 1.0, dans le manuel de référence. Une description succincte de l'environnement, nommé *8,5*, est donnée en annexe.

IV.3.1. Systèmes dynamiques discrets : itérés de l'équation logistique

Un problème classique dans le domaine des systèmes dynamiques discrets consiste à modéliser la dynamique d'une population en tenant compte des ressources qui limitent sa croissance. Soit `N(t)` le nombre d'individus de la population au temps `t`. Dans un premier temps, on suppose que la croissance de la population ne dépend que du nombre d'individus présents, i.e. :

```
N(t+1) = F(N(t))
```

Cette hypothèse est vraie si deux générations d'individus n'interagissent pas, dans le cas contraire il faudrait par exemple utiliser une fonction de la forme `G(N(t), N(t-1))`. La fonction `F` prend la forme suivante :

```
F(x) = ax +  des termes non-linéaires
```

La partie linéaire de `F` indique que la croissance de la population est proportionnelle à la population présente. Cette partie linéaire correspond au modèle malthusien[1] et induit une croissance exponentielle de la population si `a>1`. Les termes non-linéaires doivent corriger ce modèle pour prendre en compte le fait que, quand les ressources sont limitées, il y a compétition entre les individus pour se reproduire. La compétition est proportionnelle au nombre de rencontres possibles entre les individus, i.e. elle est proportionnelle à $N^2(t)$. Le modèle devient donc :

[1] Thomas Robert Malthus, pasteur et économiste anglais (1766-1834) compara la croissance exponentielle des populations à la croissance linéaire des ressources terrestres et fut partisan, en conséquence, de la limitation des naissances.

$$N(t+1) = aN(t) - bN^2(t)$$

avec `a` et `b` positifs, `a` représentant le taux de croissance et `b` un paramètre dépendant des ressources et qui modélise l'environnement de la population. La quantité `a/b` est appelée « capacité de charge » de l'environnement (carying capacity en anglais). Pour illustrer ces comportements, on va d'abord simplifier l'équation en exprimant la taille de la population en unités de capacité de charge : $y(t) = \frac{b}{a} N(t)$. Le système devient :

$$y(t+1) = k\,y(t)(1 - y(t)) = f(y(t))$$

avec `k=a`. L'équation précédente s'appelle *l'équation logistique*.

Nous aimerions calculer les résultats des premières itérations `f`, f^2, f^3, …, pour des valeurs initiales prises dans tout l'intervalle `[0, 1]` (la population est normalisée par exemple en milliers d'habitants) et pour toutes les valeurs du paramètre `k` supérieures à `1.2`.

Un problème de représentation se pose en 81/2 : il faut décider quelle variation se fera dans le temps, et quelles autres variations se feront dans l'espace. On décide par exemple de faire varier `k` dans le temps (car on ne l'a pas borné a priori), et de calculer à chaque instant pour les fonctions `f`, f^2, f^3, f^4, …, toutes les images de l'intervalle discrétisé `[0, 1]`. Afin de les représenter sur un même graphique, on va concaténer ces différentes images les unes à la suite des autres (ce qui correspond, comme on le verra, à utiliser la récursion spatiale pour exprimer l'itération des fonctions `f`).

On appelle `start` le vecteur des points de discrétisation de `[0, 1]` avec un pas de `0.05` : `start` est de la forme `{0, 0.05, 0.1, 0.15, …, 0.95}`. Le coefficient `k` est un stream qui évolue par pas de `0.1` au cours du temps. On appelle `iter1` la première itération de `f`, `iter2` la deuxième itération, etc. Le tissu `separe` est un vecteur constant de `0` qui a pour but de séparer les différentes itérations, afin d'obtenir une jolie représentation graphique en intercalant un espace entre deux itérations consécutives.

```
start = 0.05 * '20,

k@0 = 1.2,
k   = $k + 0.1 when Clock,

iter0 = start,
iter1 = k*iter0*(1-iter0),
iter2 = k*iter1*(1-iter1),
…
iter5 = k*iter4*(1-iter4),

separe[5] = 0;

map0 = iter0 # separe # iter1 # separe # iter2 # … # iter5
```

La définition de `map0` est fastidieuse : on doit écrire explicitement à la main le calcul des différentes itérations `iter1`, `iter2`, etc. Il serait plus simple d'écrire une définition récursive spatiale de `map0`. En fait, la seule différence entre ce que l'on veut faire ici, et les expressions récursives que nous avons déjà vues (par exemple factorielle), c'est que l'élément de départ `start` n'est pas un scalaire mais déjà un vecteur. Cela ne change rien au principe, et donc, nous remplaçons dans le programme précédent la définition de `map0` par la définition de `map1` :

```
map1 = (start # k*map1*(1 - map1)):[100]
```

Le tissu `map1` compte 100 éléments. Puisque le tissu `start` de départ compte 20 éléments, on a donc calculé dans `map1`, 100/20 = 5 itérations. Dans cette version de `map1`, il n'y a pas de séparation entre les itérations. On peut très bien écrire une autre version :

```
map2 = ((start # separe) # k*map2*(1 - map2)):[125]
```

mais les itérations seront séparées par `separe`, `f(separe)`, `f²(separe)`, etc. Or tous les éléments de `separe` sont à `0` et donc `k*separe(1-separe) = 0 = separe`. Par suite, le tissu `map2` est exactement comme le tissu `map0`. Au passage, remarquons que nous tronquons `map2` à 125 éléments, ce qui correspond à 125/(20+5) = 5 itérations (le tissu de départ est constitué de `start` concaténé à `sépare`).

Le dessin de la figure IV.8 est constitué du tracé des trois dernières itérations de `map2`. C'est donc un tissu de 25*3 éléments qu'il faut sélectionner à la fin de `map2`. On utilise donc une troncature négative (et on prend 80 éléments afin de garder une jolie bordure).

```
m = map2:[-80]
```

L'affichage est obtenu par la commande : « `!plot 30 m` » de l'environnement *8,5*.

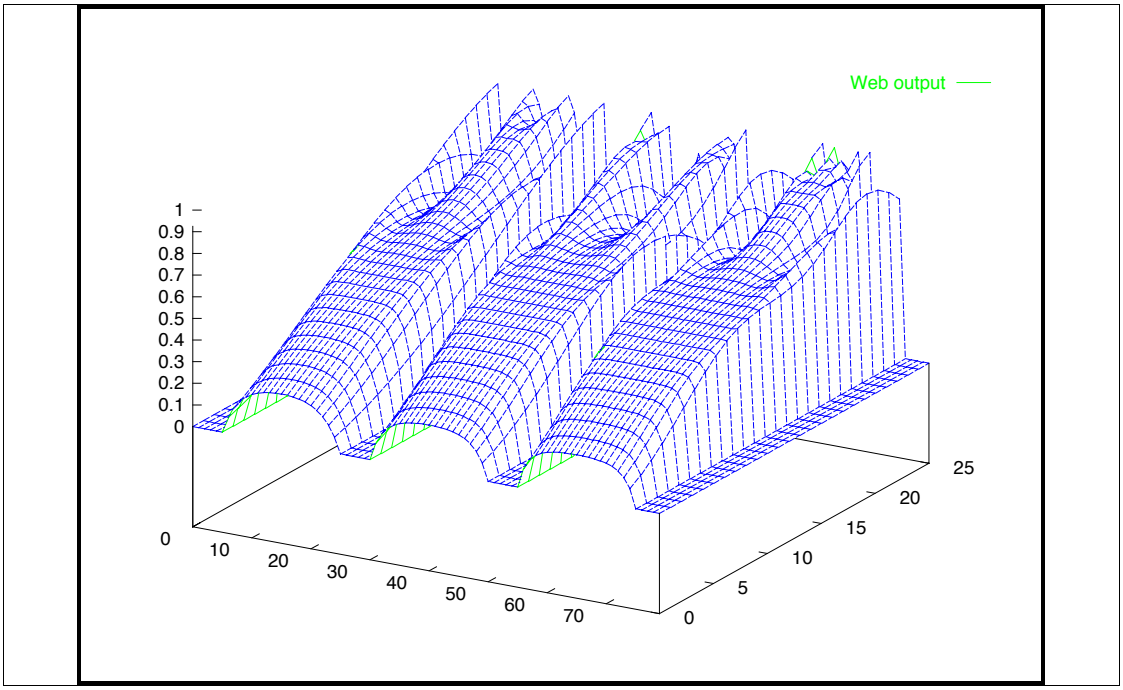


Figure IV.8 : *Représentation graphique du tissu* `m` *dont la définition est donnée plus haut. Le paramètre* `k` varie suivant l'axe "en profondeur". *L'axe vertical correspond à la valeur des points du tissu. L'aspect collection correspond à l'axe horizontal dans le plan de la figure.*

IV.3.2. Système dynamique continu : équation de diffusion en 2D et équation des ondes

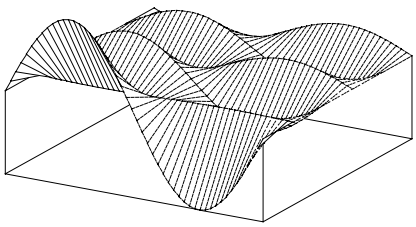
Les systèmes dynamiques continus correspondent aux équations différentielles et aux équations aux dérivées partielles. Nous avons déjà vu la discrétisation d'une équation de diffusion dans une barre de métal (équation parabolique aux dérivées partielles). L'exemple ci-dessous est un exemple similaire mais pour une équation différentielle du second-ordre, du type de celles modélisant la propagation des ondes. L'équation fait intervenir une dérivée seconde, d'où un stream retardé deux fois (`$p_v`, où `p_v` correspond déjà au reatrd de `v`). Deux retard étant utilisés, il faut fixer les deux premières valeurs de `v`.

```
pi = 3.1415926535,
start =  sin('50 * (2*pi/49)),

V = 0 # inside # 0,
V@0 = start,
V@1 = (0 # (2*start(middle)
              - start(left)
              - start(right)) # 0)
        when Clock,

float inside = -0.50 * p_v(left)
              - 0.50 * p_v(right)
              + 0.55 * p_v(middle)
              + 0.45 * $ p_v(middle),

p_v = $V when Clock,
left = '48, middle = '48+1, right = '48+2
```



IV.3.3. Les réseaux d'automates

Nous donnons deux exemples de réseaux d'automates pris dans deux grandes familles : les automates cellulaires et les réseaux systoliques.

IV.3.3.a. Un automate cellulaire : le jeu de la vie

Le monde du jeu de la vie est constitué d'un tableau de cellules organisé en grille. Chaque cellule a huit voisins possibles et est soit dans l'état vivant soit dans l'état mort. L'état des cellules évolue au cours du temps en suivant la loi : une cellule vivante reste vivante à la génération suivante si le nombre de ses cellules voisines à la génération courante est compris entre `2` et `4` ; elle meurt d'isolement ou de surpopulation si le nombre de ses voisins vivants est inférieur à `2` ou supérieur à `4` ; une cellule morte devient vivante si le nombre de ses voisines vivantes est compris entre `2` et `4`.

Le programme 81/2 utilise une collection de booléens afin de représenter si une cellule est vivante (valeur `true`) ou morte (valeur `false`). Les équations correspondantes sont les suivantes :

```
system Automate_Cellulaire =
{
   cellules@0 = {…}, /* un état initial des cellules */
   cellules[100,100]  = (nbre_voisins ≥ 2) && (nbre_voisins ≤ 4),

   /* précédent représente cellules à la génération précédente */
   précédent = $cellules when Clock,

   nbre_voisins = voisins_nord + voisins_sud
                  + voisins_est + voisins_ouest,

   voisins_nord  = if Nord(précédent,false)  then 1 else 0 fi,
   voisins_sud   = if Sud(précédent,false)   then 1 else 0 fi,
   voisins_ouest = if Ouest(précédent,false) then 1 else 0 fi,
   voisins_est   = if Est(précédent,false)   then 1 else 0 fi,

   function Nord(x, c)  = (c#x) .('x),
   function Sud(x, c)   = (x#c) .('x + {{{1, 0}}}:|'x|),
   function Ouest(x, c) = (c#^x).('x),
   function Est(x, c)   = (x#^c).('x + {{{0, 1}}}:|'x|)
}
```

Les fonctions `Nord`, `Sud`, …, complètent une collection de géométrie `[n, m]` par une ligne ou une colonne de `c` (`c` doit être une constante scalaire) puis font la sélection nécessaire pour réaliser un décalage

correspondant à l'accès aux voisins nord, sud, etc. La valeur `c` correspond donc à la valeur des voisins inexistants pour les cellules frontières.

IV.3.3.b. Réseaux systoliques

Un réseau systolique est un réseau régulier d'automates. Nous allons prendre le cas d'un réseau linéaire où chaque nœud du réseau délivre sur deux sorties, les résultats de la combinaison par `f` et `g` de ses 2 entrées (Cf. figure IV.9).

Le programme 81/2 correspondant utilise la récursion spatiale pour traduire la répétition des nœuds du réseau :

```
system Systo = {
     a  = e1 # f(sa, sb),
     b  = e2 # g(sa, sb),
     sa = a:[n-1],
     sb = b:[n-1]
}
```

le tissu `sa` représente les valeurs que l'on trouve le long des 1ère entrées des nœuds dans le réseau, le tissu `sb`, idem pour les 2ème entrées.

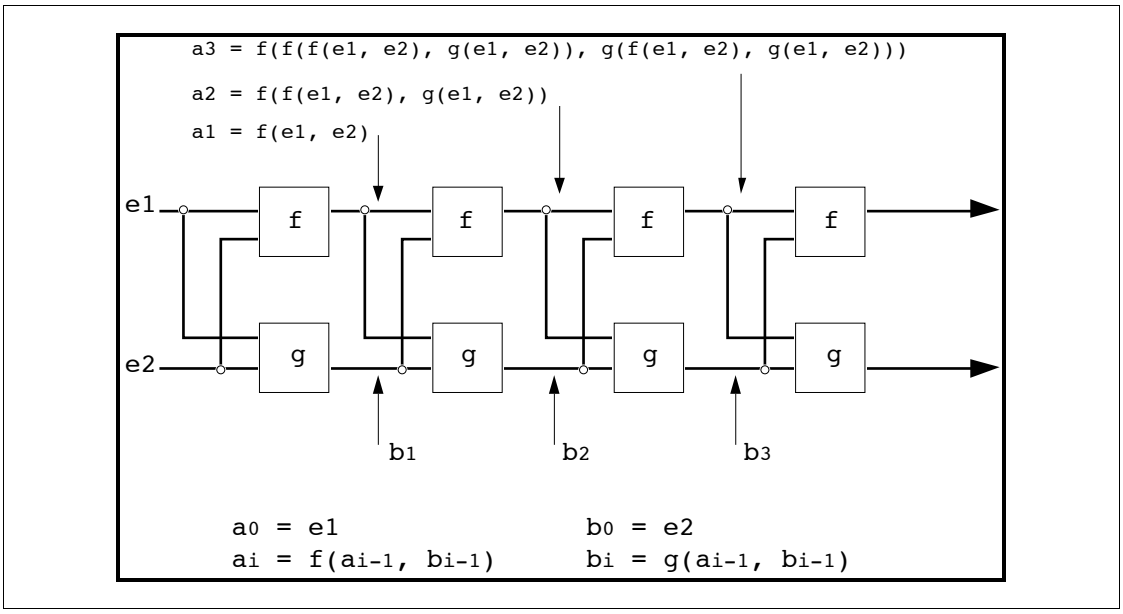


Figure IV.9 : *Illustration d'un exemple de réseau systolique linéaire*

IV.3.4. Traitement d'images : étiquetage de composantes connexes

On se donne l'image binaire `I` d'une photographie en noir et blanc représentée par une collection à deux dimensions. Une composante connexe est un sous-ensemble de l'image constitué de pixels connexes ayant tous la valeur `1`.

On désire étiqueter les composantes connexes de I, c'est-à-dire calculer un tissu `C` de même géométrie que `I` et tel que si deux points de `I` appartiennent à la même composante connexe, alors ils ont la même

valeur dans `C`. Cette valeur, l'étiquette, doit être différente pour chaque composante connexe et vaudra `-1` en tout point de `C` où `I == 0`. L'algorithme est le suivant :

1) Calculer une valeur différente pour chaque point ;
2) Propager cette valeur aux voisins qui appartiennent à la même composante connexe. Entre deux étiquettes possibles, un point choisit l'étiquette de valeur maximale.
3) Itèrer l'étape (2) tant qu'on a pas atteint un état global stable.

le programme 81/2 est le suivant :

```
/* une image binaire 256 *256. La valeur d'un point est l'entier 0 ou 1 */
I [256, 256] = { … }

/* index est un tissu constant de valeur : { {0 1 2 … 255} {256 257 … } … } */
index [I] = '256*256 + {'256}:[256, 256] ,

/* En 81/2, les entiers peuvent servir de valeurs logiques :
   0 correspond à false et tous les autres entiers à true.
   L'état initial de C correspond à l'image ou les pixels à faux sont transformé en -1
   et les pixels à vrai en l'index du point.
   Les états successifs de C correspondent aux itérations de l'étape 2.
   Ces itérations ont lieu tant que la condition d'arrêt n'est pas validée.
*/
C@0 = if I then index else -1 fi,
C   = if I then max(c1, c2, c3, c4) else $C fi
      until stop,

/* condition d'arrêt : l'étiquetage n'a pas changé entre 2 itérations */
dc = $C when Clock,
stop = && \ ($dc == dc),

/* Calcul de l'étiquette en fonction des voisins : les fonctions Nord, etc. ont déjà été introduites */
c1 = max (dc, Nord(dc, -1)),
c2 = max (dc, Sud(dc, -1)),
c3 = max (dc, Est(dc, -1)),
c4 = max (dc, Ouest(dc, -1)),
```

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

```
composantes = if (index == C after stop) then 1 else 0 fi,
nombre_composantes = + \ composantes,
```

Le tissu `composantes` permet de donner la valeur 1 à un seul point de chaque 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.

IV.3.5. Transformation du temps en espace

On veut enregistrer *n* valeurs consécutives d'un tissu scalaire `S` dans un tissu `T` de cardinal *n*, ce qui correspond à un transformateur série-parallèle (Cf. figure IV.10). Les tops de `T` se produisent tous les *n* tops de `S`.

```
function Série_Parallèle(S, n) =
{
    /* Un compteur modulo n */
    cpt@0 = 1 synchro S,
    cpt   = if (cpt == n) then 1 else $cpt + 1 fi
            synchro S,

    /* le stockage des valeurs de S se fait dans le buffer à décalage buffer */
    buffer[n] = if cpt == 1
                 then S # 0
                 else S # $buffer:[n-1]
                fi,

    /* Un échantillonnage tous les n tops */
    T = buffer when (cpt == n)
}.T
```

La fonction `SérieParallèle` utilise un système pour cacher les tissus auxiliaires `T`, `cpt` et `buffer`. Le tissu `cpt` est un compteur qui compte de `1` à `n` de manière cyclique, au rythme des tops de `S`. Le tissu `buffer` est un buffer qui est géré par décalage et qui enregistre les valeurs successives de `S` (au début, `n-1` valeurs sont initialisées à `0`). Le tissu `T` correspond au tissu `buffer` aux instants où il y a `n` nouvelles valeurs de `S`.

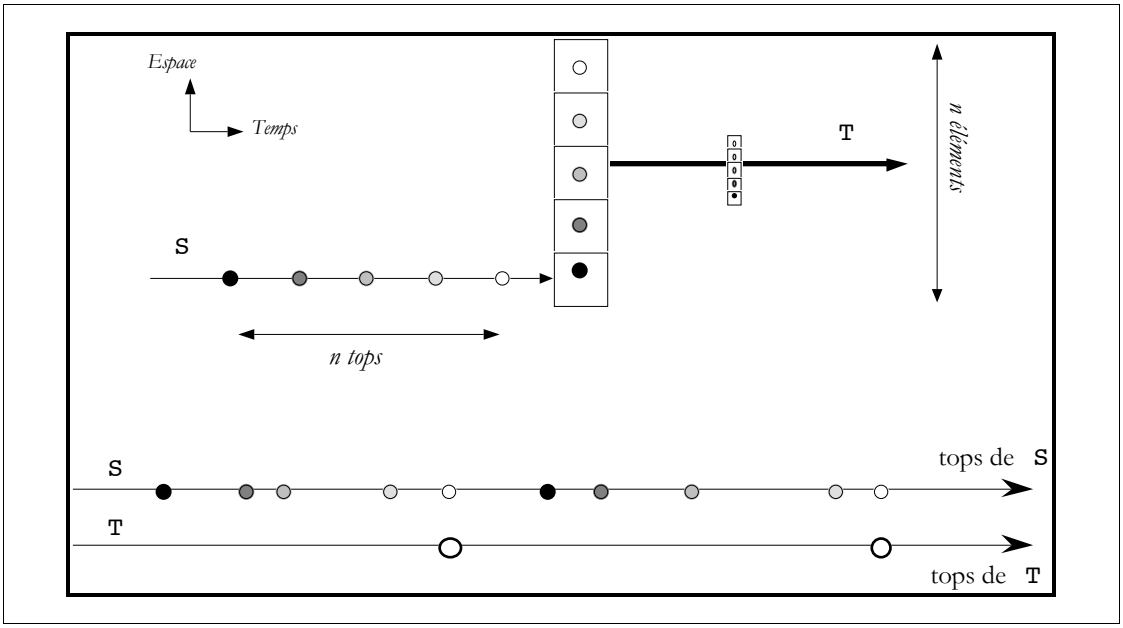


Figure IV.10 : *Un transformateur série-parallèle*

IV.3.6. Simulation par événements discrets

Une machine-outil fabrique une pièce mécanique à une cadence fixe (toutes les s secondes). Un contrôleur examine si la pièce est bonne, ce qui lui prend un temps aléatoire variant équiprobablement entre 1 et c secondes. En moyenne il rejette r% des pièces.

Un programme 81/2 simulant ce comportement peut être le suivant : on crée un système pour chacunes des entités : `Machine`, `Contrôleur`, `Buffer` et `Rebut`. `Machine` définit un stream de booléen à `vrai` quand une pièce est produite, `Buffer` et `Rejet` fournissent chacun un stream indiquant respectivement le nombre de pièces dans le buffer et dans le rebut.

```
s = 5, c = 7, r = 10,

Machine = {
     t@0 = 0,
     t   = $t + 1 when Clock,
     pièce = 0 == (t%s), /* pièce est à vrai toutes les s secondes */
},

Contrôleur =  {
     /*     cpt démarre avec la valeur  0 avec le début d'un examen et passe à 1
            avec la fin de l'examen. Un examen peut démarrer s'il y a une pièce dans le
            buffer. Le temps d'examen d'une pièce est celui qui sépare 2 tops
            de Clock -c.
     */
     cpt@0 = 0 after Buffer.nbre_pièce >= 1,
     cpt = if $cpt == 1
               then if Buffer.nbre_pièce >= 1
                         then 0
                         else $cpt
                    fi
               else $cpt + 1 when Clock -c
          fi,

     /* vaut vrai quand une pièce a été examinée */
     examen_fait = (cpt == 1) && ($cpt == 0),
},

Buffer = {
     nbre_pièce@0 = 0,
     nbre_pièce =
          if Machine & Contrôleur.examen_fait
          then $nbre_pièce                  /* production 1, consommation 1 */
          else if Machine.pièce
               then $nbre_pièce + 1         /* production 1, consommation 0 */
               else if Contrôleur.examen_fait
                    then $nbre_pièce - 1    /* production 0, consommation 1 */
                    else $nbre_pièce        /* production 0, consommation 0 */
                    fi
               fi
          fi,
},
```

```
Rebut = {
      /* Pour savoir si la pièce est un rebut, on tire au sort un nombre uniformément
         réparti entre 0 et 100. Si r  est supérieur à ce nombre, alors la pièce est un rebut. */
      nbre_pièce@0 = 0,
      nbre_pièce   = if r > random(100)
                      then $nbre_pièce + 1
                      else $nbre_pièce
                     fi when Contrôleur.examen_fait,
}
```

IV.3.7. La programmation objet en 81/2

La notion de système et la concaténation des systèmes, nous permet de manipuler des environnements. La composition permet en particulier d'implémenter des *enregistrements extensibles*. Cela nous permet donc d'émuler une programmation dans un "style objet".

Un système 81/2 représente à la fois le concept de classe des langages objets et celui de constructeur permettant de créer une instance de classe. Les arguments nécessaires au constructeur sont les variables libres du système. L'instantiation d'une classe correspond à la concaténation du système avec les arguments attendus par le constructeur. L'apport, grâce à la concaténation, d'équations supplémentaires, correspond à de *l'héritage*.

Un système complet correspond alors à un objet des langages objets. Mais ici les valeurs des "slots" sont des tissus : il n'y a pas de notion d'envoi de messages ou de méthodes. Un système 81/2 décrit les relations qui sont vraies entre les différents objets du programme, et ces relations sont maintenues à travers toutes les variations des valeurs des tissus. On peut donc dire que 81/2 implémente une certaine forme de propagation de contraintes dans un monde objet. Bien évidemment, il n'y a pas en 81/2 tous les mécanismes de protection et d'encapsulation disponibles dans les vrais langages objets.

Pour illustrer ce style de programmation, on va définir dans le style des langages objets, une classe `Mobile` d'objets mobiles dans un plan. `Mobile` est donc représenté par un système 81/2 qui possède deux variables libres : la position initiale `initial` et les déplacements élémentaires `déplacement`. À partir de ces variables libres, qui sont des vecteurs à deux éléments correspondant aux axes `Ox` et `Oy`, le système `Mobile` définit une `position` :

```
Mobile =  {
     position@0 = initial,
     position   = $position + déplacement,
},
```

On peut à partir de `Mobile`, définir les mobiles soumis à un mouvement uniforme de vitesse. La classe `TrajectoireUniforme` attend une position initiale (nécessaire à `Mobile` dont elle hérite) et un vecteur `vitesse` pour s'instancier :

```
TrajectoireUniforme = Mobile #
{
     déplacement = vitesse when Clock,
},
```

Le système `TrajectoireUniforme` est un système qui possède tous les attributs du système `Mobile`, puisqu'il s'agit de ce système étendu par la définition de `déplacement` qui permet de calculer le déplacement élémentaire d'une trajectoire uniforme (on suppose que `vitesse` est une constante et que le déplacement uniforme désiré entre deux tops de `Clock` a pour valeur cette constante). L'opérateur `#`

compose ces deux systèmes et effectue la liaison entre la variable libre `déplacement` qui apparaît dans `Mobile` et la définition de `déplacement` dans le système anonyme qui complète la définition de `TrajectoireUniforme`.

Nous allons poursuivre cet exemple en utilisant `TrajectoireUniforme` pour représenter la trajectoire circulaire d'une planète autour d'un soleil en mouvement uniforme. Cela nous amène à écrire la classe `TrajectoireCirculaire` qui attend un rayon, un centre et une vitesse angulaire :

```
TrajectoireCirculaire = Mobile #
{
     initial = {centre.0, rayon + centre.1},
     déplacement = {dx, dy},

     t@0 = 0,              /* angle de la rotation */
     t   = $t + vitesse_angulaire when Clock,

           /* déplacement circulaire élémentaire autour d'un centre mobile  */
     dx = -rayon*2*cos((t+$t)/2)*sin((t-$t)/2)
          + centre.0 - $centre.0,
     dy = rayon*2*sin((t+$t)/2)*cos((t-$t)/2)
          + centre.1 - $centre.1,
},
```

Il ne nous reste plus qu'à instantier nos classes pour décrire le système solaire :

```
soleil =   TrajectoireUniforme #
           {
              vitesse = {1, 1};
              initial = {0, 0};
           },

planète =  TrajectoireCirculaire #
           {
              rayon = 1;
              centre = soleil;
              vitesse_angulaire = 0.1;
           }
```

La figure IV.11 illustre le résultat.

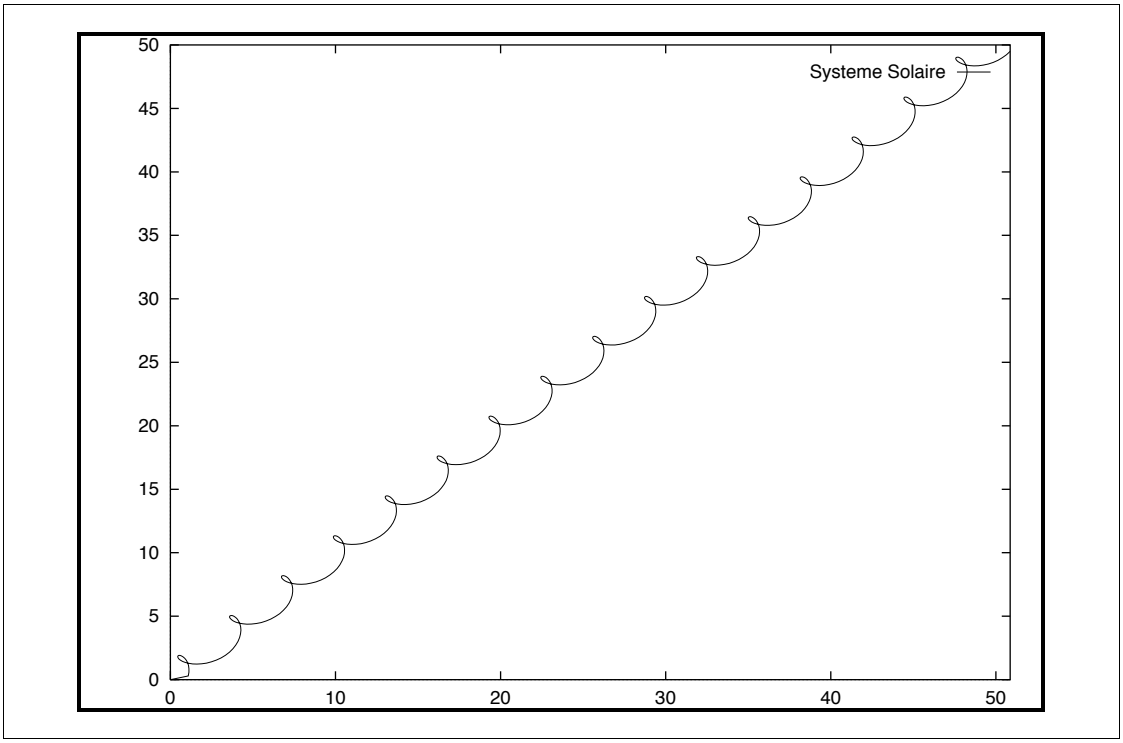


Figure IV.11 : *Tracé de la trajectoire d'une planète en mouvement circulaire autour d'un soleil animé d'un mouvement de translation uniforme.*

IV.3.8. Programmation fonctionnelle en 81/2

Les fonctions utilisateur en 81/2 ne peuvent pas être récursives ; c'est une limitation du compilateur actuel. Cependant, la récursion des fonctions apparaît en 81/2 comme un mécanisme marginal de programmation. En effet, dans un langage fonctionnel comme Lisp ou ML, les objets manipulés principalement par un programme sont les fonctions et le programmeur utilise la récursion pour construire des fonctions. En 81/2, le programmeur construit des tissus, pas des fonctions. Pour construire des tissus, il dispose de la récursion temporelle et de la récursion spatiale des tissus, de la même manière que le programmeur fonctionnel dispose de la récursion des fonctions.

Nous avons donné dans le chapitre II et III de nombreux exemples qui illustraient comment utiliser la récursion des tissus pour calculer une fonction récursive primitive. Si a un moment donné, on a besoin uniquement d'une seule valeur de la fonction, on utilisera la récursion temporelle. Si on a besoin de plusieurs valeurs de la fonction en même temps, on utilisera une récursion spatiale. Ceci est illustré avec l'exemple de Fibonacci :

```
/*interdit en 81/2 actuellement */
function fib(x) = if x == 0
                  then 0
                  else if x == 1
                       then 1
                       else fib(x - 1) + fib(x-2)
                       fi
                  fi
```

Mais on peut écrire si on a besoin d'une seule valeur à un instant

```
/*récursion temporelle sur un tissu scalaire */
fib@0 = 0 when Clock,
fib@1 = 1 when Clock,
fib   = $fib*$$fib
```

et si on a besoin de la table de `fib` a un moment donné :

```
/*récursion spatiale */
fib[n] = 0#fib:[n-1] + {0, 1}#fib:[n-2]
```

IV.3.9. Traductions dans le langage 8½ des itérations des langages impératifs

Il existe deux types de boucles dans les langages impératifs : la boucle *for* qui consiste à répéter une action un nombre de fois connu à l'avance, et la boucle *while* qui consiste à itérer une action en fonction d'un résultat calculé.

La boucle `while` des langages impératifs correspond naturellement au concept de stream, puisqu'un stream est capable, contrairement aux collections, de représenter un nombre non borné d'itérations à travers ses tops[1].

Pour la traduction des boucles *for*, on pourrait utiliser l'aspect stream d'un tissu. Elles correspondraient alors à des streams qui ont un nombre de tops limités : un top par répétition. Il semble plus logique de les représenter par une collection dont le cardinal correspond au nombre de répétitions de la boucle. S'il y a des dépendances entre les répétitions, la définition de ce tissu sera récursive. Donnons quelques exemples, en utilisant une syntaxe `C` (le `for` en `C` est utilisé ici pour son aspect répétition, bien qu'il ait la capacité d'exprimer des itérations) :

```
/* boucle en C */
float T [10];
for (i = 0; i<10; i++) T[i] = 2*i;
```

```
/* tissu 8½ correspondant */
float T [10] = 2 * 'T
```

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

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

exhibe des dépendances entre les répétitions. De ce fait, la collection correspondante est définie par une récursion spatiale (pour simplifier on utilise un tissu auxiliaire `u`) :

```
/* tissu 8½ correspondant */
float U [10] = (1 # u) : |U|,
u = U + 2.7 * 'U
```

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

---

[1] L'imbrication de boucles `while` peut se traduire par l'utilisation de l'opérateur `every` (cet opérateur est présenté dans [GIA 91b] mais n'est pas actuellement implémenté dans la version 0.1 du compilateur 8½). Notons cependant que l'on montre en théorie de la calculabilité, que tout programme contenant des `while` imbriqués peut se réécrire en un programme ne contenant qu'une seule boucle `while` englobant tout le programme.

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

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

IV.3.10. Les géométries dynamiques : exemple du triangle de Pascal

Cet exemple n'est pas traité par le compilateur version 0.1 car il définit récursivement un système dont la géométrie varie au cours du temps. Cet exemple, qui a pour but de montrer l'utilité des définitions récursives de systèmes, motive le développement d'un calcul complet sur les collections non homogènes et illustre les difficultés qu'il faut résoudre.

Il s'agit de calculer les coefficients C_n^p de l'expansion du polynôme $(a+b)^n$. Ceux-ci sont définis par la relation de récurrence :

$$C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$$

Le *triangle de Pascal* est un algorithme permettant de calculer ces coefficients de manière commode : chaque élément `p` dans ligne `n` du triangle correspond à la somme de l'élément directement au-dessus et de l'élément précédent dans la ligne `n-1` du triangle :

```
1                                  coefficients des produits a^i b^j dans le développement de (a+b)^0
1 1                                coefficients des produits a^i b^j dans le développement de (a+b)^1
1 2 1                              coefficients des produits a^i b^j dans le développement de (a+b)^2
1 3 3 1                            coefficients des produits a^i b^j dans le développement de (a+b)^3
1 4 6 4 1          coefficients des produits a^i b^j dans le développement de (a+b)^4
 …
```

On débute et on termine chaque ligne par un `1`. Il s'en suit que chaque ligne du triangle de Pascal est la somme de la ligne précédente décalée et correctement complétée par des zéro, i.e. :

```
{1, 1}           = {0, 1} + {1, 0}
{1, 2, 1}        = {0, 1, 1} + {1, 1, 0}
{1, 3, 3, 1}     = {0, 1, 2, 1} + {1, 2, 1, 0}
{1, 4, 6, 4, 1} = {0, 1, 3, 3, 1} + {1, 3, 3, 1, 0}
 etc.
```

On décide de calculer ce triangle par un stream 81/2 dont le i^ème^ top a pour valeur une collection qui représente la i^ème^ ligne du triangle de Pascal. Le triangle de Pascal `T` peut donc se calculer par le programme suivant :

```
T@0 = 1,
T   = ($T#0 + 0#$T) when Clock
```

Avec les notations du chapitre II, le résultat de l'exécution de ce programme est :

```
T ⇒ <  {1};
       {1, 1};
       {1, 2, 1};
       {1, 3, 3, 1};
       {1, 4, 6, 4, 1};  … <
```

On voit que la géométrie de `T` est dynamique : à chaque top, `T` a pour valeur une collection qui a un élément de plus que la collection calculée au top précédent. Un programme 81/2 de ce type demande la mise en place d'un mécanisme d'allocation dynamique de la mémoire, une vérification des types à l'exécution, etc.

-oOo-

V. Le langage parallèle 81/2

Dans ce chapitre, nous examinerons l'adéquation du langage 81/2 à l'expression et à l'exploitation du parallélisme offert par les nouvelles architectures parallèles. Pour cela, nous montrerons pourquoi 81/2 est un langage à la fois data-flow et data-parallèle.

Pour commencer, nous oublierons le langage 81/2 proprement dit et nous présenterons une classification des langages parallèles. Deux critères nous permettront de comparer les langages et cette classification nous amènera à étudier deux modèles de programmation parallèle : la programmation data-parallèle et la programmation data-flow. Nous examinerons ensuite les limitations dues au cadre séquentiel des langages data-parallèles classiques. Cela nous conduit à analyser d'autres modèles d'expression du séquencement et à choisir le cadre data-flow pour des raisons qui tiennent tant à l'expressivité qu'à l'efficacité.

Les problèmes posés par le choix d'un modèle d'exécution data-flow sont alors détaillés. Nous concluons à la nécessité de développer un modèle de calcul data-parallèle déclaratif et *statique* afin de capturer efficacement à la fois le parallélisme de données et le parallélisme de contrôle présents dans une application, tout en étant à même d'exploiter les ressources des nouvelles architectures massivement parallèles.

Ce modèle de calcul correspond au sous-ensemble statique de 81/2. La mise en œuvre de ce sous-ensemble sur une architecture parallèle est ensuite esquissée. Le compilateur 81/2 transforme un programme 81/2 en une *navette*. Une navette est le code dont l'exécution permet de calculer et d'énumérer dans l'ordre temporel croissant les valeurs des collections qui composent les tissus définis par le programme. Le compilateur est grossièrement composé de quatre phases : la synthèse et la vérification du typage des expressions, le calcul de l'horloge des expressions, le calcul des dépendances des tâches associées aux expressions, le placement et l'ordonnancement statiques de ces tâches. Le lecteur trouvera dans [GIA 92] et dans la bibliographie du projet 81/2 une description plus complète et plus technique.

Le langage 81/2 a été conçu pour être compilé, tant pour une machine séquentielle que pour des machines parallèles, en imposant pour la version 1.0 un modèle d'exécution data-flow qui soit statique. Cela veut dire que la réservation mémoire et l'ordonnancement des instructions sont connus à la compilation, avant l'exécution. Le code produit ne nécessite pas d'exécutif (comme par exemple un service d'allocation dynamique et de gestion de la mémoire) ni de pile d'exécution (car les expressions 81/2 sont traduites en termes de boucle de répétition).

V.1. **Une classification** de l'expression du parallélisme dans les langages de programmation

Le foisonnement des architectures et des langages parallèles amène un recouvrement des concepts qui rend difficile leur comparaison et l'établissement d'une classification universellement acceptée. C'est pourquoi nous proposons dans la table V.1 un cadre destiné à fixer les concepts présentés dans ce chapitre. Nous avons choisi de classer les langages à travers deux critères : la manière dont ils permettent au programmeur d'exprimer le contrôle et la manière dont ils lui permettent de manipuler les données.

Données \ *Contrôle*	**Langages Data Flow** *0 compteur d'instructions*	**Langages Séquentiels** *1 compteur d'instructions*	**Langages Concurrents** *n compteurs d'instructions*
Langages Scalaires	Sisal, Id, Lau Acteurs FP	Fortran, C Pascal	Ada, Occam ConcurrentPascal
Langages Collections	8 1/2 (APL, Gamma)	*Lisp, HPF, Pomp-C, CMFortran	CMFortran multi-threads

Table V.1. *Une classification des langages du point de vue du parallélisme. On ne considère ici que le noyau fonctionnel pur d'APL (les programmes d'une ligne correspondant à une expression sans effet de bord). Quant à FP, si les données correspondent à des collections à travers le concept de* séquence*, les efforts de recherche ont surtout porté sur les* formes fonctionnelles *qui permettent de combiner des fonctions existantes. C'est pourquoi nous avons fait figurer FP dans les langages scalaires.*

En ce qui concerne l'expression du contrôle, le programmeur a le choix entre trois stratégies :

- Ne pas l'expliciter : c'est le modèle d'exécution data-flow. Dans ce modèle, il n'y a pas d'expression possible du séquencement par le programmeur. C'est au compilateur (extraction statique du parallélisme), ou au support d'exécution (extraction dynamique par un interprète ou par une architecture matérielle), de construire un séquencement des calculs compatible avec les dépendances fonctionnelles entre les données du programme.
- Expliciter *la séquence* des calculs (tous les calculs se font en séquence) : c'est le modèle d'exécution séquentiel impératif classique.
- Expliciter *les calculs qui peuvent ne pas se faire en séquence* : c'est l'approche des systèmes et langages concurrents. Cette approche offre des structures de contrôle explicites comme `PAR` et `ALT` en Occam, `FORK` et `JOIN`, etc.

Pour la manipulation des données, nous distinguerons deux grandes classes de langages :

- *Les langages de collections* sont des langages qui permettent au programmeur de manipuler des ensembles de données comme des touts. On peut citer par exemple : APL, SETL, *Lisp, etc.

– *Les langages scalaires* permettent aussi au programmeur de manipuler des ensembles de données mais uniquement à travers un accès individuel aux éléments de ces ensembles. Par exemple, en Pascal standard, la principale opération sur un tableau est l'accès à un de ses éléments.

Le data-parallélisme est lié à la prise de conscience qu'il est possible d'introduire le parallélisme dans un langage séquentiel (c'est la "starisation" des langages classiques : de `C` à `C*`, de Lisp à *Lisp, de ML à NESL, etc.). Mais la table V.1 montre que la notion de collection est orthogonale à l'expression du contrôle par le programmeur. En conséquence, les langages de collections peuvent se marier aussi bien avec les langages concurrents (multi-threading) qu'avec les langages data-flow (par exemple Gamma [BAN 88] ou 81/2).

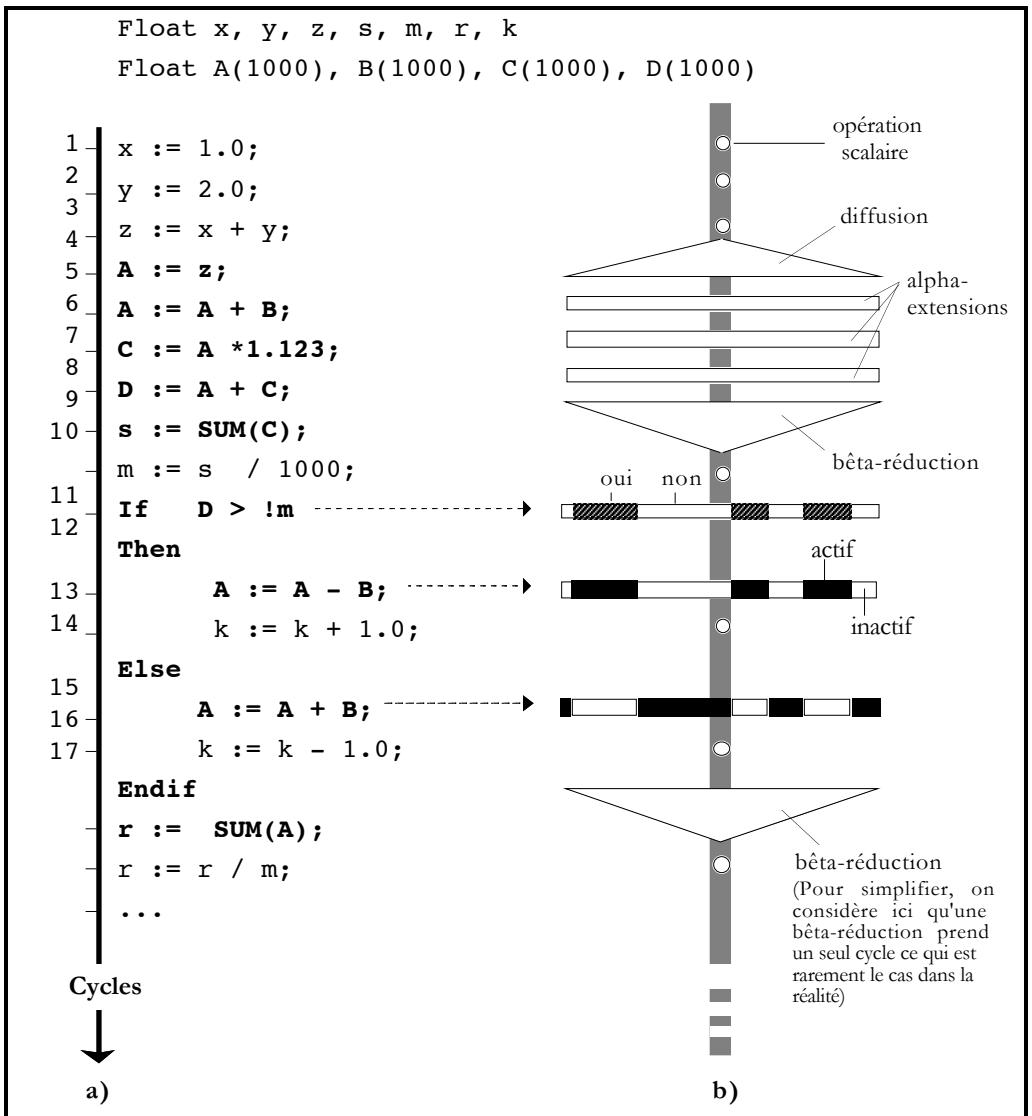
Dans la section suivante, nous allons présenter rapidement le modèle de programmation data-parallèle séquentielle. Ce modèle ne va pas sans poser des problèmes à la fois de nature sémantique et d'efficacité vis à vis des nouvelles architectures parallèles. Les problèmes d'efficacité peuvent se résoudre si on quitte un modèle à flot de contrôle strictement séquentiel, et les problèmes de nature sémantique peuvent se résoudre si on rend le contrôle implicite. C'est pourquoi, même si le mariage entre l'approche data-parallèle et l'approche concurrente a des attraits indéniables, il nous semble que la stratégie « data-parallélisme plus data-flow » est de loin la plus claire du point de vue de l'expressivité du parallélisme et la plus prometteuse quant à son exploitation efficace. Nous présenterons donc le modèle de calcul data-flow, puis nous examinerons plus précisément les bénéfices d'un mariage « data-parallélisme plus data-flow » avant de revenir à 81/2.

V.2. La programmation data-parallèle séquentielle

L'exploitation du parallélisme de données vient de la constatation naturelle que certaines applications sont composées de données de nature identiques sur lesquelles il faut effectuer simultanément une même action. Les données sont associées aux processeurs contrairement à l'approche concurrente où c'est le code des tâches qui leur est associé.

Un programme parallèle typique (Cf. figure V.2) dans le style de programmation data-parallèle classique est composé d'une suite d'instructions séquentielles. Il diffère d'un programme séquentiel classique par le fait que certaines instructions permettent au programmeur de traiter en une seule opération des tableaux de données comme des touts (alpha-extension, permutation, etc). Des opérations complémentaires comme la diffusion et la bêta-réduction permettent de passer des opérations scalaires aux opérations sur les tableaux et vice-versa.

Comme on répète une action identique en parallèle sur tous les processeurs, il est possible de centraliser le contrôle : un contrôleur unique peut envoyer de manière synchrone l'action à exécuter à tous les processeurs. C'est le modèle d'exécution SIMD. Pour le programmeur, tout se passe comme s'il avait un programme séquentiel dont les actions ne portent plus sur des données scalaires mais sur des tableaux de données. Le contrôle de flot des instructions reste séquentiel car chaque structure de contrôle séquentielle classique se double d'une structure de contrôle séquentielle data-parallèle qui a pour effet de répliquer de manière synchrone la séquence de contrôle sur chacun des processeurs de la machine.



Un programme data-parallèle calcule à la fois sur des scalaires et des tableaux. Les calculs scalaire sont figurés en maigre, les calculs data-parallèles en gras (par exemple, le point d'exclamation représente l'opérateur d'alpha-extension d'un scalaire en une collection, ou diffusion). Sur une architecture SIMD, les calculs scalaires sont effectués sur une station de travail qui pilote "un processeur de tableaux" auquel est délégué le calcul sur les collections. Ce processeur de tableaux correspond à un tableau d'unités de calcul. La station de travail diffuse les instructions à effectuer à chacune de ces unités. L'implémentation d'une conditionnelle implique que certaines unités soient inactives pendant que la station de travail diffuse le code correspondant à une des branches de la conditionnelle. L'activité des unités de calcul est gérée par un masque d'activation. Dans le cas de la conditionnelle, le masque d'activation est positionné sur chaque unité de calcul suivant le résultat de l'évaluation de la condition. Le coût d'évaluation de la conditionnelle est donc égal à la somme des coûts d'évaluation des deux branches, même si toutes les unités de calcul empruntent la même branche. En cas de conditionnelles imbriquées, ou de conditionelles multiples (case… of…) on voit que le rendement du calcul peut baisser exponentiellement avec les niveaux d'imbrication. Par comparaison, dans une architecture MIMD ou SPMD, où chaque unité de calcul dispose de son propre contrôleur et est donc autonome, le coût de l'évaluation de la conditionnelle aurait été égal au maximum des coûts de chaque branche.

Figure V.2 : *Un programme data-parallèle séquentiel typique. Le code du programme est figuré en* `a)`*. Le flot d'exécution est représenté en* `b)`*.*

Du point de vue de la programmation des machines *massivement* parallèles, la programmation data-parallèle présente l'avantage décisif suivant : il n'y a pas besoin d'écrire autant de tâches qu'il y a de processeurs. En fait, il suffit d'ajouter un « 0 » dans la déclaration d'un tableau pour qu'il y ait 10 fois plus de calculs à exécuter, alors que si on avait voulu obtenir le même résultat à travers le parallélisme de contrôle, il aurait fallu écrire 10 fois plus de tâches (s'il suffisait de dupliquer les tâches, ce serait le signe qu'on exploite alors du parallélisme de données). On voit donc que le parallélisme de données est une voie particulièrement intéressante pour le développement d'un modèle de programmation adapté aux architectures massivement parallèles (celles qui comptent plus d'un millier de processeurs).

Le langage 81/2 permet de manipuler des collections ; c'est donc un langage data-parallèle. Mais contrairement aux langages data-parallèles classiques, il n'y a pas en 81/2 de notion explicite de contrôle séquentiel du flot des instructions par le programmeur. Cette gestion séquentielle du contrôle pose deux types de problèmes dans les langages data-parallèles : des problèmes d'expressivité (sur la sémantique d'un tel langage) et des problèmes d'efficacité vis à vis des nouvelles architectures parallèles. Par ailleurs, beaucoup de langages data-parallèles ont choisi de laisser au programmeur le choix du placement des données. Nous montrerons que ce choix ne favorise pas la réutilisation et la portabilité des programmes.

V.2.1. Problèmes sémantiques posés par le parallélisme de données à contrôle de flot séquentiel

Les langages data-parallèles comme *Lisp ou Pomp-C, introduisent le data-parallélisme au moyen de structures de contrôle permettant de gérer de manière strictement synchrone l'activité parallèle des$processeurs (par exemple la structure `*when` en *Lisp ou la structure `where` en Pomp-C qui correspondent à la conditionnelle data-parallèle). Ces structures de contrôle sont la source de certaines difficultés sémantiques. Ces difficultés proviennent de la mauvaise interaction entre d'une part, le concept de collection, et d'autre part, la gestion des deux flots de contrôle (à savoir le séquencement de la partie scalaire et le séquencement de la partie parallèle du programme).

V.2.1.a. Contrôle séquentiel scalaire et contrôle séquentiel parallèle

Examinons le programme suivant (en Pomp-C) :

```
collection [512,512] Pixel;
Pixel int image;          image est une collection d'entiers
int a;                    a est un scalaire entier
…
image = FALSE;            tous les points de image sont affecté à faux  donc
where (image) {           aucun  processeur ne doit être actif dans le where.
     a = 5;               Mais cette affectation scalaire est exécutée  car elle ne
                          dépend pas du where
     everywhere {   permet de réactiver inconditionnellement tous les  processeurs
          printf("Yo");   cette instruction étant scalaire, un seul  Yo est imprimé
          image = TRUE;   ici, tous les points de image sont modifiés
     }
}
```

Dans la branche d'un `where`, ce n'est pas parce que tous les processeurs sont inactifs, que certaines instructions ne seront pas exécutées : c'est le cas par exemple d'une instruction scalaire (`a = 5`) ou bien une instruction parallèle dans la portée immédiate d'un `everywhere` (instruction `image = TRUE`).

V.2.1.b. Contrôle de flot et création de données parallèles

Dans l'exemple suivant (en *Lisp), les problèmes arrivent lors de la création d'une collection **A** :

```
(*when (…)                          on suppose que dans ce *when, certains
                                    processeurs sont inactifs
   (*let ((A (!! 1)))               on crée une nouvelle collection, A, locale au *let, et
                                    on initialise à 1 seulement les éléments de A situés
                                    sur des processeurs actifs
      (*all (*sum A))))             à cause de *all, on fait la somme de tous
                                    les éléments de A
```

En fait, la collection **A** est créée avec la géométrie courante (celle du `*when`) mais elle n'a été initialisée que partiellement lors du `*let`. La somme de tous les éléments de **A** additionnera aussi les valeurs indéterminées des éléments situés sur les processeurs inactifs lors de la création de **A**.

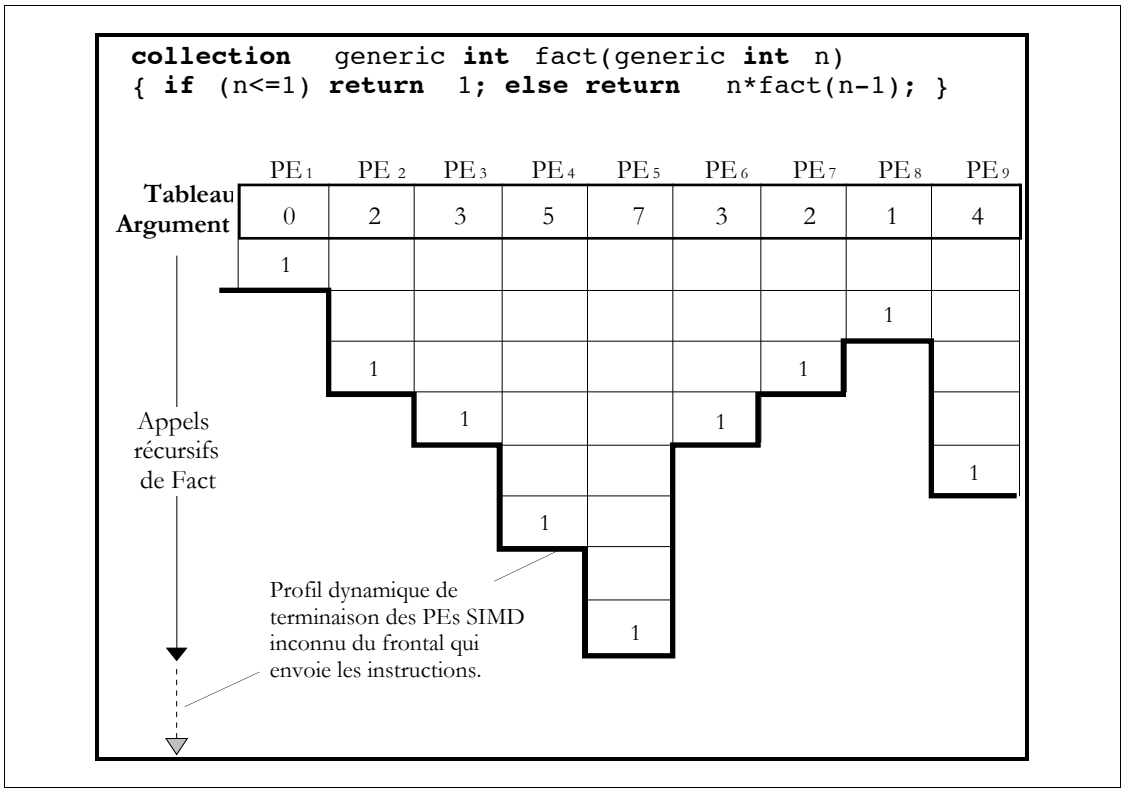


Figure V.3 : *L'expression la plus naturelle de la factorielle est incorrecte dans une implémentation SIMD. Le contrôleur doit envoyer le code correspondant aux appels récursifs de factorielle tant qu'il y a un processeur actif. Les processeurs qui atteignent le code « `return 1` » sont inactifs jusqu'à ce que tous les processeurs aient atteint la terminaison. Le problème est que le contrôleur ne sait pas détecter la terminaison des processeurs autrement qu'en faisant une opération de réduction globale, à chaque appel récursif, ce qui rend l'implémentation des appels de fonction data-parallèle prohibitivement coûteux. De plus, cela ne constitue pas réellement une solution, car comme le montre l'exemple du §V.2.1.a, un processeur inactif peut très bien être impérativement réactivé. Une implémentation correcte exigerait que le contrôleur soit capable de déterminer (à faible coût) que les processeurs élémentaires sont tous inactifs* et *le resteront.*

V.2.1.c. Alpha-extension et contrôle de flot parallèle

On voit qu'un langage data-parallèle séquentiel ne permet pas toujours au programmeur de maîtriser facilement le traitement des collections. Le premier exemple nous montre qu'un **where** ne peut pas servir à conditionner un appel de fonction situé dans sa portée : même si tous les processeurs sont inactifs, les compilateurs de *Lisp et de Pomp-C génèrent l'appel systématique à la fonction car elle pourrait contenir soit des **everywhere**, soit des instructions scalaires. Par suite, la fonction suivante (exemple extrait du manuel Pomp-C [PAR 92]) :

```
collection generic int fact(generic int n)
{ if (n<=1) return 1; else return n*fact(n-1); }
```

est incorrecte puisque l'appel récursif est systématiquement exécuté : la récursion n'est pas bornée et le programme boucle (Cf. figure V.3).

L'expression récursive correcte de factorielle doit introduire un mécanisme de surveillance stoppant la récursion dès que tous les processeurs sont inactifs. Cela peut être évité si on sait que la fonction ne contient pas de code scalaire ou d'effet de bord mais cette propriété ne peut être déterminée en toute généralité par un compilateur.

V.2.1.d. Contrôle de flot parallèle et accès en parallèle à un tableau

La boucle **FORALL** disponible en FORTRAN-D, en Vienna-FORTRAN ou en HPF permet de spécifier des accès en parallèle à un tableau. Par exemple, la boucle suivante :

```
FORALL (i=1:99)
     A(i+1) = 2*B(i)
END forall
```

va évaluer *en parallèle* les 99 affectations : **A(2)←2*B(1)**, **A(3)←2*B(2)**, …, **A(100)←2*B(99)**. Grâce aux accès en parallèle, qui correspondent aux expressions d'indices qui apparaissent dans la portée d'un **FORALL**, le programmeur peut spécifier très finement des domaines complexes de calcul. Cependant la sémantique de la boucle **FORALL** est loin d'être simple. Par exemple, dans le programme suivant :

```
FORALL (i=1:10)
     where (C(i))
          A(i+1) = B(i)
     elsewhere
          A(i-1) = B(i)
     end where
END forall
```

Supposons que **C(2)** a pour valeur le booléen **vrai** et que **C(4)** a pour valeur le booléen **faux**. Alors, l'élément **A(3)** est assigné deux fois, une fois par **B(2)** et une fois par **B(4)**. Ces affectations étant censées s'effectuer en parallèle, quel doit être le résultat ?

Suivant les langages, un programme comme le précédent est déclaré incorrect (mais le compilateur ne peut pas vérifier ce genre de propriétés en toute généralité, car elles sont indécidables), ou bien permis car il y a séquentialisation de l'évaluation du corps des **where** dans la portée d'un **FORALL**, ou encore le résultat est déclaré « dépendant de l'implémentation ».

Ces exemples montrent les difficultés sémantiques qu'il y a à *émuler la notion de collection* dans le cadre d'un contrôle de flot séquentiel.

V.2.2. Implémentation du data-parallélisme séquentiel sur les nouvelles architectures parallèles

Dans cette section nous évoquons les problèmes d'implémentation des langages data-parallèles séquentiels sur les nouvelles architectures parallèles à contrôle hybride.

V.2.2.a. Les nouvelles architectures à contrôle de flot hybride

Récemment, on a vu émerger une nouvelle classe d'architectures massivement parallèles comme la CM5 [TMC 91] ou PTAH [CAP 92]. Ces architectures sont capables d'exploiter plusieurs sources de parallélisme à la fois, en préservant la simplicité et l'efficacité des modèles SIMD tout en relâchant les contraintes de synchronisme afin d'acquérir la flexibilité et le rendement des processeurs caractéristiques des modèles MIMD. Ces architectures sont des architectures à contrôle hybride [STE 90] : SPMD, MSIMD, MIMD-fortement synchronisable… On pense généralement que ces architectures permettront d'atteindre la puissance du TeraFlops nécessaire aux applications numériques intensives du « Grand Challenge ».

Il est alors naturel, voire nécessaire, d'implémenter les langages data-parallèles sur ces nouvelles machines. En effet, les langages data-parallèles sont parfaitement adaptés aux applications numériques intensives car ils permettent de manipuler très facilement les objets classiques du calcul numérique : vecteurs, matrices, … ; et plus généralement, les langages data-parallèles sont bien adaptés au calcul massivement parallèle (bases de données, traitement d'images, etc.) grâce à leur expression d'un parallélisme de données à grain fin (Cf. §V.2. introduction).

V.2.2.b. Contrôle de flot séquentiel et data-parallélisme

Néanmoins, le modèle data-parallèle séquentiel pose de sérieuses difficultés sur les machines à contrôle hybride et plus généralement sur les machines MIMD (Cf. [HAQ 91] pour une approche de la question). La plus importante est sans doute la mauvaise utilisation des ressources de calcul liée à un contrôle de flot *strictement séquentiel* comme on le rencontre dans les implémentations SIMD pures. Autrement dit, on transporte dans les modèles MIMD les inconvénients connus des modèles SIMD [GER 91]. Par exemple, lors des calculs sur des données scalaires, les processeurs affectés aux traitements parallèles restent inactifs ; de même, l'imbrication d'instructions comme `where` peut réduire exponentiellement le nombre de processeurs actifs (Cf. figure V.2).

V.2.2.c. Communication et modèle d'exécution SIMD

Mais la principale limitation est sans doute l'incapacité à masquer les temps de communication entre processeurs par l'exécution de branches de calculs indépendantes. Cette limitation, qui ne peut être levée par les implémentations SPMD, a de graves conséquences car le réseau de communication est souvent la partie la moins performante d'une architecture. La performance du modèle d'exécution data-parallèle séquentiel est donc bornée par la partie la plus faible de l'architecture. C'est une conséquence directe du cadre strictement séquentiel imposé au data-parallélisme par les structures de contrôle de flot séquentiel.

V.2.3. Placement et ordonnancement explicite dans les langages data-parallèles séquentiels

Le compilateur d'un langage data-parallèle doit pouvoir répondre à la question : sur quels processeurs sont placés les éléments des différentes collections qui apparaissent lors de l'évaluation d'un programme ? C'est le problème du *placement des données*. La réponse à cette question peut être fournie explicitement par le

programmeur, ou bien laissée à la charge du compilateur. C'est la première solution qui est choisie par la plupart des langages data-parallèles, bien que cela ne soit en rien intrinsèque à ces langages.

Il nous semble qu'un langage parallèle de haut-niveau ne peut pas faire le choix d'un placement et d'un ordonnancement explicites. Par langage de « haut-niveau » nous qualifions ici les modèles de programmation qui permettent la réutilisation et la portabilité des programmes. Or, il est facile de montrer que le placement explicite des données va à l'encontre des objectifs de réutilisation et de portabilité.

V.2.3.a. Le placement explicite des données en HPF

Dans le langage data-parallèle HPF, le programmeur peut spécifier explicitement le placement des éléments d'un tableau à travers des *directives d'alignement* (on aligne un tableau par rapport à un repère appelé *patron*) et des *directives de distribution* (on distribue les éléments des tableaux alignés avec le patron sur les processeurs de l'architecture cible). Voici un exemple de placement malencontreux des données, cité dans [HPF 93] :

```
REAL A(1000), B(1000,1000)
…
TEMPLATE T(1000, 1000)
PROCESSORS P(10,10)
DISTRIBUTE T(BLOCK,BLOCK)
           ONTO P
ALIGN A(:) with T(1,:)
…
FORALL I=1,1000
     A(I) = …
END DO
```

On dispose de 100 processeurs et d'un patron **T**, correspondant à un repère à 2 dimensions, dont les éléments s'équidistribuent sur ces processeurs.

Les éléments de **A** sont stockés sur 10 processeurs (parmi les 100 disponibles).
L'exécution de la boucle s'exécutera donc sur 10 processeurs. Pour utiliser les 100 processeurs, il faut explicitement réaligner **A** avant le calcul.

Un programmeur conscient des implications implémentatoires de ces déclarations n'écrirait sans doute pas un tel fragment de code, même si celui-ci apparaît naturellement dans son application (par exemple **A** est une "condition aux frontières" dans un calcul qui est lui aligné sur **T**). Cependant, cet exemple illustre les mauvaises interactions inévitables entre des alignements et des distributions écrits par une première personne et du code écrit par une seconde personne. Or c'est ce cas de figure qui se produit souvent lors de l'utilisation de routines de traitement en bibliothèques.

Le placement des données est généralement explicitement spécifié par le programmeur d'application en fonction de ses propres besoins de calcul. Ces besoins peuvent être contradictoires avec les besoins spécifiques des calculs de chaque routine, avec des conséquences similaires à celles de l'exemple donné plus haut. L'exemple suivant est encore plus édifiant :

```
REAL FUNCTION Foo (X)
REAL X(:), Foo ALIGN X with *
  …
END Foo

ALIGN B(:,:) with T(:,:)

FORALL I=1,1000
     B(I,I) = Foo( B(I,1:1000) )
END DO
```

Le processeur qui stocke `A(I,I)` calculera `Foo(B(I,1:1000))`, en créant éventuellement sa propre copie locale de `B(I,1:1000)`. Une autre stratégie possible en HPF serait de distribuer les évaluations de ***Foo*** en appliquant la même règle que pour le programme. Dans ce cas, chaque invocation de ***Foo*** sera exécutée par les 10 processeurs qui stockent la colonne de **B** accédée par cette invocation.

Trois stratégies sont possibles pour éviter ces inconvénients :

- Le programmeur d'application connaît le placement requis pour une utilisation optimale des routines, et il utilise cette connaissance lors du développement de son propre programme.

– Un programmeur système a développé différentes versions de la même routine, afin de s'accorder avec chaque famille de placements possibles dans les applications[1].

– On réorganise systématiquement les données à l'entrée de chaque routine.

Dans les deux premiers cas, on a échoué dans l'encapsulation et la capitalisation d'un savoir-faire dans une routine de bibliothèque réutilisable. Dans le dernier cas, si le réarrangement systématique des données à l'entrée de chaque routine assure localement une exécution au mieux de celles-ci, le mouvement global des données dans tout le programme est, lui, loin d'être correctement géré. Le même type de remarque s'applique quand on change le nombre de processeurs ou la topologie de la machine cible : un bon placement peut devenir mauvais quand on augmente ou quand on diminue le nombre de processeurs[2].

Ces inconvénients sont causés par *une description des calculs qui n'est pas assez abstraite* : le programmeur ne spécifie pas seulement les relations spatiales et temporelles des éléments du calcul entre eux (ces calculs doivent s'évaluer en séquence, cela peuvent s'évaluer en parallèle) mais il doit spécifier aussi une implémentation ad-hoc sur une architecture cible donnée (ces calculs se dérouleront sur tel processeur, ces communications logiques correspondront à telles communications physiques, etc.). Quand l'architecture cible change, le programme est à réexaminer et parfois à réécrire.

En Fortran HPF, ce réexamen est limité aux déclarations d'alignement et de distribution. Ces déclarations sont indépendantes du code de calcul, ce qui est un premier pas vers l'abstraction et évite de devoir intervenir sur tout le texte du programme. Cependant les exemples précédents montrent que les déclarations d'alignement et de distribution ne sont pas modulaires, ce qui empêche le développement de vraies routines largement réutilisables et portables sur différentes architectures.

C'est pourquoi nous voulons développer un modèle de programmation de haut-niveau, proche des applications et suffisamment abstrait afin de laisser implicite les contraintes des architectures cibles. C'est le compilateur qui sera chargé d'utiliser au mieux les ressources matérielles pour l'évaluation du calcul.

V.3. Le modèle de calcul data-flow scalaire

Les conclusions de la section précédente poussent fortement à dégager le data-parallélisme et son support, la collection, du cadre strictement séquentiel. Il faut donc trouver un autre cadre susceptible d'accueillir les structures de données parallèles. Ce cadre devra pouvoir laisser implicite la distribution des données.

La classification de la table V.1 nous indique deux choix possibles : le mariage du data-parallélisme avec l'approche concurrente et le mariage du data-parallélisme avec l'approche data-flow. Nous avons choisi avec 81/2 d'explorer cette seconde voie.

---

[1] Du coup, on utilise des techniques sophistiquées de hiérarchie objet afin de gérer les interactions possibles. Par exemple on va développer des routines pour représenter des matrices gérées par ligne, et par colonnes, deux des sous-classes possibles pour la classe « matrice ». Le problème reste de prévoir toutes les interactions possibles et par exemple de développer une routine de multiplication matrice-ligne/matrice-ligne, matrice-ligne/matrice-colonne, matrice-colonne/matrice-ligne, etc. Les interactions entre les classes concrètes qui implémentent les classes abstraites croissent exponentiellement avec le nombre des premières.

[2] par exemple à cause d'un effet de « *stride* »

V.3.1. Le choix du modèle data-flow

Le mariage entre l'approche data-parallèle et l'approche concurrente produit des langages ayant des attraits indéniables : d'abord ces langages reflètent assez fidèlement la structure matérielle des nouvelles architectures à contrôle hybride ; ensuite ils sont faciles à appréhender par les programmeurs car ils correspondent à la juxtaposition de la notion de tâche et de la notion de collection ; enfin ils bénéficient des outils et des formalismes développés pour les systèmes concurrents. Cependant ils imposent l'expression explicite du parallélisme de contrôle ce qui minimise potentiellement le parallélisme exploitable (Cf. §V.3.3.a). L'expression explicite du parallélisme de contrôle s'accorde$plus facilement avec une distribution explicite : telle donnée est gérée par telle tâche qui est affectée à tel processeur. Par ailleurs, ces langages imposeraient aussi une hiérarchisation parfois artificielle des applications en deux couches : des tâches MIMD à forte granularité qui manipulent des tableaux gérés en mode SIMD. Enfin, le cadre naturel de ces modèles d'exécution est un cadre explicitement asynchrone avec tous les problèmes associés.

Ceci nous a amené à considérer l'approche data-flow comme le cadre naturel pour l'expression du data-parallélisme. Nous allons dans la section suivante présenter rapidement le modèle data-flow standard, qui manipule des valeurs scalaires, et montrer pourquoi un programme déclaratif 81/2 est un programme data-flow. Nous reviendrons ensuite sur les avantages et les inconvénients du modèle d'exécution data-flow.

V.3.2. Le modèle de calcul data-flow scalaire

V.3.2.a. Le principe de l'assignation unique

Les principes des modèles de calcul ressortant du data-flow[1] remontent à la fin des années 1960, [TES 68], et des architectures d'ordinateurs data-flow ont été proposées au début des années 1970 [DEN 74] comme une alternative au modèle d'exécution impératif classique. La motivation initiale était de rapprocher la notion de programme de celle d'énoncé mathématique, afin de pouvoir formaliser le traitement des programmes. Or la notion informatique de variable diffère considérablement de la notion mathématique de variable : une variable mathématique représente la dénotation d'une valeur « indéterminée mais constante » alors que la variable informatique, qui est associée à un emplacement mémoire, dénote une valeur qui change au cours des affectations.

Tesler et Enéa ont imaginé un langage dans lequel la notion informatique de variable rejoindrait la notion mathématique de variable. Pour cela, ils ont ajouté une contrainte à un langage impératif, *l'assignation unique* : chaque variable ne peut apparaître qu'une seule fois en partie gauche d'une affectation. Cette contrainte présente des difficultés quand une affectation apparaît dans un corps de boucle : il faut alors supposer qu'une variable dénote une suite de valeurs (nous reviendrons dessus un peu plus loin).

Dans un langage impératif classique, il y a parfois redondance entre le séquencement imposé par les dépendances entre données et le séquencement imposé par le programmeur. Par exemple :

```
BEGIN
     a := 2;
     b := a + 1
END
```

est un fragment de programme `PASCAL` qui vérifie le principe d'assignation unique. Le point-virgule spécifie un séquencement impliquant que la valeur de `a` doit être calculée avant celle de `b`. Mais l'expression définissant `b` dépend de la valeur de `a` : la dépendance entre les données induit naturellement un ordre

[1] [THA 87] est un recueil des articles fondateurs du data-flow et [DEN 91] propose une bonne perspective historique sur le data-flow.

d'évaluation et, dans cet exemple, *l'ordre induit par la dépendance des données suffit à déterminer le résultat du calcul.* Cette propriété est une conséquence du principe d'assignation unique. Avec l'assignation unique, le séquencement des opérations peut être implicitement assuré par le respect des seules dépendances entre les données.

Dans la suite, nous appellerons langage data-flow, les langages dans lesquels le séquencement des opérations est implicitement assuré par la dépendance entre les données. Dans un modèle data-flow, le programmeur *ne peut pas* spécifier de relation d'ordre entre les opérations du langage. Un *ordonnancement* des opérations est automatiquement déduit du programme soit *statiquement* (par un compilateur), soit *dynamiquement* (par un évaluateur logiciel ou matériel).

V.3.2.b. Définition des graphes data-flow

Un principe général des programmes data-flow est que toute variable ne dénote qu'une seule valeur tout au long de l'exécution du programme. Cette propriété permet de représenter un programme par un graphe, *le graphe data-flow* (Cf. figure V.4), où les nœuds correspondent à des expressions et les arcs représentent les dépendances entre les expressions.

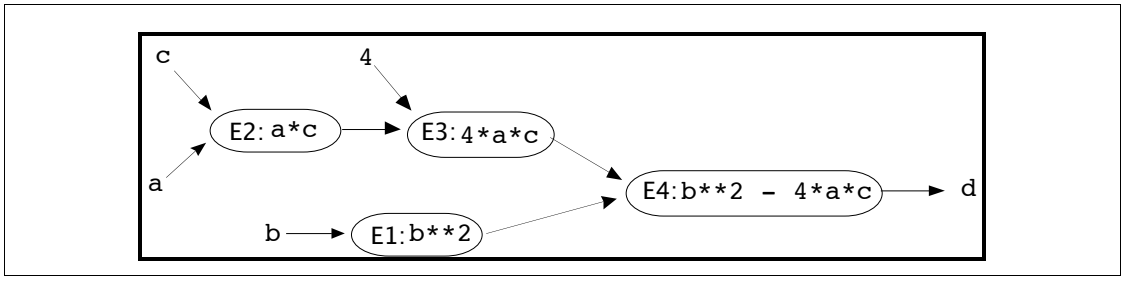


Figure V.4 : *Graphe data-flow de l'expression* `b**2-4*a*c`.
Chaque nœud correspond à une expression élémentaire. Une expression élémentaire peut être calculée dès que ses entrées sont prêtes. Lors d'une évaluation séquentielle, on peut calculer E1; E2; E3; E4*, ou bien* E2 ; E1 ; E3 ; E4*, ou encore* E2 ; E3 ; E1 ; E4*, etc. Dans le cas d'une évaluation parallèle, le calcul de* E1 *peut se faire parallèlement au calcul de la séquence* < E2 ; E3 <.

Lors de l'évaluation d'un graphe data-flow, c'est la disponibilité des données qui déclenche l'évaluation des expressions et non pas la gestion d'un (ou plusieurs) compteur(s) d'instructions. Une expression peut être évaluée dès que tous ses arguments ont été calculés (Cf. figure V.5). Avec ces règles, il est facile de réaliser un évaluateur data-flow, en implémentant le mécanisme suivant :

- les données se comportent comme des jetons qui circulent sur les arcs ;
- les nœuds sont des transformateurs de jetons et produisent un jeton sur leur arc de sortie quand les jetons nécessaires pour le calcul sur les arcs sont présents en entrée.

On peut aller plus loin dans l'utilisation des graphes data-flow et considérer des graphes quelconques, par exemple cycliques (un graphe data-flow cyclique correspond à une expression récursive, comme on le verra plus bas). La figure V.5 représente le graphe data-flow d'un compteur. Le nœud noté « `1*` » est un générateur de jetons « 1 » et le nœud « `1&` » génère le jeton « 1 » puis transmet à sa sortie les jetons qui se présentent en entrée. La suite des jetons qui passent sur l'arc `A` est la suite des entiers strictement positifs.

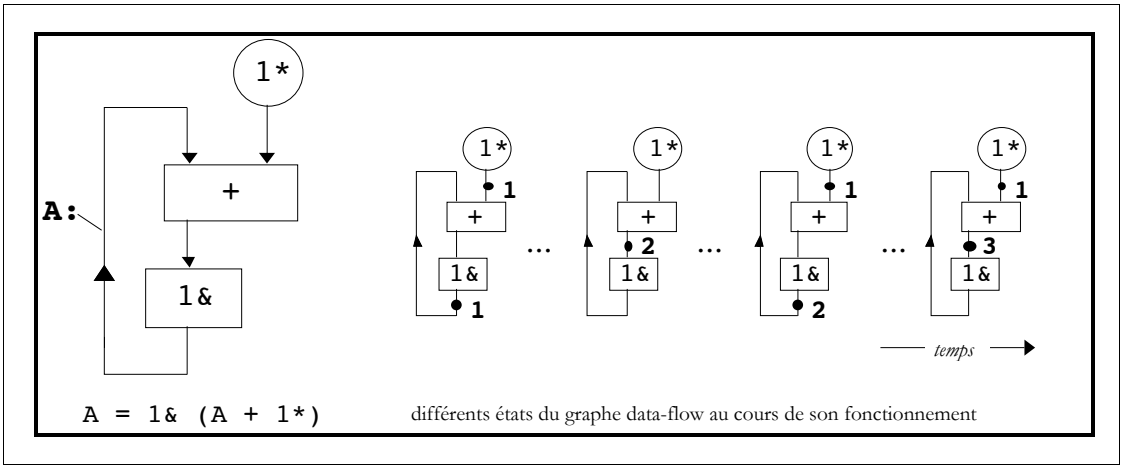


Figure V.5 : *Graphe d'un compteur data-flow (à gauche) et son fonctionnement (à droite). Les jetons sont figurés par des pastilles sur les arcs. Le nœud* `1*` *est un générateur de « 1 » et le nœud* `1&` *génère le jeton « 1 » puis transmet à sa sortie les jetons qui se présentent en entrée. La suite des jetons qui passent sur l'arc* `A` *est la suite des entiers strictement positifs.*

V.3.2.c. Graphe data-flow et programme déclaratif

On peut retrouver l'expression calculée par un graphe data-flow en redonnant une forme textuelle à ce graphe :

- on associe une variable à chaque arc :
- une variable représente la valeur du jeton transitant sur l'arc associé ;
- la valeur du jeton sur un arc sortant d'un nœud est fonction de la valeur des jetons des arcs entrants du nœud.

Par exemple, pour le graphe data-flow de la figure V.4, on a :

```
E1 = b**2
E2 = a*c
E3 = 4*E2
d  = E1 - E3
```

Ces quatre définitions peuvent et doivent être considérées comme autant d'équations : partout où une variable apparaît, on peut la remplacer par le membre droit de sa définition sans changer le résultat calculé. Cette propriété est analogue à l'assignation unique et s'appelle la *transparence référentielle*. Autrement dit, nous savons associer à un graphe data-flow un système d'équations.

Dans l'exemple du compteur de la figure V.5, en composant les transformations rencontrées sur les chemins qui mènent à la variable `A`, on obtient l'équation (en évitant de nommer les expressions intermédiaires)

```
A = 1& (A + 1*)
```

Comme le graphe précédent est cyclique, la définition de la variable `A` est une définition récursive : `A` apparaît à la fois en partie gauche et en partie droite de l'équation qui la définit.

L'arc qui est désigné par `A`, est pris dans un cycle et voit passer plusieurs jetons (en fait, une infinité de jetons). Il faut donc préciser ce que nous avons affirmé : une variable représente en fait *une suite de valeurs*. Nous retrouvons les streams 81/2 : un programme 81/2 correspond au système d'équations associé à un graphe data-flow.

V.3.2.d. Les programmes data-flow et la résolution des équations récursives

Si un programme 81/2 correspond à un système d'équations associé à un graphe data-flow, la réciproque n'est pas vraie. En effet, en 81/2, un stream **x** définit par

```
x = expression de streams
```

est toujours *une fonction* des streams qui apparaissent dans ***expression de streams***. En terme de graphe data-flow, cela veut dire que les nœuds du graphe data-flow sont tous des fonctions, donc que pour chaque nœud, le stream de sortie est fonction des streams en entrée.

Ce n'est pas le cas pour tous les graphes data-flow. Par exemple, certains langages data-flow introduisent un opérateur non-déterministe `merge` permettant de fusionner deux streams. L'opérateur `merge` prend deux entrées `a` et `b`. Si un jeton se présente sur `a` et qu'il n'y a pas de jeton sur `b`, le jeton `a` est propagé en sortie. Si un jeton se présente sur `b` et qu'il n'y a pas de jeton sur `a`, le jeton `b` est propagé en sortie. Enfin, si des jetons se présentent à la fois sur l'entrée `a` et sur l'entrée `b`, un des jetons *choisi au hasard*, est propagé sur la sortie, et l'autre jeton est éliminé. Le caractère *non-déterministe* du choix du jeton est important : c'est lui qui empêche d'associer une fonction à l'opérateur `merge`.

G. Kahn a établi, aux débuts des années 1970, un résultat permettant de caractériser ce que calcule un système d'équations correspondant à un graphe data-flow [KAH 74]. Ce résultat dit que si le graphe data-flow ne contient que des nœuds qui ont un comportement fonctionnel (cas des programmes 81/2), alors le système d'équations correspond au calcul du plus petit point fixe d'une certaine fonction[1]. M. Faustini [FAU 82] a pu établir, aux début des années 1980, le lien avec le modèle d'exécution en termes de circulation de jetons sur le graphe data-flow : les deux approches calculent la même chose.

Ces résultats nous fournissent un outil pour résoudre des équations moyennant certaines hypothèses à vérifier : pour résoudre un système d'équations « variable = expression », il suffit de faire "travailler" le graphe data-flow associé. Cependant, cette méthode de résolution peut être coûteuse à implémenter si la circulation des jetons dans le graphe est coûteuse à réaliser. Le compilateur 81/2 détermine une grande partie des paramètres de la circulation des jetons, afin de laisser le moins possible de décisions à prendre lors de l'exécution, et à abaisser autant qu'il est possible, les coûts de ces mécanismes.

V.3.3. Les avantages du modèle d'exécution data-flow

Dans cette section, nous allons examiner le modèle de programmation data-flow du point de vue de l'expression et de l'exploitation du parallélisme.

V.3.3.a. Expression implicite du parallélisme

Le premier avantage du modèle data-flow est de nature ergonomique. Dans un langage data-flow, le séquencement des instructions est implicite pour le programmeur ; il ne fait que décrire les dépendances entre les données de l'application. Ainsi, le programmeur n'a pas à décrire les opérations à effectuer en parallèle. Par exemple, il est préférable d'écrire naturellement le programme qui calcule la somme de quatre données :

```
som = a + b + c + d,      ce programme est un système d'équations où
a = …,                                l'ordre d'écriture des équations est indifférent
b = …,
c = …,
d = …
```

[1] Il faut en plus que les expressions en partie droite des équations vérifient certaines propriétés additionnelles.

et de laisser à un outil le soin d'extraire le parallélisme, plutôt que d'écrire explicitement le programme parallèle (en Occam) :

```
PAR
     PAR
          a := …
          b := …
          c := …
          d := …
     tmp1 := a + b
     tmp2 := c + d
som := tmp1 + tmp2
```

ce qui l'oblige à utiliser deux instructions **PAR**, deux variables auxiliaires et à fournir aussi un placement adéquat des tâches sur les processeurs. Cet exemple illustre bien que l'expression implicite du parallélisme est plus naturelle pour le programmeur (en contrepartie, il existe des exemples comportant beaucoup de séquencement qui sont pénibles à écrire dans un style déclaratif).

V.3.3.b. Extraction maximale du parallélisme

Le deuxième avantage du modèle data-flow, dérive directement du séquencement implicite du programme : cela permet potentiellement une exploitation optimale du parallélisme lors de la phase d'extraction et lors de l'exécution proprement dite. Un ordonnancement minimal des opérations est automatiquement déduit du programme ce qui permet en conséquence une expression maximale du parallélisme. Cet ordonnancement peut être inféré statiquement (i.e. par un compilateur, qui peut alors tenir compte des caractéristiques de la machine cible), ou dynamiquement (par le support d'exécution : interprète ou architecture data-flow [PLA 76] [GUR 85]).

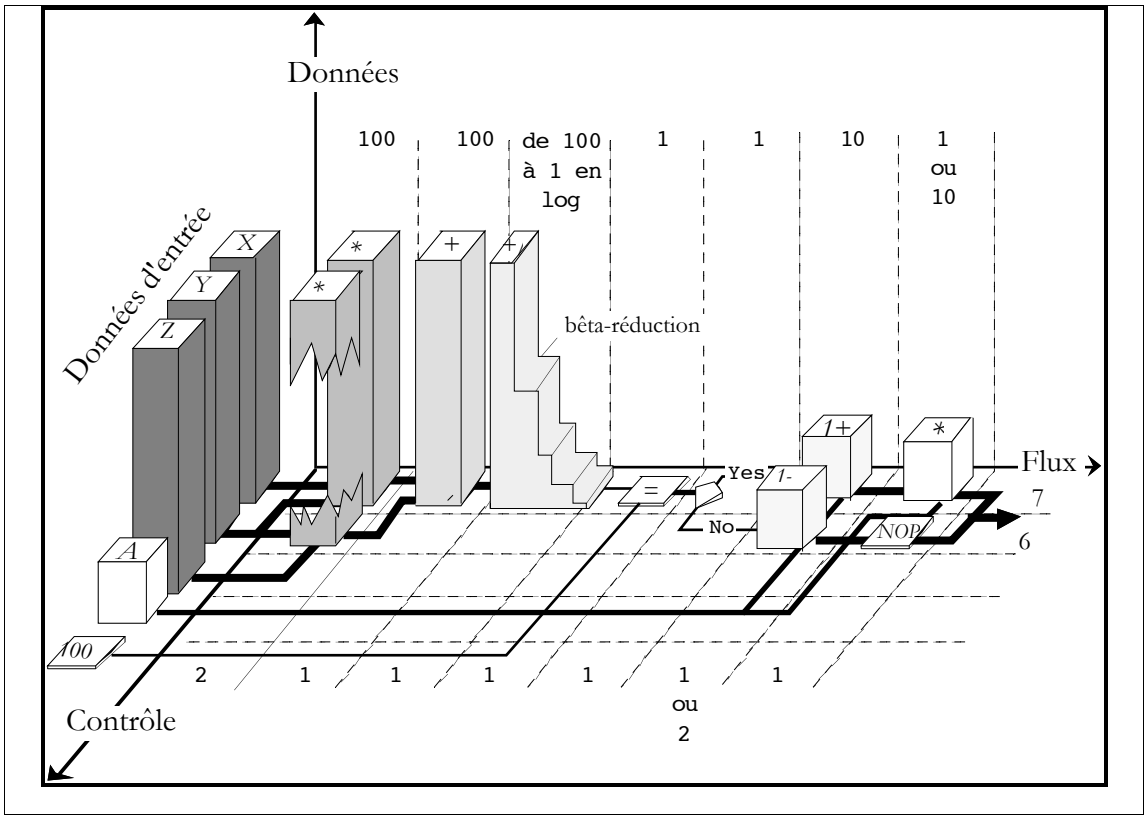


Figure V.6 : *Profil de parallélisme dans un graphe data-flow.*
*Le graphe data-flow correspond à l'expression 8*1/2 *«* `if 100 == +\(x*y + y*z) then A+1 else 1-A fi` *». Il est facile à partir de la forme data-flow de cette expression d'extraire le parallélisme des calculs.*
*Il y a trois sources de parallélisme, qui dans la figure sont représentées suivant trois axes. Suivant l'axe vertical, on a figuré le data-parallélisme, suivant l'axe horizontal, le parallélisme de flux, et suivant l'axe oblique le parallélisme de contrôle. Le parallélisme de données correspond au traitement parallèle des tableaux. Le parallélisme de contrôle correspond à la capacité de faire plusieurs choses différentes en même temps. Dans cette expression, il y a peu de parallélisme de contrôle. Le parallélisme de flux correspond à une exploitation en mode pipe-line des unités de calcul : si on a à évaluer l'expression pour plusieurs jeux de données (ce qui est le cas pour les streams 8*1/2*), alors si on dispose d'assez de ressources physiques de calcul, on peut commencer l'évaluation de l'expression pour le jeu de données n°* `i+1`*, alors que l'évaluation de l'expression pour le jeu de données n°* `i` *n'est pas encore achevée.*

La figure V.6 présente le "profil de parallélisme" d'un programme data-flow et data-parallèle :

- Le *parallélisme de contrôle* provient du graphe data-flow : il correspond à l'activation en parallèle de différent nœuds.
- Le *parallélisme de données* provient de l'exécution data-parallèle de nœuds dont les arguments sont des tableaux.
- Le *parallélisme de flux* provient de l'évaluation en mode pipe-line du programme pour toute une suite d'entrées.

On voit qu'avec une représentation data-flow data-parallèle, l'extraction de tout le parallélisme est immédiate et implicite. Avec un programme écrit dans un langage impératif parallèle, le programmeur aurait dû spécifier lui-même, à la main, *ce qui peut* s'effectuer en parallèle et *ce qui doit* s'effectuer en séquence. Un compilateur parallélisant part d'un programme impératif séquentiel (ou parallèle, dans le cas de compilateur restructurant) pour arriver à une forme similaire à un graphe data-flow (le graphes des dépendances). Mais cette transformation est difficile. Elle utilise des techniques sophistiquées d'analyse de références comme la

programmation paramétrique en nombre entiers [FEA 91] ou la résolution d'équations diophantiennes, qui sont coûteuses en temps d'exécution et qui ne sont pas toujours générales.

V.3.3.c. Déterminisme des résultats

Le troisième avantage des modèles data-flow provient du *déterminisme des résultats* d'un calcul. Dans un langage asynchrone comme Occam, l'expression du parallélisme passe par le non-déterminisme de l'exécution : le choix d'une commande gardée dans un `ALT` est théoriquement équitable et deux actions dans un `PAR` peuvent s'exécuter dans n'importe quel ordre (Cf. cependant []DIJ 88]) et donc éventuellement en parallèle. Afin d'obtenir le déterminisme des résultats du programme, le programmeur doit exprimer « du séquencement » par des sémaphores, des gardes, etc. S'il exprime trop de séquencement, il y a perte de parallélisme exploitable, par contre, s'il exprime trop peu de séquencement, le programme pourra calculer des résultats différents suivant les aléas de l'exécution. Un langage data-flow ne souffre pas de cet inconvénient : l'expression du séquencement est implicite et correspond à « juste ce qui est nécessaire » pour assurer un résultat déterminé.

V.3.3.d. Support formel

Un dernier avantage du data-flow provient de *la transparence référentielle* : un programme data-flow peut être vu comme un ensemble d'équations mathématiques et toute référence à une variable peut être remplacée par sa définition. Par conséquent, il est plus facile de vérifier et de raisonner formellement sur de tels programmes. Cela permet la transformation automatique des programmes, en vue de leur implémentation ou encore de leur optimisation (Cf. par exemple [LEI 83] et [WAT 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* [HOF 85].

V.3.4. Les inconvénients des langages data-flow

Nous allons voir dans cette section que le modèle data-flow classique, qui manipule des scalaires, n'est pas bien adapté au traitement des tableaux. De plus, les modèles data-flow classiques ont souvent utilisé un mode d'évaluation dynamique, dont la gestion coûteuse à l'exécution ne se justifie pas dans le cas des programmes à comportement statique qu'on rencontre dans le traitement numérique intensif.

V.3.4.a. Une mémoire à assignation unique

Dans un article célèbre [GAJ 82], Gajski & *al.* a critiqué l'approche data-flow vis-à-vis de la parallélisation automatique des programmes séquentiels. Si la sémantique fonctionnelle et l'absence d'effet de bord des programmes data-flow rend leur analyse facile par un compilateur, la gestion de l'assignation unique coûte très cher. Par exemple, elle implique en toute généralité l'utilisation d'un récupérateur dynamique d'espace mémoire (ou *garbage collector*). De plus elle ne permet pas une gestion efficace des tableaux.

Une règle comme l'assignation unique oblige conceptuellement à recopier tout un tableau chaque fois que l'on modifie la valeur *d'un seul* de ses éléments (cas de la modification d'un élément d'un tableau dans un corps de boucle par exemple). Cette recopie, nécessaire afin de garder au langage son caractère data-flow, peut le rendre complètement inefficace et donc inutilisable en pratique. La solution est de "regrouper" les mises à jour des éléments afin de les effectuer simultanément lors d'une opération globale sur tout le tableau.

Afin de répondre à ces critiques, les langages data-flow ont introduit des mécanismes permettant de mieux gérer les tableaux : les structures de données *mutables* comme les I-structures en Id [NIH 86], les expressions explicitement parallèles comme `forall` et `expand` en LAU, Val ou Sisal. De telles structures

permettent de dénoter par une expression unique des accès simultanés à un tableau. On voit qu'une réponse possible aux critiques de Gajski réside dans l'introduction d'opérations data-parallèles afin de manipuler les tableaux comme des touts, ce qui nous conduit à 81/2.

V.3.4.b. Modèle d'exécution dynamique versus modèle d'exécution statique

Mais l'argument principal de Gajski contre l'approche data-flow porte sur le modèle d'exécution dynamique, c'est-à-dire sur la détermination à l'exécution de l'ordre des calculs et sur la gestion dynamique de la mémoire.

Reporter la gestion des calculs pendant la phase d'exécution permet une évaluation théoriquement optimale du programme. On peut par exemple ne calculer que les résultats strictement nécessaires à l'élaboration de la valeur finale et qui dépendent des entrées du programme : on parle alors d'*évaluation paresseuse*. Mais cette gestion, théoriquement optimale, se paye d'après Gajski par un coût prohibitif du support d'exécution. En effet, 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 lorsqu'elle en a besoin). Cette disponibilité se paye en termes d'espace : messages, buffers, et de temps : attente active, délais de transmission, surcoût de gestion, etc.

C'est pourquoi Gajski défend la parallélisation automatique des langages séquentiels conventionnels car les compilateurs parallélisants déterminent statiquement les éléments nécessaires à la gestion de l'exécution et minimisent ainsi le coût de l'exécutif.

L'argumentation de Gajski repose sur l'analyse d'un exemple qui exhibe un *parallélisme statique*. 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[1] :

- 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.

On comprend alors l'argumentation : pourquoi payer le prix d'un exécutif dynamique si un exécutif statique suffit. Mais cette argumentation n'est valable que si le parallélisme présent dans une application est principalement de nature statique. Dans le cas contraire, il faudra de toute façon un modèle d'exécution dynamique.

Une analyse faite par [DEL 94] sur des programmes typiques de calcul numérique intensif (en FORTRAN) montre que la plupart des références à un tableau, qui se traduisent par des communications dans une exécution data-parallèle, sont des références statiques (Cf. figure V.7). L'hypothèse implicite faite par Gajski est donc correcte dans le cas du calcul numérique intensif : le parallélisme exhibé par ces applications est essentiellement un parallélisme de nature statique.

---

[1] Il y a bien sûr 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. [DAH 90]). Dans le modèle Acteur, un programme typique est complètement imprévisible : on ne peut pas prévoir le nombre d'acteurs qui sera créé ni qui communiquera avec qui.

V.3.4.c. Modèle d'exécution data-flow statique

Cependant la situation actuelle ne justifie plus la critique de Gajsky : en raison des problèmes d'efficacité liés à l'extraction dynamique du parallélisme, la compilation des réseaux data-flow s'est développée récemment, en particulier dans le domaine du traitement du signal [PAR 91], de la programmation temps-réel [HAL 91] [GAU 87], de la conception des algorithmes et des circuits systoliques[1] [QUI 89] [CHE 86] et de la parallélisation automatique [FEA 91]. Ces techniques permettent de développer des modèles d'exécution statique pour le data-flow et en particulier, pour le *data-flow itératif*. C'est d'ailleurs un des objectifs du projet 81/2.

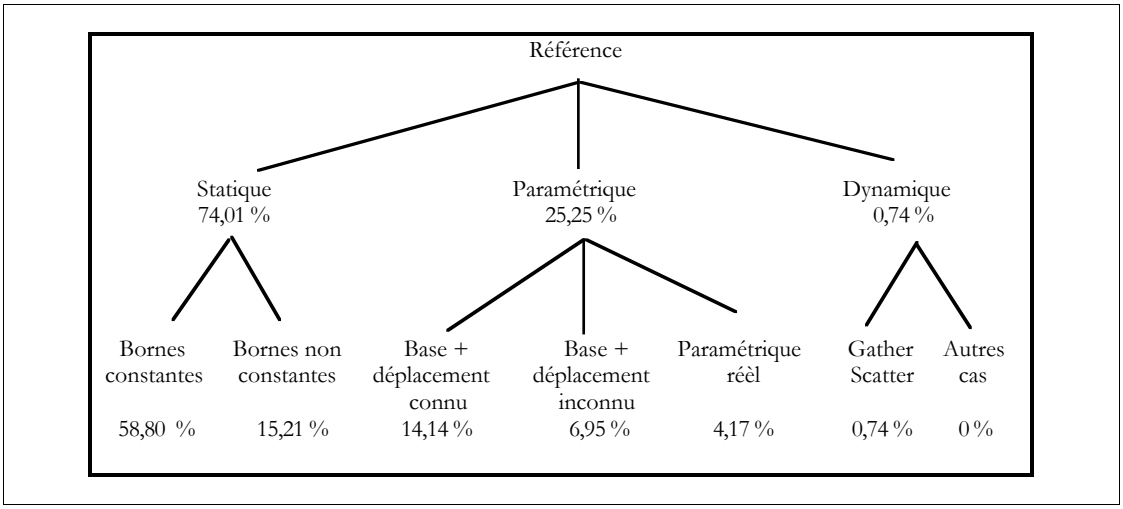


Figure V.7 : *Pourcentages approximatifs des classes de références aux données parallèles dans les applications de traitement numérique intensif. En pratique, on sait compiler les références statiques et on espère pouvoir compiler les références paramétriques.*
Les références à un tableau sont classées en fonction de leurs degré de prévisibilité. La référence à un tableau peut toujours se mettre sous la forme suivante : $\phi(I, V)$ où I est un vecteur d'indices de boucle `forall` *et V un vecteur de variables du programme. Les références statiques correspondent au cas où ϕ est une fonction affine de I uniquement. Les références paramétriques correspondent au cas où ϕ est une fonction affine de I et une translation en V (si de plus les variables de V sont des indices de boucle* `do` *alors le déplacement est connu). Les pourcentages donnés correspondent aux occurrences des expressions d'indexation des tableaux dans le texte des programmes Les programmes servant de base à cette analyse comprennent les routines LAPACK, des programmes de modélisation du climat, différents solveurs, etc. Cf. [DEL 94]*

V.3.5. Le modèle data-flow itératif : un modèle d'exécution statique

Afin de compiler un programme data-flow, il est nécessaire de connaître statiquement le graphe de ce programme. Cela interdit les *appels récursifs quelconques* de fonctions : en effet, un appel de fonction correspond à la création (dynamique) d'un (sous-) graphe data-flow. En présence de fonctions récursives, il n'est plus possible d'obtenir statiquement le graphe complet du programme par dépliage des appels fonctionnels. Si on restreint les schémas d'appels récursifs à une classe convenable, alors on peut développer des techniques d'analyse qui permettent de connaître statiquement le graphe des appels. Par exemple, dans le cas des algorithmes systoliques, on restreint les appels récursifs aux équations récursives linéaires et le graphe des appels correspond à un polyèdre convexe connu à la compilation.

[1] Un réseau systolique est généralement représenté par un système d'équations récurrentes linéaires. Ce système d'équations représente en fait la spécification d'une fonction et on peut voir le réseau systolique comme le graphe des appels à cette fonction.

Dans le cas du langage 81/2, nous avons restreint la récursion à la récursion temporelle et à la récursion spatiale. Cela nous permet de connaitre, à la compilation, un séquencement des calculs et l'implantation des objets calculés (par exemple l'espace mémoire nécessaire aux calculs). Les récursions temporelles et spatiales se traduisent alors à l'exécution par des itérations qu'il faut implémenter sur un ordinateur parallèle. C'est pourquoi nous parlons de *modèle data-flow itératif*. Les techniques mises en œuvre par le compilateur 81/2 pour déterminer automatiquement le placement des données et l'ordonnancement des itérations sur les ressources de l'architecture cible, seront présentées dans la dernière partie de ce chapitre.

V.4. 81/2 : les bénéfices d'un mariage data-flow + data-parallélisme

V.4.1. Un langage exprimant efficacement beaucoup de parallélisme utile

Nous venons de voir que

- le data-parallélisme excelle dans l'expression de l'espace,
- le data-flow dans l'expression des contraintes temporelles.

La mise en œuvre de d'un seul de ces deux modèles de programmation souffre donc de ne ne satisfaire qu'à une partie des besoins. Mais data-parallélisme et data-flow constituent des notions orthogonales et il est possible de les conjuguer : avec ces deux concepts réunis, on peut exprimer tout le parallélisme présent dans une application. De plus, chacun des deux modèles permet de remédier à des problèmes posés par l'autre.

L'introduction de structures de données data-parallèles dans un langage data-flow corrige certains des défauts des modèles data-flow initiaux, principalement en introduisant un traitement spécifique des tableaux. Par ailleurs, la collection est une structure de données qui est naturellement répartie, préoccupation habituellement absente de la problématique du data-flow classique. Inversement, le modèle d'exécution data-flow est une alternative au modèle d'exécution séquentiel des langages data-parallèles classiques, ce qui permet d'envisager des implémentations efficaces sur les nouvelles architectures massivement parallèles à contrôle hybride, cela d'autant plus que l'on développe des modèles d'exécution statiques. En particulier, l'expression à la fois du parallélisme de contrôle et du parallélisme de données permet au compilateur :

- de masquer les délais de communications par des calculs indépendants et
- de choisir les activités parallèles qui entraîneront un coût de gestion minimal.

V.4.3. Un langage parallèle de haut-niveau

En 81/2, c'est le compilateur qui est chargé de "plier" les calculs d'un programme, avec leur structure spatio-temporelle propre, sur une architecture donnée. Le programmeur n'a donc pas à prendre en compte l'architecture cible et celle-ci ne transparaît pas au niveau du programme.

Nous espérons avoir montré dans la première partie de ce chapitre que 81/2 constitue un modèle de programmation parallèle abstrait. Cela passe par l'implicitation du parallélisme et de la distribution des données qui sont des notions opératoires. Cette implicitation reste cependant en correspondance fidèle avec d'autres concepts dont le traitement est abstrait en 81/2 : le stream qui est en correspondance avec la notion de d'ordonnancement et la collection qui reflète la distribution et le parallélisme de données. Ces concepts

sont suffisamment abstraits et suffisamment universels pour être portable à travers différentes architectures parallèles. C'est ce qui permet de développer une programmation parallèle indépendante de l'architecture[1].

Le caractère *abstrait* d'un langage fondé sur des équations, ne nous semble pas un obstacle à son utilisation, comme le montre la large diffusion d'autres langages d'équations comme Prolog ou SQL. Ce style de programmation, qui se répand de plus en plus, semble aussi être particulièrement bien adapté aux numériciens comme le suggère les exemples de programmes que nous avons donnés dans le chapitre précédent, et plusieurs tentatives récentes tendent à vouloir programmer directement les architectures parallèles à partir de la description mathématique des problèmes à modéliser, plutôt qu'à travers un programme FORTRAN(Cf. [FRI 93] ou [ZIM 92]).

[1] Ou trouvera dans [SKI 90] le développement de cette idée et la preuve de sa justesse du point de vue de la complexité des programmes, pour le cas des collections.

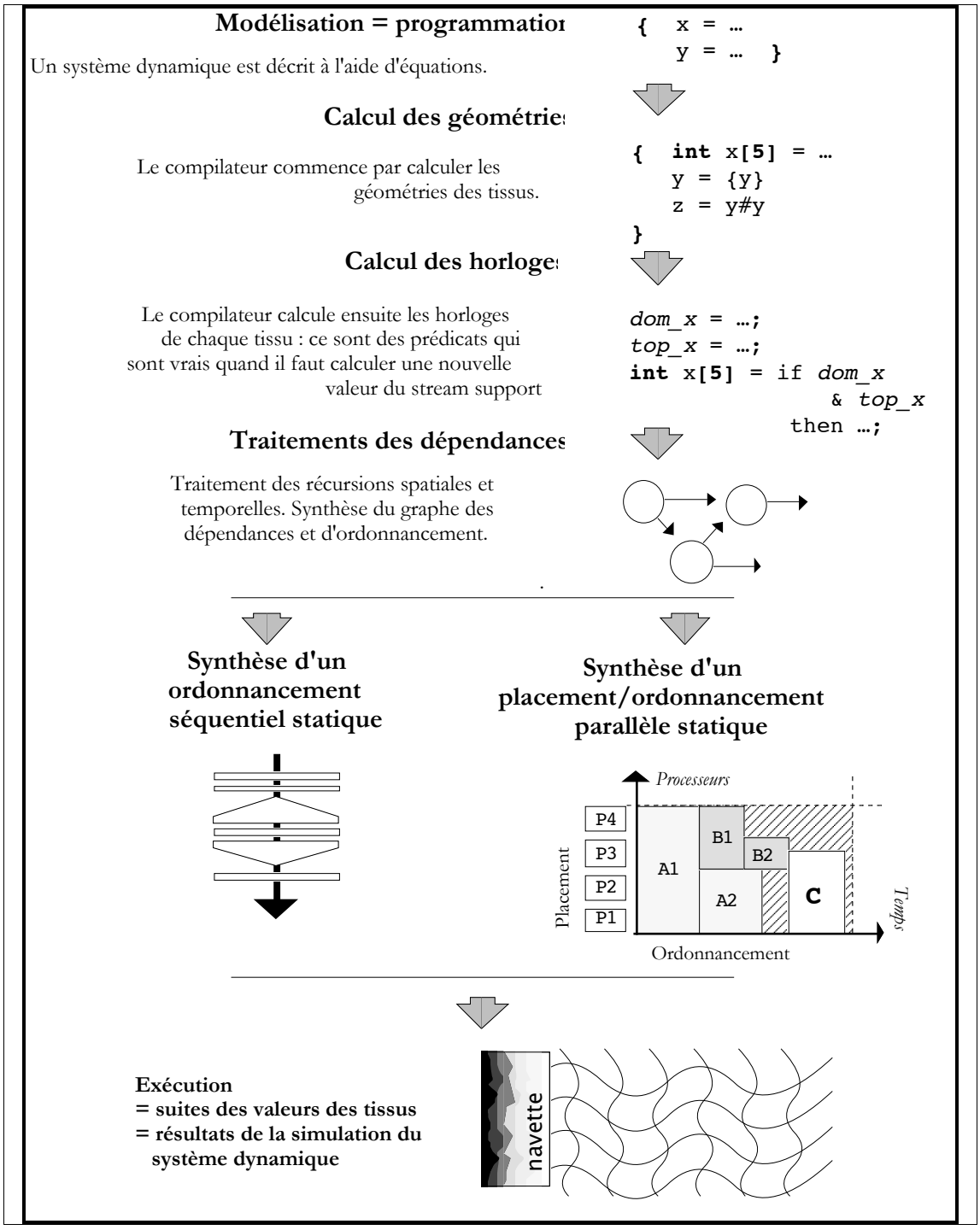


Figure V.8 : *Les étapes du processus de compilation. La navette est le code généré par le compilateur 81/2. Cette navette est soit séquentielle, soit parallèle. Dans le cas séquentiel, on obtient un ordonnancement séquentiel par tri topologique du graphe des dépendances et le code généré est un code séquentiel pour une machine virtuelle SIMD, ou bien, du code C. Dans le cas pleinement parallèle, on synthétise un placement et un ordonnancement parallèle pour une architecture hybride (MIMD, SPMD, MSIMD) à partir du graphe des dépendances, en tenant compte des communications et de la granularité des tâches requises par les caractéristiques de l'architecture. L'exécution d'une navette calcule la suite des valeurs des tissus d'un programme 81/2.*

V.5. Éléments d'implémentation du langage 81/2

Dans la dernière partie de ce chapitre, nous allons esquisser quelques éléments d'implémentation du langage 81/2 Après un panorama général du compilateur, nous présenterons une technique permettant d'éviter un ordonnancement dynamique, le calcul d'horloge, et une technique de placement automatique, qui utilise une heuristique issue des techniques de "bin-packing".

Le langage expérimental 81/2 a été conçu pour être compilé, tant pour des machines séquentielles que pour des machines parallèles, en imposant pour la version 1.0 un modèle d'exécution data-flow qui est totalement statique. Cela veut dire que la réservation mémoire et l'ordonnancement des instructions sont connus dès la compilation. Le code produit ne nécessite pas d'exécutif ni de pile d'exécution.

Le compilateur traduit un programme 81/2 en une *navette*. Une navette est le code machine dont l'exécution permet de calculer et d'énumérer dans l'ordre temporel croissant les valeurs des tissus définis par le programme. Le compilateur est grossièrement composé de quatre phases : la synthèse et la vérification du typage des expressions, le calcul de l'horloge des expressions, le calcul des dépendances des tâches associées aux expressions, le placement et l'ordonnancement statiques de ces tâches (Cf. figure V.8 ci contre).

Les grands traits de la stratégie de compilation sont les suivants :

- Les collections homogènes vont se traduire par des vecteurs dont les éléments seront manipulés par les processeurs de la machine cible.
- La notion de système s'implémentent en termes d'environnement. Mais puisque nous avons restreint la version 1.0 du langage aux géométries statiques, la liaison des variables à leur valeur est statiquement résolue par le compilateur et ne donne lieu à aucun traitement particulier dans le code généré.
- L'écoulement du temps correspond à une seule grande boucle tant-que. Le corps de cette boucle tant-que correspond à la gestion des calculs (parallèles) qui implémentent l'évaluation des valeurs à un tic donné.
- L'implémentation de l'activité des streams passe par la synthèse de gardes conditionnant tout calcul correspondant à l'évaluation d'une expression : une expression ne doit être évaluée que si le tic courant correspond à top. La synthèse de ces gardes est appelée *calcul d'horloge*.
- Il faut déterminer ensuite les dépendances entre les calculs : c'est *le calcul des dépendances* des tâches.
- Enfin, il faut *ordonnancer* en respectant les dépendances et *placer* les calculs sur les processeurs de la machine cible.

V.5.1. Calcul de l'horloge d'une expression

Le calcul des valeurs d'un tissu lors de l'exécution d'un programme 81/2 correspond aux calculs des valeurs des collections successives qui composent le stream support du tissu. La valeur d'un tissu à un instant donné est une collection (de valeurs scalaires) appelée *valeur instantanée* du tissu.

Le calcul de l'horloge h**T** d'un tissu **T** (Cf. le chapitre II) permet de décider si le calcul d'une valeur instantanée doit ou non prendre place à un tic donné. Pour permettre le calcul des horloges, il faut raisonner sur les *tissus normalisés* (v**T**, h**T**).

Le calcul de l'horloge d'une expression se fait simplement par transformation du programme initial. Chaque définition de la forme :

```
X = f(Y)
```

est transformée en

```
vX = if hx then f(vY)                                    (1)
```

où h**X** est une expression définissant l'horloge du tissu **X**. Cette expression est synthétisée par induction sur la structure de la définition de **X**. Par exemple,

```
h(A when B) = vB ∧ hB
```

(d'autres exemples de calculs d'horloges ont été donnés dans le chapitre II).

Il se peut que le système d'équations qui définit les horloges présente un cycle. Si on prend par exemple :

```
S@0 = X,
S = X + $S
```

en supposant que la définition de **X** ne fait pas intervenir **S**, on obtient le système d'équations suivant pour l'horloge de **S** :

```
hS = h@0(X) v ( ¬h@0(X) ∧ (hX v hS) )
```

Le prédicat h**@n(X)** est un stream booléen normalisé qui a la valeur **vrai** uniquement au $n^{ème}$ top de **X**. L'horloge de **S** est la disjonction de deux horloges qui correspondent à la définition quantifiée par **@0** et à la définition non quantifiée. L'horloge de cette dernière est la conjonction d'une garde (qui est la négation des horloges des définitions quantifiées) et de l'horloge du membre droit de la définition non quantifiée. Par ailleurs, l'horloge de **$S** est l'horloge de **S** sur le domaine de **$S**, ce qui explique l'expression « h**X v** h**S** » plutôt que « h**X v** h$**S** ». Pour simplifier notre exposé, ignorons le système de gardes. Il reste une équation de définition de l'horloge de **S** qui est de la forme :

```
hS = hX v hS                                             (2)
```

Cette définition est récursive. Pour résoudre ce système, on utilise une *interprétation abstraite* (Cf. [COU 77] [OKE 87]) qui permet de calculer à la compilation une expression non récursive de l'horloge, en accord avec la sémantique choisie. Cela veut dire que l'horloge du tissu solution qui est calculée, sera la plus petite au sens d'un certain ordre sur l'ensemble horloges, Cf chapitre II et [Gia 91b]. Dans notre exemple, le prédicat qui sera finalement calculé pour l'horloge de **S** sera :

```
hS = hX                                                  (3)
```

On peut vérifier que ce prédicat est bien une solution du système précédent puisque :

hS	=	hX	d'après (3)
	=	hX v hX	disjonction
	=	hX v hS	d'après (3)
	=	(2)	

Le calcul des horloges des tissus d'un programme 81/2 repose sur l'évaluation abstraite des expressions des horloges à partir de la sémantique dénotationelle du langage. Un des avantages de cette approche, outre le fait qu'elle soit complètement statique, est qu'il est aussi possible de détecter certaines expressions qui resteront constantes (on peut alors optimiser le code généré), ou bien qui ne produiront aucun calcul (et qui correspondent à des tâches en étreinte fatale, ce qui est certainement une erreur de programmation).

On obtient ainsi un nouveau programme, *le programme en forme normale*. Le programme en forme normale contient des calculs correspondant à l'évaluation des streams d'horloge et des calculs correspondant à la prodution des valeurs des tissus, les premiers conditionnant les seconds, à la manière de l'expression (1).

Grossièrement, le compilateur génère pour chaque expression de ce programme une tâche. Mais il reste à calculer explicitement les dépendances entre les tâches afin de déterminer les séquences d'activation de celles-ci.

V.5.2. Le graphe des dépendances des expressions

Le graphe data-flow associé à un programme 81/2 est extrait immédiatement du programme en forme normale. Hélas ce graphe ne peut être employé tel quel par le compilateur pour générer l'ordonnancement des tâches sur les processeurs. En effet, si dans le cas des programmes data-flow scalaires, le graphe data-flow correspond aussi au graphe de dépendances des expressions, ce n'est plus le cas en présence de collections car une collection représente un ensemble de scalaires. Par exemple, dans le programme 81/2 suivant :

```
A = B
```

chaque élément de la collection support du tissu **A** dépend de l'élément correspondant de **B** et de lui uniquement. Par contre, le programme suivant, qui correspond à la somme de tous les éléments de **B**,

```
A = +\B
```

produit un tissu **A** composé d'un seul élément qui dépend de tous les points de **B**. Pourtant les deux programmes se traduisent par le même graphe data-flow où les nœuds associés à **A** et à **B** sont reliés.

Le graphe data-flow d'un programme 81/2 n'est donc pas le graphe des dépendances des expressions du programme. Par contre, on peut le voir comme une approximation de ce graphe, si on traduit chaque arc du graphe data-flow comme une dépendance entre deux éléments quelconques des collections calculées au nœud source et au nœud but de l'arc. Un graphe *G* est une *approximation du graphe des dépendances*, si celui-ci est un sous-graphe de *G*. L'intéret d'une approximation provient du fait que le respect des contraintes de l'approximation entraîne automatiquement le respect des contraintes du graphe des dépendances. En pratique, l*e graphe des dépendances ne peut pas être construit explicitement* car il contient autant de nœuds qu'il y a de points dans les tissus du programme. Par exemple, en calcul numérique, il est fréquent de rencontrer des matrices 1000×1000, ce qui impliquerait des graphes de plus d'un million de nœuds pour chaque opération. Par contre, il peut être plus facile ou plus praticable de calculer une approximation.

En tant qu'approximation du graphe des dépendances, le graphe data-flow d'un programme 81/2 est trop grossier ; par exemple, on ne peut pas implémenter le programme 81/2 suivant :

```
iota = 0 # (1 + iota:[9])
```

sur sa seule base. En effet, le graphe data-flow de ce programme est cyclique (Cf. figure V.9) ce qui empêche son ordonnancement.

Le travail du compilateur est d'annoter le graphe data-flow afin d'obtenir une approximation plus fine du graphe des dépendances.

Nous appellerons *graphe de séquencement des tâches* l'approximation du graphe des dépendances obtenue par annotation du graphe data-flow de la manière suivante (Cf. Figure V.9) :

- Une expression quelconque **e** dépend du tissu **x** si **x** apparaît syntaxiquement dans **e**. Toutefois, on enlève les dépendances qui correspondent à des variables apparaissant dans la portée d'un opérateur de délai : ces dépendances correspondent à une dépendance par rapport à une valeur produite dans le passé (elle est donc satisfaite de facto) et non pas par rapport à une valeur instantanée d'un tissu.

– La dépendance (instantanée) entre une expression et une variable est étiquetée « `p` » si la valeur en un point `i` de `e` dépend uniquement de la valeur du point `i` de `x` (on l'appelle alors une dépendance point à point).

– La dépendance est étiquetée « `t` » si un point `i` de `e` dépend de la valeur de tous les points de `x` (on l'appelle alors une dépendance totale).

– La dépendance est étiquetée « `+` » si la valeur du point `i` dépend de la valeur des points `j` de `x` avec `j<i` (on l'appelle alors une dépendance positive).

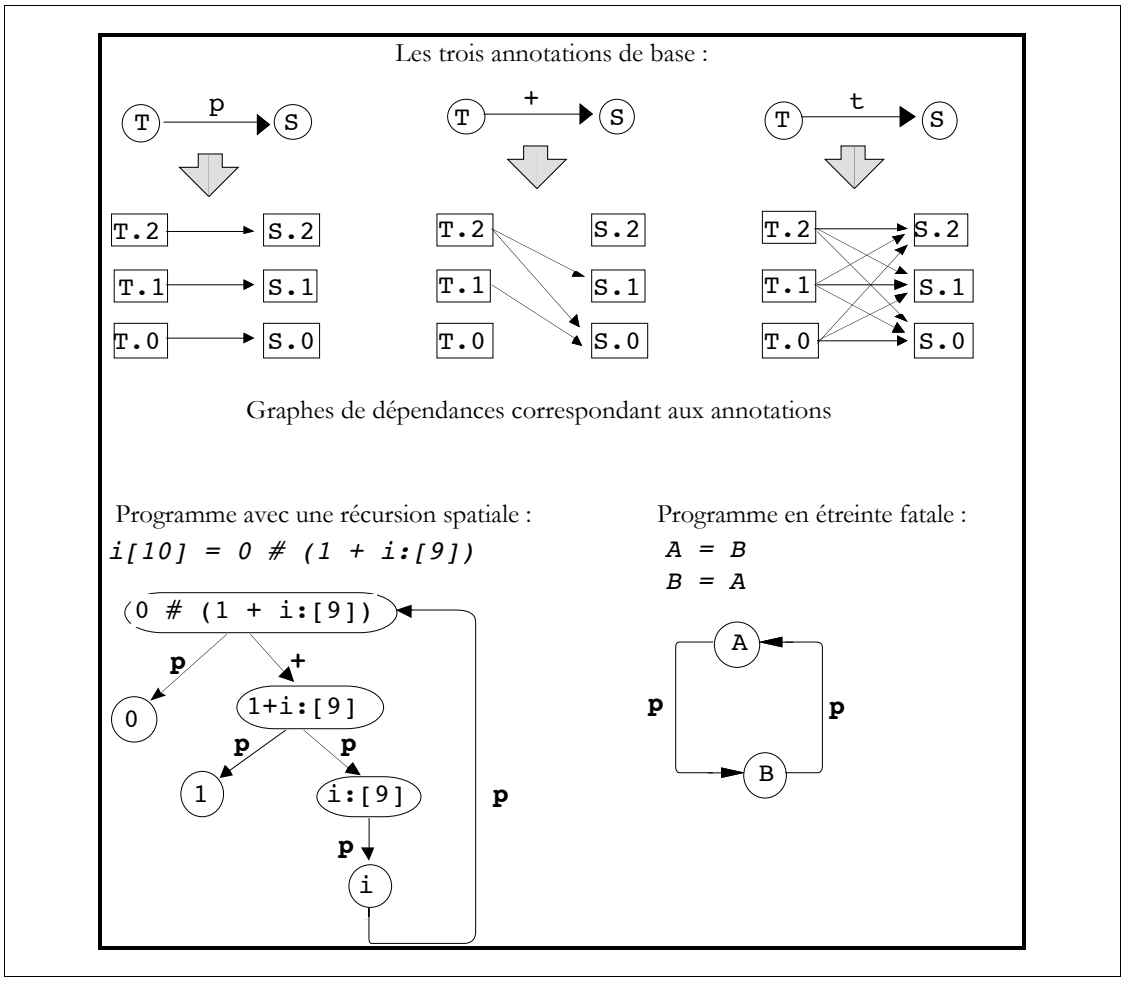


Figure V.9 : *Représentation des trois types d'annotations utilisées pour construire les graphes de séquencement. Application à deux exemples.*

Dans le graphe de séquencement, les cycles contenant un arc de type `t` ou bien aucun arc de type `+`, correspondent à des expressions en etreinte fatale. Ces cycles sont appelés *cycles instantanés*. Les tissus définis dans les cycles instantanés voient leur valeurs toujours indéfinies. Les cycles restant (qui comportent des arcs `+` et pas d'arc `t`) correspondent à des expressions récursives spatiales qui nécessitent une implémentation séquentielle. Les expressions qui n'apparaissent pas dans un cycle sont des expressions data-parallèles. Elles peuvent être calculées dès que les expressions dont elles dépendent ont été calculées.

Le graphe de séquencement des tâches étant une approximation du graphe réel des dépendances, il se peut que l'on détecte comme incorrects des programmes qui ont une valeur bien définie. Par exemple, avec l'approximation précédente :

```
T[2] = { T.1, 0 };
```

le séquencement de `T` dépend totalement de lui-même car `T.`*`i`* dépend totalement de `T`. Par suite, ce programme présente un graphe de séquencement contenant un cycle total, alors que la sémantique du langage donne la solution `{ 0, 0 }`.

D'autres approximations plus fines que le graphe de séquencement sont possibles[1] afin de ne pas rejeter un tel programme, mais leur mise en œuvre est plus gourmande en temps de calcul [GIA 91b].

Notons que ce que nous avons interprété comme une approximation du graphe des dépendances, peut se voir comme la preuve de la "productivité" d'une définition récursive (Cf. § III et [SIJ 89]). La notion de productivité d'une définition récursive est liée au calcul d'un élément maximal dans le domaine sémantique. Donc, bien que le traitement présenté semble faire appel à des concepts opérationnels (l'existence d'un ordre total sur le graphe des dépendances), notre traitement peut se reformuler entièrement en termes de sémantique dénotationelle abstraite et s'interpréter en termes de résolution d'équations.

V.5.3. Placement et ordonnancement statiques des tâches

Une fois le graphe de séquencement des tâches établi, le compilateur peut placer les tâches sur les processeurs d'une machine cible et choisir sur chaque processeur un ordonnancement compatible avec le graphe de séquencement. Pour résoudre ce problème, nous nous limitons à un type d'ordonnancement qui prend une forme particulière : *l'ordonnancement cyclique*. Dans ce cas, un tel ordonnancement correspond à la répétition par les processeurs (répétition éventuellement infinie) de l'exécution d'un morceau de code que nous appelons *navette*. Ce morceau de code correspond à l'ordonnancement et au placement de chaque expression d'un programme 81/2 : ce placement et cet ordonnancement est appelé*motif*. Pour pouvoir générer la navette, le compilateur doit donc générer ce motif.

Pour générer un motif, le compilateur associe à chaque tâche un rectangle dans un espace (temps × processeur). La largeur du rectangle correspond au temps d'exécution de la tâche et sa hauteur au nombre de processeurs idéalement requis pour une exécution pleinement parallèle de la tâche (Cf. figure V.10). Par exemple, si la tâche correspond à l'addition data-parallèle de deux tableaux de 100 éléments, la hauteur du rectangle associé sera de 100.

Avec cette représentation, le problème de la distribution optimale et de l'ordonnancement optimal des tâches revient à trouver un placement des rectangles qui minimise la longueur des empilements et qui est borné en hauteur par le nombre de processeurs de la machine cible. Des heuristiques très efficaces existent pour ce problème d'empilement de rectangles qui est NP-complet dans le cas général, et qui est connu sous le nom de « bin-packing » en deux dimensions [GAR 78].

---

[1] Dans le compilateur actuel, l'approximation mise en œuvre est une légère amélioration de celle que nous avons présentée : elle permet d'accepter l'exemple ci-dessus et elle n'est pas beaucoup plus coûteuse. Il s'agit principalement de remplacer la dépendance `+` par une dépendance `+`n ou `-`n, le nombre n représentant un décalage de la collection argument de n crans vers la droite (dépendance `+`) ou vers la gauche (dépendance `-`) par rapport à la collection résultat.

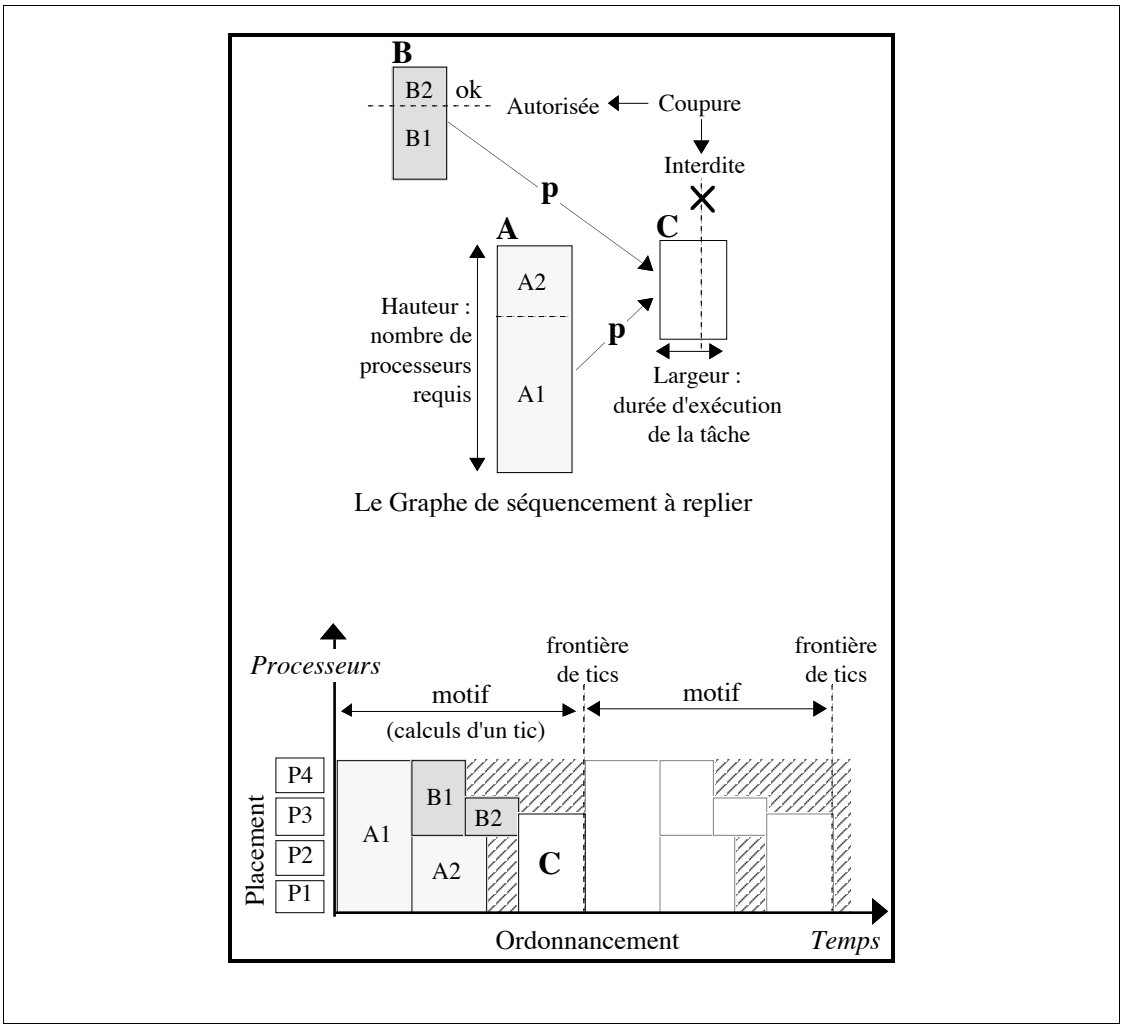


Figure V.10 : *Ordonnancement et placement d'un graphe de séquencement par une méthode de bin-packing en deux dimensions.*

Nous testons actuellement une stratégie gloutonne [MAH 92] consistant à placer au plus tôt la plus grande part possible d'une des tâches disponibles le plus tôt : une tâche devient disponible à la date de terminaison des tâches dont elle dépend, date augmentée des temps de communication nécessaires aux transferts des données entre les processeurs ; si plusieurs tâches sont disponibles en même temps, un critère de choix additionnel est utilisé comme par exemple toujours choisir la tâche se trouvant sur le chemin critique ; si la largeur de la tâche choisie excède le nombre de processeurs libres, on "coupe" la tâche en une première partie qui est ordonnancée et en une seconde partie qui est remise dans l'ensemble des tâches disponibles (afin d'être ordonnancée et placée ultérieurement). Nous admettrons qu'il est possible de couper en deux parties dans le sens horizontal un rectangle de calcul (Cf. figure V.10). En fait, c'est possible car les tâches data-parallèles requérant *n* processeurs correspondent à *n* tâches scalaires indépendantes.

Un résultat établi dans [HAW 89] peut être directement utilisé pour borner inférieurement les performances de cette stratégie. Ce résultat borne le comportement au pire et garantit la très bonne qualité de l'heuristique qui est employée ici.

€

Perspectives : la version 2.0 du langage 8½

Pour terminer ce document, nous évoquons quelques unes des directions de recherche que nous poursuivons. Ces directions s'articulent suivant les trois axes qui définissent le langage : un langage de programmation *parallèle* et de *haut-niveau* pour la modélisation et simulation des *systèmes dynamiques*.

Modélisation et simulation des systèmes dynamiques

Géométrie dynamique et modélisation des phénomènes de croissance

Nous avons choisi dans le chapitre I de ne pas nous préoccuper des relations dynamique entre temps et espace. Cette propriété se traduit entre autre par le fait que la géométrie de la collection support du tissu doit rester identique au cours du temps. Cette propriété n'est pas une propriété vérifiée a priori par un programme 81/2 et nous avons vu dans le chapitre IV, à travers l'exemple du triangle de Pascal, un programme 81/2 dont la géométrie varie au cours du temps.

Les tissus à géométrie variable, ou encore à géométrie dynamique, ont beaucoup de domaines d'applications :

- *physique* : résolution numériques des équations aux dérivées partielles par des méthodes adaptatives, modèles multi-échelles, … ;
- *automatique* : système multiphases ;
- *morphogénèse* : en biologie (croissance des plantes L-systèmes), en physique (croissance des cristaux), en mathématiques (pavage) ;
- *informatique* : optimisation combinatoire, exploration des arbres d'états, analyse de données, simulations paramétriques, systèmes de réécriture parallèle.

Par ailleurs, nous n'avons pas étudié la notion de voisinage : in fine, une collection permet d'implémenter n'importe quelle structure de voisinage à travers l'accès indifférencié à tous les éléments de la collection. La voie qui consiste au contraire à étudier les différentes géométries permises par différents types de voisinages, n'a pas encore été abordée.

Représentation et construction d'un espace et d'un temps continus

Le temps est représenté en 81/2 à travers la notion de stream, et l'espace est représenté à travers la notion de collection. Ces deux structures de données et les opérations qui les accompagnent, correspondent à une représentation essentiellement événementielle et finie du temps et à une représentation discrète et finie de l'espace. Il est toujours possible de discrétiser un système continu afin de le simuler de manière discrète sur un ordinateur numérique. Cependant, l'expression directe de modèles continus et leur traitement spécifique offre au programmeur un langage de très haut-niveau d'expression. De plus, les modèles de simulation *combinés* discrets-continus sont de plus en plus nécessaires, par exemple si l'on veut modéliser aisément des systèmes qui combinent des processus naturels continus avec une commande discrète (contrôle/commande industriel par exemple).

Notre but est donc de développer des représentations continues du temps et de l'espace. Trois approches semblent pouvoir être explorées.

La première consiste à permettre l'insertion arbitraire d'instants entre deux instants donnés (ou l'insertion d'un point d'espace entre deux points donnés). Une forme naïve de cette approche a déjà été explorée à travers la hiérarchisation des collections. Une proposition pour un temps hiérarchique, fondé sur la notion de sous-stream, a aussi été avancée avec l'opérateur `every`. Mais les conséquences d'une telle introduction doivent être étudiées de manière plus approfondie.

La deuxième approche consiste à introduire un nouveau type de données scalaires correspondant aux variables temporellement ou/et spatialement continues. L'évolution de ces variables est décrite par des équations différentielles ou/et aux dérivées partielles. Il faudra résoudre deux principaux types de problèmes : l'intégration des équations de définition des variables continues et l'interaction entre les variables continues et les variables discrètes. Cette interaction est de deux types : la loi de comportement d'une variable continue peut subir un changement dépendant d'une variable discrète (changement de régime, de phase, commande "bang-bang", etc.) et un événement discret peut être déclenché par une variable continue lorsque sa valeur atteint un certain seuil.

La troisième approche du continu se base sur les concepts développés par l'analyse non-standard qui modélise le continu à travers le discret et la notion d'échelle de grandeur. Cette approche a déjà été utilisée avec succès en informatique graphique pour la discrétisation de courbes continues et leur tracé digitalisé.

Représentation et construction d'un temps asynchrone

Le modèle de temps développé en 81/2 à travers les streams est un modèle de temps synchrone. La compilation d'un programme 81/2 conduit donc à la synthèse d'une fonction dont l'itération produit les résultats de la simulation. Cette fonction correspond en fait à la fonction d'évolution du système dynamique modélisé par le programme. L'évolution est synchrone en ce sens que

$$\forall\, t \in I\!N,\ 1 \le i \le n,\ x_i\,(t+1) = f_i\,(x_1\,(t), \ldots, x_n\,(t))$$

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* (encore appelé *itérations chaotiques)*. 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 :

$$\forall\, t \in T_i,\ 1 \le i \le n,\ x_i(t+1) = f_i\,(x_1(\tau_1^i(t)), \ldots, x_n(\tau_n^i(t)))$$
$$\forall\, t \notin T_i,\ 1 \le i \le n,\ x_i(t+1) = x_i\,(t)$$
$$\forall\, t \in \bigcup T_i,\ 1 \le i, j \le n,\ 0 \le \tau_j^i(t) \le t$$

Ce schéma d'itération esr appelé asynchrone car la mise à jour de la valeur des variables se fait de manière asynchrone. Les fonctions τ_j^i correspondent à l'*horizon temporel* du calcul de la variable x_i : elles décrivent comment le processus de calcul de x_i perçoit la valeur des autres variables au cours du temps. L'itération asynchrone est le mode de fonctionnement primitif des réseaux de neurones formels de type Hopfield. C'est aussi le schéma utilisé en physique par beaucoup de simulations dites de Monte-Carlo, ou par le fonctionnement des circuits VLSI asynchrones. Ce modèle d'exécution est aussi utilisé pour rendre compte d'un délai de communication non-déterministe dans un réseau.

Le but est de développer un modèle d'exécution asynchrone pour 81/2 et d'étudier sous quelles conditions on peut relier les calculs effectués par le même programme dans les deux modes d'exécution. Savoir relier le comportement des deux modèles permettrait de simplifier l'analyse de nombreux systèmes et de rendre moins coûteuse l'implémentation des synchronisations des programmes 81/2 (par exemple en les supprimant).

La programmation paramétrique

Les réseaux de neurones formels ou les automates cellulaires nous ont habitué à l'idée qu'un mécanisme de calcul simple et extrêmement régulier pouvait néanmoins être suffisamment puissant pour servir de modèle de programmation. Le langage 81/2, par sa capacité à manipuler des collections comme des touts, est particulièrement bien adapté à la simulation de tel systèmes. Ces simulations conduisent à considérer des programmes uniformes et homogènes. Ces programmes correspondent aux membres d'une même famille et diffèrent entre eux par la valeurs de paramètres.

L'étude de ces familles de programmes, dont la structure est à préciser[1], nous intéresse pour plusieurs raisons :

- Programmer un tel système revient à déterminer les paramètres du "bon" programme. L'activité de programmation prend alors une tournure différente. On peut illustrer notre propos par l'exemple des réseaux de neurones formels, où la programmation du réseau se fait à travers un algorithme d'apprentissage qui permet d'ajuster les paramètres.
- Il est possible d'étudier la structure de la famille, sa topologie, quand on se déplace dans l'espace des paramètres. Cela permet d'introduire précisément des notions de robustesse d'un programme, de transformations continues d'un programme en un autre, etc.
- Un membre de la famille étant repéré par des paramètres numériques, et l'exécution d'un programme donnant des résultats numériques, il y a une dualité entre programmes et données. Cette dualité ne semble pas s'interpréter simplement en termes de lambda-calcul. Les familles paramétriques sont donc une nouvelle voie pour explorer les équivalences entre programmes et données.

[1] Par exemple, cette structure pourrait être celle engendrée par des opérateurs similaires à ceux impliqués par les opérations de calculs tensorielles, qui permettent de manipuler de manière régulière des tableaux structurés.

– Une famille paramétrique présentant une structure régulière, il est possible d'en tirer des propriétés utiles et intéressantes pour la compilation. Par exemple, les prédicats logique peuvent tous s'écrire sous la forme d'une somme de produits ; les circuits de type PLA implémentent matériellement sur VLSI la famille paramétrique des sommes de produits de booléens.

Un langage de haut-niveau

La simulation graphique et interactive

Actuellement, un programme 81/2 est un texte ASCII décrivant un système dynamique et l'exécution de ce programme correspond à la simulation du système dont on peut observer l'évolution grâce à un outil d'affichage graphique des résultats.

La plupart des outils de simulation vont plus loin dans l'utilisation du medium graphique. Par exemple Extend™ ou Interactive-Physics™ permettent la construction graphique des modèles de simulation. Avec Extend, le programmeur construit graphiquement une représentation hiérarchique de son modèle sous la forme d'un graphe data-flow. Avec Interactive-Physics, le programmeur construit un modèle de système mécanique en assemblant les différents objets de la simulation (ressorts, poids, amortisseurs, fils, axes, etc.). La simulation elle-même va animer ces différents objets.

Nous souhaiterions développer un environnement *graphique* et *interactif* de simulation de haut-niveau. Pour cela il faudrait développer des éditeurs graphiques de programmes 81/2 et des interactions avec la simulation en cours de fonctionnement. Dans un premier temps, il s'agirait de développer la notion de *stream externe*. Les valeurs d'un stream externe ne sont pas définies par une équation 81/2 mais proviennent d'une entité extérieure au système 81/2 : fichier, processus UNIX, autre programme 81/2, fenêtre interactive de dialogue avec l'utilisateur…

Dans un deuxième temps, nous voulons imaginer et développer un système qui permette, interactivement, d'attacher à des fragments de programmes 81/2 une représentation et une animation de cette représentation. Une fois cet attachement fait, il sera possible de créer graphiquement des programmes 81/2 et d'observer leur exécution à travers une animation.

Typage temporel

Un stream 81/2 correspond à une suite de valeurs dans le temps. Ces valeurs sont produites à certains instants que l'on appelle des tops. L'ensemble de tous les tops de tous les streams d'un programme correspond aux instants d'une horloge globale, instants qui sont désignés sous le nom de tics. Pour chaque stream, le compilateur 81/2 synthétise des prédicats qui indiquent si le tic est un top du stream. Ces prédicats correspondent à un premier système de typage temporel. Un *type temporel* est un type qui décrit la structure des calculs d'un stream dans le temps. Si deux streams ont le même type temporel, alors ils sont synchrones. Le calcul d'un type temporel correspond au calcul d'horloges de SIGNAL ou de LUSTRE, mais il est étendu à des expressions non-synchrones c'est-à-dire que le programmeur peut combiner librement des streams qui ont des ensembles de tops différents.

Nous voulons à présent développer un système de typage temporel correspondant à une interprétation abstraite plus globale : il s'agit de déterminer les segments temporels sur lesquels sont localisées les valeurs d'un stream. En effet, certains opérateurs temporels 81/2 permettent de geler le "démarrage" d'un stream, de geler sa progression ou de ne s'intéresser qu'à la production d'une seule valeur. Une application immédiate de

ce système de typage est de détecter certaines étreintes fatales et de déterminer des activités exclusives l'une de l'autre et donc d'allouer plus efficacement les ressources de calcul (CPU, mémoire).

Un langage parallèle

Étude et implémentation d'un opérateur de séquencement temporel

En programmation déclarative, on utilise la récursion pour rendre compte de la répétition des calculs. La récursion temporelle s'exprime en 81/2 à travers un opérateur de délai (ou retard) noté `$`. Une suite retardée correspond à une suite décalée d'un cran vers la droite. Par exemple, l'équation « `T = $T + 1` » exprime une suite dont la valeur à chaque instant est égale à la valeur à l'instant précédent augmentée de 1.

Ce type de récursion temporelle porte sur le stream en son entier. Il devient alors difficile et artificiel d'exprimer des répétitions dépendant de façons complexes de conditions logiques. Pour exprimer plus naturellement les programmes où le séquencement est important, nous voulons introduire un opérateur permettant un nouveau type de récursion temporelle. Pour cela, il faut raffiner le concept de stream afin de distinguer *les streams à support borné*. Les streams à support borné sont ceux qui ont un nombre fini de tops. Le nouvel opérateur de séquencement « `→` » permet de séquencer des streams à support borné. On retrouve ainsi un formalisme proche de CSP et on peut directement écrire des équations comme : « `a = 1 → 2 → a` » qui génère le stream `<1, 2, 1, 2, 1 …<`. Avec ce type de formalisme, il sera possible par exemple d'exprimer les mêmes séquencements qu'en OCCAM.

Ordonnancement conditionnel et extension des expressions récursives effectives

L'exécution d'un programme 81/2 consiste à résoudre parallèlement un système d'équations. De manière imagée, la compilation consiste à donner une forme triangulaire à ce système d'équations afin de le résoudre facilement, de manière analogue à la résolution d'un système d'équations linéaires par la méthode du pivot de Gauss qui commence par trianguláriser la matrice du système avant de le résoudre. Les expressions récursives correspondent à des cycles dans le graphe des dépendances des équations et empêchent la triangularisation. Actuellement le compilateur traite les cycles mais souffre de deux limitations : la détection de faux cycle et la détection conservatrice d'un sous-ensemble des équations récursives effectives.

La détection des cycles se fait actuellement sur la base d'un critère purement syntaxique. Cela amène par exemple à considérer comme cycliques les deux expressions suivantes :

```
B = if A then 1 else C
C = if A then B else 2
```

Si `A` est `vrai`, alors le graphe des dépendances est « `A → B → C` » alors que si `A` est faux, le graphe des dépendance est `A → C → B`. Comme le graphe des dépendances actuellement construit ne tient pas compte de la valeur de `A`, on détecte un cycle `B ↔ C`. Il s'agit donc, dans un premier temps, d'inventer et de développer un concept de graphe des dépendances conditionnelles, qui permette de traiter correctement ce cas de figure, et d'étendre la méthode à toutes les fonctions partielles.

Le compilateur 81/2 sait traiter certains cycles correspondant à des expressions récursives *effectives*. Une expression récursive effective correspond à la définition d'un objet unique dont chaque élément est calculable en un temps fini par substitution de valeurs. Par exemple, l'équation suivante, qui correspond à un tableau de 10 éléments discrétisant une fonction harmonique, est effective :

```
f(0) = 0
f(1) = 1
f(n) = 2f(n-1) - f(n-2), 2≤n≤9.
```

Par contre l'équation récursive suivante, qui ne diffère que par les éléments initialisés, n'est pas effective :

```
f(0) = 0
f(9) = 9
f(n) = 2f(n-1) - f(n-2), 1≤n≤8
```

(essayez !) et le compilateur distingue correctement les deux cas présentés. Techniquement, en termes de sémantique dénotationelle, l'effectivité d'une équation récursive est liée à une propriété de maximalité dans le treillis du domaine sur lequel on cherche la solution de l'équation. Actuellement le compilateur 81/2 fait une analyse d'effectivité qui rejette conservativement certaines équations récursives effectives, ceci pour des raisons analogues à celles qui motivent le développement d'un graphe des dépendances conditionnelles.

Ces voies de recherches, outre qu'elles étendent la classe des programmes 81/2 compilables efficacement, permettent de cerner une nouvelle notion de collection, incluant les collections partielles (i.e. non maximales dans le treillis des collections). Les collections partielles sont indispensables si on veut traiter corrrectement les collections à structure dynamique.

ANNEXE

L'environnement de programmation 8,5

Le compilateur interactif **8,5** implémente un environnement de programmation minimal pour un sous-ensemble du langage 81/2 version 1.0 décrit dans ce document. Le numéro de version de cet environnement au premier trimestre 1994 est `1.01`.

8,5 est un système interactif : l'utilisateur spécifie interactivement et incrémentalement ses équations et lance la compilation complète ou seulement une phase de celle-ci (par exemple l'inférence des géométries ou l'ordonnancement des calculs). Il peut exécuter dans la foulée le code produit.

Deux types de codes sont générés :

- Le compilateur peut générer un code pour une machine virtuelle SIMD (écrite en C) qui manipule des vecteurs. Cette machine virtuelle est émulée au vol par l'environnement, donnant ainsi à l'utilisateur, l'impression de disposer d'un interprète du langage.
- Le compilateur peut générer un programme C séquentiel statique, i.e. sans appels de fonction ni allocation mémoire dynamique. Ce programme doit être ensuite compilé par n'importe quel compilateur C standard, avant de pouvoir être exécuté. Ce programme ne nécessite pas de "run-time" pour s'exécuter et peut être embarqué dans n'importe quel autre programme `C`.

8,5 tourne sous UNIX et est écrit en CAML-Light (un dialecte de ML). Il est relié à un traceur de courbes (GNUPLOT) ce qui permet de visualiser les résultats des exécutions : les figurations graphiques des résultats des programmes de ce document ont été obtenues par ce moyen.

Le lecteur de l'environnement est un petit éditeur ligne qui accepte deux types d'entrées : une équation 81/2 ou une commande. Les commandes du compilateur débutent toutes par un point d'exclamation suivi d'un mot-clé et de différents arguments. On peut lire des fichiers de définition d'équations, lancer des phases de la compilation, évaluer un programme, etc.

La syntaxe des équations sous 8,5 accepte les programmes donnés dans ce document. Le points-virgule est synonyme de la virgule : en effet, nous avons déjà dit que la notation `<;<` ne correspond pas à un opérateur 81/2 et c'est pourquoi le point-virgule peut être utilisé sous 8,5 comme alternative au séparateur spatial, ce qui permet une ponctuation à deux niveaux des programmes. On trouvera dans [MIC 94] la syntaxe précise acceptée par 8,5 et la description des opérateurs implémentés.

Distribution de la version 0.1 de l'environnement 8,5

Le système 8,5 version 1.01 dont il est question dans ce document est disponible sur simple demande par courrier électronique à l'adresse suivante : `[michel|giavitto]@lri.lri.fr`. Il est disponible soit sous la forme de sources, soit sous la forme d'un exécutable pour station SUN-MicroSysteme Sparc10 tournant sous le système d'exploitation SOLARIS 1.4.1 et SOLARIS 2.1. Afin de générer le système complet à partir des sources, il faut disposer d'une installation du système CAML-LIGHT version 0.5 ou 0.6 (accessible par ftp à l'adresse `ftp.inria.fr`, de GMAKE (version GNU de Make), de GNUPLOT, et d'un compilateur `C`.

Avertissement : l'environnement 8,5 version 1.01 n'est pas maintenu.

III.1.b. Les restrictions de la version 1.01 du compilateur

La version 1.01 du compilateur est restreinte à un sous-ensemble du langage 81/2 version 1.0 tel que nous l'avons présenté dans ce document.

- Les équations récursives sont restreintes aux collections régulières ; on ne peut pas définir récursivement un système.
- Les fonctions définies par l'utilisateur ne peuvent pas être récursives : la définition d'une fonction peut faire appel à une autre fonction, mais il faut que cette fonction soit déjà complètement définie. On ne peut pas passer une fonction en paramètre à une autre fonction. Attention, cela n'empêche pas de définir des tissus récursivement dans le corps d'une fonction, par récursion spatiale ou temporelle des tissus.
- On ne peut pas alpha-appliquer, bêta-réduire ou balayer avec une fonction définie par l'utilisateur. Par contre, il est possible de le faire avec les fonctions prédéfinies (par exemple les fonctions arithmétiques). Les fonctions arithmétiques prédéfinies sont implicitement alpha-étendues afin de s'appliquer à des arguments de n'importe quelle géométrie. On peut bêta-réduire mais pas balayer avec l'opérateur de composition.
- Il n'est pas possible d'alpha-entendre une bêta-réduction ou un balayage. Par contre, les opérations de bêta-réduction et de balayage peuvent être qualifiées par l'axe de la bêta-reduction ou du balayage : ⊕ `n\ A` est équivalent à : (⊕ `(n-1)\)` ^ A, et ⊕ `1\ A` correspond à ⊕ `\ A`. Il en va de même pour le balayage. De manière similaire, la concaténation admet la qualification d'un axe : `A #^ B` correspond à `# ^ (A, B)`, `A #^^ B` correspond à `(#^) ^ (A, B)`, etc.
- L'implémentation de la sélection généralisée ne supporte que des vecteurs d'adresse sélectionnant un scalaire et non pas une sous-collection (Cf. §III.3.2.a). Autrement dit, si $[x_1, \ldots, x_p]$ est la géométrie de `X`, alors la sélection généralisée `X.(I)` n'admet que des collections `I` de géométrie $[i_1, \ldots, i_q, p]$ et le résultat a pour géométrie $[i_1, \ldots, i_q]$.

Bibliographie du projet 8½

- A. Mahiout, J.-L. Giavitto, J.P. Sansonnet, *Distribution and scheduling data-parallel dataflow programs on massively parallel architectures*, Software for Multiprocessors and Supercomputers, September 21-23, 1994, Moscow
- A. Mahiout, J.-L. Giavitto, J.-P. Sansonnet, *Placement et ordonnancement de graphes data-flow data-parallèles*, Actes des 5èmes Rencontres sur le Parallélisme, Brest, mai 1993.
- A. Mahiout, *Ordonnancement et placement statique de grands graphes data-flow pour les architectures MSIMD*, DEA d'Architecture des Machines Informatiques Nouvelles de l'université de Paris-Sud, LRI, Septembre 1992.
- C. Germain, J.-L. Giavitto, J.-P. Sansonnet, *Implémentation d'un Paradigme de Programmation Fonctionnelle sur une Machine Massivement Parallèl*e, Actes des Premières Journées Francophones des Langages Applicatifs, 28-29 Janvier 1991, Gresse-en-Vercors, édités par BIGRE.
- 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.
- J.-L. Giavitto, *81/2 : Un modèle MSIMD pour la simulation massivement parallèle*, mémoire de thèse présentée pour obtenir le grade de Docteur en Sciences de l'Univeristé de Paris-Sud, soutenue le 10 décembre 1991 à Orsay. Numéro d'ordre : 1872.
- 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 avril 1991, Münich, LNCS 487.
- J.-L. Giavitto, J.-P. Sansonnet, *81/2 : data-parallélisme et data-flow*, Technique et Science Informatique, Octobre 1993.
- J.-L. Giavitto, J.-P. Sansonnet, O. Michel, *Inférer rapidement la géométrie des collections*, Workshop on Static Analysis, WSA'92, Septembre 1992, Bordeaux, proc. édité par Bigre
- J.-L. Giavitto, *Typing geometry of homogeneous collection*, Internationnal workshop on Array Manipulation, ATABLE-92, Montréal June 1992.
- 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, Paris.
- Jean-Louis Giavitto, *A Synchronous Data-Flow Language for Massively Parallel Computers*, Parallel Computing'91, Septembre 1991, Londres.
- Jean-Louis Giavitto, *Un langage Data-flow synchrone pour la simulation massivement parallèle*, Actes des 2ième Journées francophones des Langage Applicatifs, Perros-Guirec Janvier 1992, Bigres+Globule.
- O. Michel, J.-L. Giavitto, J.P. Sansonnet, *A data-parallel declarative language for the simulation of large dynamical systems and its compilation*, Software for Multiprocessors and Supercomputers, September 21-23, 1994, Moscow

Bibliographie

[AGH 85] G. AGHA, Actors: a model for concurrent computation in distributed systems, AI tech. rep. 844, MIT, 1985.

[AMS 93] J. AMSTERDAM, Automatic Qualitative Modeling of Dynamical Physical Systems, AI Lab Technical Report N°1412, MIT, January 1993.

[ASH 76] E.A. ASHCROFT and W. W. WADGE, « Lucid - A formal system for writing and proving programs », *SIAM Journal on Computing* (3):336-354, Sept., 1976.

[AUL 89] A. AULIN, *Foundations of mathematical system dynamics*, IFSR Int. Series on System Sciences and Engineering, Pergamon Press, 1989.

[BAC 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 (repris dans [THA 87]).

[BAN 88] J.-P. BANATRE, A. COUTANT, D. LE METAYER, « A parallel machine for multiset transformation and its programming style », *Future Generation Computers Systems*, N° 4, 1988.

[BER 87] G. BERRY, P. COURONNE, G. GONTHIER, Synchronous programming of reactive systems, an introduction to ESTEREL, INRIA technical report 647, 1987.

[BER 89] G. BERRY, G. BOUDOL, The chemical abstract machine, Rapport de Recherche N°1133, INRIA, Décembre 1989.

[BIR 87] R. S. BIRD, « An Introduction to the Theory of Lists », *Logic of Programming and Calculi of Discrete Design*, NATO ASI Series, vol. F36, editor. M. Broy, Springer-Verlag 1987

[BRO 93] R. A. BROOKS, L. A. STEIN, Building brains for bodies, MIT A.I. Memo N° 1439, August 1993, MIT.

[CAP 92] F. CAPPELLO, J.-L. BECHENNEC, J.-L. GIAVITTO, « PTAH: Introduction to a new architecture for highly numeric processing », *proc. of PARLE'92*, Juin 1992, Paris, LNCS 605, Springer-Verlag.

[CAS 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.

[CHA 89] K. CHANDY, J. MISRA, *Parallel Program Design - a Foundation*, Addison Wesley, 1989.

[CHE 86] M. C. CHEN, « A Parallel Language and its compilation to multiprocessor machines or VLSI », *POPL'86*, Florida, pp 131-139.

[CLI 81] W.D. Clinger, Foundation of Actor Semantics, PhD thesis, MIT May 1987 (ai-tr-633).

[COM 78] D. COMTE, G. DURRIEU, O. GELLY, A. PLAS, J. C. SYRE, « Parallelism, Control and Synchronization Expressions in a single assignment language », *Sigplan Notice*, Vol. 13, N° 1, January 1978.

[COU 77] P. COUSOT, R. COUSOT, « Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints », *Proc. of the 4th ACM Symp. on POPL* (1977), pp238-252

[DAH 90] E. D. 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.

[DEL 86] C. DELGADO-KLOOS, *Semantics of digital circuits* LNCS 285, Springer-Verlag 1986.

[DEL 94] D. G. DELAHAUT, C. GERMAIN, « Static communication in parallel scientific programms », *Proc. of PARLE'94*, to appear in LNCS, Springer-Verlag, 1994

[DEN 74] J. B. DENNIS, « First Version of a Data Flow Procedure Language », *proc. of the Programming Symposium*, Paris, April 9-11, 1974, LNCS 19, Springer-Verlag.

[DEN 91] J. B. DENNIS, « Static Dataflow Architecture », in Jean-Luc Gaudiot, Ludomir Bic, editors, *Advanced Topics in data-flow computing*, Prentice Hall, 1991.

[DEW 84] A. DEWDREY, « Résolution de problèmes de manière analogique », Pour la Science, N° spécial informatique, 1984.

[DIJ 88] E. Dijkstra, « Position paper on "Fairness" », *Software Engineering Notes*, *ACM Sigsoft* Vol. 13, N° 2, April 1988, Cf. also the other papers in the same n°, « Another position paper on "Fairness" » by F. B. Schneider & L. Lamport, « Another view of "Fairness" » by K. M. Chandy & J. Misra.

[FAU 82] A. A. FAUSTINI, « An operational semantics for pure dataflow », *Automata, Language and Programming, 9th Colloquium*, LNCS 140, pp212-224, Springer-Verlag, 1982

[FEA 91] P. FEAUTRIER, « Dataflow Analysis of Array and Scalar References », *International Journal of Parallel Programming*, 20 (1), February 1991.

[FRI 93] P. FRITZSON, N. ANDERSSON, « Generating Parallel Code from equations in the ObjectMath Programming environment », *Parallel Computation - Second Int. ACPC Conference,* Gmunden Austria 1993, LNCS 734

[GAJ 82] D.D. GAJSKI, D.A. PADUA and D.J. KUCK, R. H. KUHN « A second opinion on data flow machines and languages », *IEEE Computer*, february 1982 (repris dans [THA 87]).

[GAR 78] M.R. GAREY, R.L. GRAHAM, D.S. JONSON, « Performance Guarantees for Scheduling Algorithms », *Operation research*, Vol. 26, N° 1, January-February 1978.

[GAU 87] T. GAUTIER, P. LE GUERNIC, L. BESNARD, « SIGNAL : a declarative language for synchronous programming of real-time systems », G. Kahn editor : *Functionnal Programming languages and computer architecture*, LNCS 274, Springer-Verlag 1987.

[GAU 91] J.-L. GAUDIOT, L. BIC, editors, *Advanced Topics in data-flow computing*, Prentice Hall, 1991 (ISBN 0-13-006503).

[GER 91] C. GERMAIN-RENAUD, J.-P. SANSONNET, *Les Ordinateurs massivement parallèles*, collection 2AI, Armand-Colin 1991.

[GIA 91a] J.-L. GIAVITTO, « A synchronous data-flow language for massively parallel computers », *Proc. of Parallel Computing'91*, 3-6 September 1991, London.

[GIA 91b] J.-L.GIAVITTO, Un langage pour la simulation massivemement parallèle des systèmes dynamiques discrets, Thèse de l'Université de Paris-Sud, decembre 1991.

[GIA 92] J.-L. GIAVITTO, J.-P. SANSONNET, O. MICHEL, « Inférer rapidement la géométrie des collections », *Workshop on Static Analysis*, WSA'92, Septembre 1992, Bordeaux, proc. édité par Bigre.

[GIA 93] J.-L. GIAVITTO, *Les langages Parallèles*, Support de cours du module LAP du DEA Informatique et du DEA Architecture des Machines Nouvelle de l'Université d'Orsay, LRI 1993.

[GUR 85] J. R. GURD, C. C. KIRKHAM, I. WATSON, « The Manchester prototype data-flow computer », *CACM* January 1985.

[HAL 91] N. HALBWACHS, P. CASPI, P. RAYMOND, D. PILAUD, « Programmation et vérification des systèmes réactifs : le langage LUSTRE », *Technique et Science Informatiques*, Vol. 10, N° 2, 1991.

[HAQ 91] P. J. HATCHER, M. J. QUINN, *Data-parallel programming on MIMD computers*, Scientific and engineering computation series, MIT Press.

[HAW 89] J.-J. HAWANG, Y.-C. CHOW, F. ANGERS, C.-Y. LEE, « Scheduling precedence graphs in systems with interprocessor communication times », *SIAM J. Comp.*, Vol. 18, N°2, pp 244-257, April 1989.

[HOF 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.

[HPF 93] HIGH PERFORMANCE FORTRAN FORUM, High Performance Fortran language specification - version 1.0, Rice University, Houston, may 1993

[HUD 92] P. Hudak, S. L. Peyton Jones, P. Wadler (editors), « Report on the programming language Haskell, a non-strict purely functional language (version 1.2) », *ACM SIGPLAN Notices*, Mar. 1992.

[IEEE 89] *Special issue on dynamics of discrete event systems,* Proc. of the IEEE, Vol. 77, N°1, pp1-232, January 1989.

[JOH 83] S. D. JOHNSON, *Synthesis of Digital Designs from Recursion Equations*, ACM Distinguished Dissertation 1983, MIT Press 1983.

[KAH 74] G. KAHN, « The semantics of a simple language for parallel programming », *proc. of IFIP Congress'74*, 1974

[LEG 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.

[LEI 83] C. E. LEISERSON, J. B. SAXE, « Optimizing Synchronous Systems », *Journal of VLSI and Computer Systems*, vol. 1, N°1, pp. 41-67, 1983.

[MAH 92] A. MAHIOUT, Ordonnancement et placement statique de grands graphes data-flow pour les architectures MSIMD, DEA d'Architecture des Machines Informatiques Nouvelles de l'université de Paris-Sud, LRI, Septembre 1992.

[MAU 89] C. MAURAS, Alpha : un langage équationel pour la conception et la programmation d'architectures parallèles synchrones, thèse de l'université de Rennes I, 15 décembre 1989.

[MCG 82] J. R. MCGRAW, « The VAL Language », *ACM Trans. on Programming Languages and Systems*, Vol. 4, N° 1, January 1982.

[MCG 85] J. R. MCGRAW, S. K. SKEDZIELEWSKI, S. ALLAN, R. OLDEHOEFT, J. GLAUERT, C. KIRKHAM, W. NOYCE, R. THOMAS, SISAL : Stream and iteration in a single assignement language, Lawrence Livermore National Laboratory report, Livermore CA USA, March 1985 Version 1.2 M-146.

[MEY 90] J.-A. MEYER, editor, *Simulation of Adaptative Behavior - From Animals to Animats*, proceedings of the 1st SAB conf., 24-28 September 1990, Paris

[MIC 94] O. MICHEL, *Manuel de référence du langage* 81/2, rapport technique du LRI, à paraître 1994.

[MIL 89] R. MILLER, Q. F. STOUT, « Mesh computer algorithms for computational geometry », *IEEE Transactions on computers*, Vol. 38, N° 3, March 1989pp 321-340

[NIH 86] R. S. NIKHIL, K. PINGALI, ARVIND, Id nouveau, CSG Memo 265, MIT Laboratory for Computer Science, Cambridge MA, July 1986.

[OKE 87] R. A. O'Keefe, *Finite Fixed Point Problem*, proc. of the 4th ICLP'87, J.-L. Lassez edt., pp729-743, Melbourne, 1987 MIT Press.

[PAG 88] H.R. PAGELS, *The dreams of reason: The computer and the rise of the sciences of complexity*, Simon and Schuster, New-York 1988

[PAR 91] K. K. PARHI, D. G. MESSERSCHMITT, « Static rate-optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding », *IEEE Trans. on Comp.*, Vol 40, N°2, February 1991.

[PAR 92] N. PARIS, Definition de POMP-C (version 1.99), rapport LIENS 92-5, Département de Mathématiques et d'Informatique de l'École Normale Supérieure, Mars 1992.

[PDP 86] D. E. RUMELHART, J. 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)

[PLA 76] A. PLAS, D. COMTE, O. GELLY, J.C. SYRE, « LAU system architecture: a parallel data-driven processor based on single assignement », *Proc. 1976 Int. Conf. Parallel Processing*, pp 293-302.

[PUG 70] A. L. PUGH, *Dynamo II. User's manual*, MIT Press 1970

[QUI 89] P. QUINTON, Y. ROBERT, *Algorithmes et architectures systoliques*, Masson, 1989.

[ROZ 87] M.-N. ROZIN, Etude pratique des ressources de la programmation plane, Mémoire de Maitrise de l'université de Paris-8-Vincennes, juillet 1987

[SAN 90] J.-P. SANSONNET, *Architectures de Machines Parallèles*, cours de DEA de l'Université d'Orsay, LRI 1991.

[SCH 84] N. SCHMITZ, J. GREINER, « Software aids in PAL circuit design, simulation and verification », *Electronic design*, 32(11), May 1984.

[SCH 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

[SIJ 89] B. A. SIJTSMA, « On the productivity of recursive list definitions », *ACM Transactions on Programming Languages and Systems*, Vol. 11, N° 4, October 1989, pp 633-649

[SIP 91] J. M. SIPELSTEIN, G. E. BLELLOCH, « Collection-oriented languages », *proc. of the IEEE*, Vol. 79, N° 4, april 1991.

[SKI 90] D. SKILLICORN, « Architecture-Independent Parallel Computation », *Computer*, December 1990.

[SMI 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.

[SPE 76] F. H. SPECKHART, W. L. GREEN, *A guide to using CSMP : the continuous system modeling program*, Prentice-Hall 1976

[STE 64] H. STEINHAUS, *Mathématiques en instantanés*, Flammarion Paris 1964.

[STE 90] G. STEELE, « Making asynchronous parallelism safe for the world », *POPL'90*, San-Franscisco, 17 January 1990, pp 218-231.

[STE 94] L. A. Stein, « Imagination adn situated cognition » *Journal of Experimental and theoretical Artificial Intelligence* (to appear).

[TES 68] L.G. TESLER, H.J. ENEA, « A language design for concurrent processes », *AFIPS Conference Proceedings*, vol. 32, pp 403-408, 1968.

[THA 87] S. S. THAKKAR editor, *Selected reprints in dataflow and reduction architectures*, IEEE Computer Society Press, 1987.

[TMC 91] THINKING MACHINE CORPORATION, The Connection-Machine CM5 technical Summary, October 1991.

[VAN 92] J. VAN LEUWEN editor, *Handbook of Theoretical computer science*, Volume B: Formal Models and Semantics, article *Algebraic Specification* by M. WIRSING, article *Domain theory* by D. SCOTT, North-Holland.

[WAD 81] W. W. WADGE, « An extensional treatment of dataflow deadlock », *Theoretical Computer Science,* 13 (1), 1981, pp 3-15.

[WAD 85] W. W. WADGE and E.A. ASHCROFT, *Lucid, the Data flow Programming Language*, Academic Press U.K., 1985.

[WAT 91] R. 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.

[ZIM 92] E. V. ZIMA, « Recurrent relations and speed-up of computation using computer algebra systems », *Design and Implementation of symbolic computation systems*, Int. Symposium DISCO'92, Bath U.K. April 1992, LNCS 721, Springer-Verlag, 1992