ORSAY
N° d'ordre : 4596

UNIVERSITÉ DE PARIS-SUD
U.F.R. SCIENTIFIQUE D'ORSAY

THÈSE

présentée pour obtenir

le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS SUD (XI)

Spécialité : INFORMATIQUE

par

Olivier MICHEL

Sujet : **Représentations dynamiques de l'espace dans un langage déclaratif de simulation**

Soutenue le 11 Décembre 1996 devant le jury composé de :

MM.	Laurence PUEL	*Président*
	Gilles BERNOT	*Rapporteur*
	Patrick SALLÉ	*Rapporteur*
	Edward ASHCROFT	*Examinateur*
	Jean-Louis GIAVITTO	*Examinateur*
	Jean-Paul SANSONNET	*Directeur de la Thèse*

Remerciements

Je remercie vivement Madame Laurence PUEL, Professeur, Directrice du L.R.I., qui a bien voulu présider le jury de cette thèse.

Je veux sincèrement remercier Monsieur Gilles BERNOT, Professeur à l'Université d'Évry Val d'Essonne et Monsieur Patrick SALLÉ, Professeur à l'I.R.I.T., d'avoir accepté d'être rapporteur de ma thèse. Je leur suis particulièrement gré d'avoir lu, avec toute l'attention et l'intérêt dont ils ont fait preuve, un bien épais manuscrit.

Je suis très honoré de la présence dans le jury de Monsieur Edward ASHCROFT, Professeur à l'Arizona State University, père du langage Lucid, dont les idées nous ont toujours inspirés et stimulés.

Monsieur Jean-Paul SANSONNET, Directeur de recherche au C.N.R.S., a accepté d'être le directeur de cette thèse. Les discussions que nous avons eues ont toujours été la source d'un grand plaisir et je l'en remercie sincèrement.

Monsieur Jean-Louis GIAVITTO, Chargé de recherche au C.N.R.S., a encadré cette recherche pendant trois ans ; qu'il reçoive ici mes remerciements les plus vifs.

Mon travail s'est déroulé dans un environnement fertile : l'équipe *Architectures Parallèles* du L.R.I. Je remercie Monsieur le Professeur Daniel ÉTIEMBLE, responsable de l'équipe, de m'y avoir accueilli.

J'ai passé trois ans avec Abderrahmane MAHIOUT et Dominique DE VITO sur le projet 81/2. Je les remercie de leur constante gentillesse et de la qualité des discussions que nous avons eu pendant cette période. Je suis redevable à Julien SOULA qui a effectué son stage de D.E.A. au sein de l'équipe et dont la maîtrise de Mathematica nous a permis de disposer d'un premier évaluateur de GBF. Franck CAPPELLO et Franck DELAPLACE ont à la fois été des exemples de rigueur et ont su dispenser entrain et bonne humeur. Je les en remercie vivement.

Introduction

L'objectif de ce travail est de concevoir et d'étudier des représentations dynamiques de l'espace dans le cadre d'un langage déclaratif. Le but est d'intégrer dans un langage informatique des structures de données proches des objets mathématiques utilisés dans la modélisation des systèmes dynamiques (SD), afin de simplifier l'expression des algorithmes et les programmes.

Ce travail a pour cadre le projet 81/2. Issu de l'équipe *Architectures Parallèles* du LRI, l'ambition initiale du projet était de développer un langage parallèle pour la simulation des SD. Ce type d'applications représente la grande majorité des programmes exploités sur les super-ordinateurs.

Cela a conduit à développer des structures de données dédiées à la simulation des SD : le *stream* et la *collection* 81/2, et surtout, à étudier la compilation efficace des programmes déclaratifs définissant de telles structures de données. Cet objectif a été en partie atteint, avec le développement d'un compilateur et l'étude de la distribution automatique des calculs.

Une constatation est alors venue réorienter les directions de recherche du projet 81/2 : l'*expressivité* d'un langage de simulation est au moins aussi importante que son *efficacité*. En effet, les objets modélisés par les simulations actuelles sont de plus en plus complexes et leurs interactions compliquées. Les algorithmes de simulation deviennent de plus en plus sophistiqués. Le peu d'expressivité d'un langage constitue alors un obstacle à la mise en œuvre de ces nouvelles simulations. Avec la montée en puissance des microprocesseurs, et la standardisation des logiciels de base permettant de travailler sur des réseaux hétérogènes de stations de travail, *le nouveau challenge n'est plus seulement celui de la puissance de calcul mais aussi celui de la programmation des modèles*.

Le problème posé au départ était d'étendre la notion de collection en 81/2 afin de pouvoir programmer la simulation de processus de croissance. L'exemple type visé est la simulation des L systèmes, un formalisme de réécriture parallèle utilisé dans la modélisation de la croissance des plantes. La simulation des L systèmes conduit à représenter un arbre dynamique et à évaluer des attributs sur cet arbre.

Cela m'a amené directement à la nécessité de manipuler des fragment de code, sous forme de termes ouverts. La systématisation de cette idée et sa formalisation a donné lieu à la notion d'*amalgame*. Les amalgames constituent un mécanisme puissant pour structurer et paramétrer les programmes déclaratifs sous la forme de «paquet d'équations». Il est possible d'adopter plusieurs points de vue sur les amalgames : le point de vue qui nous intéresse plus particulièrement est le point de vue spatial. Un amalgame définit une notion d'espace *irrégulier*. En effet, pour raisonner sur les amalgames, il est commode de les représenter sous la forme d'un graphe data-flow : toutes les opérations de construction d'amalgames admettent alors une représentation graphique simple en termes de construction de graphe.

Dans le même temps, une autre partie de l'équipe 81/2 développait les concepts nécessaire à la compilation de définitions récursives de concaténation de tableaux. Ce problème nécessite de décrire de manière compacte le graphe data-flow des calculs entre les éléments du tableau. Nous nous sommes alors rendus compte que dans les deux cas on raisonnait sur le même objet, un graphe des dépendances, mais que dans un cas c'était un objet qui présentait une grande régularité (définition récursive de concaténation de tableaux) et que dans l'autre cas, il n'en présentait pas (les amalgames).

L'étude des régularités d'un graphe de dépendances d'une définition récursive de tableau a montré qu'on pouvait décrire ce graphe par la présentation d'un groupe mathématique, ce qui permet d'unifier le traitement d'autres structures de données récursives, comme par exemple les arbres. Par ailleurs, l'analogie avec les amalgames m'a convaincu de la nécessité de représenter des tableaux partiels, c'est-à-dire des tableaux dont tous les éléments n'ont pas une valeur définie. Cela donne lieu à la notion de *champ de données basé sur une structure de groupe* ou GBF. Les GBF permettent la représentation d'espaces *réguliers*, où chaque point de l'espace a la même structure de voisinage. Cette représentation permet la définition de structures spatiales complexes exprimant en plus une information de type.

Si on revient à présent aux problèmes de simulation des SD, on voit que les GBF et les amalgames permettent de représenter deux sortes d'espaces (continu et discret) qui interviennent dans la description d'un SD. Les GBF sont particulièrement aptes à représenter la discrétisation d'une portion d'espace, comme on le fait par exemple pour la résolution numérique d'une équation aux dérivées partielles. Et les amalgames sont particulièrement aptes à représenter les relations structurelles entre les variables qui décrivent un SD. Mais ces nouveaux types de données dépassent leur motivation initiale : les amalgames sont utilisables dans le domaine de la programmation incrémentielle et les GBF s'inscrivent dans la ligne des travaux autour des types de données algébriques.

Organisation du document

Ce document se structure en quatre parties.

La première partie décrit le langage **$8_{1/2}$** . Les notions de stream, de collection et leur intégration dans un langage déclaratif sont présentées. En particulier, les opérations sur les streams et les collections sont brièvement décrites, afin de permettre au lecteur de comprendre les exemples donnés. Nous rappelons à cette occasion l'approche *intensionnelle* du langage $8_{1/2}$. Le dernier chapitre de la première partie donne des exemples de SD. Ces exemples mettent en évidence l'adéquation du langage à la simulation des SD et nous donnons, tout au long du document, des exemples illustrant notre propos. Nous espérons qu'ils aideront à la compréhension des concepts décrits et des problématiques abordées.

La seconde partie de ce document définit et étudie la notion de **GBF**. Elle commence par la critique de la notion classique de tableau que l'on rencontre dans les langages évolués. Cette notion de tableau est trop pauvre pour la description de structures spatiales plus complexes que la grille euclidienne. Nous proposons d'étendre la notion de tableau en considérant un tableau comme une *fonction partielle* de l'ensemble de ses index vers un ensemble de valeurs. On munit cet index d'une structure de groupe. Cette structure définit les GBF. Nous détaillons dans cette partie les opérations sur les GBF et illustrons notre propos par de nombreux exemples pris dans des domaines aussi divers que un calcul de gaz sur un réseau, un processus de croissance ou un calcul de relaxation. Un schéma général d'implémentation des GBF est donné et détaillé dans le cas abélien. Les mécanismes décrits et les exemples donnés sont validés par des programmes en Mathematica.

La troisième partie de ce document est consacrée aux **amalgames**. Nous mettons en évidence que la structure d'espace d'un GBF, décrite par la présentation d'un groupe, correspond aussi au graphe des dépendances d'un système $8_{1/2}$, mais que ce graphe n'est pas forcément homogène. Un système $8_{1/2}$ est une collection d'équations et nous nous posons la question de construire ces objets comme résultat d'un calcul. Pour cela, nous nous appuyons sur des systèmes *ouverts*, c'est-à-dire des systèmes impliquant des identificateurs auxquels ne correspond aucune définition. La définition des expressions ouvertes et d'un mécanisme de clôture des expressions par capture de nom donne lieu à la définition des amalgames. Trois opérateurs seulement permettent de construire des amalgames. Ces trois opérateurs ont une interprétation simple en termes de mécanismes de programmation. Une sémantique formelle des amalgames est ensuite développée dans le chapitre X. Cette sémantique, exprimée dans le style SOS, donne lieu à une implémentation en Mathematica d'un interprète du calcul sur les amalgames. Le chapitre suivant illustre par des exemples l'utilisation des amalgames. Nous montrons en particulier la puissance expressive du formalisme à l'occasion du codage de l'arithmétique. Nous utilisons les amalgames pour la modélisation d'un processus de croissance.

Enfin, la dernière partie du document se propose d'**intégrer** les deux nouvelles structures de données dans le langage $8_{1/2}$. Nous proposons une solution qui consiste à développer des *streams d'amalgames de GBF*. Nous n'intégrons pas l'ensemble des GBF, mais nous nous restreignons à un sous-cas correspondant à des tableaux dynamiques. Cela nous oblige à définir une nouvelle sémantique $8_{1/2\mathcal{D}}$, dans le style opérationnel, car la sémantique initiale de $8_{1/2}$ ne supporte pas les extensions dynamiques. Une implémentation en CAML est actuellement en cours. Malgré toutes ces restrictions, nous donnons dans le dernier chapitre des exemples significatifs de programmes $8_{1/2\mathcal{D}}$, et en particulier, la simulation d'une classe de L systèmes.

Table des matières

Liste des exemples

[...] au fond, j'ai été victime d'un préjugé analogue à celui de Newton qui pensait que toutes les fonctions avaient le temps comme variable indépendante. [...]

René THOM

(in *Paraboles et catastrophes*)

L'espace est une représentation nécessaire *a priori* qui sert de fondement à toutes les intuitions extérieures. On ne peut jamais se représenter qu'il n'y ait pas d'espace, quoique l'on puisse bien penser qu'il n'y ait pas d'objets dans l'espace. Il est considéré comme la condition de la possibilité des phénomènes, et non pas comme une détermination qui en dépende, et il est une représentation *a priori* qui sert de fondement, d'une manière nécessaire, aux phénomènes extérieurs.

Emmanuel KANT

(in *Critique de la raison pure*)

- Garçon !
- Monsieur ?... Le bar est fermé, Monsieur... Monsieur désire quelque chose ?
- Oui..., de l'espace...
- Pardon ?
- Un peu d'air... Vous avez compris ?... Barrez-vous !

Henri-Georges CLOUZOT,
Jean FERRY

(in *Quai des Orfèvres*)

Tout autre est le rhizome, *carte et non pas calque*.[...] La carte est ouverte, elle est connectable dans toutes ses dimensions, démontables, renversable, susceptible de recevoir constamment des modifications. Elle peut être déchirée, renversée, s'adapter à des montages de toute nature, être mise en chantier par un individu, un groupe, une formation sociale. [...]

Gilles DELEUZE,
Félix GUATTARI

(in *MILLE PLATEAUX*)

À mes parents...

Première partie

La simulation et les langages déclaratifs

Chapitre I

Simulation des systèmes dynamiques et langages déclaratifs : l'approche 81/2

Dans cette première partie nous défendons l'idée d'un langage de programmation de haut-niveau pour la simulation des systèmes dynamiques. Les systèmes dynamiques sont un cadre très général pour modéliser des phénomènes décrits dans l'espace et qui évoluent dans le temps. L'étude de ces modèles, qui se retrouvent dans tous les domaines, passe très souvent par la simulation.

Le langage 81/2 a développé des structures de données adaptées à la représentation de l'état d'un système et à la représentation de sa trajectoire (la suite des états dans le temps). Ces structures de données sont intégrées dans un cadre déclaratif, ce qui rapproche le programme des formalismes mathématiques utilisés naturellement dans les modèles.

Cependant, les structures de données offertes par 81/2 sont encore trop rudimentaires, surtout lorsqu'il s'agit de traiter les aspects dynamiques de la représentation spatiale d'un système. Ces aspects interviennent quand il faut simuler des processus de morphogénèse ou bien quand il faut calculer l'espace des états conjointement à l'évolution du système.

L'objet de ce travail est de développer de nouvelles représentations informatiques de l'espace, aptes à répondre à ces problèmes.

Structure de la partie. Cette partie est composée de trois chapitres. Dans le premier chapitre nous examinons la problématique d'un langage de programmation pour la simulation, qui doit répondre à la fois à des problèmes d'efficacité et d'expressivité.

Dans le deuxième chapitre, nous présentons succinctement le langage 81/2. En particulier, les opérations sur les streams et les collections sont brièvement décrites, afin de permettre au lecteur de comprendre les exemples donnés tout au long de ce document. Un stream est la représentation informatique d'une trajectoire, et une collection correspond à la représentation informatique d'un domaine de l'espace.

Le troisième chapitre illustre les notions introduites à travers des exemples. Deux exemples serviront de « fil rouge » tout au long de ce document : il s'agit de la modélisation du comportement d'une créature artificielle rudimentaire, le *wlumf*, et de la simulation grossière de la croissance d'un coquillage, l'*ammonite*.

Structure du chapitre. Ce chapitre commence par introduire la notion de système dynamique, puis compare les objectifs d'efficacité et d'expressivité d'un langage de programmation dédié à la simulation des systèmes dynamiques. Le cadre général du projet 81/2 est ensuite présenté.

I.1 Les systèmes dynamiques

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.

Nous donnons le nom général de *système dynamique* (ou SD) aux modèles qui décrivent dans l'*espace* un état qui évolue dans le *temps*. La modélisation et la simulation des SD représente un domaine d'application très important. En effet, les besoins en simulation de SD se retrouvent par exemple dans tous les domaines où on peut difficilement effectuer une expérimentation :

- soufflerie numérique,
- simulations nucléaires,
- résistance des matériaux,
- évolution des éco-systèmes,
- modélisation des semi-conducteurs et des supra-conducteurs,
- chromo-dynamique quantique,
- dynamique moléculaire et réactions chimiques,
- croissance de cristaux et problèmes de morphogénèse,
- et finalement toutes les applications de «réalité virtuelle» correspondant à la simulation de phénomènes dans un monde abstrait.

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 décrit 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.

Il existe de nombreux formalismes différents pour décrire un SD. Par exemple, les physiciens classent traditionnellement les modèles de systèmes dynamiques suivant le type *continu* ou *discret* de l'état, du temps et de l'espace. La table 1 donne des exemples de formalismes utilisés pour décrire ces SD.

Initialement application de l'informatique, la simulation devient elle-même un paradigme de programmation et, de ce fait, un langage de haut-niveau pour la simulation développe un nouveau modèle de programmation. En effet, les phénomènes du monde physique constituent, sous une forme idéalisée, les objets d'un

`C`: continu, `D`: discret.	Équation aux dérivées partielles	Équation différentielle	Réseau d'itérations couplées	Automate cellulaire
Espace	`C`	`D`	`D`	`D`
Temps	`C`	`C`	`D`	`D`
État	`C`	`C`	`C`	`D`

TAB. 1 –: *Divers formalismes utilisés pour la modélisation de SD et leur caractérisation (continu, discret) par rapport à l'espace, le temps et l'état. Le seul moyen de décrire et d'étudier le comportement de ces formalismes passe souvent par la simulation. Celle-ci étant faite sur un ordinateur digital, il est nécessaire de discrétiser le temps, l'espace et les valeurs décrivant l'état. Cependant, le programme à écrire pour la résolution d'une équation aux dérivées partielles est très éloigné du programme d'un automate cellulaire : le nombre d'états en jeu dépasse de plusieurs centaines d'ordres de grandeur l'espace des états d'un automate cellulaire et un flottant* `IEEE` *est une bonne approximation d'une variable réelle. Mais il existe un* cadre commun *à tous ces formalismes, et que doit prendre en compte un langage dédié à ce type de simulation, c'est la* représentation du temps et de l'espace.

nouveau calcul. On peut identifier les relations qui existent entre les étapes de modélisation et de simulation d'un SD et les étapes de construction d'un programme :

description d'un monde	≡	*programme* informatique
paramètres de la description	≡	*données* du programme
phénomènes dans le monde décrit	≡	*résultats* du calcul

Ce point de vue fertile est à l'origine de l'invention de nouveaux algorithmes (comme par exemple les procédures de recuit-simulé, les algorithmes évolutionnistes, les réseaux neuro-mimétiques...), et participe plus généralement d'un domaine de recherche nouveau, identifié sous le nom «Parallel Problem Solving from Nature» [SM91].

I.2 Langages expressifs ou efficaces pour la simulation des systèmes dynamiques

Les applications de simulation des SD nécessitent de grandes puissances de calcul. Elles représentent d'ailleurs la majorité des applications des super-ordinateurs actuels. L'initiative du «Grand Challenge» [NSF91] qui a pour but le développement de l'infrastructure matérielle et *logicielle* pour atteindre le tera-ops, rappelle que l'expérience numérique, devenue indispensable dans toutes les disciplines scientifiques, n'est envisageable que si on dispose des ressources de calcul nécessaires. Cela est vrai surtout dans le domaine de la prédiction par simulation. Par exemple, les premières simulations, en 1950 par VON NEUMANN, d'un modèle de circulation atmosphérique barotrope (c'est-à-dire négligeant les variations de température le long des surfaces isobares), effectuée sur l'`E.N.I.A.C.`, ont permis de calculer l'évolution météorologique sur une durée de 24 heures dans une région donnée [CFVN61]. Hélas, la simulation avait nécessité 24 heures de calculs !

Un langage de programmation ne peut avoir aucun impact sur l'efficacité du *modèle*. Il peut par contre permettre une utilisation optimale des ressources fournies par l'architecture sur laquelle est exécuté le programme. Cependant, un langage pour la simulation se doit aussi d'être *expressif*.

La notion d'expressivité correspond aussi à un critère d'efficacité, mais cette fois-ci elle concerne la tâche de programmation. En effet, la simulation correspond à une méthode où l'on extrait un *modèle* d'un phénomène que l'on désire étudier, afin de pouvoir effectuer des expériences à partir de ce modèle, et étudier son comportement dans des situations diverses. Un modèle est toujours une représentation simplifiée du phénomène que l'on désire simuler, mais il doit en conserver les caractéristiques principales, afin que les conclusions de la simulation soient en rapport avec le système étudié. Pour pouvoir définir *efficacement* un modèle, il est nécessaire de définir un formalisme qui soit le plus proche possible de la classe d'applications que l'on désire modéliser. Si on essaye de formaliser un SD à l'aide d'un langage impératif classique, il est évident que la majeure partie du temps sera consacré à la résolution de problèmes d'intendance tels que :

- la représentation des entités de la simulation,
- la gestion de la mémoire,
- la représentation du temps logique et sa gestion,
- la gestion du séquencement des activités des entités de la simulation.

Un langage pour la simulation des SD doit donc offrir des concepts de haut-niveau pour faciliter la programmation des applications. Cet objectif n'est pas nécessairement antinomique avec un objectif d'efficacité. En effet :

- Il est rappelé dans [NSF91] que l'augmentation de la taille des problèmes traités est due *en parts égales* à :
  1. l'augmentation des performances du matériel,
  2. la sophistication des algorithmes.

Par conséquent, en permettant d'exprimer des algorithmes plus sophistiqués, on améliore aussi l'efficacité de la simulation. Par ailleurs, il faut prendre conscience [FKB96, Zim92, GG96, Can91, MGS94] que l'utilisation de langages rudimentaires comme Fortran77 empêche le développement de nouveaux algorithmes (cf. le développement des nouveaux solveurs et des nouvelles librairies scientifiques en C++ [BLQ92, KB94, ASS95, HMS+94, PQ93, PQ94, LQ92]).

- Un langage de haut-niveau, puisqu'il est indépendant d'un architecture matérielle particulière, conduit *de facto* au développement de programmes plus portables puisqu'ils ne sont pas liés à une implémentation particulière, et sont donc plus aptes à profiter des évolutions technologiques.

- Un langage de haut-niveau peut se compiler efficacement, tout au moins pour la partie du langage qui peut s'implémenter efficacement ! Si les concepts offerts par un langage de haut-niveau sont proches d'une application, mais que ces concepts s'implémentent mal sur une architecture donnée, alors la probabilité est grande que cette application sera inhéremment inefficace, quel que soit le langage utilisé pour la coder.

I.3 8$_{1/2}$: un langage expressif pour la simulation

Le but poursuivi par le projet 8$_{1/2}$ est l'étude et le développement de structures de données et de contrôle permettant une programmation expressive des applications de SD. L'implémentation efficace de ces structures est aussi étudiée, et donne lieu au développement d'un langage expérimental nommé 8$_{1/2}$.

Le langage 8$_{1/2}$ a adopté un *style déclaratif de programmation*. En effet, l'utilisation d'un style déclaratif permet de considérer que les lois d'évolution du SD, exprimées sous la forme de relations fonctionnelles entre variables, constituent un programme.

Nous avons vu que la simulation d'un SD nécessite l'expression de l'évolution des variables d'un système dans deux dimensions particulières : l'*espace* et le *temps*. La variation, en 8$_{1/2}$, d'une valeur dans le temps est rendue par une structure de données adaptée : la série ou *stream*. La représentation de l'état d'un domaine de l'espace conduit au concept de *collection* : un ensemble structuré de valeurs qui est manipulé comme un tout homogène. La combinaison de ces deux structures de données est appelée un *tissu* : un tissu représente la trajectoire, dans le repère ⟨temps, espace, valeur⟩ d'un SD (voir la figure 1).

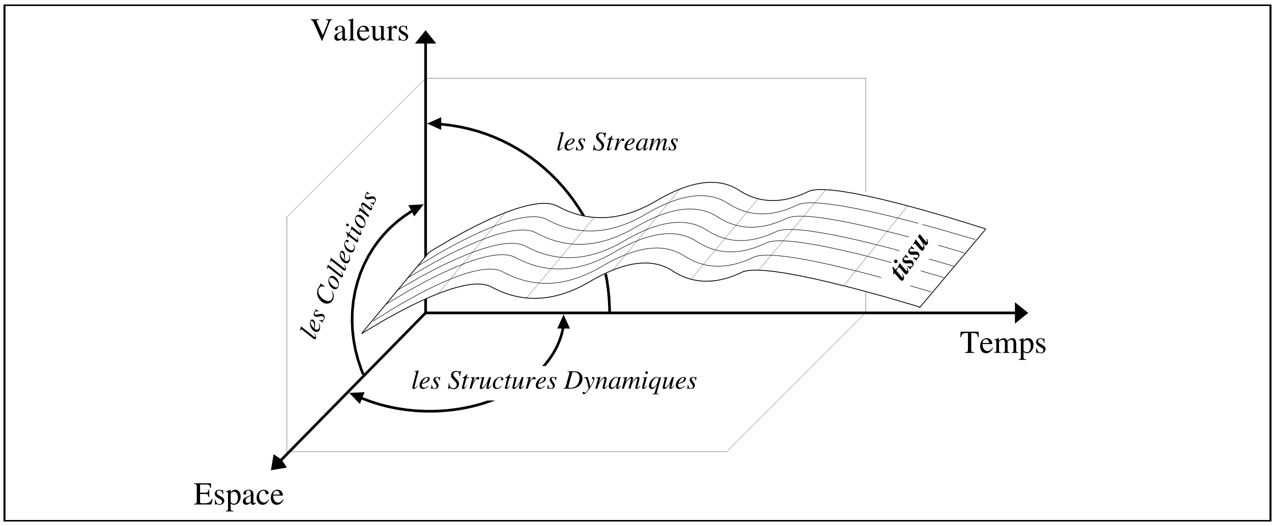


FIG. 1 –: *Un tissu spécifié par une équation 8$_{1/2}$ est un objet dans le repère ⟨temps, espace, valeur⟩. Un stream est une valeur variant dans le temps ; une collection représentant la variation d'une valeur dans l'espace. La variation de l'espace dans le temps détermine une structure dynamique.*

Objectif de ma thèse

C'est dans cette perspective que s'inscrit le travail de cette thèse. L'objectif du travail que nous présentons dans ce document est de concevoir et d'étudier des représentations dynamiques de l'espace dans un cadre déclaratif. En effet, l'intégration de modèles de données proches des objets mathématiques utilisés, doit permettre de réduire d'avantage le fossé entre modèles et langages de programmation, afin de simplifier les algorithmes et les programmes.

Notre motivation principale est de pouvoir développer facilement :

- la simulation des phénomènes hautement dynamiques, comme les phénomènes de morphogénèse ;
- la simulation de SD pour lesquels l'espace des états n'est pas connu *a priori*, mais doit être calculé conjointement avec l'évolution temporelle du système.

La suite de cette partie est consacrée à la présentation du langage 81/2 tel qu'il existe et est défini dans [Gia91a, GS93]. Les concepts que nous développons sont motivés et illustrés par des exemples de taille restreinte en lignes de code, mais néanmoins représentatifs des problèmes que l'on rencontre en simulation. Nous les illustrons aussi par des exemples purement informatiques, car les concepts étudiés dépassent le champ d'application de la simulation de SD et présentent un intérêt «pour eux-mêmes».

La deuxième partie de ce document étudie la représentation d'un *espace homogène* à l'aide d'une nouvelle structure de donnée déclarative : le *champ basé sur un groupe*.

La troisième partie étudie la représentation d'un *espace hétérogène* à travers le concept d'*amalgame*.

Enfin, la quatrième partie intègre ces deux nouvelles structures de données dans le langage 81/2, définissant le langage **81/2**$_{\mathcal{D}}$.

Chapitre II

Le langage déclaratif 81/2

II.1 Description du langage déclaratif 81/2

Le langage 81/2 est un langage déclaratif : un programme est un ensemble d'équations qui définissent des *tissus*. Un tissu est la combinaison de la notion de *stream* et de *collection*. La dimension « stream » d'un tissu permet de représenter la trajectoire d'un SD ; la dimension « collection » permet de représenter son état.

Une équation 81/2 est de la forme :

$$id = expr$$

où id est un identificateur et $expr$ une expression pouvant faire référence à d'autres identificateurs de tissus. On dit que id est le *definiendum* et $expr$ le *definiens*. Cette équation peut se lire comme une définition (celle de id) mais aussi comme une relation qui doit être maintenue vraie entre le tissu id et les tissus qui apparaissent dans $expr$.

Pour des raisons de simplicité, nous allons d'abord présenter la notion de stream, puis celle de collection. Les opérations définies sur un stream s'appliquent à un tissu, en considérant la dimension stream du tissu. Il en va de même pour les opérations agissant sur les collections.

II.2 La notion de stream

Pour simplifier le traitement formel des programmes, Tesler et Enea [TE68] ont défini la notion de *langage à assignation unique*. Afin de prendre en compte les structures de contrôle itératives, une variable ne pouvait alors plus dénoter une valeur unique, mais devait être étendue pour pouvoir dénoter une suite (éventuellement infinie) de valeurs. De telles suites infinies de valeurs, appelées *streams*, peuvent être manipulées intensionnellement, c'est-à-dire *comme des touts* sans faire explicitement référence aux éléments la constituant.

La notion de stream permet de représenter les itérations d'une manière «mathématiquement respectable» pour citer [WA85] et même [Wat91] : « les expressions sur les suites infinies de valeurs sont aux boucles ce que la programmation structurée est aux `gotos` ».

Cette approche a donné lieu au développement du paradigme data-flow sous des formes très diverses et dans des domaines d'applications variés. Dans un modèle data-flow, ou plus généralement dans les langages de programmation déclaratifs, les données sont spécifiées par des équations (dans un langage data-flow, ce sont les streams qui sont spécifiés par des équations). Depuis le langage Lucid [WA76], bien des langages s'inspirant du modèle data-flow ont été proposés : Lustre [CPHP87] et Signal [GBBG86] dans le domaine de la programmation temps-réel, Crystal [Che86] pour la conception et le développement d'algorithmes systoliques, Palasm [SG84] pour la programmation des PLDs, Daisy [Joh83] pour la conception des circuits VLSI, 81/2 [Gia91b, MDVS96] pour la simulation parallèle, Unity [CM89] pour la spécification abstraite des programmes parallèles, etc. De plus, la définition déclarative de streams est un sous-produit naturel de l'analyse de dépendances d'un langage impératif conventionnel comme Fortran (dans ce cas, un stream correspond aux valeurs successives prises par une variable dans une boucle).

II.2.1 Les streams synchrones

Les streams synchrones ont été proposés dans les langages Lustre et Signal comme un outil pour la programmation des systèmes réactifs temps-réels. Dans ces deux langages, la succession des éléments dans un stream est fortement liée à la notion du temps : l'ordre de calcul des éléments dans le stream est le même que l'ordre de succession de ses éléments [PKL93].

Cette propriété n'est pas vraie en Lucid : le calcul de l'élément i du stream A peut nécessiter le calcul de la valeur de l'élément $i+1$ de A. Par ailleurs, le calcul de l'élément i du stream A n'est pas synchronisé avec le calcul de l'élément i d'un stream B. A l'opposé, la synchronisation des calculs est un contrainte fondamentale en Lustre ou en Signal[1]. Par exemple, l'expression

$$A + B$$

où A et B sont des streams (le résultat est un stream dont le i^{e} élément a pour valeur la somme des i^{e} éléments des streams A et B), n'est permise que si le calcul des éléments de A et de B prend place au même instant.

Ce type de contrainte requiert que les streams A, B et $A+B$ partagent une référence temporelle commune : l'*horloge* du stream. De plus, on restreint les expressions de streams de telle manière que le calcul d'un élément du stream ne fasse référence qu'à un ensemble statiquement limité de valeurs précédentes du stream [Cas92]. De cette manière, on peut identifier la succession des éléments dans un stream avec le passage du temps dans le programme.

II.2.2 Les streams 81/2

Les streams du langage 81/2 partagent avec l'approche précédente l'idée de comparer l'ordre d'occurrence d'un élément dans des streams différents, mais il est toujours possible de composer deux streams. Ainsi, le stream $A+B$ est toujours défini. Pour que cette expression ait un sens bien défini, il est nécessaire que tous les streams partagent une référence temporelle commune : un *temps global partagé*. Les instants de ce temps global partagé sont appelés *tics*. L'ensemble des instants pour lesquels un élément du stream A est évalué est appelé *horloge* de A et correspond aux *tops*[2] du stream. Les valeurs d'un stream étant « égrenées » dans le temps, nous appellerons *valeur instantanée d'un stream* la valeur de l'élément correspondant à l'instant courant.

L'horloge d'un stream correspond à l'ensemble des instants sur lesquels on doit déclencher un calcul afin de continuer à satisfaire la définition du stream. Par exemple, l'horloge de $A+B$ est constituée de l'union des horloges de A et de B puisque la valeur instantanée de $A+B$ est *susceptible* de changer quand la valeur instantanée de A ou de B change, c'est-à-dire si l'instant courant appartient à l'horloge de A ou de B. La valeur d'un stream est cependant observable à tout instant et sa valeur instantanée est la valeur calculée au premier tic précédent dans son horloge.

On peut dire que le concept de stream en Lustre ou Signal correspond à une logique de signal instantané tandis que le stream 81/2 correspond à une logique de niveau. La notion d'horloge en 81/2 correspond donc plutôt à la spécification des instants où ces valeurs doivent être produites plutôt qu'aux instants où elles doivent être consommées. De plus, la valeur courante d'un stream est toujours utilisable. Cela ne permet donc pas à 81/2 d'exprimer des contraintes temps-réel (comme par exemple d'exprimer que l'horloge de deux streams doit être la même, comme le permet la primitive `synchro` du langage Signal). En revanche, il est plus facile de composer arbitrairement des streams, ce qui est nécessaire par exemple pour l'expression des trajectoires d'un système dynamique [MGS94].

1. Elle permet de spécifier par exemple qu'une action doit avoir lieu à une heure fixe, midi pile par exemple, un genre de contrainte qu'on veut usuellement exprimer dans les applications temps-réel, cf. [BB91, Hal93] pour une introduction générale à la programmation temps-réelle synchrone.
2. La dénomination « tic » et « top » provient de l'analogie avec une horloge : entre deux « tops » se produisent des « tics ».

II.2.3 Les opérations sur les streams

Dans cette section, nous allons présenter succinctement un certain nombre d'opérateurs qui agissent sur les streams. Nous utiliserons la notation suivante :

$< \underline{1}; \underline{2}; 2; 2; \underline{3}; \underline{4}; \ldots <$

pour désigner un stream dont le premier élément vaut 1, le deuxième 2, etc. Les tics faisant partie de l'horloge du stream sont représentés par des valeurs soulignées. Dans l'exemple précédent, les tics 0, 1, et 4 (la numérotation des éléments d'un stream commence à 0) sont des tics de l'horloge mais aussi des tops.

II.2.3.1 Les constantes

Une constante n dénote le stream :

$< \underline{n}; n; n; n; n \ldots <$

La constante `Clock` 0 dénote le stream :

$< \underline{true}; \underline{true}; \underline{true}; \underline{true}; \underline{true}; \ldots <$

Plus généralement, la constante `Clock` n (avec $n \geq 1$) dénote le stream de booléens vrais ayant un top tous les n tics ; pour une valeur négative de n, le stream de booléens ayant un top tous les tics avec une probabilité de $1/n$.

II.2.3.2 Le retard ou délai

L'opérateur de retard permet de référencer la valeur d'un stream au top précédent. Un stream A retardé se note $\$A$. Nous donnons dans la table 1 un exemple de stream retardé.

Numéro de tic	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
A	$\underline{1}$	$\underline{2}$	2	2	$\underline{3}$	3	$\underline{4}$	...
$\$A$	nil	$\underline{1}$	1	1	$\underline{2}$	2	$\underline{3}$	...

TAB. 1 –: *Exemple de stream retardé*

L'horloge d'un tissu A retardé est identique à l'horloge de A au premier top près. La valeur de $\$A$ est définie à partir du tic consécutif au premier tic de A.

II.2.3.3 La quantification des équations

Il est parfois nécessaire de spécifier une valeur à un top donné (par exemple pour spécifier une valeur initiale). Pour ce faire, on utilise la *quantification* des équations. La définition :

$S@0 = A;$

permet d'indiquer que la valeur du premier top de S est donnée par la valeur du premier top de A. L'opérateur @ (*at*) limite la portée d'une équation à un top donné : celui de son argument, qui est obligatoirement une constante entière.

Le programme suivant :

$$\begin{aligned} S@0 &= A; \\ S &= B; \end{aligned}$$

définit la valeur de S comme étant celle de A sur le premier top et celle de B après. L'équation $S = B$ n'est pas quantifiée et s'applique quand aucune équation quantifiée ne peut s'appliquer. Par exemple, si :

$A \quad \Rightarrow \quad < \underline{0}; 0; 0; 0; 0 \ldots <$

$B \quad \Rightarrow \quad < \underline{3}; \underline{4}; 4; 4; \underline{5}; \ldots <$

alors :

$S \Rightarrow < \underline{0}; \underline{4}; 4; 4; \underline{5}; \ldots <$

Le signe $\Rightarrow$ se lit : « s'évalue en ». Il est possible de définir autant de quantifications que nécessaire sur un stream.

II.2.3.4 L'échantillonneur

Dans l'équation :

$S = A \,\texttt{when}\, B;$

le stream B doit être un stream de valeurs booléennes. L'horloge du stream résultat est constituée des tops de B sur lesquels B est vrai. La valeur du stream résultat est la valeur du tissu A sur les tops de la nouvelle horloge.

L'opération d'échantillonnage est une opération de contrôle : elle peut être considérée comme une vanne permettant de faire (ou de ne pas faire) passer le flot des valeurs de A. C'est pourquoi cet opérateur est appelé un *échantillonneur*, cf. table 2.

Numéro de tic	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
A	$\underline{1}$	$\underline{2}$	2	$\underline{3}$	$\underline{4}$	4	$\underline{5}$	$\underline{6}$	6
B	nil	nil	$\underline{\texttt{false}}$	$\underline{\texttt{true}}$	$\underline{\texttt{false}}$	$\underline{\texttt{true}}$	$\underline{\texttt{true}}$	$\underline{\texttt{false}}$	`false`
A `when` B	nil	nil	nil	$\underline{3}$	3	$\underline{4}$	$\underline{5}$	5	5

TAB. 2 –: *Exemple de stream échantillonné par l'opérateur* `when`

En 81/2, il existe d'autres opérations d'échantillonnage sur les streams (`until`, `after`, `at`) que nous ne présenterons pas ici. Le lecteur se reportera à [Gia91a] pour une définition des variantes existantes.

II.3 La notion de collection en 81/2

II.3.1 Collections et opérations intensionnelles

Une collection est un agrégat d'éléments qu'on peut manipuler comme un tout. Par exemple, les tableaux en APL [FI73] sont des collections. Par contre, les tableaux en Pascal ne sont pas des collections car la principale opération permise sur un tableau est l'accès à un seul élément : par conséquent il n'est pas possible de les manipuler globalement (par une opération d'addition de deux tableaux par exemple).

La structure de collection est présente dans de nombreux langages de programmation. Nous citerons, par exemple, l'ensemble (SETL [SDDS86]), le multi-ensemble (GAMMA [BCM88]), le vecteur (*LISP [Thi86]), l'imbrication de vecteurs (NESL [Ble93], 81/2 [Gia91a]) et bien sûr le tableau multidimensionnel (APL [Ive87], HPF [Ric93], MOA [HM91], indexical Lucid [AFJW95], etc.)

II.3.2 Les collections en 81/2

Les éléments d'une collection se notent entre accolades $\{\ldots,\ \ldots\}$, chaque élément étant séparé par une virgule ou un point-virgule. En 81/2, il y a deux types de collections. Le premier type de collection correspond à des imbrications homogènes de vecteurs, et le deuxième type à un ensemble labellé de valeurs quelconques (elle peuvent donc aussi être imbriquées).

La figure 1 donne un exemple de représentation arborescente d'une imbrication de collection. Les nœuds internes de l'arbre sont des collections et les feuilles de cet arbre correspondent aux éléments scalaires de la

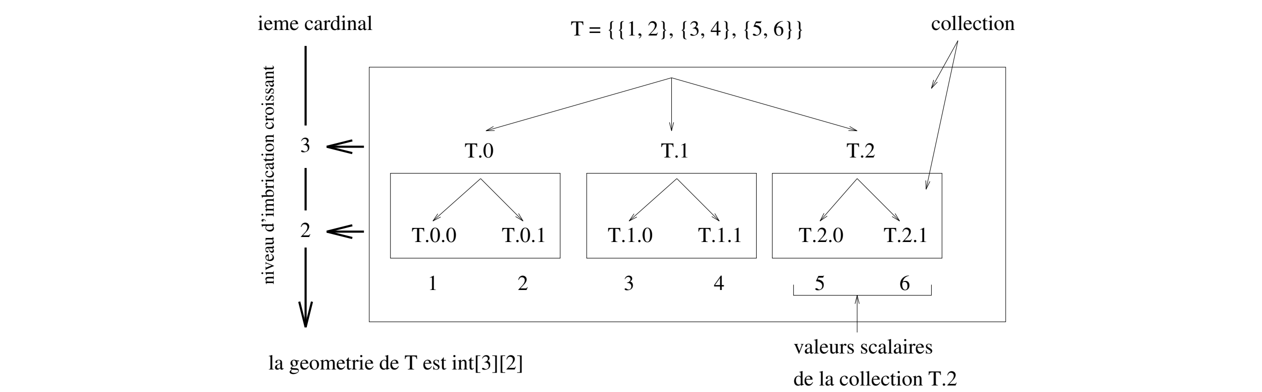


FIG. 1 –: *La géométrie des collections imbriquées homogènes en 81/2*

collection. Cette représentation permet de définir la *dimension d'une collection* dont la valeur est égale à la profondeur de l'arbre qui lui est associé, moins un. On appelle *rang* d'une collection, la liste des nombres de fils d'un nœud à chaque niveau. Par exemple, la collection :

$$\{\{1,2\},\{3,4\},\{5,6\}\}$$

a une dimension égale à 2 (il y a deux niveaux d'imbrication) et le rang est égal à $\{3,2\}$ (le premier niveau comporte 3 éléments, le second niveau en comporte 2).

II.3.3 Les collections homogènes

Une collection est dite *homogène* si toutes les régions de l'espace définies par ses imbrications sont semblables. Autrement dit, les éléments d'une collection homogène ont tous la même dimension ; si l'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 2 est un exemple de collection homogène et de collection non-homogène (une collection non-homogène est dite *hétérogène*). Pour simplifier, on considère qu'une collection homogène correspond à un *tableau* (à la `C`).

Le type d'une collection homogène en 81/2 consiste en la définition du type scalaire de ses éléments (collections feuilles) ainsi que de la *géométrie* de la collection. Le type scalaire est un élément de $\{\texttt{Int}, \texttt{Bool}, \dots\}$; la géométrie est une liste d'entiers positifs où la n^{e} valeur représente le nombre d'éléments du n^{e} niveau d'imbrication. Le nombre d'éléments dans la géométrie est le *cardinal* de la collection.

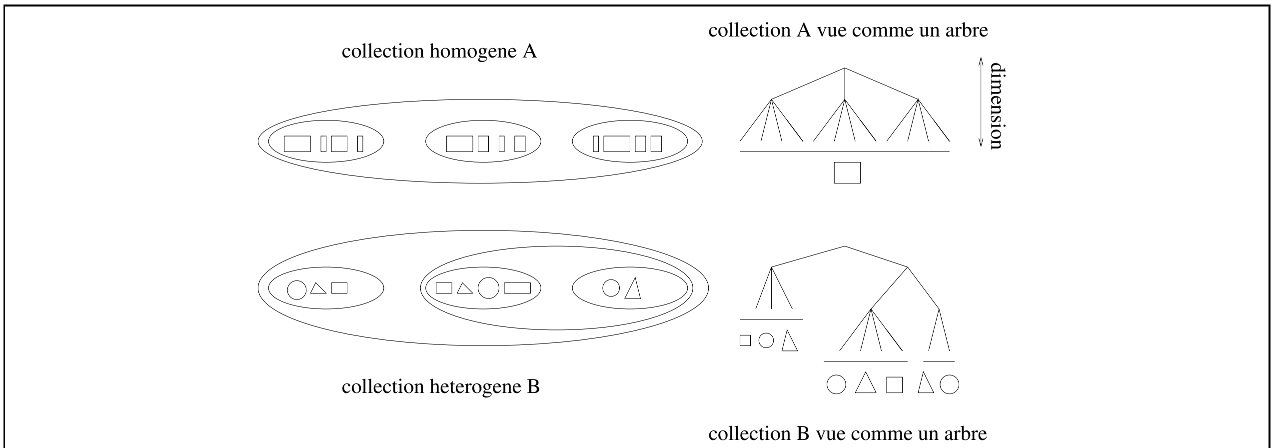


FIG. 2 –: *Imbrication homogène et hétérogène de collections. La collection A est homogène car tous les éléments de A ont la même dimension et chaque élément de A a le même nombre de frères dans la structure. On peut considérer que A est un tableau. Ce n'est pas le cas de la collection B : les éléments de B n'ont pas tous la même dimension et le nombre de frères d'un élément, à un niveau donné, n'est pas constant.*

Le typage des équations 8$_{1/2}$ est implicite : on ne spécifie pas (ou au minimum) les informations de type et de géométrie sur les collections. Toute constante scalaire peut se transformer implicitement en une collection constante. Par exemple, la constante 5 correspond à une collection homogène, de géométrie quelconque, dont la valeur en tout point est égale à 5. La géométrie d'une (collection) constante est automatiquement inférée par le compilateur quand c'est possible, c'est-à-dire quand le programme fournit suffisamment d'informations sur les géométries. Une constante scalaire permet donc de dénoter de manière uniforme plusieurs tissus constants de géométries différentes.

Dans la suite de cette section, nous allons présenter un certain nombre d'opérateurs permettant de construire des expressions. Ces opérateurs, sauf mention contraire explicite, s'appliquent uniquement à des tableaux.

II.3.3.1 L'α-extension des opérateurs scalaires

L'α-extension, notée « ^ » , est un opérateur qui permet d'étendre une opération scalaire sur des éléments de type S en une opération sur des collections d'éléments de type S. L'α-extension se fait par l'application de la fonction élément par élément. Par exemple, l'expression suivante applique l'opérateur d'addition scalaire + aux deux collections en argument :

$$+\ \hat{}\{1,2,3\}\{4,5,6\} \Rightarrow \{5,7,9\}$$

En 8$_{1/2}$, l'α-extension est implicite pour tous les opérateurs arithmétiques (+, $*$...), les opérateurs relationnels (<, >...), les opérateurs logiques ($\wedge$, $||$...), et la conditionnelle `if`...`then`...`else`.... La notation infixe usuelle est utilisée pour les opérateurs standards. L'α-extension de l'addition des collections précédentes s'écrit alors simplement :

$$\{1,2,3\} + \{4,5,6\}$$

Les opérateurs implicitement α-étendus sont naturellement surchargés pour permettre une application à des expressions, qu'elles soient scalaires ou de géométrie quelconque.

II.3.3.2 La β-réduction

La β-réduction[1], notée « \ », 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. Par exemple, dans l'expression suivante :

$$\begin{array}{lll} +\backslash\{1,2,3\} & \Rightarrow 1+(2+3) & \Rightarrow\ 6 \\ \mathtt{max}\backslash\{1,2,3\} & \Rightarrow \mathtt{max}(1,\mathtt{max}(2,3)) & \Rightarrow\ 3 \end{array}$$

l'utilisation de la β-réduction avec des opérateurs + ou `max` permet de sommer les éléments d'une collection et respectivement d'en sélectionner le plus grand élément.

L'ordre de réduction n'étant pas précisé, la fonction f doit être associative si l'on veut un résultat déterministe. En 8$_{1/2}$, cet ordre est dépendant de l'implémentation et il est de la responsabilité du programmeur de n'utiliser que des fonctions associatives[2].

Si la collection est imbriquée, la β-réduction s'applique aux éléments de la première dimension. Par exemple :

$$\mathtt{max}\backslash\ \{\{7,12,14,8\},\{16,10,11,10\}\} \Rightarrow \{16,12,14,10\}$$

La β-réduction de chacun des éléments du vecteur se fera à l'aide de l'α-extension :

$$(\mathtt{max}\backslash)\hat{}\{\{7,12,14,8\},\{16,10,11,10\}\} \Rightarrow\ \{14,16\}$$

1. La β-réduction du data-parallélisme qui nous concerne ici ne doit pas être confondue avec la β-réduction du λ-calcul.
2. On laisse au lecteur le soin d'imaginer les différents résultats de la β-réduction d'une collection par un opérateur de division.

II.3.3.3 L'opération de balayage ou scan

La β-réduction se généralise grâce à l'opérateur de balayage noté «\\» : cet opérateur permet de calculer une collection dont le i^{e} élément a pour valeur la réduction des i^{es} premiers éléments de la collection. Par exemple, sur des collections de dimension 1 (des vecteurs) :

$$\begin{array}{lcl} +\backslash\backslash\ \{1,1,1,1\} & \Rightarrow & \{1,2,3,4\} \\ \mathtt{max}\backslash\backslash\ \{1,3,2,4\} & \Rightarrow & \{1,3,3,4\} \end{array}$$

et, sur une collection de dimension 2 :

$$max\backslash\backslash \left\{ \begin{array}{l} \{\ 7,\ 12,\ 14,\ 8\,\} \\ \{\,16,\ 10,\ 11,\ 10\,\} \\ \{\ 1,\ 2,\ 3,\ 20\,\} \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \{\ 7,\ 12,\ 14,\ 8\,\} \\ \{\,16,\ 12,\ 14,\ 10\,\} \\ \{\,16,\ 12,\ 14,\ 20\,\} \end{array} \right\}$$

II.3.3.4 Coupure et extension d'une collection

L'opérateur de coupure et d'extension « : [ ] » permet d'étendre et de restreindre les dimensions d'une collection. L'expression $e : [c_1, \ldots, c_n]$ permet de modifier le nombre d'éléments dans chacune des dimensions, par troncature ou réplication. Le terme $[c_1, \ldots, c_n]$ représente le rang du résultat. Par exemple :

$$\begin{array}{lcl} A & \equiv & \{1,2,3,4\} \\ B & \equiv & \{\{1,2\},\{3,4\},\{5,6\}\} \end{array}$$

alors :

$$\begin{array}{lcl} A : [2] & \Rightarrow & \{1,2\} \\ B : [2,3] & \Rightarrow & \{\{1,2,1\},\{3,4,3\}\} \end{array}$$

II.3.3.5 La concaténation des collections

L'opérateur de concaténation, noté « $\#_n$ » permet de concaténer deux collections en les mettant « bout à bout » suivant la n^{e} dimension (on abrège les opérateurs « $\#_0$ » et « $\#_1$ » respectivement par « # » et « $\#\hat{}$ ». Par exemple, la concaténation des expressions A et B suivantes :

$$\begin{array}{lcl} A & \equiv & \left\{ \begin{array}{c} \{1,1,1\} \\ \{2,2,2\} \end{array} \right\} \\ B & \equiv & \left\{ \begin{array}{c} \{3,3,3\} \\ \{4,4,4\} \end{array} \right\} \end{array}$$

suivant la première dimension a pour résultat :

$$A \#_0 B \quad \Rightarrow \quad \left\{ \begin{array}{c} \{1,1,1\} \\ \{2,2,2\} \\ \{3,3,3\} \\ \{4,4,4\} \end{array} \right\}$$

et suivant la seconde dimension :

$$A \#_1 B \quad \Rightarrow \quad \left\{ \begin{array}{c} \{1,1,1,3,3,3\} \\ \{2,2,2,4,4,4\} \end{array} \right\}$$

II.3.3.6 La sélection

Il est possible d'accéder à la valeur d'un point d'une collection (homogène ou hétérogène) grâce à l'opérateur de sélection qui se note «e . n» où e est une collection et n un entier positif. L'indexation des collections commence à 0. Par exemple, pour $A = \{\{1,2\},\{3,4\}\}$:

A . 0	$\Rightarrow$	$\{1,2\}$
A . 1	$\Rightarrow$	$\{3,4\}$
A . 2	$\Rightarrow$	opération invalide car elle correspond à un accès « en dehors des bornes »
A . 0 . 1	$\Rightarrow$	2

II.3.3.7 La permutation généralisée

L'opérateur de *permutation* permet de ré-ordonner les éléments d'une collection suivant les indices d'une autre collection. Soit $A = \{1,2,3,4,5,6\}$ une collection et $I = \{0,0,2,1\}$ la collection des indices des éléments à sélectionner dans A. Alors :

$$A(I) \Rightarrow \{1,1,3,2\}$$

et $\forall i \in [0,5], A(I)\ .\ i = A\ .\ (I\ .\ i)$. La permutation se note avec des parenthèses. Voici un autre exemple où les éléments de I ne sélectionnent pas des scalaires, mais des collections. Si $B = \{\{1,2,3\},\{4,5,6\},\{7,8,9\}\}$, alors

$$B(I) \Rightarrow \{\{1,2,3\},\{1,2,3\},\{7,8,9\},\{4,5,6\}\}$$

L'opération de permutation en $8_{1/2}$ est plus générale : la collection des indices peut être une collection imbriquée. Par exemple :

$$\begin{array}{lcl} J & = & \{\{0,1\},\{2,0\}\} \\ A(J) & \Rightarrow & \{\{1,2\},\{3,1\}\} \\ B(J) & \Rightarrow & \{\{\{1,2,3\},\{4,5,6\}\},\{\{7,8,9\},\{1,2,3\}\}\} \end{array}$$

II.3.3.8 L'opérateur iota

L'opérateur *iota* [1] est noté «$'$» en $8_{1/2}$. Si l'opérateur iota a pour argument un entier, alors le résultat d'une expression $'n$ est la collection des 0 à $n-1$ premiers entiers. Dans le cas où on applique cet opérateur à une collection (on parle alors de l'opérateur iota *généralisé*), celui-ci retourne la collection des indices correspondant à la géométrie de l'argument. Si on parle en termes de coordonnées d'un point pour désigner la suite des indices permettant d'accéder à un point, alors l'opérateur iota permet de construire la collection des coordonnées.

L'utilisation conjointe des opérateurs iota et de permutation permet d'effectuer le décalage des éléments d'une collection. Par exemple, $'A + \{\{\{1,0\}\}\}$ est une collection qui indice les voisins nord de A, en considérant la direction nord comme la direction où l'indice i d'un élément de position (i,j) est décrémentée. Une sélection de A suivant cette collection permet de décaler la collection A dans le sens des lignes décroissante (correspondant à un décalage vers le «nord» dans un voisinage NEWS), c'est-à-dire de construire la collection où tous les éléments occupent une position au «nord» de leur position initiale dans A.

II.3.4 Les collections hétérogènes : les systèmes

Un *système* est une collection d'éléments nommés. Par exemple, l'expression suivante :

$$A \equiv \{a = 1, b = \{2,2\}\}$$

définit une collection de deux éléments, dont le premier élément a pour valeur la collection constante 1 et le second élément a pour valeur la collection $\{2,2\}$. On peut maintenant accéder aux éléments de cette collection, de façon classique, par une position :

$$A\ .\ 0 \Rightarrow 1$$

1. L'opérateur iota provient du langage APL.

$$A \,.\, 1 \quad \Rightarrow \quad \{2,2\}$$

mais aussi par l'identificateur qui est attaché à la définition :

$$\begin{array}{lll} A \,.\, a & \Rightarrow & 1 \\ A \,.\, b & \Rightarrow & \{2,2\} \end{array}$$

Par ailleurs, on assimile une collection hétérogène à un système en utilisant un mécanisme de nommage implicite : chaque élément d'une collection hétérogène est identifié par sa position. Ainsi, l'expression :

$$\{1, \{2,2\}\}$$

est une collection qui n'est pas un tableau car ses éléments sont de types différents. Cette collection est un système, l'identificateur du premier élément étant le label numérique 0, l'identificateur du second élément étant le label 1. On surcharge naturellement les opérateurs de concaténation, de définition (les accolades) et de sélection, afin de faciliter l'écriture d'expressions impliquant les systèmes et les tableaux.

Le nommage des éléments d'un système ressemble à une équation en 81/2 : en fait, c'est une équation 81/2. Par exemple, le système :

$$\{a = 1,\, b = \{a+1, a+1\}\}$$

s'évalue en :

$$\{a = 1,\, b = \{2,2\}\}$$

Un système correspond donc à un paquet d'équations et permet de structurer le code 81/2.

II.4 Combinaison de collections et de streams : le tissu 81/2

La programmation en 81/2 se fait à travers une structure de données unique, le *tissu*. Un tissu est :

– soit un stream dont la valeur instantanée est une collection homogène, dont la géométrie est indépendante de l'instant considéré,

– soit un système dont les éléments sont des streams.

Ces deux aspects étaient plus ou moins confondus dans [Gia91a]. En effet, on peut remarquer que :

– un stream de collections homogènes peut se voir comme une collection de streams (un stream par élément) qui ont tous la même horloge,

– une collection homogène est un cas particulier de collection hétérogène (grâce au schéma de nommage implicite).

Pour le moment, nous maintiendrons une claire distinction entre les notions de tableau et de système. Dans le chapitre suivant, nous décrivons quelques exemples d'expressions 81/2.

Chapitre III

Exemples d'expressions 81/2

Dans le chapitre précédent nous avons décrit rapidement le langage déclaratif 81/2. Nous décrivons dans ce chapitre des exemples de programmes 81/2 naturellement orientés vers des applications de simulation et par conséquent la manipulation d'espaces. Le plan de ce chapitre est le suivant :

- Le premier exemple (cf. section III.1) décrit la simulation d'un système réactif dont les niveaux internes varient au cours du temps.
- Le second exemple (cf. section III.2) correspond à une implémentation en 81/2 d'équations différentielles couplées sur un réseau : le modèle chimique de diffusion–réaction d'A. Turing développé initialement pour tenter d'expliquer la formation de motifs dans un substrat homogène.
- Nous modélisons ensuite un phénomène de croissance simple, celui de la coquille d'un mollusque (cf. section III.3). Ce phénomène de croissance, discrétisé dans l'espace et dans le temps, utilise une collection support de dimension suffisante pour représenter les étapes successives de la simulation.
- Enfin, nous montrons la capacité du langage 81/2 à définir récursivement des structures de données (cf. section III.4).

Tous les exemples décrit dans ce chapitre ont été implémentés et exécutés dans un environnement 81/2 permettant la compilation des équations [MGS94, Mic96a]. Nous avons intégré à l'environnement d'exécution des programmes 81/2 une liaison avec un outil de tracé graphique, le logiciel Gnuplot [WKC+90]. Cette interface est décrite dans l'annexe D.

III.1 Le wlumf

Nous souhaitons décrire un système réactif particulier, un «wlumf» [Leg91] : c'est une créature artificielle dont le comportement, qui consiste principalement à manger, est régulé par le niveau de certains états internes (on se référera à [Mae91] pour un tel modèle dans le cadre de la simulation en éthologie).

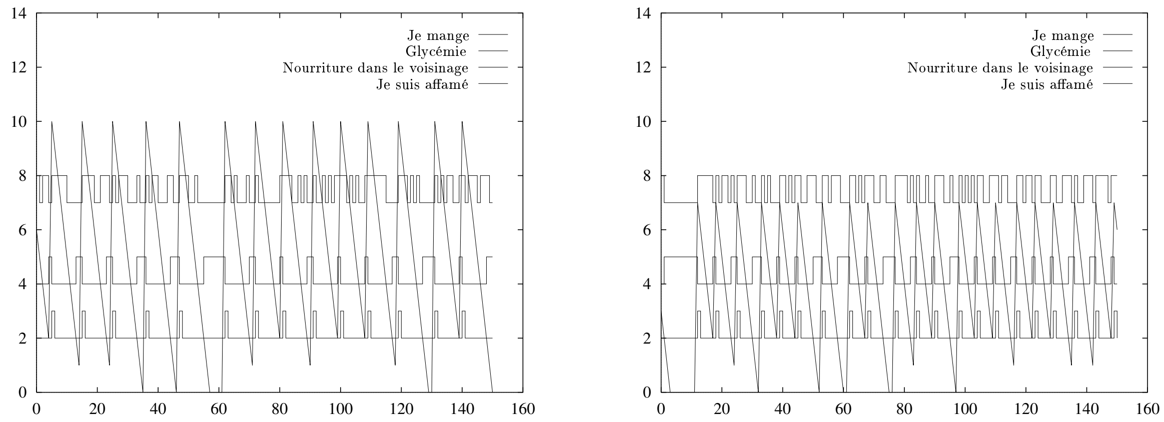
Plus précisément, un wlumf est affamé quand sa glycémie est inférieure à une valeur $limg_{inf}$. D'autre part, il peut manger s'il dispose de nourriture dans son environnement (les valeurs des variables de l'environnement changent de façon aléatoire). Le métabolisme du wlumf est tel que quand il mange, sa glycémie atteint la valeur $limg_{sup}$, puis décroît jusqu'à zéro à une vitesse de dec_G par unité de temps. Au début de la simulation, le wlumf a une glycémie égale à $init_G$. Toutes les variables décrivant le comportement du wlumf sont scalaires. Essentiellement, le wlumf est constitué de compteurs et de bascules qui sont déclenchés et ré-initialisés à des vitesses différentes. La définition du wlumf consiste en deux définitions ; un environnement, où sont définis un tissu entier *cpt* qui représente un compteur, incrémenté avec une probabilité de 0.5 et un tissu booléen *nourriture* qui à une valeur vraie quand *cpt* est un multiple de 2 :

$$
\begin{array}{lcl}
\multicolumn{3}{l}{env = \{} \\
\quad cpt & = & \$cpt + 1 \,\texttt{when}(\texttt{Clock} - 2);\ cpt@0 = 0; \\
\quad \texttt{boolean}\ nourriture & = & 0 == (cpt\%2); \}
\end{array}
$$

Ces définitions sont utilisées par la définition du wlumf :

$$
\begin{array}{lcl}
\multicolumn{3}{l}{wlumf = \{} \\
\quad affame & = & (glucose < 3); \\
\quad affame@0 & = & \texttt{false}; \\
\quad glucose & = & \texttt{if}\ mange\ \texttt{then}\ 10\ \texttt{else}\ \max(0, \$glucose - 1)\ \texttt{when}\ \texttt{Clock}; \\
\quad glucose@0 & = & 6; \\
\quad mange & = & \$affame\ \&\&\ env\ .\ nourriture; \\
\quad mange@0 & = & false; \}
\end{array}
$$

où le système $wlumf$ définit : le tissu booléen $affame$ qui a une valeur vraie quand la valeur du tissu $glucose$ passe en dessous d'un seuil limite ; le tissu entier $glucose$ qui a pour valeur la glycémie du wlumf à un instant donné et enfin le tissu booléen $mange$ qui a une valeur vraie quand le wlumf est affamé et que de la nourriture se trouve dans son environnement (cf. figure 1).



(a) Une première évaluation

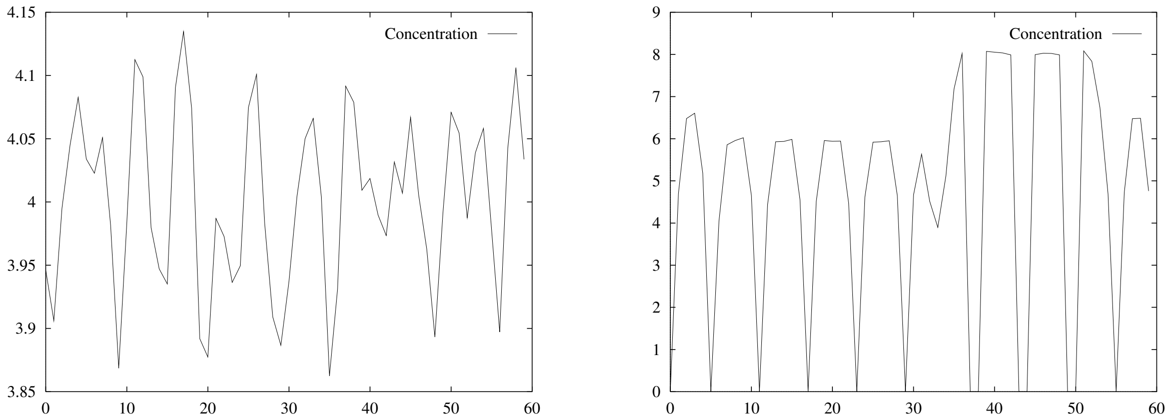
(b) Une évaluation avec des initialisations différentes

FIG. 1 –: *Comportement d'un système dynamique hybride. Simulation du comportement d'un « wlumf » dans des environnements où les valeurs initiales sont différentes.*

III.2 L'équation de diffusion–réaction d'A. Turing

Nous donnons ici un exemple d'équations différentielles couplées sur un réseau, exemple correspondant à la deuxième colonne dans la classification donnée table 1 de la page 4.

Le mathématicien A. TURING souhaitait savoir si la forme des taches sur un pelage d'animal pouvaient avoir une origine autre que génétique. Son modèle montre que des taches *réalistes* peuvent être produites comme le résultat d'un processus ne faisant pas intervenir de matériel génétique. Il a proposé un modèle de réaction chimique pour expliquer la formation de motifs, comme le résultat de concentrations de certaines substances chimiques, dans un milieu initialement homogène. TURING appelle ces agents chimiques des *morphogènes* [Tur52] et le modèle de réaction chimique, le modèle de *diffusion–réaction*. La forme générale des équations de TURING est [Tur91] :



(a) État du système après 100 pas de simulation (b) État du système après 1000 pas de simulation

FIG. 2 –: *Équation de diffusion–réaction dans un tore, d'après A.* TURING. *Le résultat de la simulation est représenté après 100 pas de temps élémentaires (figure (a)), et après 1000 pas de temps, alors que le phénomène décrit par le système dynamique est stabilisé (figure (b)). La variation constatée peut correspondre à une quantité de colorant et le résultat peut s'interpréter comme des taches périodiquement disposées le long du tore.*

$$
\begin{aligned}
\frac{\partial x}{\partial t} &= F(x,y) + D_x \nabla^2 x \\
\frac{\partial y}{\partial t} &= G(x,y) + D_y \nabla^2 y
\end{aligned}
$$

où les équations définissent que le changement de concentration pour un morphogène x à un instant t dépend d'une fonction F des concentrations locales des morphogènes x et y et d'un processus de diffusion de x. La constante D_x exprime la vitesse de diffusion du morphogène x et $\nabla^2 x$ est une mesure de la variation de la concentration de x par rapport à ses voisins. Depuis l'introduction de ces idées, de nombreux systèmes ont été étudiés afin de mettre en évidence l'action de morphogènes dans un processus de diffusion–réaction lors de l'apparition de motifs.

Par exemple, le système différentiel de départ [BL74] qui gouverne l'état de la cellule r est :

$$
\frac{\Delta x_r}{dt} = \frac{1}{16}(16 - x_r y_r) + (x_{r+1} - 2x_r + x_{r-1}) \quad (1)
$$

$$
\frac{\Delta y_r}{dt} = \frac{1}{16}(x_r y_r - y_r - \beta) + (y_{r+1} - 2y_r + y_{r-1}) \quad (2)
$$

où x et y représentent deux espèces chimiques en réaction, diffusant sur un tore de cellules indexées par r. Dans ce modèle, le temps de la réaction chimique est continu alors que l'espace de la diffusion correspond à l'espace discret des cellules. La programmation en 8½ de ce modèle utilise bien sûr l'aspect stream d'un tissu pour représenter le temps et l'aspect collection pour représenter l'espace des cellules. Le programme 8½ est le suivant :

$$
\begin{array}{lcl}
iota & = & '60; \\
droite & = & \mathtt{if}(iota == 0)\,\mathtt{then}\,59\,\mathtt{else}(iota - 1); \\
gauche & = & \mathtt{if}(iota == 59)\,\mathtt{then}\,0\,\mathtt{else}(iota + 1); \\
 & & \\
rsp & = & 1.0/16.0; \\
diff1 & = & 0.25; \\
diff2 & = & 0.0625;
\end{array}
$$

$$
\begin{array}{lcl}
x & = & \$x + \$dx \ \texttt{when}\,\texttt{Clock}; \quad x@0 = 4.0; \\
y & = & \max(0.0, \$y + \$dy) \ \texttt{when}\,\texttt{Clock}; \quad y@0 = 4.0; \\
beta & = & 12.0 + rand(0.05 * 2.0) - 0.05; \\
xdiff & = & x(droite) + x(gauche) - 2.0 * x; \\
ydiff & = & y(droite) + y(gauche) - 2.0 * y; \\
dx & = & rsp * (16.0 - x * y) + xdiff * diff1; \\
dy & = & rsp * (x * y - y - beta) + ydiff * diff2;
\end{array}
$$

Dans le programme écrit en $8_{1/2}$, on retrouve exactement les deux mêmes équations (1) et (2) sous la forme des tissus dx et dy, les autres équations correspondant à des calculs intermédiaires $xdiff$..., au calcul de la condition initiale *beta* ou à l'accès aux voisins par une opération data-parallèle de *gather* : $x(droite)$... Une simulation est illustrée par la figure 2.

À titre de comparaison, nous avons reproduit en annexe A le programme C effectuant les même calculs que le programme $8_{1/2}$ que nous venons de décrire. La puissance expressive de $8_{1/2}$ est évidente :

1. le programme $8_{1/2}$ reflète *exactement* la décomposition du problème et les équations (1) et (2) sont directement traduites en $8_{1/2}$ dans les équations dx et dy,
2. 70 lignes C sont nécessaires à l'expression du problème en C contre 13 en $8_{1/2}$.

L'efficacité du code $8_{1/2}$ est excellente car le ratio du temps d'exécution $8_{1/2}$ par rapport au temps d'exécution en C n'est que de 1.72, en faveur de C [DV96, pp 21].

III.3 Modélisation et simulation d'un processus de croissance simple : l'ammonite

Nous décrivons dans cette section la modélisation et la simulation d'un processus de croissance simple : celui de l'ammonite. Nous détaillons les étapes de la modélisation, qui extrait d'un processus complexe une description informatique simple, puis nous donnons une version $8_{1/2}$ du programme correspondant à la simulation et enfin nous donnons le résultat de la simulation.

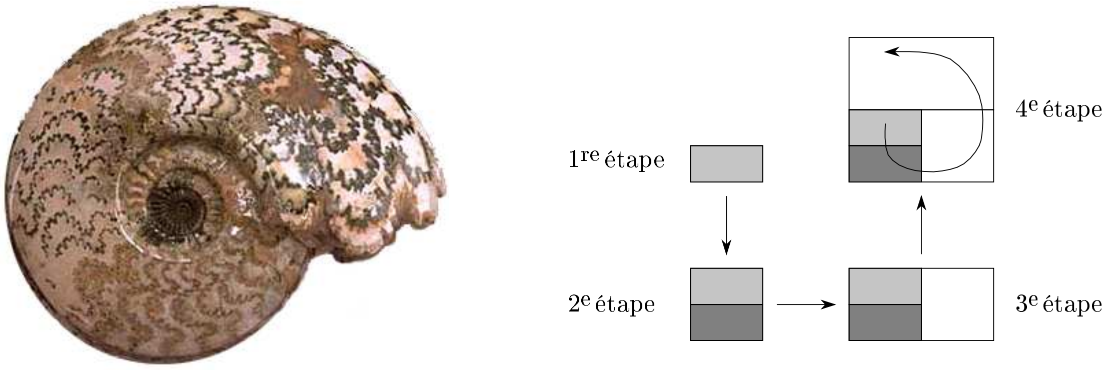
III.3.1 Modélisation du processus de croissance

La spirale logarithmique est le modèle paradigmatique de la croissance d'un mollusque fossile : l'*ammonite*, un membre du groupe des céphalopodes éteint au crétacé. On s'intéresse uniquement à la modélisation de la croissance de la coquille du mollusque.

Nous présentons ici une version stylisée du modèle, où la coquille est représentée sous la forme d'un carré dans un espace discret, et où les développements successifs se font suivant un des côtés du carré. On discrétise le temps suivant lequel la croissance va se dérouler : on considère qu'à chaque étape de la simulation, une transformation de la coquille a lieu. Le côté où cette transformation s'opère est différent à chaque étape et se déroule suivant un sens anti-horaire. Par exemple, si on choisit un maillage rectangulaire de l'espace muni d'un voisinage de VON NEUMANN, la première étape de développement ayant lieu suivant le coté sud de la coquille, la seconde étape aura lieu suivant le côté est, la troisième suivant le côté nord, etc. De plus, à une étape n du développement, la nouvelle portion de la coquille a une surface égale à la surface de la coquille à l'étape $n-1$. La figure 3 illustre la modélisation du processus de croissance que nous venons de décrire.

III.3.2 Expression du processus de croissance en $8_{1/2}$

Le domaine de croissance de l'ammonite est représenté sous la forme d'un tableau dont la taille borne supérieurement toutes les générations au cours de la simulation. Ce tableau représente le *support* de la simulation. Cependant tous ses éléments ne sont pas significatifs : les valeurs différentes de 0 «représentent» l'objet modélisé. Le programme $8_{1/2}$ correspondant à cette simulation est le suivant :



(a) Une ammonite.

(b) Représentation des quatre premières étapes du processus de développement de la coquille de l'ammonite.

FIG. 3 –: *Modélisation du processus de croissance de la coquille de l'ammonite.*

$$
\begin{array}{lcl}
cpt & = & \$cpt + 1 \;\texttt{when Clock}; \quad cpt@0 = 0; \\
dir & = & cpt\%4; \\
w@0 & = & \{\{\{2,0\}\}\} : [32,32,2]; \\
w[32,32,2] & = & \texttt{switch}(dir == 1)\;\texttt{then}\;\$w\;\texttt{else}\;\texttt{switch}(dir == 2)\;\texttt{then}\;\$w + \$w\;\texttt{else} \\
 & & \texttt{switch}(dir == 3)\;\texttt{then}\;\$w\;\texttt{else}\;\$w + \$w; \\
l@0 & = & \{\{\{0,2\}\}\} : [32,32,2]; \\
l[32,32,2] & = & \texttt{switch}(dir == 1)\;\texttt{then}\;\$l + \$l\;\texttt{else}\;\texttt{switch}(dir == 2)\;\texttt{then}\;\$l\;\texttt{else} \\
 & & \texttt{switch}(dir == 3)\;\texttt{then}\;\$l + \$l\;\texttt{else}\;\$l; \\
A@0 & = & (0 : [15,32] \,\#\, (0 : [2,15] \,\#\hat{}\, 1 : [2,2] \,\#\hat{}\, 0 : [2,15]) \,\#\, 0 : [15,32]; \\
A[32,32] & = & \texttt{if}\;\$A\;\texttt{then}\;\$A + 1\;\texttt{else}\;\texttt{if}(\$A)\;\texttt{.}\;dep\;\texttt{then}\;1\;\texttt{else}\;0\;\texttt{when Clock}; \\
North & = & \texttt{if}\;north > 31\;\texttt{then}\;31\;\texttt{else}\;\texttt{if}\;north < 0\;\texttt{then}\;0\;\texttt{else}\;north; \\
West & = & \texttt{if}\;west > 31\;\texttt{then}\;31\;\texttt{else}\;\texttt{if}\;west < 0\;\texttt{then}\;0\;\texttt{else}\;west; \\
East & = & \texttt{if}\;east > 31\;\texttt{then}\;31\;\texttt{else}\;\texttt{if}\;east < 0\;\texttt{then}\;0\;\texttt{else}\;east; \\
South & = & \texttt{if}\;south > 31\;\texttt{then}\;31\;\texttt{else}\;\texttt{if}\;south < 0\;\texttt{then}\;0\;\texttt{else}\;south; \\
north & = & {}'A - \$w; \quad east = {}'A - \$l; \quad south = {}'A + \$w; \quad west = {}'A + \$l; \\
dep & = & \texttt{switch}(dir == 1)\;\texttt{then}\;East\;\texttt{else}\;\texttt{switch}(dir == 2)\;\texttt{then}\;South\;\texttt{else} \\
 & & \texttt{switch}(dir == 3)\;\texttt{then}\;West\;\texttt{else}\;North;
\end{array}
$$

Le tissu entier *cpt* implémente un compteur qui va représenter le numéro de génération de la coquille. Le tissu entier *dir* spécifie suivant quelle direction il est nécessaire de faire croître la coquille : c'est un tissu qui est égal au reste de la division entière de *cpt* par 4. Les tissus w et l représentent la taille de l'ammonite suivant un axe est–ouest ou un axe nord–sud. En fonction du côté suivant lequel la croissance doit avoir lieu, le tissu *dep* est égal à la portion croissance de l'ammonite. Enfin, le tissu A spécifie la collection support de la croissance de l'ammonite où les valeurs différentes de 0 représentent les générations successives de la croissance. Les générations les plus anciennes correspondent à des points de la collection support dont la valeur est la plus elevée.

Le programme 81/2 n'est pas immédiat, pour deux raisons :

1. il est nécessaire de construire explicitement les permutations qui correspondent à un déplacement suivant les quatres directions ;
2. il faut gérer explicitement les éléments définis et indéfinis de la représentation de l'ammonite dans le

tableau.

Ces deux problèmes seront résolus par l'introduction des GBF dans la partie suivante. On comparera avec le programme équivalent proposé dans la quatrième partie (cf. section XIV.4.3, page 219).

III.3.3 Simulation du processus de croissance

La figure 4 illustre le résultat après 7 pas de simulations. Il apparaît que la taille de la région, utilisée pour représenter les 7 premières générations, est inférieure à la taille du tableau. En effet, dans la mesure où on ne connaît pas la taille du domaine de l'espace qui représente la coquille à l'issue de la simulation, il est nécessaire d'utiliser un tableau dont les dimensions excèdent les besoins de la représentation du phénomène.

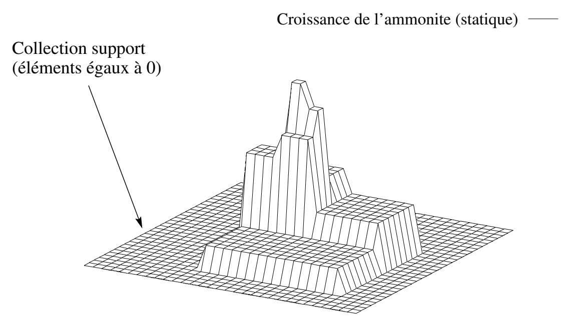


FIG. 4 –: *Simulation du processus de croissance de l'ammonite. Les cellules correspondant à une génération de la coquille se situent à la même hauteur.*

III.4 Définition récursive de tableaux

Le langage 81/2 permet la définition récursive de structures de données. Par exemple, le programme *iota* suivant :

$$iota \quad = \quad 0 \,\#\, (1 + iota : [4]); \tag{3}$$

définit une équation récursive. Il s'agit d'une forme particulière de récursion, la récursion *spatiale* car la référence récursive *iota* n'apparaît pas comme opérande d'un opérateur de délai \$ (voir le compteur *cpt* dans le programme du processus de croissance de l'ammonite pour un exemple de récursion *temporelle* classique).

Oublions l'aspect stream des tissus et expliquons pourquoi l'équation (3) a pour solution la collection $\{0, 1, 2, 3, 4\}$. L'équation est valide pour chaque élément de la collection et on peut par conséquent affirmer que :

$$iota \,.\, i \quad = \quad (0 \,\#\, 1 + iota : [4]) \,.\, i$$

pour $0 \leq i < 5$ (c'est l'inférence de la géométrie des tissus qui permet de définir les bornes de i, on verra à ce sujet [Gia91a]). On obtient, grâce aux propriétés de la concaténation :

$$\begin{aligned} iota \,.\, 0 &= 0 \\ iota \,.\, i &= 1 + iota \,.\, (i-1) \qquad 1 \leq i < 5 \end{aligned}$$

et par conséquent :

$$\begin{aligned} iota \,.\, 1 &= 1 + iota \,.\, 0 = 1 + 0 = 1 \\ iota \,.\, 2 &= 1 + iota \,.\, 1 = 1 + 1 = 2 \end{aligned}$$

$$\begin{aligned} iota \,.\, 3 &= 1 + iota \,.\, 2 = 1 + 2 = 3 \\ iota \,.\, 4 &= 1 + iota \,.\, 3 = 1 + 3 = 4 \end{aligned}$$

Remarquons que pour calculer la valeur de la collection $iota \,.\, i$, il est nécessaire de connaître la valeur de $iota \,.\, (i-1)$. Nous exprimons cette contrainte par le fait que les éléments i et $i-1$ sont *voisins* [1]. De ce point de vue, les voisins d'un élément P sont les éléments accédés pour calculer la valeur de P. Dans l'équation (3), les voisins sont définis *implicitement* par l'action de la concaténation de 0 à gauche, qui décale la collection vers la droite, créant ainsi un ordre linéaire entre les éléments. La figure 5 illustre la progression des calculs.

Nous ne désirons pas réduire le concept de tableau (récursif) à celui de fonction. Ainsi, on ne permet que les équations récursives dont le schéma de calcul correspond à une progression dans le calcul des valeurs, d'un point à un point voisin : le calcul doit se propager le long d'un déplacement (pour un indice donné, le calcul se fait soit sur les indices croissants soit sur les indices décroissants). Pour les tableaux 81/2, cette contrainte a pour conséquence d'interdire les expressions récursives qui impliquent une permutation, et de n'admettre que les expressions impliquant une concaténation et/ou une opération de coupure-extension. La théorie associée est présentée dans [Gia91a, GS93].

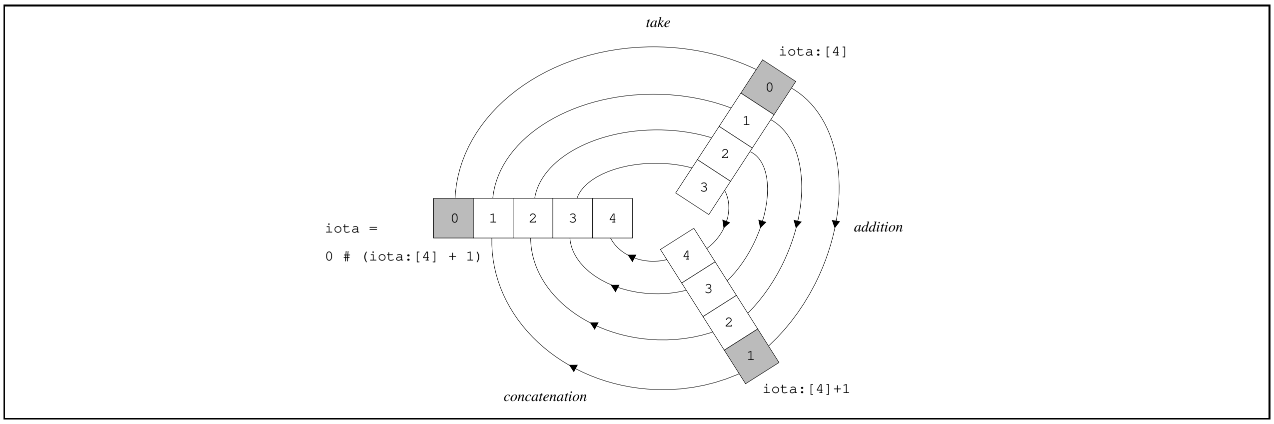


FIG. 5 –: *Calcul de iota[5] par une méthode de récursion spatiale. L'indice correspond au numéro de génération de la valeur de la collection iota : à l'initialisation, seule la valeur de iota* . 0 *est connue ; ensuite, par propagation du calcul vers la droite, l'ensemble des valeurs de la collection est calculé.*

1. Cette terminologie est motivée par le postulat fondamental que, en physique, les actions sont locales dans l'espace (pas d'actions instantanées à distance) et dans le temps (causalité). Cela signifie que l'état d'un point dépend uniquement de l'état de ses voisins spatio-temporel. En conséquence, si la valeur en un point est nécessaire pour calculer la valeur en un autre point, ces deux points sont forcément voisins dans un certain espace, l'espace (peut-être abstrait) dans lequel le phénomène a lieu.

Deuxième partie

GBF

Chapitre IV

Tableaux et représentation des espaces homogènes

La deuxième partie de ce document est consacrée à la représentation d'un espace homogène. De telles structures de données se rencontrent quand on discrétise de manière régulière un domaine de l'espace physique, par exemple pour la résolution numérique d'une équation aux dérivées partielles.

Le tableau est habituellement utilisé pour la représentation informatique des espaces homogènes. Mais il présente de nombreux défauts : la topologie n'est pas explicite et elle reste trop simple, il ne permet pas la représentation de formes arbitraires, etc.

Notre idée est de se concentrer sur la notion de voisinage d'un point et sur les déplacements permis dans l'espace homogène. Un voisinage peut alors se coder par la présentation d'un groupe. Cela constitue le fondement de la notion de *forme*. Du point de vue informatique, une forme correspond au type d'un tableau. Mais ce type est beaucoup plus riche que les types habituels.

Représenter un tableau comme une fonction sur $\mathbb{Z}^n$ est l'approche des *champs de données*. Cette vision permet au programmeur de manipuler les tableaux en intension. Dans cette lignée, nous étendons alors le concept de tableau à celui de fonction partielle sur une forme. Le résultat est un *champ de données basé sur une structure de groupe* ou GBF (en anglais).

Bien que motivé initialement par des applications concrètes, la riche structure dont sont munis les GBF ouvre des perspectives très intéressantes dans le domaine du typage et de la définition récursive des données. En particulier, ils permettent d'unifier les concepts de tableau et d'arbre.

Structure de la partie. Cette deuxième partie se compose de quatre chapitres. Le premier chapitre commence par préciser la notion d'espace à laquelle nous faisons référence. Nous poursuivons par la critique de la notion de tableau, du point de vue de son utilisation dans la simulation des SD. Le chapitre se termine par un état de l'art.

Le chapitre suivant est consacré à la notion de forme et de GBF, puis aux problèmes de la définition récursives de GBF. On ne permet que les schémas récursifs qui «respectent la notion de voisinage» spécifiée par la forme du GBF. Nous abordons ensuite le problème de l'implémentation des GBF. Un algorithme général est proposé, qui repose sur l'existence de quelques mécanismes de base. Ces mécanismes de base n'existent pas pour tous les groupes (il faut par exemple savoir résoudre le problème du mot).

Le troisième chapitre détaille les mécanismes de base dans le cas des GBF correspondant à un groupe abélien. Ces mécanismes ont été implémentés sous l'environnement de calcul symbolique Mathematica et ils montrent la validité de l'approche adoptée.

Le dernier chapitre de la partie propose plusieurs exemples d'utilisation des GBF. En particulier, nous montrons l'apport des GBF dans le codage d'une méthode de relaxation, dans un problème de gaz sur réseaux et pour la simulation de la croissance d'une ammonite sur un maillage hexagonal.

Structure du chapitre. Nous commençons par préciser la notion d'espace auquelle nous faisons référence : c'est un ensemble de points, mais chaque point à des voisins. Cette notion n'apparaît pas que dans les applications de physique et nous faisons le parallèle avec la notion de graphe des dépendances des calculs.

Nous poursuivons par la critique de la notion de tableau, suivant trois point de vue : la représentation de la topologie, la représentation d'une forme arbitraire, et les définitions récursives.

Enfin, nous passons en revue quelques formalismes qui ont été proposés pour modéliser la notion de tableau. Nous examinons aussi le concept de tableau offert dans deux langages de haut-niveau. Et nous finissons par un rapide examen de nouvelles bibliothèques de manipulation data-parallèle de tableaux.

IV.1 La représentation de l'espace dans un langage pour la simulation des SD

Nous avons déjà mentionné qu'un langage dédié à la simulation des SD devait offrir des structures de données permettant la représentation de l'espace. Le chapitre précédent fournit des exemples d'espaces que l'on veut représenter : un tore de cellules pour la simulation de l'équation de TURING, un plan pour la croissance simplifiée de l'ammonite, etc.

Quand nous parlons d'espace, nous faisons par conséquent référence à la représentation d'un concept qui intervient dans la simulation des SD. Quand ces SD modélisent un phénomène naturel, il s'agit souvent de notre espace usuel.

En 81/2, la représentation de l'espace se fait à travers le concept de collection : une collection est un ensemble de données qu'on manipule comme un tout. Cette propriété est importante car elle rend compte :

- *du caractère homogène de l'espace* : en première approximation, les mêmes lois s'appliquent en tout point de l'espace. La représentation informatique doit donc permettre d'appliquer le même calcul en tout point de cet espace. Ceci est une approximation car le traitement aux frontières peut différer.
- *de la simultanéité* : c'est l'espace qui rend possible l'existence de phénomènes simultanés (ils ont lieu en même temps, mais en des lieux différents). La représentation informatique de l'espace doit donc permettre l'expression, même implicite, de cette simultanéité.

Cependant, pour le modélisateur, un espace ne se réduit pas à un ensemble de points : il s'intéresse aux voisinages de ces points et aux mouvements possibles dans cet espace. C'est pourquoi, en accord avec le postulat fondamental en physique que les actions sont locales dans l'espace (pas d'actions instantanées à distance), les expressions récursives de collections sont restreintes à celles qui peuvent se calculer en propageant les valeurs connues de proche en proche (cf. l'exemple de *iota* dans la section III.4).

Hélas, la représentation de l'espace en 81/2 présente des insuffisances que nous allons détailler. Mais, avant de critiquer la notion de tableau en 81/2, nous voulons faire apparaître un point de vue fertile pour l'informaticien : *l'ensemble des données d'un programme constitue* aussi *un espace*, indépendamment du fait que ces données représentent ou non une partie d'un domaine spatio-temporel d'une simulation physique.

En effet, pour un programmeur, il existe une notion de *voisinage logique des données*. Par exemple, dans un arbre, un nœud père est voisin de ses fils, car on peut accéder, à partir du père, à un de ses fils. Dans une matrice, à partir de l'élément d'index (i, j) on accède souvent aux éléments d'index $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$ et on les considère comme étant voisins : ce schéma d'accès se nomme d'ailleurs «voisinage de VON NEUMANN». Dans une liste simplement chaînée, on passe séquentiellement de la tête de liste à la queue de la liste. Dans un enregistrement, les différents champs sont logiquement regroupés et ce regroupement a une existence tangible, puisqu'on peut le désigner par un identificateur. Les exemples peuvent se multiplier pour nous convaincre que la notion de voisinage logique est essentielle dans la définition des structures de données.

Cette notion de voisinage n'est pas une notion, purement logique, ayant un sens seulement pour le programmeur, mais invisible à l'exécution. En effet, le traitement dans un programme respecte la notion de voisinage logique qui apparaît dans les structures de données, le calcul se propageant généralement de proche en proche. Par exemple, si on considère la définition récursive d'une fonction `map` sur les listes, on s'aperçoit que le traitement se propage de la tête vers la queue de la liste. Plus généralement, les traitements récursifs respectent les voisinages induits par une structure de données au point qu'il apparaît nécessaire et possible de définir automatiquement des *schémas de récursion naturellement liés à la structure de données* : c'est l'objet des *catamorphismes* [FS96, NO94].

Cela signifie qu'un calcul sur une structure de données respecte une propriété de localité : le calcul associé à un élément dans une structure de données (par exemple le calcul de la valeur d'un attribut dans un arbre) dépend uniquement du calcul sur les voisins. On peut renverser cette proposition pour affirmer que si un calcul est nécessaire à un autre calcul, alors ces deux calculs sont forcément voisins dans un certain espace. Cet espace du calcul peut être abstrait, mais il est souvent matérialisé par la structure de données. Remarquons que ce point de vue n'est pas nouveau puisque l'étude de la notion de voisinage dans un ensemble de calculs constitue le fondement de l'approche de la sémantique dénotationelle [Vic88, vL90].

Nous reviendrons dans la partie suivante sur cette façon de voir l'ensemble des données et des calculs d'un programme comme un espace. Dans cette partie, nous allons nous intéresser à la représentation d'espaces qui se révèlent nécessaires pour les applications de simulation de SD. De ce point de vue, la notion de tableau, même manipulé comme un tout, est insuffisante. Une propriété caractéristique des espaces auxquels nous nous intéressons dans cette partie, est qu'ils sont *homogènes*, c'est-à-dire qu'ils présentent des propriétés de *régularité*. Le problème auquel nous nous intéressons consiste donc en la description concise d'espaces homogènes les plus variés possibles.

Nous commençons par critiquer dans la section suivante la notion de tableau. Puis nous présentons diverses tentatives menées pour remédier à ces critiques. Le prochain chapitre présente notre solution et des éléments d'implémentation. Ces éléments d'implémentation sont détaillés pour un cas particulier, puis nous donnons des exemples.

IV.2 Une notion de tableau insuffisante

Nous établissons la critique de la notion de tableau du point de vue des applications de simulation :

- Une structure de tableau est généralement utilisée pour la discrétisation d'une région de l'espace dans le cadre de la simulation d'un SD. Or, un tableau est une représentation trop pauvre de l'espace.
- Un tableau est une structure de données dont la géométrie est contraignante : il est difficile de représenter des formes arbitraires avec un tableau, ou des formes dynamiques excédant les bornes.
- La description des phénomènes à simuler conduit naturellement à spécifier des définitions récursives de tableaux. Mais le traitement de celles-ci en 8½ n'est pas satisfaisant.
- Enfin, le tableau est une structure de données mal adaptée au parallélisme.

IV.2.1 Tableau et représentation de l'espace

Du point de vue des applications de simulation, les collections sont essentiellement utilisées pour modéliser la discrétisation d'une portion d'espace : on utilise, par exemple, un vecteur pour enregistrer la variation de la température aux points de discrétisation d'une barre dans le cadre d'une résolution numérique d'un problème de diffusion de chaleur. Les collections, manipulées comme des touts, sont particulièrement adaptées à la simulation de phénomènes physiques, car les mêmes lois s'appliquent en chaque point de l'espace. Cette propriété se traduit par des calculs pouvant être exprimés globalement sur des collections [1].

La structure de tableau, est sans doute une des structures de données parmi les plus expressives pour représenter la discrétisation d'un morceau d'espace, car elle correspond naturellement à une discrétisation

1. Le lecteur intéressé par des exemples de langages utilisant la notion de collection se référera à la section II.3.1, page 12.

en grille euclidienne. Mais elle présente plusieurs inconvénients : par exemple une opération naturelle dans un espace, qui est d'aller d'un point à un de ses points voisins, doit être codée en termes de manipulations d'index ; d'autre part, la topologie de l'espace discrétisé n'est pas explicite et la représentation de topologies plus générales que la grille euclidienne est loin d'être évidente ; de plus, un tableau correspond à une région connexe de l'espace dont les bornes doivent être fixées à l'avance ; etc.

Intéressons nous aux problèmes de topologie. Dans l'exemple de la simulation d'un processus de diffusion/réaction (cf. section III.2, page 20), le tore des cellules est représenté par un vecteur dont un élément correspond à une cellule (plus exactement, un élément du vecteur correspond à une variable décrivant une des cellules). La loi d'évolution nécessite l'accès à l'état des cellules voisines. Les cellules voisines dans le tore sont « presque voisines » dans le vecteur. En effet, excepté pour les cellules d'index 0 et N qui se trouvent aux extrémités du vecteur, des cellules voisines correspondent à des données contiguës dans le vecteur (cf. figure 1).

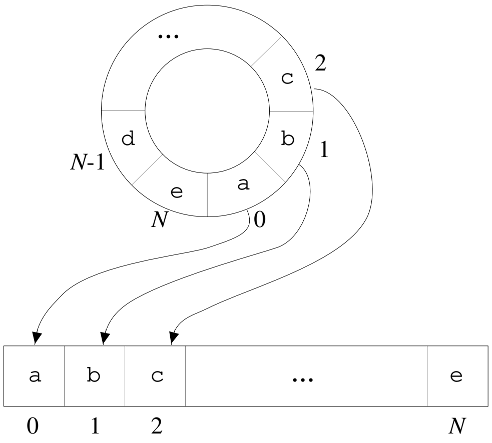


FIG. 1 –: *Représentation d'une structure torique par un vecteur*

Examinons de plus près cette notion de contiguïté : ce n'est pas d'une contiguïté physique dans la mémoire (`RAM`) qu'il s'agit, car la mémoire est obligatoirement une structure à une dimension, alors qu'un tableau peut être à deux ou plusieurs dimensions. Dans ce cas, deux éléments (i, j) et $(i + 1, j)$ que l'on considère naturellement comme contigus ne sont pas contigus dans la mémoire, si le tableau est rangé par ligne. On est donc enclin à qualifier deux éléments de voisins dans un tableau, si on passe de l'index de l'un, à l'index de l'autre, par une manipulation simple d'indices, comme par exemple l'incrémentation de l'un des indices.

Dans le cas de l'exemple du tore, il apparaît que la numérotation des cellules se fait par un entier, et que l'accès à partir d'une cellule à une de ses cellules voisines, se fait par l'incrémentation ou la décrémentation modulo N de l'entier numérotant la cellule. Cette représentation est artificielle : en effet, l'indexation des cellules, est un artifice de codage qui n'est absolument pas nécessaire. La seule opération qui soit réellement nécessaire consiste à pouvoir désigner, à partir d'une cellule l'une ou l'autre de ses voisines.

Cependant l'artifice d'implémentation, qui consiste à affecter un index à chaque point de l'espace et à calculer le point voisin est un « artifice universel » dans le sens suivant :

- On ne représente que des portions finies d'espaces discrétisés : il est par conséquent toujours possible de numéroter les points de l'espace que l'on souhaite représenter et manipuler.

- le coût d'accès aux éléments d'un tableau est uniforme : il n'est pas plus coûteux[1] d'accéder à l'élément (i,j) qu'à l'élément (k,l). Par conséquent, si on néglige le coût du calcul de l'index du voisin ($k = f(i,j), l = g(i,j)$) à partir de l'index (i,j) du point courant, on est capable d'émuler, sans sur-coût, n'importe quelle topologie avec un tableau.

1. Cette assertion est correcte en première approximation. En effet, sur une machine séquentielle l'accès à la mémoire est présenté comme uniforme, mais l'existence de hiérarchie mémoire remet en cause ce résultat. De plus, sur une architecture à mémoire distribuée, les accès distants sont bien évidemment plus coûteux que les accès locaux.

La représentation de l'espace par un tableau, consiste donc à *coder* la topologie de l'espace à deux niveaux :

1. dans un système de numérotation des points,
2. dans les fonctions de calcul des indices des voisins.

Dans le cas le plus extrême, il n'est même pas nécessaire que la dimension de l'espace à représenter soit en adéquation avec la dimension du tableau : dans ce cas, le codage du voisinage est totalement arbitraire. C'est le cas pour les méthodes d'éléments finis : l'espace est discrétisé en éléments et l'ensemble des éléments est manipulé comme un vecteur ; le calcul des voisins, se réduit au stockage explicite, dans une liste, des voisins pour chaque élément [Bai90] (cf. figure 2).

Nous désirons malgré tout rendre la topologie de l'espace explicite dans la structure de données. En effet :

- C'est une information sémantique de haut-niveau, connue du programmeur et qu'il est nécessaire de transmettre à un lecteur éventuel du programme.
- Cette information est utilisable par un compilateur, ou un interprète. En effet, la topologie utilisée conditionne l'accès aux données et permet de choisir une représentation optimale des éléments en mémoire. L'extraction des schémas d'accès, à partir des opérations sur les indices, nécessite des analyses sémantiques difficiles et parfois totalement infructueuses.
- La codage de la topologie dans les index est une opération complexe. Par exemple, quel schéma le programmeur doit-il adopter pour une discrétisation hexagonale de l'espace ? La représentation avec des tableaux n'est naturelle que pour les discrétisations correspondant à des grilles euclidiennes multi-dimensionelles.
- La manipulation explicite des indices s'oppose à l'approche intensionnelle des tableaux, où on veut les manipuler comme des touts. Faire apparaître les indices, même à travers une opération globale comme la permutation, laisse transparaître le point de vue extensionel du tableau.
- Il est possible d'expliciter simplement la topologie, quand celle-ci est régulière (ou encore homogène).
- Beaucoup d'opérations sont paramétrées par la topologie (par exemple, en APL, l'opération de réduction est paramétrée par l'axe, qui permet de spécifier suivant quelle dimension une opération de réduction doit avoir lieu) ou peuvent être naturellement étendues si on les paramétrise par la topologie.

La solution que nous proposons pour rendre explicite la topologie sous-jacente à une collection, est de considérer explicitement l'ensemble des déplacements. Un déplacement élémentaire consiste à passer d'un point à un des points voisins. Si le voisinage est régulier, on peut appliquer un déplacement élémentaire à n'importe quel point et on peut donc l'appliquer globalement au tableau par α-extension. Il est naturel d'ajouter un déplacement nul permettant de «rester sur place» ; de composer des déplacements élémentaires et d'inverser un déplacement. Autrement dit, on peut munir l'ensemble des déplacements d'une structure de groupe et l'application d'un déplacement à un tableau correspond à l'*action du groupe*.

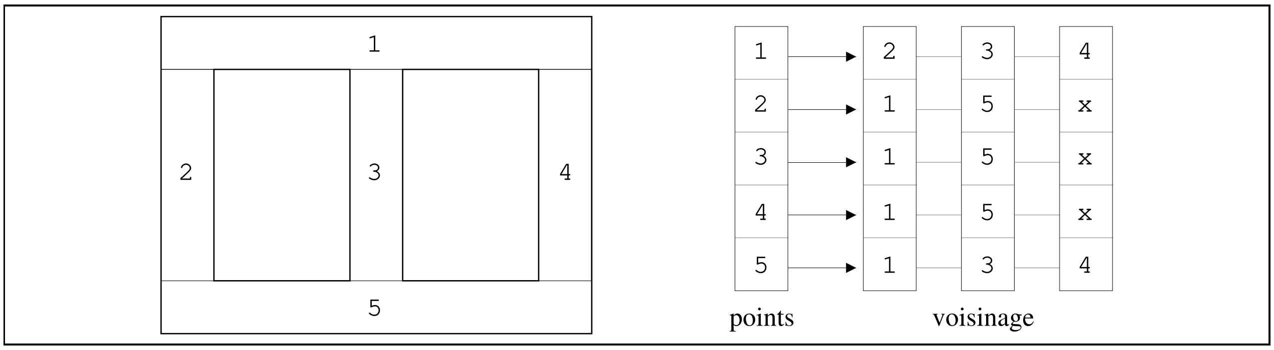


FIG. 2 –: *Représentation de type éléments finis d'une région de l'espace : la structure de la figure de gauche peut être codée en utilisant des vecteurs suivant la représentation de la figure de droite.*

Donnons un exemple. On peut décrire un groupe par une *présentation* qui consiste en l'énumération des générateurs (les éléments qui correspondent aux déplacements élémentaires) et une série d'équations contraignant les compositions de déplacements. Un élément du groupe est un mot composé de générateurs. Pour le tore de cellules que nous avons décrit précédemment, la présentation qui définit le groupe des déplacements possibles est :

$$\langle v \; ; \; v^n = e \rangle$$

où e note le déplacement nul et où v représente l'accès à un voisin. Le mot $v.v$ ou v^2 représente l'accès à un 2-voisin, tandis que v^{-1} représente l'accès au « voisin opposé » au voisin v. La figure 3 illustre l'effet de l'action de v, v^{-1} et v^2 sur un tableau.

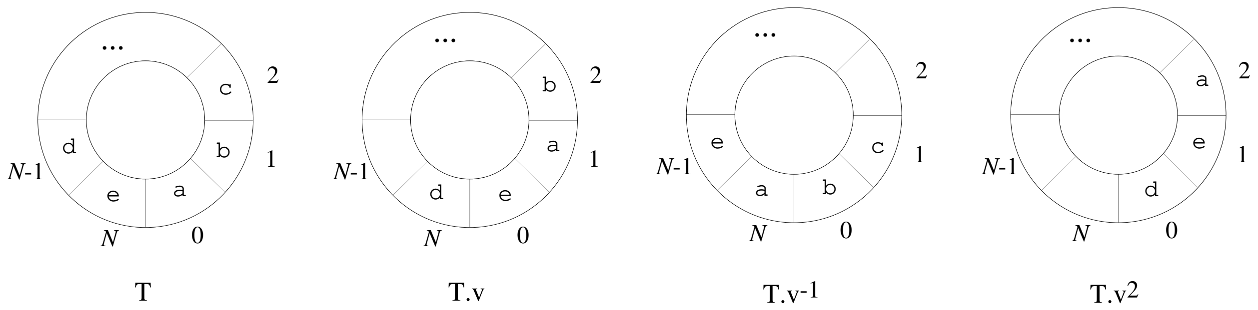


FIG. 3 –: *Exemple de l'action du groupe* $\langle v \; ; \; v^n = e \rangle$ *sur un tableau*

IV.2.2 Tableau et représentation des formes

Dans le chapitre précédent nous avons développé une simulation du processus (simplifié) de croissance de l'ammonite. L'appartenance d'un point à l'ammonite doit être explicitement codée par une valeur particulière du tableau. Cette représentation rend possible la représentation de formes arbitraires sur le fond du tableau, (voir la figure 4), mais nécessite une gestion explicite. Par ailleurs, le codage des déplacements est particulièrement pénible car il faut calculer explicitement la permutation correspondante des index.

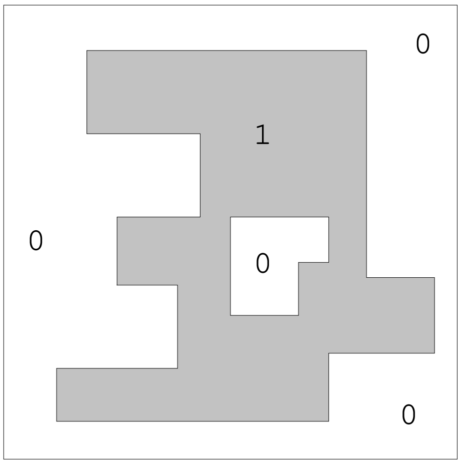


FIG. 4 –: *Représentation d'une forme arbitraire* dans *un tableau. On ne peut pas représenter cette forme* par *un tableau, car elle ne correspond pas une région rectangulaire, il faut la représenter* dans *un tableau par codage.*

Pour rendre facilement utilisable le codage d'une forme comme « région » dans un tableau, il faudrait :

– gérer implicitement la distinction forme/fond,

– simplifier la gestion des déplacements.

La représentation explicite des déplacements, introduite à la section précédente, répond en partie au second point.

Pour le premier point, on peut remarquer que le codage du fond comme valeur spéciale 0 d'un élément, est un codage arbitraire. La seule chose que l'on doit pouvoir exprimer, est l'appartenance d'un point à la forme (et éventuellement lui spécifier une valeur) où alors l'appartenance d'un point au fond, et il n'y a dans ce cas aucune valeur à représenter. De cette constatation provient l'idée de considérer le tableau comme une *structure de données partielle*, c'est-à-dire une structure de données dont certains éléments peuvent avoir une valeur indéfinie.

On est donc amené à l'idée que le concept de tableau peut être étendu au concept de *champ de données*. Un tableau est considéré comme une fonction stricte de $[n_1 \ldots m_1] \times \cdots \times [n_d \ldots m_d]$ et un champ de données est une fonction (partielle) d'un ensemble d'index.

Nous demandons à ce qu'un groupe agisse sur l'ensemble d'index : ce groupe s'interprète comme le groupe des déplacements. Nous verrons dans le chapitre suivant que l'ensemble des index peut naturellement être choisi comme égal au groupe des déplacements.

IV.2.3 Tableaux, fonctions et définitions récursives

Il existe un lien étroit entre les notions de tableau et de fonction. En effet, on peut voir un tableau comme une fonction qui associe à un index, un élément du tableau :

$$\mathit{Tableau}[0 \,..\, n_1] \times [0 \,..\, n_2] \times \cdots \times [0 \,..\, n_d] \rightarrow \mathit{Valeur}$$

Traditionnellement, il existe deux façons de concevoir les fonctions [Bar84, pp 3] :

1. le point de vue *intensionnel* assimile la notion de fonction à celle de règle de calcul (un exemple d'une telle approche est le λ-calcul) ;
2. le point de vue *extensionnel*, attribué à Dirichlet, considère une fonction comme un ensemble de paires (argument, valeur) ; un exemple de cette approche est la notion de fonction continue en mathématique.

Selon ces distinctions, il est clair que les tableaux 81/2 sont des structures de données extensionnelles, car on calcule et on stocke en mémoire les paires (index, valeur). En fait, l'index n'est pas explicitement stocké, mais codé dans la position de la valeur en mémoire. Mais un tableau 81/2 est aussi un objet intensionnel, car la manipulation des tableaux ne fait pas de référence explicite à ses constituants (on les manipule comme des touts).

L'analogie entre tableau et fonction n'est pas purement formelle : par exemple, en 81/2, on définit récursivement des tableaux (c'est le cas aussi pour les langages systoliques comme Crystal [Che86], Alpha [Mau89] ou Pei [VP92, VP93]), comme on peut définir des fonctions par récursion. Cela montre bien la similarité des deux concepts.

Cependant, la notion de tableau comporte plus d'information que la notion de fonction. Par exemple, l'ensemble de définition d'une fonction est quelconque, alors que l'ensemble des index d'un tableau correspond à un n-uplet d'entiers ; on ne sait généralement pas dire si une fonction est partielle ou pas, alors qu'un tableau doit avoir une valeur bien définie pour chaque index ; etc.

En 81/2, une *différence cruciale* est faite entre fonction et tableau : le schéma récursif d'une définition de fonction peut prendre n'importe quelle forme, alors que le schéma récursif d'une définition de tableau est contraint (cf. l'exemple de *iota* section III.4). En effet, le schéma récursif sur un tableau utilise la concaténation suivant un des axes du tableau. Par suite, le processus de résolution de l'équation se propage suivant une dimension, de proche en proche. La figure 5 illustre ce processus. Par contre, la définition récursive d'une fonction peut prendre n'importe quelle forme et le processus de résolution est quelconque : il suffit de considérer les dépendances du calcul de la fonction f pour s'en convaincre :

$$f(x) = \begin{cases} x/2 & \text{si } x \bmod 2 = 0 \\ 3x+1 & \text{sinon} \end{cases}$$

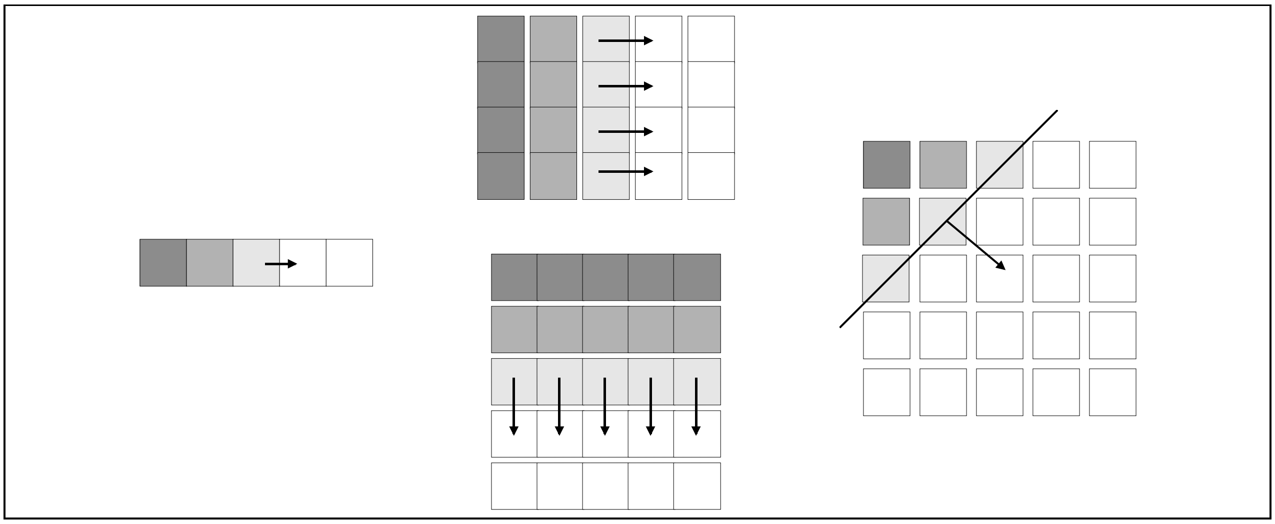


FIG. 5 –: *Propagation des calculs dans la définition récursive d'un tableau en 81/2. Le premier schéma illustre la propagation des calculs lors de la résolution d'une définition récursive d'un vecteur, comme par exemple la définition de iota (voir figure 5). Le deuxième schéma illustre une propagation pour un tableau à deux dimensions. Il faut noter que pour chaque axe, la propagation peut se faire de l'index 0 vers les index croissants, ou de la borne supérieure du tableau vers l'index 0. Le troisième schéma montre un processus de calcul qui se propage* en diagonale. *Il n'est pas possible de définir un tel processus en 81/2, alors que cela semble être naturel.*

Les considérations précédentes trouvent une formalisation idéale dans le cadre de la sémantique dénotationelle [vL90]. La structure du domaine sur lequel on résout l'équation récursive $f = \varphi(f)$ permet de contraindre plus ou moins le processus de calcul. Par exemple, pour les fonctions, on choisit généralement de chercher la solution sur l'ensemble des fonctions munies d'un ordre de SCOTT $\sqsubseteq_S$:

$$f \sqsubseteq_S g \quad \leftrightarrow \quad f(x) \sqsubseteq g(x)$$

pour x appartenant à l'ensemble de départ de f et g muni de la relation d'ordre $\sqsubseteq$. Le processus de calcul consiste à calculer des approximations de la solution, approximations obtenues par itération de φ à partir de $\perp$, la fonction indéfinie partout. Si on regarde l'ensemble de définition de deux itérations successives, par exemple $\varphi^n(\perp)$ et $\varphi^{n+1}(\perp)$, on sait que $\texttt{Def}(\varphi^n(\perp)) \subseteq \texttt{Def}(\varphi^{n+1}(\perp))$, mais ces deux ensembles peuvent différer arbitrairement.

Pour les tableaux 81/2, il ne peuvent pas différer arbitrairement car on *spécialise* l'ordre sur le domaine. Par exemple, pour des vecteurs, on utilise un *ordre préfixe* (comme pour les listes). Par suite, $\texttt{Def}(\varphi^n(\perp))$ doit être un préfixe de $\texttt{Def}(\varphi^{n+1}(\perp))$: autrement dit, le calcul se propage de l'index 0 vers les index croissants. On peut reconnaître cette méthode de traitement des définitions récursives d'une structure de données dans les travaux présentés dans [Wad81] pour les streams data-flow et dans [Sij89] pour les listes. C'est cette approche qui a été utilisée pour la définition récursive des tableaux 81/2 [Gia91a, GS93].

Cependant, ce traitement des récursions permises est insuffisant. En effet, notre intuition est d'autoriser les schémas récursifs qui permettent de propager le calcul de voisins en voisins. C'est bien le cas pour les schémas permis en 81/2, mais par exemple, on ne peut pas propager un calcul le long d'une diagonale, alors que ce schéma respecte bien une certaine notion de voisinage. Il faut donc élargir la notion de voisinage et permettre les définitions récursives «qui respectent le voisinage».

IV.2.4 Le tableau est une structure de données mal adaptée au parallélisme

Comme nous l'avons mentionné précédemment, l'extraction des mouvements de données, à partir des manipulations d'index, est une tâche difficile pour un compilateur. Mais l'analyse des mouvements de données est cruciale pour implémenter efficacement l'accès à un tableau, quand l'accès à un tableau n'est pas uniforme [Del94]. Dans les langage Vienna-Fortran, Fortran-D et HPF, le programmeur peut spécifier des directives ayant une influence sur l'alignement et la distribution des éléments d'un tableau. Cette facilité est offerte pour permettre l'optimisation des accès par un placement judicieux des données. Cependant, la solution adoptée, si elle a le mérite de la simplicité, n'adresse pas le problème réel, qui est la gestion efficace des mouvements de données : un programmeur peut spécifier un placement, qui contredise les accès effectivement réalisés lors du calcul. Il est important de capturer les mouvements logiques de données impliquées par un algorithme et de décider d'un placement qui en minimise le coût. De ce point de vue, la représentation explicite des déplacements élémentaires des données par l'utilisation des champs de données, est une approche qui ne peut que se révéler fructueuse.

IV.3 Les théories étendant la notion de tableau

IV.3.1 Les formalismes

IV.3.1.1 Les algèbres de Bird-Meertens

BIRD et MEERTENS (B&M) ont développé une algèbre fonctionnelle sur les liste [Bir87]. Cette algèbre permet de décrire une classe de programmes par un formalisme équationnel. Elle hérite en cela de l'approche de BACKUS [Bac78].

Les données considérées sont des listes d'éléments d'un certain type noté $[a_1, \ldots, a_n]$ (dont les éléments peuvent être aussi des listes, ce qui autorise l'imbrication de listes et donc les tableaux). Les principales fonctions qui définissent l'algèbre sont (on suppose que les éléments, fonctions et opérateurs sont tous d'un type compatible avec l'opération décrite) :

- La concaténation (notée $\sharp$) : en prenant deux listes de même type, on obtient une liste constituée des éléments de la première puis, à la suite, ceux de la deuxième. Par exemple : $[a_1, \ldots, a_n]\sharp[b_1, \ldots, b_m] = [a_1, \ldots, a_n, b_1, \ldots, b_m]$.
- L'α-extension (notée $*$) : elle nécessite deux arguments, une fonction f et une liste $[a_1, \ldots, a_n]$. Le résultat est une liste constituée des éléments de la liste argument sur lesquels on a appliqué la fonction : $[f.a_1, \ldots, f.a_n]$.
- La β-réduction (notée $\backslash$) : elle s'applique à la liste en suivant une opération et rend l'élément obtenu par application successive de cet opérateur sur les éléments de la liste. Par exemple : $\oplus\backslash[a_1, \ldots, a_n] = (\ldots(a_1 \oplus a_2) \oplus \cdots \oplus a_n)$.

Le formalisme décrit par B&M repose sur l'étude des homomorphismes d'un certain ensemble dans l'ensemble des listes muni de la loi de concaténation :

$$h : (\text{Liste}, \sharp) \rightarrow (E, \oplus) \quad \text{tel que} \quad h(x\sharp y) = h(x) \oplus h(y)$$

Les propriétés de cette algèbre permettent la définition de règles de transformation et d'identité sur ces fonctions. Il est alors possible de décrire un programme par un ensemble d'équations sur les données grâce à ce formalisme puis de les simplifier, en utilisant les identités de l'algèbre (ces simplifications peuvent être faites semi-automatiquement). La théorie nous assure que les transformations qui ont eu lieu ne modifient pas la sémantique du programme original. On trouvera dans [Gib94] un exemple d'une telle simplification pour le problème de recherche du maximum de la somme des éléments des segments d'une liste d'entiers. La description la plus naturelle impliquait une complexité cubique (trouver les segments d'une liste est quadratique et trouver le maximum d'un ensemble d'entiers est linéaire) et les simplifications l'ont ramené à une complexité linéaire.

IV.3.1.2 L'algèbre MOA

L'algèbre MOA développée sur les tableaux multi-dimensionels garde à peu près le même formalisme que celle de B&M, c'est-à-dire qu'elle est basée sur les opérations de concaténation de listes, extension, réduction de listes et les homomorphismes. L'intérêt de ce formalisme est de trouver des identités permettant la vérification et la simplification automatique des programmes.

Cependant, la différence avec l'approche de B&M réside dans la façon de représenter les données manipulées. En effet, les objets ne sont plus des listes imbriquées mais des paires de listes (f, c) dont le premier élément spécifie la structure de l'objet et le second élément spécifie le contenu. La liste f est :

- soit une liste de la forme [0],
- soit une liste d'entiers ne contenant pas les entiers 1 et 0. Le nombre d'éléments dans la liste définit le nombre de dimensions de la structure et chaque élément spécifie la taille de chacune des dimensions. La longueur de la liste des valeurs, c, doit être égale à la taille de l'objet qui est le produit des tailles des dimensions.

Par exemple, $([3], [a, b, c])$ représente le vecteur $[a, b, c]$, $([2, 3], [a, b, c, d, e, f])$ la matrice $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$, $([\,], [a])$ le scalaire a et $([0], [\,])$ l'objet vide.

Le principal avantage de cette représentation est que la manipulation des structures est indépendante de la dimension de celles-ci (les tableaux ont un type commun quelles que soient leurs dimensions). Cela permet de prévenir la multiplication des règles algébriques et rend la manipulation de la *forme* des structures possibles (dénombrement des éléments, décompte des dimensions, reconfiguration de l'objet...) sans qu'il ne soit nécessaire de fournir des procédures d'exploration récursive.

IV.3.1.3 Les champs de Lisper

Lisper [Lis93, LC94, Lis96] propose de considérer les données comme des fonctions d'un certain index dans un ensemble de valeurs (par exemple, une matrice entière $n \times m$ devient une fonction de $[1..n] \times [1..m]$ dans $\mathbb{Z}$). Cette nouvelle structure est appelé un *champ*. Cette approche permet de s'abstraire des problèmes d'implémentation des données puisqu'on ne décide d'aucune forme privilégiée pour la représentation des données. De plus, l'index de ces fonctions pouvant être plus complexe que celui donné en exemple, les structures data-parallèles classiques que sont les tableaux ne forment plus qu'une classe particulière des données décrites par les champs. Ainsi, on peut espérer exprimer des algorithmes data-parallèles plus portables.

De la même façon que précédemment, un programme est vu comme une équation fonctionnelle avec les opérateurs classiques (α-extension, β-réduction...) à la différence que chez Lisper, ces opérateurs agissent sur la fonction qui définit la donnée donnant une forme purement fonctionnelle aux équations.

Le choix d'un index quelconque pour les champs pose néanmoins le problème du calcul des valeurs de celui-ci, en particulier s'il est infini (le problème n'est pas tant celui de la formalisation du programme que celui de l'implémentation effective de celui-ci). On doit alors considérer que la fonction du champ est définie partiellement sur l'index. Cela introduit une nouvelle opération fondamentale qui est la *restriction explicite* de l'index d'un champ : f/b, avec f un champ quelconque et b un champ booléen, signifie que les valeurs du champ f n'ont de chance d'être définies que sur les points de l'index où le champ b prend la valeur vraie. Il faut alors trouver les propriétés et les identités de cet opérateur pour pouvoir l'utiliser ce qui est loin d'être évident dans le cas général.

Les études menées par Lisper se sont uniquement concentrées sur les champs comme fonctions de $\mathbb{Z}^n$.

IV.3.1.4 Discussion

Les formalismes algébriques développés permettent la manipulation et la simplification d'expressions de tableaux. L'approche algébrique est particulièrement puissante pour formaliser une notion intensionelle de tableau, mais les approches citées ne nous suffisent pas : seul les champs de Lisper permettent de définir des structures partielles sur des algèbres d'index plus complexes que les entiers munis de l'addition.

Cependant, considérer une structure de données comme une fonction d'un ensemble d'index vers un ensemble de valeurs est une approche *trop générale* : on est dans le domaine d'étude générale des fonctions (par exemple avec une théorie comme le λ-calcul) et on ne dispose pas de structures plus spécifiques qui soient attachées à l'idée de tableau (ou aux généralisations de cette idée). Il faut donc «spécialiser» la notion de champ. Les travaux menés sur les champs se sont, en conséquence, consacrés exclusivement à des champs sur $\mathbb{Z}^n$ et ont réutilisé des résultats classiques sur les polyèdres convexes, initialement développés pour le traitement des tableaux systoliques et l'analyse de dépendances des nids de boucles [LC94].

Remarquons que le choix d'une algèbre particulière est délicat : par exemple, suivant qu'on utilise la concaténation ou bien le constructeur `cons` sur les listes, on obtient une structure de données aptes au parallélisme ou induisant des calculs intrinsèquement séquentiels [Ski93, Mis94], alors que le choix initial semble de prime abord assez innocent.

IV.3.2 Les automates cellulaires

IV.3.2.1 Automates cellulaires sur graphes de Cayley

Dans ses travaux [Rók93, Rók94a, Rók94b, Rók95b, Rók95a], Róka a commencé l'étude des automates cellulaires (AC) sur graphes de Cayley.

Un automate cellulaire correspond à un réseau de machines élémentaires toutes identiques, modélisées par des automates finis. Ces machines sont situées dans des cellules disposées régulièrement, communiquant avec leurs voisines et évoluant en parallèle de manière synchrone. Les AC ont été introduits dans les années 40 par Von Neumann.

L'idée de Róka est de modéliser les réseaux de cellules par un graphe de Cayley, c'est-à-dire par un graphe dont les sommets sont les éléments d'un groupe et où les arcs sont étiquetés par un générateur du groupe (voir le chapitre suivant). Les graphes de Cayley sont décrits de façon compacte par exemple par la présentation finie du groupe associé, et cette structure permet d'inférer des propriétés importantes du graphe. Ils ont été ainsi beaucoup étudiés comme réseaux d'interconnection [Hey96].

Róka a mené ses travaux suivant trois directions principales : la simulation d'AC sur graphes de Cayley « bi-directionnel » (la relation de voisinage est symétrique) par des AC unidirectionnels (l'inverse des générateurs n'apparaît pas dans les étiquettes des arcs) ; plus généralement, la condition de simulation d'un AC sur graphe de Cayley par un AC sur un autre graphe de Cayley ; et finalement, le problème algorithmique de la synchronisation globale d'un ensemble de cellules.

Nous avons pris connaissance des travaux de Róka après avoir développé les GBF, mais il existe de nombreux liens entre les deux approches : dans les deux cas, on est amené à étudier la propagation de calculs dans un espace décrit par un graphe de Cayley. Cependant, nous ne sommes intéressé par ce point que pour l'implémentation de définitions récursives de GBF, nos motivations et nos développements étant très différents. Nous cherchons principalement à construire des graphes de Cayley comme type d'un tableau et à développer une algèbre d'opérations sur ce nouveau type. Ce n'est pas le cas des travaux cités, qui, mis à part le problème de la synchronisation, établissent des résultats de complexité pour le formalisme spécifique des AC sur graphe de Cayley. Quand au problème de la synchronisation des cellules, nous ne pouvons à vrai dire même pas nous le poser puisque le point de vue local des calculs effectués sur un sommet du graphe n'apparaît pas (nous avons privilégié l'approche qui manipule les GBF intensionnellement, c'est-à-dire comme des touts). Inversement, Róka ne peut pas se poser le problème de la définition récursive de GBF, le point de vue global n'apparaissant pas dans son approche.

IV.3.2.2 La programmation plane

Rozin a proposée une nouvelle algorithmique sur les automates cellulaires en introduisant des opérateurs permettant de représenter directement le déplacement d'informations [Roz87, Roz90] comme les rotations et les permutations planes, ou la circulation d'un front d'onde dans une mémoire SIMD.

Ses travaux n'abordent que les grilles de dimensions deux, avec un voisinage de Moore ou de Von Neumann. Ils sont donc éloignés des extensions de tableaux que nous considérons. Mais nous partageons avec son approche l'accent mis sur la circulation de proche en proche des données lors d'un calcul. Nous nous sommes surtout consacrés à l'aspect « définition d'un voisinage » et donc à spécifier la notion de « proche en proche ». Bien que l'algorithmique développée fasse apparaître des objets géométriques discrets, comme des segments ou des partitions du plan, ces travaux restent dans le cadre des automates cellulaires dans $\mathbb{Z}^2$ avec la préoccupation de l'aspect local.

IV.3.3 Les langages manipulant une structure de données alternative à la notion de tableaux

IV.3.3.1 Indexical Lucid

Le langage Lucid [AW77] de Ashcroft et Wadge est le premier langage à avoir définit la notion de *stream* à travers les opérateurs `next` et `fby` (Lucid était, à ces deux opérateurs près, identique à ISWIM de Landin [Lan66]). La structure de données disponible en Lucid était le stream de scalaires, permettant de modéliser l'évolution d'une valeur au cours du temps (le stream représentant l'aspect temporel de la variable). La notion de *contexte* est définie en Lucid. Le contexte est un entier représentant la valeur du stream à l'endroit considéré.

Lucid à beaucoup évolué depuis sa première définition et sa dernière évolution Lucid à donné lieu à ***Indexical Lucid*** [AFJW95] (IL) qui est *multi-dimensionnel.* Les contextes en IL sont maintenant un couple (identificateur, entier). L'identificateur est appelé la *dimension* dans laquelle la valeur (l'entier) se trouve. Tous les opérateurs de Lucid peuvent être paramétrés par la dimension afin de pouvoir changer le contexte en n'importe quelle dimension. Par exemple, soit un tableau bi-dimensionnel A où la dimension `da` est horizontale vers la droite et la dimension `db` est verticale vers le bas :

$$A = \begin{matrix} 1 & 3 & 4 & \dots \\ 3 & 1 & 6 & \dots \\ 1 & 9 & 2 & \dots \\ \vdots & \vdots & \vdots & \dots \end{matrix}$$

la somme des éléments du tableau suivant l'une ou l'autre dimension du tableau s'effectue simplement en paramétrant la fonction $\texttt{SommeCourante.}p\texttt{(A)}$ suivant p ($p \in \{\texttt{da}, \texttt{db}\}$) :

```
SommeCourante.d(N) = M
where
  M = N fby.d M + next.d N;
end;
```

L'opérateur `next.d` permet d'accéder à la valeur suivante dans le contexte `d`, `X fby.d Y` spécifie dans le contexte `d` la valeur (ici `Y`) qui viendra après `X`. En fonction du contexte spécifié, nous avons :

$$\texttt{SommeCourante.da(A)} \Rightarrow \begin{matrix} 1 & 4 & 8 & \dots \\ 3 & 4 & 10 & \dots \\ 1 & 10 & 12 & \dots \\ \vdots & \vdots & \vdots & \dots \end{matrix} \qquad \texttt{SommeCourante.db(A)} \Rightarrow \begin{matrix} 1 & 3 & 4 & \dots \\ 4 & 4 & 10 & \dots \\ 5 & 13 & 12 & \dots \\ \vdots & \vdots & \vdots & \dots \end{matrix}$$

En IL, les dimensions sont des *pseudo* valeurs : on peut passer une dimension en argument (mais on ne peut pas, par exemple, générer un stream de dimensions).

IV.3.3.2 FIDIL

FIDIL[1] est un langage de programmation parallèle et impératif, à parallélisme implicite et à contrôle de flot séquentiel développé par Hilfinger et Colella [HC89, HC93], supportant les calculs de différence finie et les méthodes de particules. FIDIL tente d'apporter une réponse à la complexité croissante des programmes en fournissant aux programmeurs des structures de données de haut-niveau pour le calcul scientifique, et plus particulièrement pour la résolution des équations aux dérivées partielles. Ces structures de données ont pour but de rapprocher le niveau sémantique des programmes de celui des algorithmes dont ils dérivent.

1. Pour FInite DIfference Language.

La principale innovation de ce langage réside dans l'introduction de deux nouveaux types de données appelés `domain` et `map`. Un `domain` est un ensemble fini de n-uplets d'entiers, c'est-à-dire à un sous-ensemble fini de $\mathbb{Z}^n$ et un `map` consiste en un `domain` où à chaque point de $\mathbb{Z}^n$, i.e. à chaque n-uplets d'entiers, est associé une certaine valeur. Le type de données `map` est une extension de la notion de tableau car le `domain` associé à un `map` n'a pas forcément une forme rectangulaire contrairement aux tableaux.

Les `domains` et les `maps` sont des collections car ils correspondent à des agrégats de données pouvant être manipulés comme des touts. FIDIL encourage la manipulation intensionnelle de ces objets à travers un certain nombre d'opérations prédéfinies. Ces opérations peuvent être qualifiées de calculatoires (par exemple, alpha-extension, β-réduction...) ou de géométriques (e.g. translation, contraction, union, intersection...). Une opération calculatoire est une opération dont le résultat correspond au produit d'un calcul mettant en jeu les valeurs associées aux objets arguments et les opérations géométriques touchent à l'organisation d'un objet indépendamment des éléments qui le composent.

Un `domain` quelconque peut être construit à partir des `domains` de base rectangulaires et d'un certain nombre d'opérateurs fournis par le langage (par exemple, union, différence, intersection...). Un `map` est défini à partir d'un `domain` et d'un type de données. Le `domain` d'un `map` est par défaut constant, sauf si le map considéré a été défini comme étant flexible. Son type n'étant pas forcément scalaire, il est possible de définir des `maps` d'enregistrements ou des `maps` imbriqués. Toutefois, dans ce dernier cas, les éléments d'un tel map doivent tous avoir même dimension. Enfin, les fonctions de FIDIL peuvent être génériques. Il est en effet possible de définir des fonctions polymorphes suivant les arguments de type `domain` et `map` pour la dimension et seulement suivant les arguments de type `map` pour le type de données associé.

IV.3.3.3 Discussion

Les deux langages évoqués ci-dessus offrent tous les deux une approche *intensionnelle* de la notion de tableau, c'est-à-dire qu'on peut les manipuler comme des touts. D'autres langages auraient pu être décrits comme par exemple NIAL, SISAL95 [FMSD95], NESL [Ble93], etc. Nous citons ces deux langages car ils présentent une caractéristique intéressante : ils sont dynamiques.

Indexical Lucid est un langage dynamique car le nombre de dimensions d'un tableau n'est pas fixé *a priori* : le programmeur dispose d'autant de dimensions qu'il peut avoir besoin au cours du calcul, ou, pour voir les choses autrement, les tableaux d'Indexical Lucid ont un nombre infini de dimensions. La structure de données manipulée par Indexical Lucid est donc un champ sur $\mathbb{Z}^*$.

Le langage FIDIL est un langage dynamique dans le sens où on peut représenter des domaines très riches dans $\mathbb{Z}^n$. La richesse des domaines du langage FIDIL est due à la machine virtuelle FIDEL [Sem93, Sem94].

Ici encore, on retrouve la balance entre généralité des notions mises en œuvre et spécificité de la théorie résultante : la structure de données d'Indexical Lucid est trop générale pour ne pas être traitée comme une fonction, même si l'implémentation de cette fonction est *paresseuse et mémoïsée*. La notion de topologie des données a donc disparue. Par contre les domaines de FIDIL sont des structures de données particulères avec une algèbre de régions dédiées (on peut translater, symétriser, unir... des régions). Mais la topologie des données reste celle de $\mathbb{Z}^n$.

IV.3.4 Les bibliothèques spécialisées

Dans cette section nous présentons quelques bibliothèques spécialisées qui ont été développées pour fournir un nouveau type de données, au programmeur, à travers un API (*Application Programmer Interface*). Il est très significatif que ces nouvelles bibliothèques aient été développées en C++ : leur développement dans un contexte Fortran étant bien plus difficile et l'interface moins agréable.

IV.3.4.1 LPARX

LPARX est une librairie C++ développée principalement par Baden et Kohn, supportant un modèle de programmation data-parallèle, à gros grain, correspondant à un modèle d'exécution SPMD [BKF94]. Cette librairie fournit un support dynamique pour les décompositions irrégulières de données basées sur la notion de bloc.

LPARX définit trois types de structures de données différentes : `Region`, `Grid` et `XArray`. Un objet de type `Region` est équivalent à un sous-ensemble rectangulaire de $\mathbb{Z}^n$. Aucune donnée n'est associée à un tel objet. Un objet de type `Grid` correspond à un objet de type `Region` où à chaque point de $\mathbb{Z}^n$ est associée une valeur. De plus, un tel objet est associé à un et un seul processeur. Enfin, un objet de type `XArray` est un ensemble distribué d'objets de type `Grid` ou de tout autre type d'objets possédant la notion de processeur associé, un tel objet pouvant être créé de manière dynamique. Des constructeurs de boucles séquentielles et parallèles sont aussi supportées par LPARX de manière à pouvoir itérer sur les éléments des objets de type `Region`, des objets de type `Grid` et `XArray`.

Un ensemble d'objets de type `Region` peut être utilisé pour décrire un sous-ensemble fini de $\mathbb{Z}^n$. Toutefois, LPARX ne fournit pas l'équivalent de la notion de `domain` proposée par FIDIL, i.e. ne fournit aucune structure de données correspondant à un sous-ensemble fini de $\mathbb{Z}^n$, sous-ensemble de forme quelconque, pas forcément rectangulaire. De plus, un tel objet n'est pas une collection car il n'est pas manipulable comme un tout. De manière générale, les structures de données proposées par LPARX ne sont pas d'aussi haut niveau que celles de FIDIL mais peuvent en constituer les briques de base.

IV.3.4.2 KeLP

KeLP est une librairie C++ succédant à LPARX et développée par les mêmes auteurs [FKB96]. Cette librairie hérite des concepts fournis par LPARX mais introduit un nouveau modèle de communication basé sur le *paradigme inspecteur/exécuteur*, paradigme introduit par SALTZ dans MultiBlock PARTI [ASS95] et CHAOS [HMS$^+$94]. Les extensions proposées ont pour motivation essentielle la gestion précise et efficace des synchronisations et des communications sur une machine parallèle à passage de messages.

IV.3.4.3 A++/P++

A++ et P++ sont deux librairies C++ de tableaux, développées entres autres par QUINLAN, librairies permettant de supporter le développement d'algorithmes indépendants de quelque architecture que ce soit [PQ94, LQ92]. A++ est un ensemble de classes correspondant à un modèle d'exécution séquentiel et P++ correspond au pendant parallèle de A++, augmenté d'un ensemble de directives de placement pour machines parallèles et utilisent un contrôle de flot séquentiel. Les interfaces de ces deux librairies sont identiques ; par conséquent, le fait de substituer P++ à A++ lors d'une compilation permet de paralléliser un code séquentiel, aux directives de placement près.

De manière plus précise, A++, sur laquelle s'appuie P++, permet de manipuler les tableaux à la manière de Fortran 90 [Saw92]. De ce fait, les tableaux des librairies A++ et P++ sont des collections car ils peuvent être manipulés comme des touts. Une sous-partie d'un tableau peut être accédée via l'utilisation d'index, c'est-à-dire de triplet : borne inférieure, borne supérieure et pas. Il est à noter que contrairement à FIDIL, ni A++, ni P++ ne mettent à la disposition de l'utilisateur une structure de données correspondant à un sous-ensemble fini de $\mathbb{Z}^n$ de forme quelconque, pas forcément rectangulaire. De plus, à la différence de LPARX et de ses prédécesseurs, il n'y a pas de séparation conceptuelle entre la forme d'un tableau et les données associées : il n'est pas possible de manipuler la forme d'un tableau indépendamment de ses données.

IV.3.4.4 Discussion

Les librairies que nous venons de présenter [1] offrent toutes la possibilité de définir des domaines un peu plus riches que ne le permettent les tableaux classiques. Cependant, la topologie sous-jacente reste celle de $\mathbb{Z}^n$. Les motivations de développement de ces libraires sont initialement la simulation parallèle d'équations aux dérivées partielles : la gestion du parallélisme est un point central et la représentation des données est quelque peu passée au second plan.

1. Nous remercions Dominique DE VITO pour les informations qu'il nous a fourni concernant le langage FIDIL et les bibliothèques spécialisées.

Chapitre V

Champs basés sur une structure de groupe

Nous avons exposé dans le chapitre précédent les motivations qui nous amènent à développer une généralisation de la notion de tableau. Dans ce chapitre nous allons répondre aux problèmes posés par la notion classique de tableau de deux manières :

1. en considérant un tableau comme une fonction partielle et non plus totale, de l'ensemble de ses index vers un ensemble de valeurs : c'est la notion de *champ* ;
2. en munissant l'ensemble des index d'une structure de groupe : c'est la notion de *champ basé sur un groupe*.

La traduction anglaise de champ basé sur un groupe est «Group Based Field» et nous utiliserons les initiales GBF pour désigner cette nouvelle structure de données.

La situation que nous avons actuellement est la suivante :

$$\mathit{Tableau} \in [0\ldots n_1] \times \cdots \times [0\ldots n_d] \mapsto \textsc{Val}$$

Rappelons que $A \mapsto B$ désigne l'ensemble des fonctions totales de A dans B, et $A \to B$ l'ensemble des fonctions (y compris partielles) de A dans B. Le concept de tableau est étendu à celui de champ :

$$\mathit{Champ} \in \mathbb{Z}^d \to \textsc{Val}$$

Un tableau est donc un cas particulier de champ : c'est un champ défini pour un index élément de $[0\ldots n_1] \times \cdots \times [0\ldots n_d]$ et indéfini ailleurs. La différence essentielle entre un champ et une fonction provient de l'implémentation extensionnelle du champ où les images d'un champ sont explicitement représentées alors qu'une fonction représente ses valeurs par une règle de calcul (généralement du code qu'il est nécessaire d'exécuter à chaque accès à l'image d'un index). Néanmoins, l'extension d'un champ pouvant être infinie, le calcul d'un champ doit obligatoirement être paresseux. Ce point sera abordé en section V.4.

Notre seconde généralisation consiste au remplacement de $\mathbb{Z}^d$ par un ensemble que l'on nomme une *forme* :

$$GBF \in forme \to \textsc{Val}$$

Une forme est un ensemble muni d'une structure de groupe (remarquons que $\mathbb{Z}^d$ est un groupe pour l'addition et donc un champ est un cas particulier de GBF). De la même façon que $[0\ldots n_1] \times \cdots \times [0\ldots n_d]$ représente le *type* d'un tableau, une forme représente le *type* d'un GBF.

Munir une forme d'une structure de groupe poursuit deux buts, selon qu'on adopte le point de vue du programmeur (et on s'intéresse dans ce cas à l'expressivité du langage) ou de l'implémenteur (et on s'intéresse dans ce cas à l'analyse des programmes) :

– Pour le programmeur, la structure de groupe lui permet de spécifier un *voisinage logique* des données.

– Pour l'implémenteur, la structure de groupe lui permet d'expliciter et de *raisonner sur les dépendances* et les mouvements de données.

La structure de groupe est un outil puissant pour la spécification de voisinages homogènes. Nous avons choisi cet outil car il est suffisamment général pour inclure à la fois la notion classique de tableau et d'arbre.

Les formes sont introduites dans la section V.1, les GBF dans la section V.4 et les opérations sur les GBF sont décrites en section V.5.

Nous nous intéresserons ensuite aux problèmes de la définition récursive de GBF, de manière analogue aux définitions récursives de tableaux permises par 81/2. Là encore, une distinction sera faite entre le traitement d'une fonction et le traitement d'un GBF. Nous avons mentionné dans la section III.4 page 24 qu'on ne permettait pas n'importe quel schéma récursif pour la définition d'un tableau, mais uniquement ceux qui amenaient à propager le calcul suivant un des axes naturels du tableau. Il en va de même pour les GBF : on n'admet que les schémas récursifs qui «respectent la notion de voisinage» spécifiée par la forme du GBF.

Nous aborderons ensuite le problème de l'implémentation des GBF. Des solutions sont proposées et l'implémentation des GBF abéliens est présentée dans le chapitre suivant. Enfin, des exemples illustrant les notions que nous avons conçues et développées sont donnés au chapitre VII.

V.1 Les formes

Une forme est un ensemble muni d'une certaine structure et qui va servir d'ensemble d'index à un GBF. Une forme représente donc une partie du type d'un tableau, le type complet comportant en plus la spécification du type des valeurs du tableau. L'information $[0 \ldots n_1] \times \cdots \times [0 \ldots n_d]$ est souvent résumée par le n-uplet $(n_1 \ldots n_d)$ et le type d'un tableau, par exemple `Array [`n_1 `...` n_d`] of Valeur` en Pascal, est donc un type paramétré par un n-uplet d'entiers. Cependant, les langages data-parallèles (comme *Lisp, HPF, POMP-C...) ont été amenés à introduire l'objet $[0 \ldots n_1] \times \cdots \times [0 \ldots n_d]$ comme un objet à part entière. Cet objet est appelé *form* en MOA, *geometry* en C*, *shape* en POMP-C... Cet objet permet de factoriser la déclaration de la structure du tableau entre plusieurs déclarations de type.

Dans cette section, nous étudions la notion de forme et donnons dans la section suivante des exemples de formes.

V.1.1 Voisinage d'un point, déplacement et composition de déplacements

Un espace est avant tout un ensemble d'éléments que l'on appelle des *points*. Cependant, quand on parle d'espace dans ce document on sous-entend quelque chose de plus que la seule union de points. Par exemple, dans le cadre de la simulation de l'équation de diffusion-réaction de TURING (cf. section III.2, page 20), nous avons défini une notion de cellule et de position de cellule (*à droite* et *à gauche* d'une cellule donnée). Dans l'exemple de la croissance de l'ammonite (cf. section III.3, page 22), il apparaît une notion de croissance suivant des directions privilégiées (le nord, le sud, l'est...). Dans l'exemple de la résolution d'équations récursives par la méthode de SHANLAN (cf. section XIV.2, page 210) ou de la traduction en 81/2$_{\mathcal{D}}$ des D0L systèmes (cf. section XIV.4.2.2, page 217), l'évolution de la collection support se fait de même suivant une direction privilégiée. Cela nous amène à définir la notion de voisinage et à la rendre explicite dans le type de la collection.

Un espace est donc un ensemble de points muni d'une notion de voisinage. Oublions un instant le problème de la spécification de l'ensemble des points et regardons comment définir la notion de voisinage.

Dans la suite, nous allons nous restreindre à des espaces *homogènes* (on dira aussi *réguliers*). De manière classique, un espace est dit homogène quand chaque point à le même voisinage (par exemple, «un voisin à gauche» et «un voisin à droite»). Nommons a, b, c... les directions pour se déplacer vers les voisins d'un point et $P{<}a{>}$ le voisin a d'un point P. Nous pouvons considérer a comme étant le déplacement à partir d'un point vers un de ses voisins. La figure 1 décrit plusieurs exemples de déplacements vers les voisins d'un point. Les opérations de déplacement peuvent être composées : en utilisant la notation multiplicative, nous écrivons $P<a.b>$ pour $(P<a>)<b>$. L'opération de composition des déplacements est aussi associative :

$$(a.b).c \quad = \quad a.(b.c)$$

car :

$$P<(a.b).c> = \ (P<a.b>)<c> = \ ((P<a>)<b>)<c> = \ P<a><b.c> \ = \ P<a.(b.c)>$$

Nous notons e le déplacement nul tel que $P<e> =$ P. De plus, nous définissons un déplacement inverse a^{-1} pour chaque déplacement a tel que $P<a.a^{-1}> = \ P<a^{-1}.a> = P$.

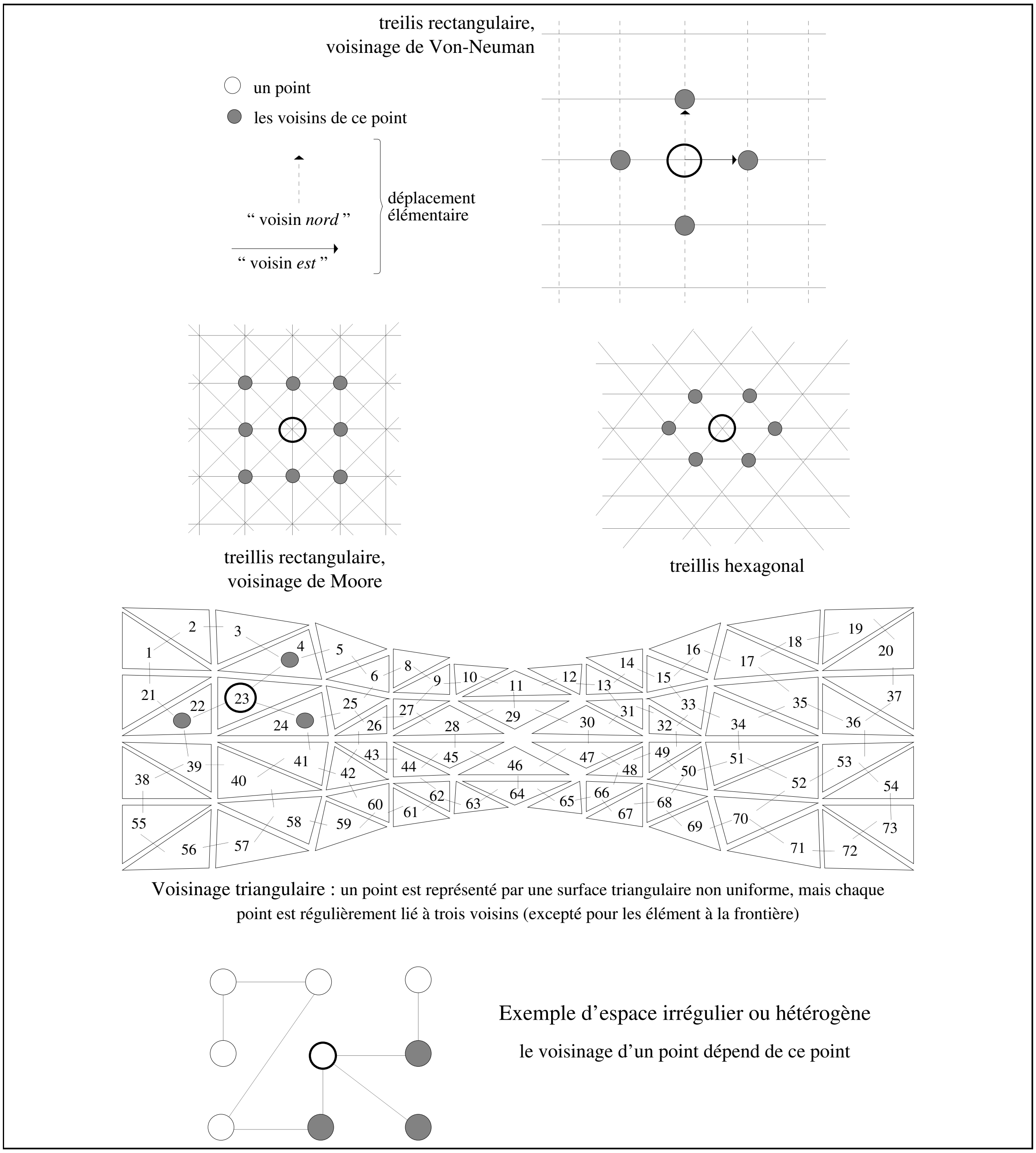


FIG. 1 –: *Quatre exemples d'espaces homogènes et un exemple d'espace hétérogène.*

En d'autres termes, les déplacement forment une structure de *groupe* pour la composition, et l'application des déplacements à un point est l'*action* du groupe sur l'espace des points.

V.1.2 Rappel mathématique

Rappelons [col95] qu'un groupe en mathématique est un couple $(G, .)$ où G est un ensemble et «.» une loi interne à G, associative, qui vérifie l'existence d'un élément neutre, noté e, et d'un inverse pour tout $x \in G$, noté x^{-1}. Un groupe G est dit *abélien* ou *commutatif* si sa loi est commutative : $\forall v, w \in G, v.w = w.v$.

On dit qu'un groupe G *opère* sur un ensemble E, et on parle de l'*action de G sur E*, si E est muni d'une loi externe φ dont le domaine d'opérateur est G :

$$(g, x) \quad \mapsto \quad \varphi(g, x)$$

de telle sorte que :

$$\varphi(g, \varphi(h, x)) = \varphi(g.h, x) \qquad \text{et} \qquad \varphi(e, x) = x$$

pour $g, h \in G$ et $x \in E$.

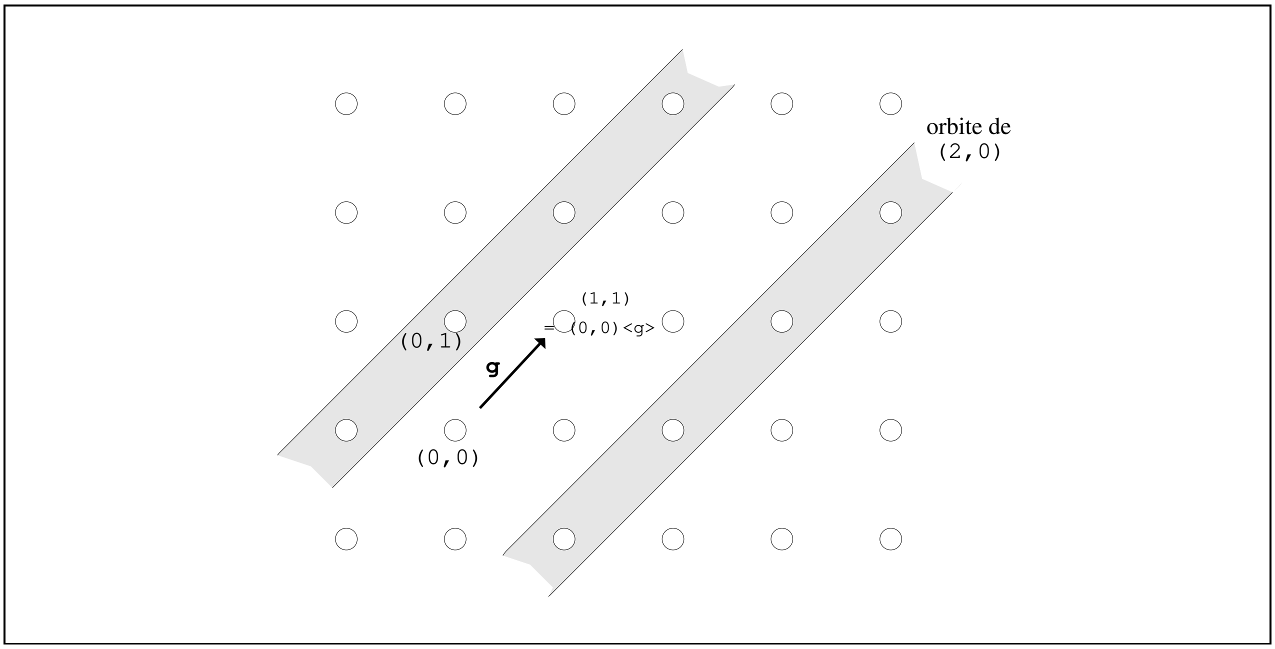


FIG. 2 –: *Action du groupe $G = \langle\!| g |\!\rangle$ dont les éléments sont les puissances de g, sur l'espace $\mathbb{Z}^2$. L'action de g sur $\mathbb{Z}^2$ consiste à se « déplacer en diagonale ». G n'opère pas transitivement sur $\mathbb{Z}^2$: une orbite correspond à une droite diagonale et il y a autant d'orbites que de droites diagonales distinctes.*

Soit $x \in E$, on appelle *orbite de x* l'ensemble des éléments $\varphi(g, x)$ pour $g \in G$. La figure 2 illustre la notion d'orbite. Deux orbites sont toujours disjointes ou confondues car la relation $x \sim y$ définie par :

$$x \sim y \quad \Leftrightarrow \quad \exists\, g \in G, y = \varphi(g, x)$$

est une relation d'équivalence sur E. Une classe d'équivalence de $\sim$ est une orbite. Le groupe G est dit *transitif* s'il n'existe qu'une seule classe d'équivalence (E tout entier). Cela signifie que si x et y sont deux éléments quelconques de E, il existe au moins un élément $g \in G$ tel que $y = \varphi(g, x)$.

Enfin, on appelle *espace homogène* un ensemble E muni d'un groupe transitif d'opérateurs.

V.1.3 Les points de l'espace

Intéressons-nous à présent à l'ensemble des points. Comment le spécifier ? La figure 2 nous montre qu'un ensemble de points, sur lequel G n'opère pas transitivement, ne nous convient pas : en effet, notre intuition est que la valeur d'un point P doit dépendre de la valeur des points atteignables à partir de P. Autrement dit, la valeur d'un point dépend uniquement des points de son orbite. S'il existe plusieurs orbites distinctes, alors les calculs des points de chacune de ses orbites sont complètement indépendants. Il est donc artificiel de réunir ces calculs dans la même structure de données.

Nous voulons donc spécifier un ensemble de points sur lequel l'ensemble des déplacements agit transitivement, ce qui signifie qu'à partir d'un point on peut rejoindre tout autre point par une composition de déplacements. Si on laisse le choix d'un ensemble arbitraire au programmeur, on est confronté à un double problème : spécificier l'action du groupe sur cet ensemble, s'assurer que cette action est transitive. Heureusement, il existe un ensemble sur lequel on est sûr que G agit transitivement : c'est G lui-même. Il suffit de prendre $\varphi(g, x) = g.x$, ou bien, dans notre cas $P{<}a{>} = P.a$.

V.1.4 Définition d'une forme

Si G représente l'ensemble des déplacements possibles, parmi ces déplacements, un sous-ensemble $S \subset G$ nous intéresse plus particulièrement. Nous appellerons les éléments de S les *déplacements élémentaires*. Nous ferons l'hypothèse que S génère G, c'est-à-dire que tout déplacement de G peut s'écrire comme composition des déplacements élémentaires et de leurs inverses.

Nous définissons $Forme(G, S)$ comme le graphe ayant G comme ensemble de ses sommets et $G \times S$ comme ensemble de ses arcs. Pour chaque arc $(g, s) \in G \times S$, le sommet de départ est g et celui d'arrivée est $g.s$. La *direction* ou *l'étiquette* de l'arc (g, s) est s. Chaque élément du sous-groupe généré par S correspond en même temps à un *chemin* (une succession de déplacements élémentaires) et à un point (le point atteint en partant du point identité e de G en suivant ce chemin). Nous utilisons alors la notation $P.s$ plutôt que $P{<}s{>}$ pour le voisin s de P. En d'autres termes, $Forme(G, S)$ est un graphe où :

- chaque sommet représente un élément du groupe,
- un arc étiqueté s se trouve entre les nœuds P et Q si $P.s = Q$,
- les étiquettes des arcs sont dans S et représentent un déplacement élémentaire.

Pour définir une forme $Forme(G, S)$ nous avons besoin de spécifier le groupe G et un ensemble fini S de générateurs. Nous allons utiliser pour cela une *présentation finie*.

Une présentation finie est spécifiée par une liste finie de générateurs et une liste finie d'équations contraignant l'égalité de deux mots. Une équation a la forme $v = w$ où v et w sont les produits de générateurs et de leurs inverses. La présentation d'un groupe n'est pas unique : différentes présentations peuvent définir le même groupe. Néanmoins, une présentation définit de façon unique $Forme(G, S)$: nous utilisons la liste des générateurs de la présentation pour spécifier S. Ainsi, les générateurs de la présentation sont les éléments distingués du groupe représentant les déplacements élémentaires d'un point vers ses voisins.

Remarque On supposera que les générateurs de deux formes différentes ont des noms différents. Par contre, le même symbole e désigne l'élément neutre dans tous les groupes indistinctement. Cela n'est pas gênant car le contexte permet de lever toute ambiguïté.

Donnons deux exemples. Soit C un groupe cyclique d'ordre n généré par $S = \{a\}$. $D1 = Forme(C, S)$ représente la discrétisation d'un cercle où n est le nombre de points de la discrétisation. Pour spécifier $D1$, nous spécifions entre les signes $\langle\!|$ et $|\!\rangle$, la liste des générateurs suivie de la liste des équations entre mots :

$$D1 = \langle\!| a \,;\; a^n = e |\!\rangle$$

Dans le cercle $D1$, nous pouvons toujours nous déplacer dans la même direction a. Il y a une seule équation qui spécifie qu'au bout de n déplacements, on se retrouve au même point. D'autres équations sont valides dans le groupe, par exemple $a^p = a^{p+n}$ mais toutes les équations valides dans $D1$ peuvent se déduire à partir de $a^n = e$ et des propriétés de groupe uniquement. Par exemple, en appliquant uniquement les lois de groupe, on a $a^{p+n} = a^p.a^n$. En appliquant la relation de $D1$, on déduit alors $a^{p+n} = a^p.e$ et la structure de groupe permet de déduire a^p. Si nous voulons aussi nous déplacer dans la direction inverse, a^{-1} doit être ajouté à la liste des générateurs.

REMARQUE Dans la suite, une présentation dénote indifféremment la forme correspondante ou bien le groupe sous-jacent en fonction du contexte. Ainsi, quand on écrit $g[D1]$ on veut dire que la forme de g est $D1$, alors que quand on écrit $x \in D1$ on veut dire que x est un élément du groupe associé à $D1$.

Une grille « bi-directionnelle » en deux dimensions est spécifiée par :

$$\begin{aligned} G2 \quad = \langle\!| \quad & Nord, Est, Sud, Ouest \; ; \\ & Nord.Est = Est.Nord, \; Sud = Nord^{-1}, \; Ouest = Est^{-1} \quad |\!\rangle \end{aligned}$$

Quatre générateurs sont définis, mais du fait des deux dernières équations, Sud et $Ouest$ ne sont que des renommages pour des raisons de facilité de $Nord$ et Est. La première équation spécifie que $Nord$ et Est commutent, c'est-à-dire que le groupe défini est *abélien*. Les groupes abéliens sont particulièrement intéressants comme structure de données, et on dispose d'un grand nombre de résultats théoriques sur eux. Nous utilisons la spécification entre les signes $\langle$ et $\rangle$ pour la présentation d'un groupe abélien, ce qui nous permet d'omettre les équations de commutation en les laissant implicites. Ainsi $G2$ peut se spécifier par :

$$\begin{aligned} G2 \quad = \langle \quad & Nord, Est, Sud, Ouest \; ; \\ & Sud = Nord^{-1}, \; Ouest = Est^{-1} \quad \rangle \end{aligned}$$

La figure 5 illustre d'autres exemples de formes abéliennes.

V.1.5 Groupe, forme et graphe de Cayley

Nous avons défini une forme comme un graphe. La représentation d'un groupe comme un graphe est connue sous le nom de *graphe de* CAYLEY et nous allons détailler les liens qui existent entre les notions de forme et de graphe de CAYLEY.

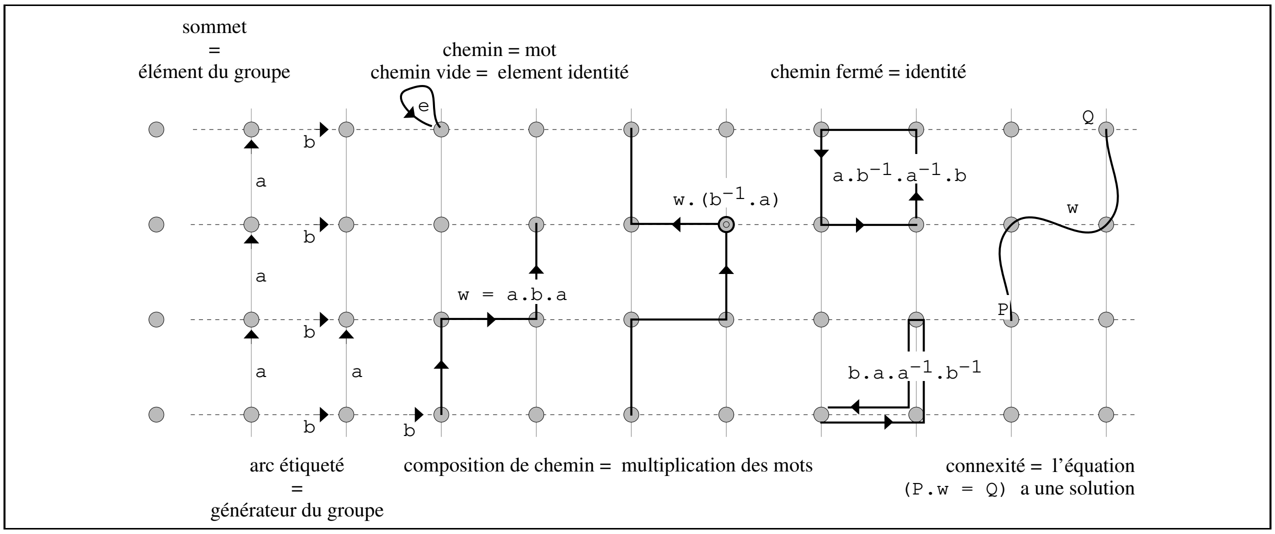


FIG. 3 –: *Représentation graphique des relations entre les notions de graphe de* CAYLEY *et la théorie des graphes.*

On dit que S est une *base* de G si tout élément de G est un produit des éléments de S et que S *génère* G si $S \cup S^{-1}$ est une base de G. Les propositions suivantes effectuent un lien entre la structure globale de $Forme(G, S)$ et les relations entre G et S :

- Pour que $Forme(G, S)$ soit connecté, il est nécessaire et suffisant que S génère G. Les composants connectés de $Forme(G, S)$ sont les cosets $g.H$ où H est le sous-groupe généré par S (un *coset* $g.H$, ou bien *classe à droite* dans le cas d'un groupe non abélien, est l'ensemble $\{g.h : h \in H\}$).

– Pour que *Forme*(G, S) ait une boucle (i.e. un circuit de longueur 1), il est nécessaire et suffisant que e soit un élément de S.

– *Forme*(G, S) à un circuit de longueur ≥ 2, si et seulement si $S \cap S^{-1} = \emptyset$.

Dans ce qui suit, ainsi que nous l'avons indiqué précédemment, nous nous restreignons au cas ou le sous-ensemble S génère G. Si S est une base de G, on appelle *Forme*(G, S) le graphe de CAYLEY du groupe G. Si S n'est pas une base de G, alors *Forme*(G, S) est un sous-graphe du graphe de CAYLEY de G. On notera qu'il existe des graphes réguliers[1] qui ne sont pas des graphes de CAYLEY d'un groupe [Whi73].

Le dictionnaire suivant, illustré par la figure 3, établit une correspondance entre la théorie des graphes et les concepts de la théorie des groupes :

Graphe de CAYLEY		*Groupe*
sommet	$\leftrightarrow$	élément du groupe
arc étiqueté	$\leftrightarrow$	générateur
composition de chemins	$\leftrightarrow$	multiplication de mot
chemin fermé (cycle)	$\leftrightarrow$	mot égal à e
connexité	$\leftrightarrow$	résolution de $P.x = Q$
(il existe un chemin de tout P vers tout Q)		

V.2 Exemples de formes

Avant de présenter des exemples de formes abéliennes et non-abéliennes, nous avons besoin d'introduire une nouvelle notion : on dit qu'un groupe est *libre*, ou qu'une forme est libre, s'il n'y a pas d'équation dans la présentation associée. Un groupe abélien libre, ou une forme abélienne libre, correspond à une présentation où les seules équations sont les équations de commutation des générateurs.

V.2.1 Exemples de formes abéliennes

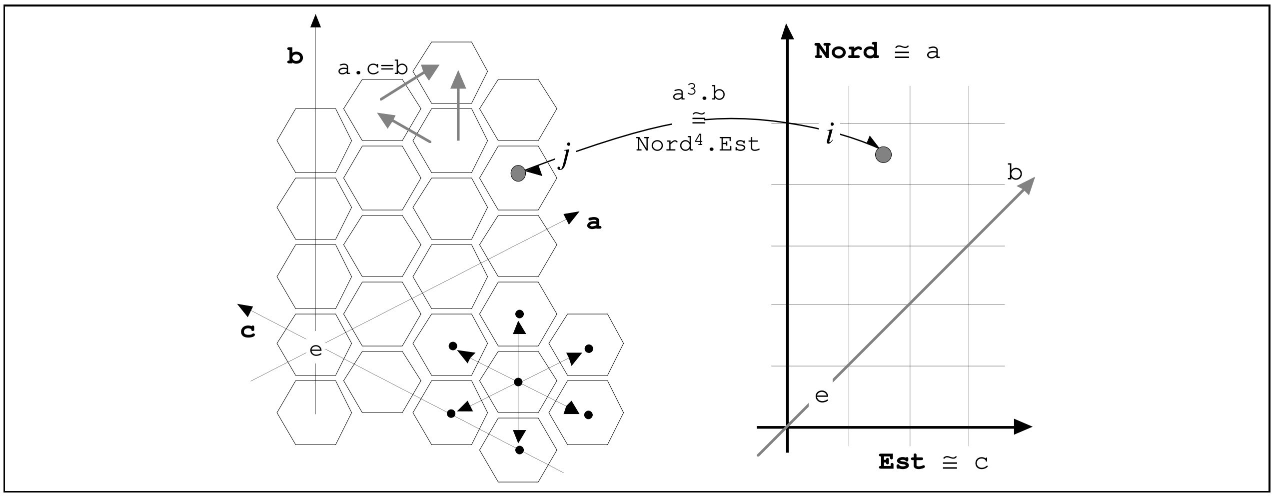


FIG. 4 –: *Représentation de H2 et G2 et de l'isomorphisme entre eux.*

Un théorème fondamental sur les groupes abéliens énonce que tout groupe abélien G est isomorphe à un produit de $\mathbb{Z}$-modules :

$$G \simeq \mathbb{Z}^n \times \mathbb{Z}/n_1\mathbb{Z} \times \mathbb{Z}/n_2\mathbb{Z} \times \ldots \times \mathbb{Z}/n_q\mathbb{Z}$$

où n_i divise n_{i+1} (cf. [LB78] ; pour une approche plus orientée informatique, on se reportera à [Coh93]). L'outil de base pour expliciter l'isomorphisme entre la présentation finie d'un groupe abélien et les $\mathbb{Z}$-modules est

1. Un graphe régulier est un graphe où chaque sommet a le même nombre de sommets voisins.

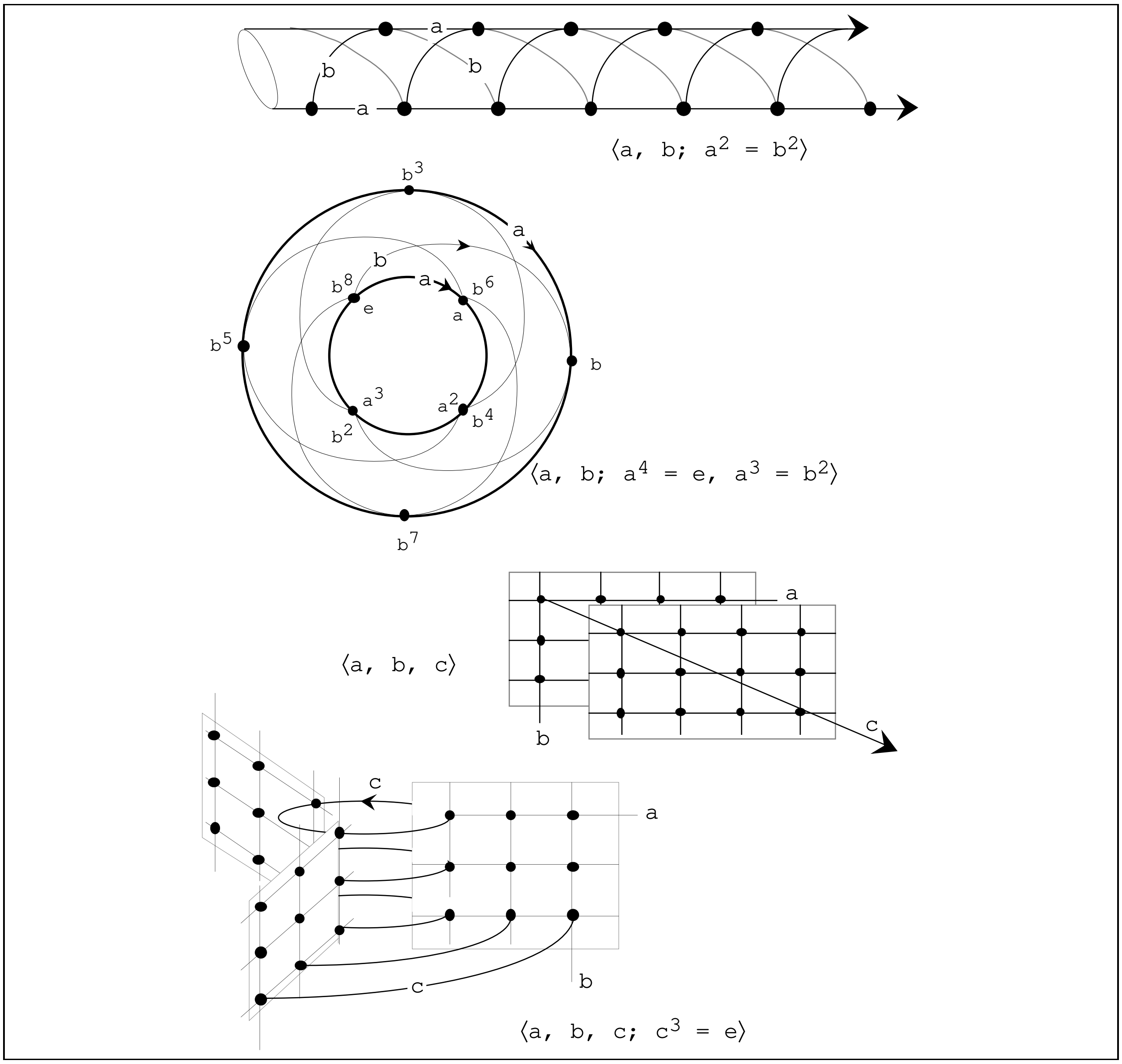


FIG. 5 –: *Quatre présentations de groupes abéliens et leur graphe Forme(G, S) associé.*

le calcul de la *forme normale de* SMITH (FNS) (cf. [Smi66] et le chapitre suivant). En d'autres termes, les formes abéliennes correspondent à une combinaison de grilles et de tores n-dimensionnels.

Dans la mesure où les tableaux (tels que les tableaux PASCAL par exemple) sont essentiellement des grilles finies, notre définition de champs basés sur les groupes englobe naturellement le concept usuel de tableau comme un cas particulier de régions bornées sur une forme abélienne libre. Par exemple, les champs de Indexical Lucid [AFJW95], les tableaux systoliques, les champs de données de LISPER [LC94] entrent dans ce cadre. Cette représentation permet la réutilisation de tous les travaux effectués dans le domaine de l'implémentation (séquentielle ou parallèle) des tableaux (e.g. [Fea91, Tor93]) pour implémenter les régions bornées sur des champs abéliens libres. Avec un peu de travail additionnel, il est possible de les adapter à la manipulation de champs abéliens finis.

Remarquons que les listes paresseuses entrent aussi dans le cadre des formes abéliennes comme région non-bornée sur une forme abélienne libre à un seul générateur.

Voici des exemples de groupes abéliens. Nous avons déjà rencontré *G2* qui correspond à une grille NEWS à deux dimensions, et soit $H2 = \langle a, b, c;\ b = a.c \rangle$ qui correspond à une grille hexagonale. La figure 4 présente

$H2$ et $G2$ dans une convention différente de celle utilisée dans la figure 5 : un sommet est représenté par une cellule plutôt que par un point et deux sommets sont voisins si les cellules correspondantes sont adjacentes.

$H2$ représente une partition hexagonale du plan qui a d'intéressantes propriétés en topologie discrète. Par exemple, une courbe fermée dans $H2$ est dite de JORDAN, c'est-à-dire quelle sépare le plan en deux régions connexes (ce n'est pas le cas dans $G2$ ce qui explique qu'on utilise $H2$ en traitement d'image).

$H2$ est isomorphe à $G2$ par les injections suivantes :

$$\begin{array}{llll} i: & H2 \longrightarrow G2, & a^p.b^q.c^r \rightarrow North^{p+q}.East^{q+r} \\ j: & G2 \longrightarrow H2, & North^s.East^t \rightarrow a^s.c^t \end{array}$$

On notera que si $H2$ et $G2$ sont des groupes isomorphes, ils n'en ont pas la même forme pour autant dans le sens ou l'image des $H2$-voisins d'un point P dans $H2$ ne sont pas les $G2$-voisins de l'image de P dans la forme $G2$.

D'autres exemples d'isomorphismes sont donnés par les formes de la figure 5. Pour le premier exemple :

$$\langle a, b\,;\ a^2 = b^2 \rangle \simeq \langle a, c\,;\ c^2 = e \rangle = \mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$$

(avec $c \leftrightarrow a.b^{-1}$). Pour le second exemple

$$\langle a, b\,;\ a^4 = e, a^3 = b^2 \rangle \simeq \langle d\,;\ d^8 = e \rangle = \mathbb{Z}/8\mathbb{Z}$$

(avec $d \leftrightarrow a^{-2}.b$).

V.2.2 Deux exemples de formes non abéliennes

Les groupes abéliens représentent un ensemble important, mais particulier, de groupes. Nous donnons ici deux exemples significatifs de formes non-abéliennes. Rappelons que la présentation de formes non-abéliennes est spécifié entre les symboles ⟨| et |⟩.

Le premier exemple (cf. [Rók94a]) est un *voisinage triangulaire* : les sommets de T sont au centre de triangles équilatéraux et les voisins d'un sommet sont les nœuds situés au centre des triangles adjacents

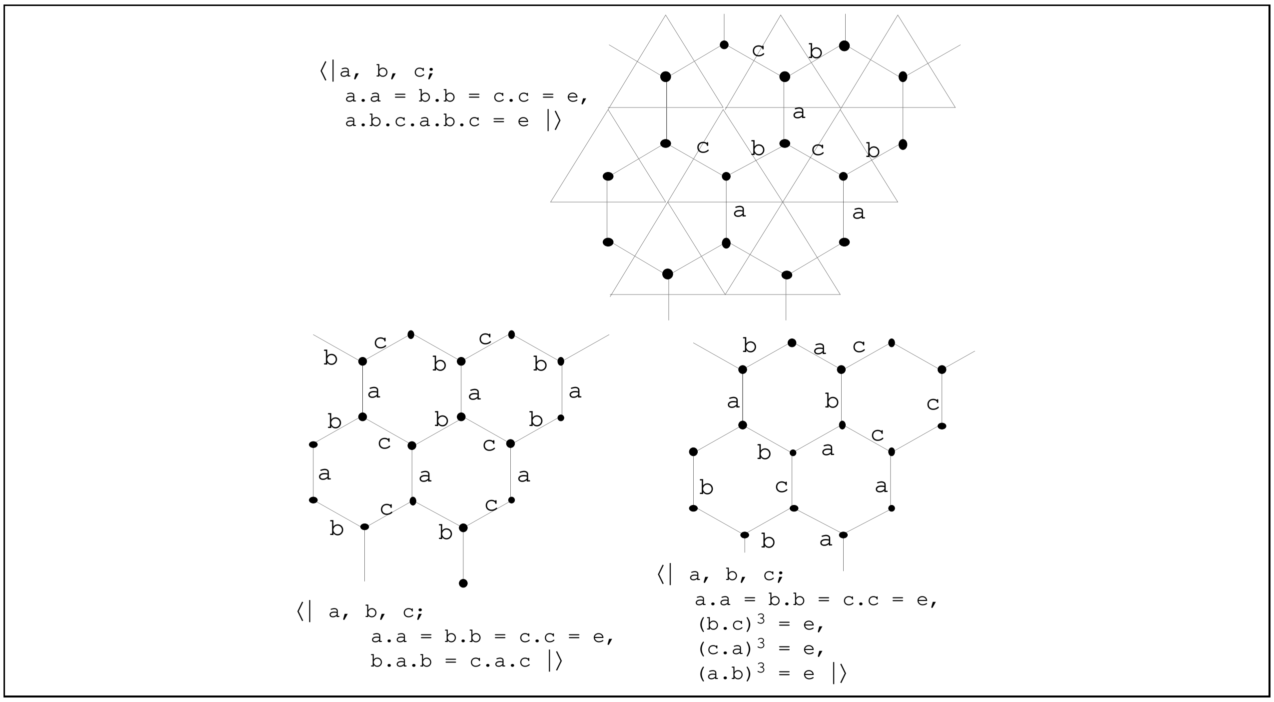


FIG. 6 –: *Trois exemples de formes à trois voisins. Ces formes ne sont pas abéliennes.*

côtés par côtés :

$$T = \langle\!|a, b, c\,;\; a^2 = b^2 = c^2 = e, (a.b.c)^2 = e|\!\rangle$$

On rencontre un treillis de cette sorte, par exemple, dans les problèmes de dynamique des fluides car leur symétrie mime efficacement la symétrie des lois des fluides. La figure 6 illustre une représentation de T ainsi que deux autres présentations pour une partition triangulaire du plan. Il est aisé de voir que, par exemple, $a.b \neq b.a$, et par conséquent que ce groupe n'est pas abélien.

Notre second exemple est simplement un groupe libre. On rappelle qu'un groupe libre est un groupe sans contraintes entre les générateurs. La figure 7 illustre le groupe non-abélien libre suivant :

$$F2 = \langle\!|x, y|\!\rangle$$

Nous voyons que l'espace correspondant peut être représenté par un arbre, c'est-à-dire un graphe non-vide sans circuit. En fait, il existe un résultat plus général exprimant le fait que si $Forme(G, S)$ est un arbre alors G est un groupe libre généré par S.

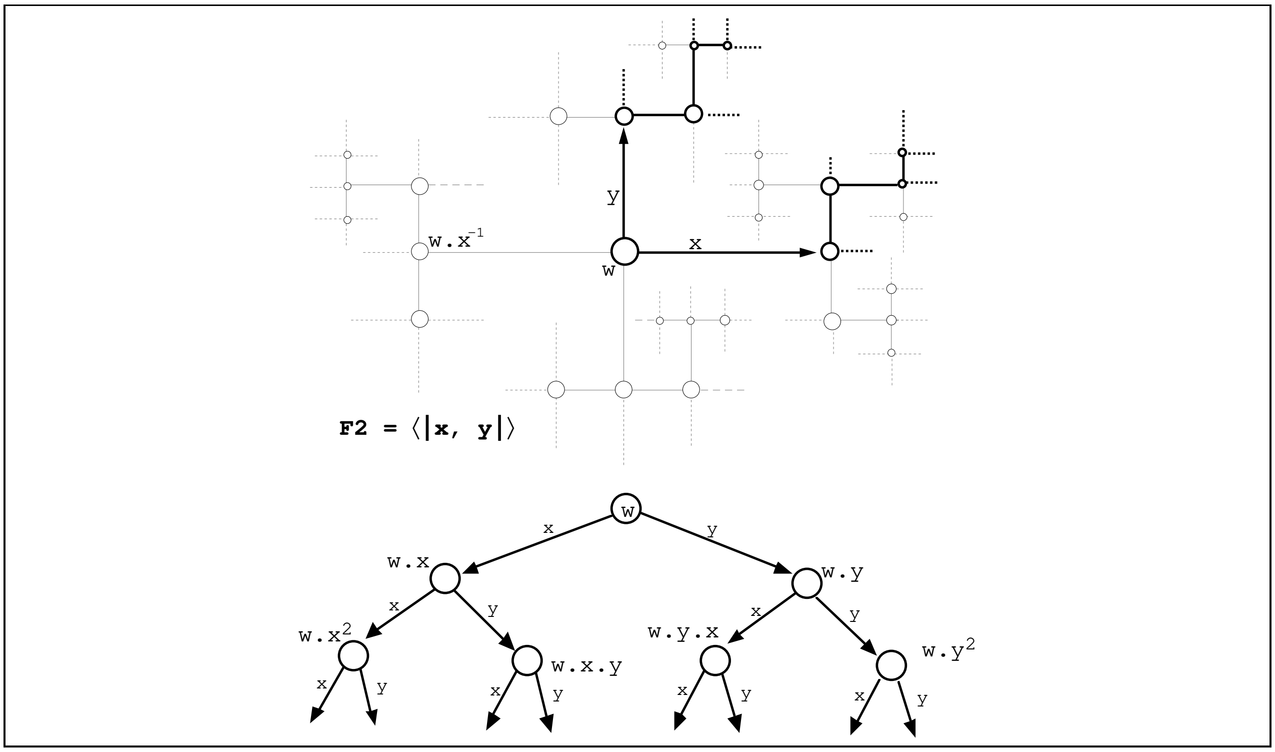


FIG. 7 –: *Un groupe non-abélien libre avec deux générateurs. Les lignes en gras correspondent à des points qui peuvent être atteints à partir du point w en suivant les déplacements élémentaires x et y.*

Cela permet l'incorporation de certaines classes d'arbres dans notre cadre. Soit $Forme(G, S)$ où G est un groupe libre et S un ensemble minimal de générateurs, i.e. tel qu'aucun sous-ensemble propre de S ne génère G. Alors, $Forme(G, S)$ est un arbre. On observera que cet arbre n'a aucun nœud sans prédécesseur. Cette situation est peu commune en informatique où les arbres infinis ont une racine et « grandissent » par les feuilles. La figure 7.b illustre les points accédés à partir d'un point w de $F2$: c'est un arbre binaire ayant pour racine w. Il n'est pas possible de rapprocher la notion de générateur de la notion de père (en effet, pour le nœud $w.x$, le père est x^{-1}, tandis que c'est y^{-1} pour le nœud $w.y$).

V.3 Construction de formes

Nous avons introduit la notion de forme et nous avons spécifié des formes par une présentation finie. Dans cette section, nous présentons des opérations de construction de formes à partir d'autres formes. Suivant le

Présentation finie d'un groupe	$\underbrace{\mathrm{G} = \langle x, y \,;\, \rangle,\ \mathrm{F} = \langle\!\langle a, b, c \,;\, a^2 = b \rangle\!\rangle, \ldots}$
Construction de forme	$G \times F, \quad G * F, \quad G/F, \ldots$

FIG. 8 –: *Types de base et constructeurs de types.*

point de vue «forme = type», ces opérations sont des constructeurs de types (voir figure 8). Mais avant de présenter quelques constructions nous allons discuter de l'égalité de deux formes.

V.3.1 Égalité de deux formes

Si on adopte le point de vue qui consiste à voir les formes comme des types, et que l'on construit des formes par des expressions, le problème de l'égalité de deux formes se pose naturellement.

Dans les langages de programmation, plusieurs sortes d'égalités de types sont utilisées : égalité par nom (deux types sont égaux s'ils réfèrent la même définition) ou égalité structurelle (deux types sont égaux s'ils définissent la même structure de données à un isomorphisme près). Par exemple, devons nous identifier $\langle a \rangle$ et $\langle b \rangle$ (égalité structurelle des formes) ou bien les considérer comme deux types distincts ? Il y a une notion d'isomorphisme naturel entre les formes, qui consiste à dire que des formes sont isomorphes si l'une se déduit de l'autre à un renommage des générateurs près.

Plus précisément, soit f une bijection de S dans S'. Si S est un ensemble de générateurs de G, alors tout élément g de G peut s'écrire comme un mot de S : $g = s_1. \ldots .s_n$. L'homomorphisme $\overline{f} : G \to G$ associé à f est défini par :

$$\overline{f}(g) \quad = \quad f(s_1). \ \ldots \ .f(s_n)$$

On dit qu'une forme $F = \mathit{Forme}(G, S)$ est isomorphe à une forme $F' = \mathit{Forme}(G', S')$ et on note $F \equiv F'$, si et seulement si il existe f bijection de S vers S', telle que $\forall w_1, w_2 \in G$:

$$w_1 = w_2 \quad \Rightarrow \quad \overline{f}(w_1) = \overline{f}(w_2)$$

et $\forall {w_1}', {w_2}' \in G'$:

$$w_1{}' = w_2{}' \quad \Rightarrow \quad \overline{f^{-1}}(w_1) = \overline{f^{-1}}(w_2)$$

Par exemple, $\langle\!\langle a \rangle\!\rangle \equiv \langle\!\langle b \rangle\!\rangle$. Par contre, peut-on dire que $\langle a \,;\, a^2 = e, a^4 = e \rangle$ est isomorphe à $\langle b \,;\, b^2 = e \rangle$? Dans ce cas, on peut le prouver, mais montrer l'isomorphisme de deux groupes, ce qu'implique l'isomorphisme de deux formes, n'est pas décidable dans le cas général. C'est pourquoi on ne considère que l'égalité «par nom» sur les formes.

V.3.2 Sous-formes

Un *sous-groupe* H d'un groupe G est une partie non vide telle que le composé de deux éléments de H est encore un élément de H et telle que H soit un groupe pour la loi de composition induite par G.

Une forme $E = \mathit{Forme}(H, S')$ est une *sous-forme* de $F = \mathit{Forme}(G, S)$ si et seulement si H est un sous-groupe de G. On note alors :

$$E = S' : F$$

Par exemple, $\langle \mathit{Est} \rangle : \mathit{G2}$ représente «l'axe horizontal» de la grille $\mathit{G2}$.

REMARQUE On voit que dans cette définition on ne demande pas $S' \subset S$ et par conséquence, les déplacements élémentaires dans E ne sont pas nécessairement des déplacements élémentaires du point de vue de F.

V.3.3 Produit cartésien de deux formes

Soit $F = Forme(G, S)$ et $F' = Forme(G', S')$. Alors le produit cartésien $F \times F'$ est défini par :

$$F \times F' \quad = \quad Forme(G \times G', (S \times \{e\}) \cup (\{e\} \times S'))$$

où le produit $H = G \times G'$ de deux groupes G et G' est un groupe dont l'ensemble des éléments est le produit cartésien de G et de G', et dont la loi de composition est :

$$(x, x').(y, y') \quad = \quad (x.y, x'.y')$$

L'injection de $G \to G \times G'$, qui à x associe (x, e) permet d'identifier canoniquement G avec une partie de $G \times G'$ (qui peut alors se voir comme un sous-groupe de $G \times G'$), et il en va de même avec G' avec l'injection $x \mapsto (e, x)$. Les générateurs considérés pour la forme produit sont donc l'union des générateurs des formes arguments. Regardons ce qui se passe sur les présentations, si :

$$\begin{aligned} F &= \langle\!| a, b, \ldots\ ;\ w_1 = w_2, \ldots |\!\rangle \\ F' &= \langle\!| a', b', \ldots\ ;\ {w_1}' = {w_2}', \ldots |\!\rangle \end{aligned}$$

alors :

$$\begin{aligned} F \times F' \quad = \quad & \langle\!| (a, e), (b, e), \ldots, (e, a'), (e, b'), \ldots ; \\ & (w_1, e) = (w_2, e), \ldots, (e, {w_1}') = (e, {w_2}'), \ldots |\!\rangle \end{aligned}$$

Prenons l'exemple suivant :

$$\begin{aligned} F_1 &= \langle a \rangle \\ F_2 &= F_1 \times F_1 = \langle (a, e), (e, a) \rangle \end{aligned}$$

Remarquons que cette forme est isomorphe à la forme $\langle x, y \rangle$ (voir figure 9).

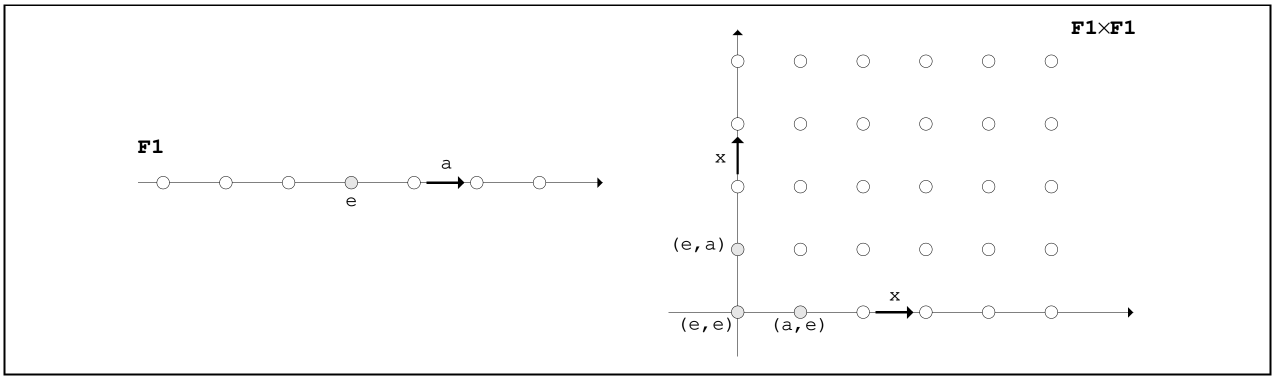


FIG. 9 –: *Le produit cartésien de deux formes $F_1 = \langle a \rangle$ et $F_2 = F_1 \times F_1$.*

V.3.4 Produit libre de deux formes

Nous définissons le produit libre à partir des présentations, mais une définition plus abstraite à partir du produit libre de deux groupes est possible.

Soient :

$$\begin{aligned} F &= \langle\!| a, b, \ldots\ ;\ w_1 = w_2, \ldots |\!\rangle \\ F' &= \langle\!| a', b', \ldots\ ;\ {w_1}' = {w_2}' |\!\rangle \end{aligned}$$

Le produit libre $F * F'$ est défini par :

$$\begin{aligned} F * F' \quad = \quad & \langle\!| a, b, \ldots, a', b', \ldots ; \\ & w_1 = w_2, \ldots, {w_1}' = {w_2}', \ldots |\!\rangle \end{aligned}$$

REMARQUE Le produit libre de deux formes n'est pas isomorphe au produit cartésien de deux formes (les éléments correspondant à G et G' dans $G \times G'$ commutent alors que ce n'est pas le cas dans $G * G'$). Si les deux formes sont abéliennes, on définit le produit libre abélien où tous les générateurs commutent.

V.3.5 Quotient d'une forme

Nous commençons par définir le quotient G/H d'un groupe G par un sous-groupe H. G/H est l'ensemble des classes d'équivalences de la relation d'équivalence :

$$x \simeq y \quad \Leftrightarrow \quad x^{-1}.y \in H$$

La classe d'équivalence de x est le coset $x.H = \{x.h, h \in H\}$. Afin de munir G/H d'une structure de groupe, il est nécessaire que H vérifie la condition :

$$\forall x \in G, \quad x.H = H.x$$

Si le sous-groupe H vérifie cette condition, on dit que c'est un *sous-groupe normal* (ou *distingué*). Nous pouvons maintenant définir une structure de groupe sur G/H : H est l'élément neutre, et si u et v sont deux représentants des classes H_1 et H_2, alors $H_1.H_2$ est défini comme la classe d'équivalence de $u.v$.

Cette définition ne dépend pas des représentations u et v qui sont choisies. En effet, supposons $u, u' \in H_1$ et $v, v' \in H_2$. Alors, $u' = u.h_1$ et $v' = v.h_2$ avec $h_1, h_2 \in H$. Nous voulons montrer que la classe de $u.v$ est aussi la classe de $u'.v'$, c'est-à-dire que $u.v \simeq u'.v'$. Or :

$$\begin{aligned} w &= (u'.v')^{-1}.(u.v) \\ &= v'^{-1}.u'^{-1}.u.v \\ &= h_2^{-1}.v^{-1}.h_1^{-1}.u^{-1}.u.v \\ &= h_2^{-1}.v^{-1}.h_1^{-1}.v \end{aligned}$$

Comme H est un sous-groupe normal, $h^{-1}.v = v.h^{-1}$ et par conséquent :

$$\begin{aligned} w &= h_2^{-1}.v^{-1}.v.h_1^{-1} \\ &= h_2^{-1}.h_1^{-1} \end{aligned}$$

donc $w \in H$ car il est le produit de l'inverse de deux éléments de H. D'où $u.v \simeq u'.v'$.

La figure 10 illustre le quotient de $G = \langle a; a^4 = e\rangle$ par $H = \langle a^2\rangle : G$. Il apparaît que le quotient G/H consiste à identifier les sommets de G qui sont joignables par un déplacement dans H.

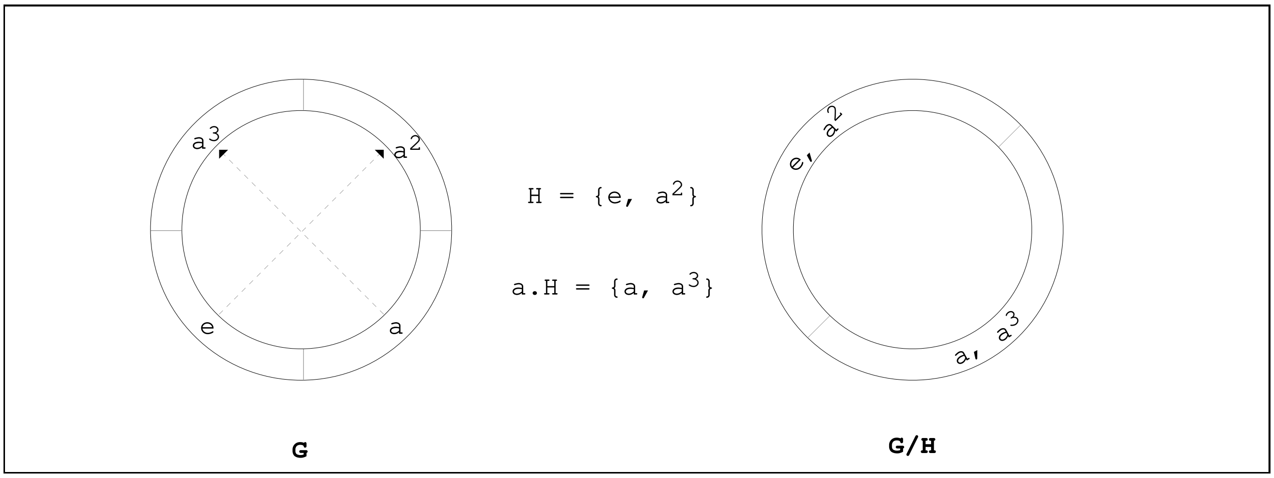


FIG. 10 –: *Quotient de $G = \langle a; a^4 = e\rangle$ par $H = \langle a^2\rangle : G$.*

On se place dans le cas des groupes abéliens pour lesquels on sait que tout sous-groupe est normal. À partir du quotient d'un groupe par un sous-groupe, on définit le quotient d'une forme $F = \mathit{Forme}(G, S)$ par un sous-groupe normal H par :

$$F/H \quad = \quad \mathit{Forme}(G/H, S)$$

En termes de présentation, si on a :

$$G \quad = \quad \langle a, b, \ldots; w_1 = w_2, \ldots\rangle$$

$$H \quad = \quad \langle h_1, \ldots \rangle : G$$

alors :

$$G/H \quad = \quad \langle a, b, \ldots ; w_1 = w_2, \ldots, h_1 = e, \ldots \rangle$$

V.4 Les champs basés sur des groupes (GBF)

Un champ basé sur un groupe, que nous abrégeons par GBF ou plus simplement par *champ*, est une fonction d'une forme dans un ensemble de valeurs. Si $g : F \to \text{VAL}$, on écrira $g[F]$ pour spécifier que g est un champ sur la forme F.

Une forme $\mathit{Forme}(G, S)$ étant un graphe, un GBF est en fait une fonction de l'ensemble des sommets G du graphe. La structure supplémentaire dont une forme est munie est utilisée pour contraindre les opérations possibles sur le GBF. Ces opérations sont décrites en section V.5. Dans cette section, nous allons nous intéresser aux « domaines de calcul » d'un GBF.

On dit que $F = \mathit{Forme}(G, S)$ est infini si G n'est pas un ensemble fini (S est toujours un ensemble fini car on utilise une présentation finie pour spécifier F). Dans le cadre des GBF, F joue le rôle d'un ensemble d'index et donc, de toute évidence, seules les valeurs d'un champ sur un domaine fini sont d'un intérêt pratique. Cela pose donc un problème.

Une première approche pour résoudre ce problème est de spécifier explicitement un sous-domaine fini dans une forme, de la même manière que $[0, \ldots, n_2] \times \cdots \times [0, \ldots, n_d]$ est la spécification d'un domaine d'itération d'un nid de boucles dans le traitement d'un tableau. Cette approche n'est pas satisfaisante, et cela pour trois raisons :

1. La réponse à la question « Quel est le langage qui remplit le mieux les conditions pour la spécification de sous-ensembles finis dans un groupe ? » est loin d'être évidente. De plus, quelque soit le langage choisi, il est douteux qu'il puisse définir de façon concise une région quelconque comme nous l'avions souhaité (cf. section IV.2.2, page 34).

2. Que signifie la spécification d'un domaine fini sur un groupe fini ? Illustrons le problème par un exemple. Supposons que l'on définisse un champ sur *D1*(4), le cercle discrétisé avec 4 éléments. On le restreint à l'intervalle $\{a^\alpha \mid 0 \leq \alpha < 3\}$. La structure cyclique de *D1*(4) est perdue et le programmeur peut utiliser indifféremment *G1* comme domaine sous-jacent pour ce champ. À l'inverse, si on le restreint à $\{a^\alpha \mid 0 \leq \alpha < 8\}$ on « re-boucle » deux fois (voir la figure 11). Ce n'est certainement pas un comportement souhaitable, car le choix de ce sous-ensemble contredit la structure de la forme.

3. Si nous savons de façon sûre où nous désirons obtenir un résultat (dans l'exemple de la diffusion de la chaleur, par exemple, cf. section VII.2, ce peut être la température au centre de la barre au 100$^{\text{e}}$ pas de temps) nous ne connaissons pas l'ensemble des points qui doivent être évalués pour calculer le résultat désiré. Dans un tel contexte, spécifier explicitement un sous-domaine du champ nous conduit à une erreur du type « accès en dehors des bornes » lors du calcul, ou bien à des calculs inutiles.

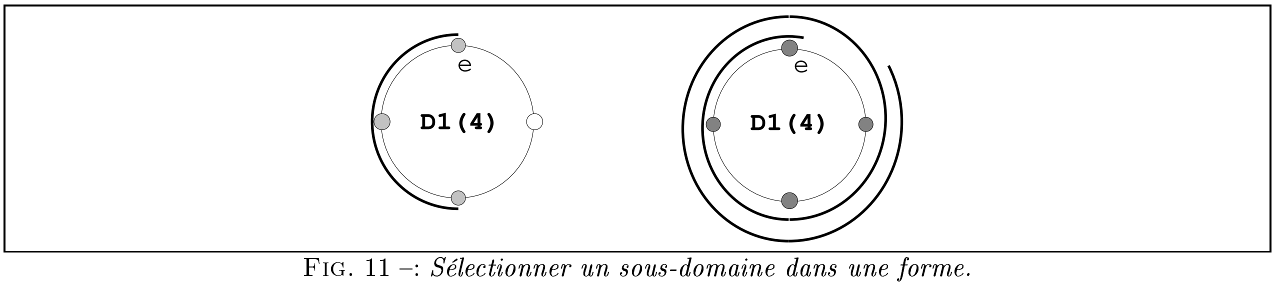


FIG. 11 –: *Sélectionner un sous-domaine dans une forme.*

En conséquence, nous adoptons une *approche paresseuse* pour le calcul d'un champ : un champ est défini virtuellement sur sa forme entière, même si celle-ci à une infinité d'éléments. Les valeurs des points du champ ne sont calculées que *si elles sont requises.*

Si l'utilisation d'une structure de données partielle et d'une stratégie d'évaluation paresseuse nous permet de résoudre le problème de la spécification d'un sous-domaine de calcul, il ne résout pas le problème de la spécification du domaine sur lequel le résultat est demandé car on désire certainement connaître la valeur du champ en plusieurs points. Néanmoins, le problème est bien plus simple car nous disposons maintenant d'un objet mathématiquement plus « propre ». La spécification du sous-ensemble requis des valeurs est analogue à la spécification d'un bon format pour la fonction `print`, exactement comme `"%.8f"` dans le langage `C` permet l'affichage des 8 premières décimales d'un `float`.

Comme premier «langage de formattage», nous faisons la proposition suivante. Les éléments d'un groupe abélien sont tous énumérés par $g_1^{n_1}.g_2^{n_2}. \dots .g_p^{n_p}$ où les g_i sont les générateurs de la forme. Un sous-ensemble est spécifié en donnant l'intervalle pour chaque n_i.

Pour les champs non-abéliens, tous les éléments du groupe sont énumérés par $g_{i_1}^{n_1}.g_{i_2}^{n_2}. \dots .g_{i_p}^{n_p}. \dots$ où g_{i_j} est un générateur et $g_{i_j} \neq g_{i_{j+1}}$. Ainsi, nous spécifions un nombre maximal p de générateurs et un intervalle pour la puissance de chaque générateur.

La figure 12.a illustre un domaine borné dans *H2* où chaque point est décrit par $a^\alpha.b^\beta.c^\gamma$ avec $0 \leq \alpha < 4$, $-1 \leq \beta < 2$, $0 \leq \gamma < 2$. La figure 12.b illustre un domaine dans T, le nombre maximal de générateurs étant 2 avec $0 \leq n_j < 2$.

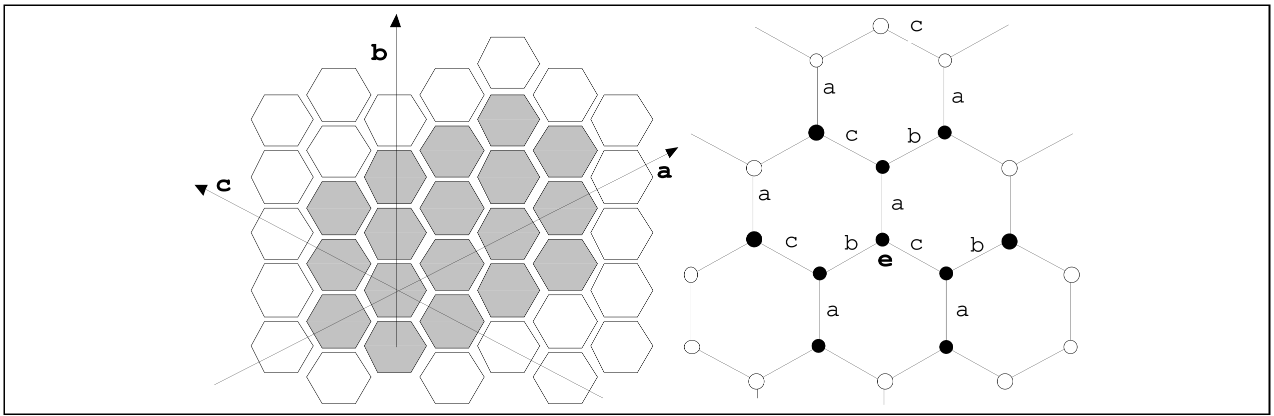


FIG. 12 –: *Deux domaines décrits par un format dédié à l'énumération d'une région dans un champ paresseux. Dans H2, le domaine correspond à* $\{a^\alpha.b^\beta.c^\gamma \mid 0 \leq \alpha < 4, -1 \leq \beta < 2, 0 \leq \gamma < 2\}$ *et pour T il s'agit de* $\{g^\alpha, g^\alpha.g'^\beta \mid g, g' \in \{a,b,c\}$ *et* $0 \leq \alpha, \beta < 2\}$.

V.5 Opérations sur les GBF

Nous détaillons dans cette section les opérations définies sur les GBF. Ces opérations dépendent essentiellement des formes sous-jacentes. Les fonctions présentées doivent être comprises comme une première algèbre sur les GBF qui pourra être étendue suivant les besoins.

V.5.1 Restriction du domaine d'un GBF

Nous venons de voir qu'un GBF est une structure nécessitant un modèle de calcul paresseux et qu'un certain langage est nécessaire à la spécification d'un domaine où un calcul est requis. Indépendamment de cela, un GBF est une structure *partielle*, c'est-à-dire que la valeur en certains points peut être indéfinie. Nous voulons traiter deux aspects différents par la notion de valeur indéfinie :

– Une valeur peut être implicitement indéfinie car le calcul qui la produit ne se termine pas.

– Une valeur peut être explicitement indéfinie parce que, par exemple, elle représente une valeur dont on ne se soucie pas (cf. section IV.2.2, page 34).

On est amené à distinguer les deux notions et, pour traiter le second aspect, nous introduisons explicitement une valeur spéciale, notée `nil`, et une opération de *restriction*.

Soit $g[F]$ un GBF et $h[F]$ un GBF à valeur dans $\textsc{Bool} \cup \{\texttt{nil}\}$, alors :

$$g' \quad = \quad g \,\texttt{where}\, h$$

est le GBF de forme F tel que :

$$\forall p \in F, \quad g'(p) \quad = \quad \begin{cases} g(p) & \text{si } h(p) = \texttt{true} \\ \texttt{nil} & \text{si } h(p) \in \{\texttt{false}, \texttt{nil}\} \end{cases}$$

et on dit que g' est la *restriction* de g à h.

Par extension, si h n'est pas un champ à valeurs booléennes, l'expression $g' = g \,\texttt{where}\, h$ dénote :

$$g' \quad = \quad g \,\texttt{where}(h \neq \texttt{nil})$$

où $(h \neq \texttt{nil})$ est le GBF de forme F et dont la valeur en un point p est égale à `true` si la valeur de h en p est définie et différente de `nil`, et égale à `false` si elle est égale à `nil`.

Nous définissons aussi la notion de *restriction explicite*, qui est une variante de la restriction, où le domaine est spécifié par le langage évoqué en section V.4. Si $g[F]$ est défini avec $F = \mathit{Forme}(G, S)$ et $s_j \in S$, alors :

$$g' \quad = \quad g \,\texttt{on}\, s_1^{i_1}.\ldots.s_n^{i_n} \,\texttt{where} \quad \alpha_1 \leq i_1 < \alpha'_1, \,\ldots,\, \alpha_n \leq i_n < \alpha'_n$$

est le GBF de forme F tel que :

$$\forall p \in G, \quad g'(p) \quad = \quad \begin{cases} g(p) & \text{si } p \in \{s_1^{i_1}.\ \ldots\ .s_n^{i_n},\ \ldots,\ \alpha_j \leq i_j < \alpha'_j, \ldots\} \\ \texttt{nil} & \text{sinon} \end{cases}$$

V.5.2 Union de deux GBF

Soit $g[G]$ et $g'[G]$ deux GBF, alors l'*union* $(g \# g')[G]$ est définie par :

$$\forall p \in G, \quad (g \# g')(p) \quad = \quad \begin{cases} g(p) & \text{si } g(p) \neq \texttt{nil} \\ g'(p) & \text{sinon} \end{cases}$$

REMARQUE Les opérateurs de restriction et d'union sur les GBF sont réminiscents de la structure de contrôle data-parallèle `where` en POMP-C ou `*when` en `*Lisp` qui permet de restreindre un traitement à une sous-région d'un tableau. Par exemple, l'expression :

```
(*when (a < 0) (!! -a) a)
```

qui calcule le tableau des valeurs absolues du tableau `a` en `*Lisp`, peut se traduire en GBF par l'expression :

$$(-a \,\texttt{where}\, a < 0) \# (a \,\texttt{where}\, a \geq 0)$$

(l'utilisation de l'extension d'une conditionnelle permettrait d'obtenir le même résultat, cf. la section suivante).

REMARQUE Du point de vue des tableaux, la concaténation des tableaux ne coïncide plus avec la concaténation des GBF. En fait, la concaténation des GBF est *plus élémentaire* : la concaténation des tableaux peut être exprimée par une union plus une translation de GBF. Si A est un tableau $[2, 3]$ et B est un tableau $[3, 3]$, alors on peut considérer les GBF A' et B' associés à A et B et définis sur la forme $\mathit{G2} = \langle i, j \rangle$. Alors, il vient :

$$A \#_{\mathit{tableau}} B \quad \simeq \quad A' \#_{GBF} \,\texttt{translate}(B', i^2)$$

cf. figure 13, où `translate` est l'opérateur de translation d'un GBF introduit dans la section suivante.

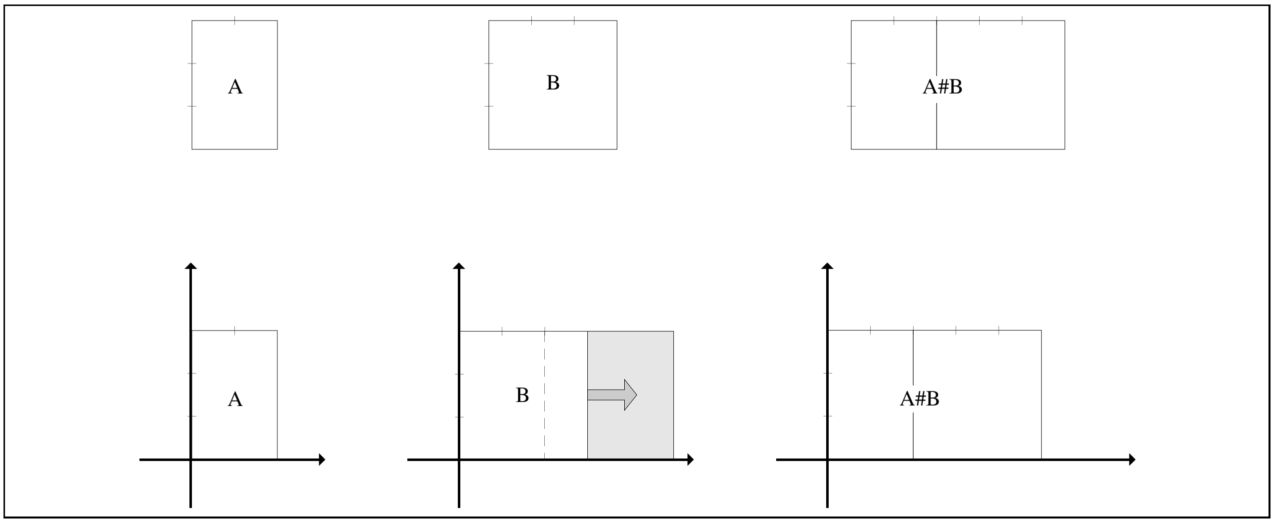


FIG. 13 –: *Union de deux formes après translation afin d'obtenir un résultat similaire à la concaténation de deux tableaux. La première ligne représente les tableaux à deux dimensions A et B et leur concaténation. La seconde ligne représente ces tableaux comme des régions positionnées dans $\mathbb{Z}^2$. La concaténation se traduit par une translation de B de la largeur de A, suivie d'une « union » de régions de $\mathbb{Z}^2$.*

V.5.3 Translation d'un GBF

Soit $g[F]$ avec $F = \mathit{Forme}(G, S)$, alors pour tout $w \in G$, on définit $g.w$, *translation à droite de g par w* tel que :

$$\forall p \in G, \quad (g.w)(p) \quad = \quad g(p.w)$$

Dans le cas où F est une forme non-abélienne, on définit la *translation à gauche* par w de manière symétrique.

L'exemple que nous avons présenté dans la section précédente s'écrit donc finalement :

$$A \#_{tableau} B \quad \simeq \quad A' \#_{GBF} B'.i^2$$

V.5.4 Plongement d'un GBF

Soit $g[F]$ avec $F = \mathit{Forme}(G, S)$ et $C = x.H$ un coset de G. Alors $(g|C)$ est un GBF de forme H défini par :

$$\forall p \in H, \quad (g|C)(p) \quad = \quad g(x.p)$$

Si C est le coset $H.x$, alors :

$$\forall p \in H, \quad (g|C)(p) \quad = \quad g(p.x)$$

Les deux notions coïncident si C est abélien ou plus généralement si H est un sous-groupe normal.

Cette opération permet de définir un champ comme sous-champ d'un champ donné. Par exemple, on peut ainsi accéder à la valeur d'un point unique x. Ainsi, $g|x.\langle\rangle$ est un champ composé d'un unique point, dont la forme est réduite à $\mathit{Forme}(\{e\}, \emptyset)$ et dont la valeur en e est égale à $g(x)$.

V.5.5 Extension d'une fonction scalaire

Une fonction scalaire est une fonction qui opère dans VAL. On ne considère pas de champ imbriqué, on peut donc étendre implicitement (cf. section II.3.3.1) une fonction scalaire sur des champs de même forme.

Soit $f : \mathrm{VAL}^d \to \mathrm{VAL}$ une fonction d'arité d, et $g_1[G], \ldots, g_d[G]$ des GBF, alors :

$$\forall p \in G, \quad (f(g_1, \ldots, g_d))(p) \quad = \quad f(g_1(p), \ldots, g_d(p))$$

V.5.6 Réductions

La réduction sur un tableau à n dimensions dans le langage APL est paramétrée par un *axe* [Ive87] : par exemple, une matrice peut être réduite suivant une ligne ou suivant une colonne. La projection du tableau le long d'un axe est encore un tableau de dimension $n-1$, résultat de la réduction. Nous généralisons cette situation de la façon suivante.

Soit H un sous-groupe normal de G : H sera l'axe de la réduction. L'expression $\oplus\backslash H\ f$ dénote la réduction d'un champ $f[G]$ suivant l'axe H par la fonction $\oplus$. On suppose que $\oplus$ est une fonction dyadique commutative et associative :

- La forme de $\oplus\backslash H\ f$ est G/H.
- La valeur de $\oplus\backslash H\ f$ en un point w est la réduction de $\{f(v) \mid v \in w\}$ par $\oplus$ (cet ensemble n'a pas d'ordre canonique, c'est pourquoi il est nécessaire d'imposer la commutativité de $\oplus$).

La figure 14 illustre quelques exemples de réductions sur *G2*. Seul le premier exemple est exprimable en APL. Un point intéressant est que H n'est pas restreint à n'être généré que par un unique générateur. Par exemple, $+\backslash G\ f$ où G est la forme de f calcule la somme globale de tous les éléments dans G.

Si G est infini, cette opération a-t-elle un sens ? Une telle opération peut avoir un sens, par exemple, quand il n'existe qu'un nombre fini de valeurs définies dans le champ. Dans ce cas, effectuer une opération globale de somme signifie réellement « faire la somme de toutes les valeurs définies et ignorer les valeurs indéfinies ». La situation est la même pour un tableau borné dynamiquement : il est potentiellement infini mais la somme des éléments signifie la somme des éléments actuellement définis. En suivant cette approche, le programmeur travaille implicitement sur une partie finie d'un champ infini pour ne pas être gêné avec la

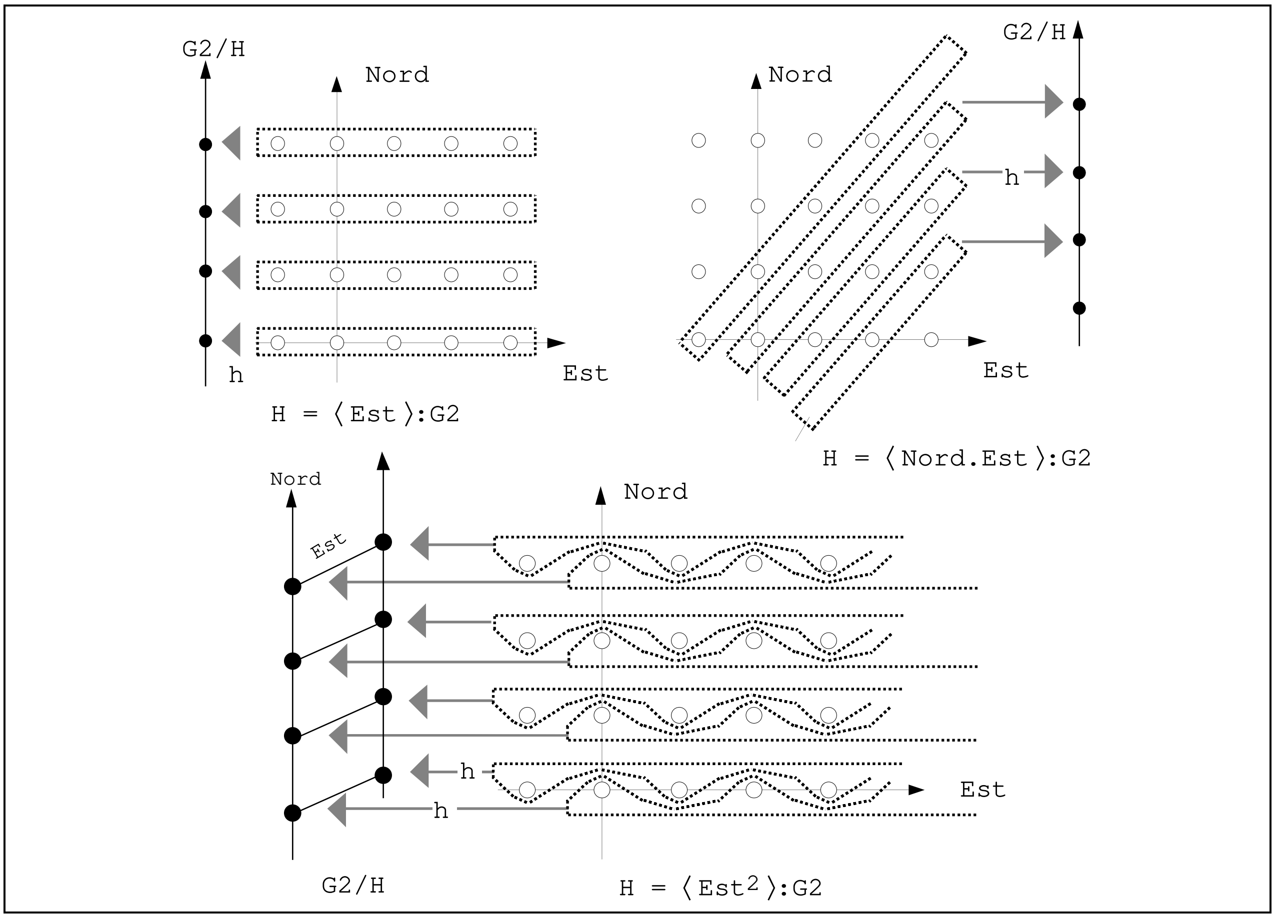


FIG. 14 –: *Trois exemples de réductions sur la forme G2.*

manipulation des bornes ou parce que la topologie de l'application est plus en accord avec la topologie d'un champ infini. Le problème est alors de définir explicitement, ou d'inférer au moment de la compilation, ou de maintenir à l'exécution, une région bornée en dehors de laquelle toutes les valeurs sont nécessairement indéfinies (i.e. égales à `nil`).

Un autre sens possible d'une réduction infinie est l'existence du résultat du point de vue mathématique (par exemple, lors du calcul d'une série convergente). Nous pouvons dans ce cas calculer une approximation du résultat si nous savons que, en dehors d'une certaine région bornée, les valeurs qui doivent être réduites sont évanescentes.

Cependant, afin de rester dans le cadre de situations décidables, on demandera que l'argument d'une réduction soit explicitement le résultat d'un champ abélien explicitement restreint, par exemple :

$$\oplus\backslash H\ (g\ \texttt{on}\ldots\texttt{where}\ldots)$$

Les opérations de *scan* [Ble89] semblent plus problématiques à définir. Par exemple, le scan suivant un axe H avec plus d'un générateur n'a pas grand sens.

V.6 Définition récursive d'un GBF

Les opérations présentées dans la section précédente permettent de calculer des GBF par composition des opérateurs. Nous aimerions faire plus et, comme en 81/2 pour les tableaux, permettre des définitions récursives. Cependant, et c'est une des distinctions entre les GBF et les fonctions, nous ne permettons pas n'importe quelle expression récursive : la valeur d'un point ne peut dépendre que des points voisins.

V.6.1 Définition d'un GBF respectant le voisinage

Soit h une fonction scalaire et $g[F]$ un GBF tel que $F = \mathit{Forme}(G, \{s_1, \ldots, s_n\})$. Si *$g$ respecte le voisinage élémentaire* défini par F, alors la valeur de g en un point p ne dépend que de la valeur de g aux points voisins de p. Autrement dit, on peut établir une relation du type

$$\forall p \in G, \quad g(p) \quad = \quad h(g(p.s_1), \ldots, g(p.s_n))$$

où h représente la dépendance fonctionnelle entre la valeur en un point et la valeur des voisins du point. Cette relation doit être vraie pour tous les points p. De ce fait, on la rend implicite et on écrit :

$$g \quad = \quad h(g.s_1, \ldots, g.s_n)$$

en faisant de cette équation une équation sur les champs et non pas sur les valeurs des champs. Les générateurs $s_1, \ldots, s_n$ ne sont pas toujours suffisants pour permettre l'inférence du domaine de g (par exemple, si les noms des générateurs n'apparaissent pas, comme dans l'équation $g = 0$). Aussi, nous écrivons :

$$g[F] \quad = \quad h(g.s_1, \ldots, g.s_n)$$

pour le champ g de forme F.

Remarque On ne considère que les définitions récursives qui prennent cette forme. En particulier, on ne permet pas n'importe quelle expression sur les GBF dans une définition récursive.

Avec ces conventions, un programme possible pour un champ sur une ligne uni-dimensionnelle où les valeurs progressent par pas de 1 entre deux voisins est :

$$\begin{array}{lcl} \mathit{G1} & = & \langle \mathit{gauche} \rangle \\ \mathit{iota}[\mathit{G1}] & = & 1 + \mathit{iota.gauche} \end{array}$$

Nous avons évidemment besoin de spécifier la valeur de *iota* en un certain point. De façon plus générale, nous avons besoin de définir la partition d'une forme et définir le champ en donnant une équation pour chaque élément de la partition.

V.6.2 Définition quantifiée par un coset

La remarque précédente implique que chaque élément de la partition peut être vu comme une forme en elle même. Nous pourrions utiliser des sous-groupes du groupe initial pour partitionner le domaine initial mais cela serait trop restrictif. Nous allons utiliser des *cosets.*

Un coset $g.H = \{g.h, h \in H\}$ est la «translation» par g du sous-groupe H. Dans le cas d'un groupe non-abélien, on distingue le coset droit $g.H$ et le coset gauche $H.g$. Pour spécifier un coset, nous donnons le mot g et le sous-groupe H. La notation $\{g_1, g_2, \ldots, g_p\} : G$ définit un sous-groupe de G généré par $\{g_1, g_2, \ldots, g_p\}$ (les g_i sont des mots de G). Il n'y a pas d'équation particulière liant les générateurs du sous-groupe mais ils sont sujets aux équations du (sur-)groupe, si elles s'appliquent. En reprenant l'exemple de *iota*, nous écrivons :

$$\begin{array}{lcll}
\mathit{G1} & = & \langle \mathit{gauche} \rangle & \\
\mathit{init} & = & e.(\langle\rangle : \mathit{G1}) & (1) \\
\mathit{iota@init} & = & 0 & (2) \\
\mathit{iota}[\mathit{G1}] & = & 1 + \mathit{iota.gauche} & (3)
\end{array}$$

L'équation (1) définit le coset $\{e\}$ car $\langle\rangle : \mathit{G1}$ est réduit à $\{e\}$ par convention (et $e.e = e$). L'équation (2) spécifie la valeur 0 pour chaque point du coset et l'équation (3) est valide pour tous les autres points.

Nous disons que l'équation (2) est *quantifiée* sur *init* et que (3) est la définition générale de *iota*.

REMARQUE On aurait pu écrire :

$$\mathit{iota}@\langle\rangle \quad = \quad 0$$

car un interprète ou un compilateur performant sait qu'après la signe de quantification @, un coset ou bien un sous-groupe est attendu. En effet, un sous-groupe H de G correspond au coset $e.H$ de G. La notation $\langle\rangle$ abrège alors $\langle\rangle : \mathit{G1}$ car on peut le déduire de $\mathit{iota}[\mathit{G1}]$.

V.6.3 Partie de forme bien formée

Dans l'exemple *iota*, *init* est inclus dans *G1* ce qui implique que les équations (2) et (3) s'appliquent toutes deux pour définir la valeur des points de *init*. Plus généralement, pour lever l'ambiguïté, nous assumons que l'équation valide est celle qui est définie sur le plus petit domaine. Les domaines sont ordonnés par inclusion et représentent un ordre partiel. Par exemple, si on a :

$$\begin{array}{lcl}
g@A & = & \ldots \\
g@B & = & \ldots
\end{array}$$

avec $A \cap B \neq \emptyset$, on suppose qu'un domaine $C = A \cap B$ a été défini et qu'on a donc aussi une équation :

$$g@C \quad = \quad \ldots$$

C'est toujours possible car l'intersection de deux cosets est soit vide soit un coset. On notera que l'ensemble des points où la définition générale $g = \ldots$ s'applique n'est pas un coset mais le complément d'une union de cosets.

V.6.4 Sémantique dénotationelle des équations sur les GBF

Dans la mesure où un champ peut être vu comme une fonction sur une forme, la sémantique d'un système d'équations de champs est la même que la sémantique des définitions récursives de fonctions en sémantique dénotationelle [vL90].

L'approche est classique mais rappelons là pour mémoire. Pour se placer dans le cadre de la sémantique dénotationelle, on ajoute à l'ensemble VAL un élément $\perp$ afin d'en faire un domaine[1] plat [vL90]. On considère alors l'ensemble $\mathcal{F}$ des fonctions sur ce domaine, qu'on munit de l'ordre de SCOTT :

$$f \sqsubseteq g \;\Leftrightarrow\; \forall x, f(x) \sqsubseteq g(x)$$

1. Dans ce paragraphe, le mot *domaine* a le sens qu'on lui donne en sémantique dénotationelle, et ne désigne pas seulement un sous ensemble dans un groupe.

Avec cet ordre, $\mathcal{F}$ est un domaine. L'expression récursive $g[F] = \varphi(g)$ d'un GBF définit un opérateur *continu* φ sur $\mathcal{F}$. En effet, φ est continu car l'expression récursive d'un GBF est une composition d'opérateurs continus. Par exemple,

$$f \mapsto \lambda x.f(x.g)$$

est un opérateur continu. De même, si C est un sous-ensemble d'index tel qu'on sache tester [1] si $x \in C$, alors

$$f \mapsto \lambda x.(\texttt{if}\, x \in C\, \texttt{then}\, f(x)\, \texttt{else}\, v)$$

est un opérateur continu. Autrement dit, on peut utiliser translation et définition quantifiée pour l'expression des GBF récursifs. Mais remarquons que l'opérateur # *n'est pas continu* et nous ne le permettons pas dans une définition récursive de GBF.

Par suite, résoudre des définitions de champs sur $\mathcal{F}$ consiste à calculer le point fixe d'un opérateur continu. Le formalisme de la théorie dénotationelle nous assure de l'existence d'une solution et du calcul de la plus petite solution comme résultat d'une itération [vL90] (la solution est la limite de $\varphi^n(\lambda x.\ \perp)$ quand n tend vers l'infini).

V.6.5 Exemple de résolution d'une équation : la définition récursive de *iota*

Nous reprenons l'exemple de la section V.6.2. Nous pouvons montrer que les équations (2) et (3) définissent la fonction :

$$\texttt{letrec}\ iota(gauche^n) \quad = \quad \texttt{if}\, n == 0\, \texttt{then}\, 0\, \texttt{else}\, 1 + iota(gauche^{n+1})$$

sur *G1*. Cette fonction a une valeur définie pour les points $\{gauche^n, n \leq 0\}$ et une valeur indéfinie $\perp$ pour les autres points (le calcul boucle). Le champ *iota* est illustré par la figure 15).

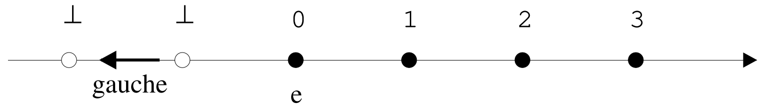


FIG. 15 –: *Le calcul de iota par un GBF.*

Si on essaie de calculer la valeur de *iota* en $gauche^{-2}$, alors :

$$iota(gauche^{-2}) \quad = \quad 1 + iota.gauche(gauche^{-2})$$

car $gauche^{-2} \notin init$. Or :

$$\begin{aligned} iota.gauche(gauche^{-2}) &= iota(gauche^{-2}.gauche) \\ &= iota(gauche^{-1}) \end{aligned}$$

Mais $iota(gauche^{-1}) = 1 + iota.gauche(gauche^{-1})$ car $gauche^{-1} \notin init$. De plus :

$$\begin{aligned} iota.gauche(gauche^{-1}) &= iota(gauche^{-1}.gauche) \\ &= iota(e) \end{aligned}$$

Comme $e \in init$, nous avons $iota(e) = 0$ et par conséquent $iota(gauche^{-1}) = 1{+}0 = 1$ et $iota(gauche^{-2}) = 2$. Par contre, pour $w = gauche^p$ avec $p > 1$, le calcul va boucler car $iota(gauche^p)$ va demander le calcul de $iota(gauche^{p+1})$.

V.6.6 Puissance d'expression des définitions récursives de GBF

Une question immédiate est de savoir si l'itération au point fixe permettant de résoudre une équation récursive sur les GBF converge en un nombre fini d'itérations. Pour des fonctions quelconques, résoudre ce

1. Autrement dit, $(x \in C)$ est une fonction totale des index dans les booléens.

problème est équivalent au problème de l'arrêt d'une machine de TURING, mais nous nous sommes restreints, ici, aux champs basés sur les groupes. Néanmoins, l'expressivité des champs basés sur les groupes est suffisante pour rencontrer le même problème : supposons un champ défini de la façon suivante :

$$g[E] = f(g.a, g.b, \dots)$$

les points accédés pour le calcul de la valeur de p sont : $p.a, p.b, \dots$. En fait, si le calcul de la valeur d'un champ en un point p dépend de lui-même, l'itération au point fixe ne peut évidemment pas converger. Nous nous trouvons donc face au problème de décider si $p.a = p$, $p.b = p$, $p.a.b = p$, ... En d'autres termes, nous devons décider si deux mots dans une présentation finie représentent le même élément du groupe. Ce problème est connu sous le nom de *problème du mot* pour les groupes et n'est pas décidable en général, mais il est décidable pour des présentation finies de groupes abéliens, pour les groupes libres et pour un grand nombre d'autres groupes intéressants.

V.7 Éléments d'implémentation

Par souci de simplicité et de clarté, nous allons prendre pour convention que les définitions de champs ont la forme suivante :

$$\begin{array}{ll} g@C1 & = c_1 \\ & \dots \\ g@Cn & = c_n \\ g[G] & = h(g.s_1, g.s_2, \dots, g.s_p) \end{array}$$

où les Ci sont des cosets, les c_i des constantes et h une extension d'une fonction scalaire quelconque. On nomme l'ensemble $\mathcal{D}g = \{s_1, \dots, s_p\}$ l'ensemble des dépendances de g.

On suppose qu'il existe un mécanisme pour ordonner les cosets et pour établir qu'un mot $w \in G$ appartient à un coset. On suppose de plus que l'on dispose d'un mécanisme pour décider de l'égalité entre deux mots. Par exemple, ces mécanismes existent pour les groupes libres et les groupes abéliens. Rappelons qu'il n'existe pas d'algorithme général de résolution du problème du mot pour les groupes non-abéliens. Nous proposons donc que les formes non-abéliennes permises fassent partie d'une librairie équipée des différents mécanismes nécessaires. Un travail futur à effectuer est de développer des familles utiles de formes non abéliennes.

Avec ces restrictions, une première stratégie pour implémenter des champs est l'utilisation de fonctions mémoïsées. Un champ $g[G]$ est stocké comme un dictionnaire où on associe à une entrée $w \in G$ une valeur $g(w)$. Si la valeur $g(w)$ de w est requise, on vérifie d'abord si w se trouve dans le dictionnaire : cela est rendu possible grâce au mécanisme fourni pour tester l'égalité de deux mots. Si w ne se trouve pas dans le dictionnaire, nous devons alors décider quelle est la définition qui s'applique, c'est-à-dire si w appartient ou n'appartient pas à un coset Ci. Dans le premier cas (w appartient à un Ci), nous avons fini le calcul et nous pouvons retourner c_i en stockant le couple (w, c_i) dans le dictionnaire. Dans le second cas (w n'appartient à aucun Ci), nous devons calculer la valeur de g aux points $w.s_1, \dots, w.s_p$ en appliquant récursivement ce processus de calcul. Puis on combine les résultats par h, on stocke le résultat dans le dictionnaire et on le renvoie. La figure 16 illustre les différentes phases du calcul d'un GBF.

V.7.1 Problèmes à résoudre sur les groupes

Les phases 1) et 2) de l'algorithme décrit par la figure 16, supposent que l'on dispose de mécanismes permettant :

1. de résoudre le problème du mot, c'est-à-dire de décider de l'égalité de deux mots (il faut en effet décider si la clé w est présente dans le dictionnaire ou non).

2. de tester l'appartenance à un coset (ainsi que l'ordre d'inclusion des cosets).

Pour le premier problème, de nombreuses familles de groupes, par exemple les groupes abéliens ou les groupes libres, ont un problème du mot décidable. Dans la section VI.1, nous développons les outils théoriques

```
Pour un w ∈ G ou bien pour les w ∈ R ⊂ G

1. Si w ∈ Dico_G

        alors retourner Dico_G(w)
        sinon (w ∉ Dico_G)
         2. si w ∈ Ci
                alors Dico_G(w) ← c_i et retourner c_i
                sinon ((w ∉ Dico_G) ∧ (w ∉ Ci))
                   calculer les v_i = G(w.s_i)
                   calculer v = h(v_1, ..., v_p)
                   Dico_G(w) ← v et retourner v
```

FIG. 16 –: *L'algorithme de calcul d'un GBF récursif. R est un ensemble de points décrit par le langage de formattage proposé en section V.4. La réalisation de cet algorithme nécessite des mécanismes évoqués en section V.7.1.*

permettant d'implémenter complètement les groupes abéliens. Pour le second problème (tester l'appartenance à un coset), il en va de même. Décider d'un ordre d'inclusion des cosets est un problème moins critique : on peut se reposer sur un mécanisme permettant d'ordonner les cosets par inclusion, si un tel mécanisme existe (c'est le cas pour les groupes abéliens), sinon on peut se fier à l'ordre textuel des déclarations. Cette situation est analogue pour le «pattern matching» des fonctions définies par cas.

V.7.2 Optimisations

Nous pouvons être plus efficace si chaque mot w peut être réduit en forme normale $\overline{w}$. Une forme normale peut être calculée pour les groupes abéliens (la forme normale de SMITH) ainsi que pour les groupes libres. Dans ce cas, le dictionnaire peut être optimisé en une *table de hash-code* avec une clé $\overline{w}$ pour w.

Dans le cas des groupes abéliens G, nous pouvons améliorer encore l'implémentation en utilisant l'isomorphisme fondamental entre G et un produit de $\mathbb{Z}$-modules (cf. [Coh93, Ili89]). En fait, une fonction sur un $\mathbb{Z}$-module est simplement implémentée comme un vecteur. La difficulté dans ce cas est la manipulation de $\mathbb{Z}^n$ qui correspond à un tableau non borné.

V.7.3 Chemin de dépendances et évaluation dirigée par les données

Le schéma d'évaluation que nous venons de définir (cf. figure 16) correspond à une stratégie d'évaluation *à la demande* : par exemple, pour évaluer $iota(gauche^{-2})$, nous devons calculer $iota(gauche^{-1})$ qui déclenche le calcul de $iota(e)$ qui enfin retourne 0.

Nous pouvons donc associer à tout point $w \in G$ un ensemble $p(w)$ de chemins dirigés correspondant aux points rencontrés pour calculer $g(w)$. Un élément p de $p(w)$ est un mot du sous-groupe généré par $\mathcal{D}g$. L'évaluation de $g(w)$ échoue si un $p \in p(w)$ a une longueur infinie. Deux cas peuvent se présenter :

1. p est cyclique,
2. p a un nombre infini de sommets.

Borner le nombre de sommets visités dans un calcul est similaire à la limite sur la taille de la pile des appels de fonctions. Une analyse statique peut être utilisée pour caractériser les domaines de G dotés de chemins finis (cf. [LC94]). Des conditions suffisantes peuvent aussi être vérifiées au moment de la compilation pour détecter des chemins cycliques. Par exemple, un critère brutal peut être $\mathcal{D}g \cap (\mathcal{D}g)^{-1} = \emptyset$. Ou bien, on peut détecter un chemin cyclique à l'exécution par un mécanisme d'«occur check».

Les outils développés pour l'ordonnancement d'équations récurrentes uniformes peuvent aussi être utilisés pour implémenter une stratégie d'évaluation gouvernée par les données mais une telle stratégie d'évaluation

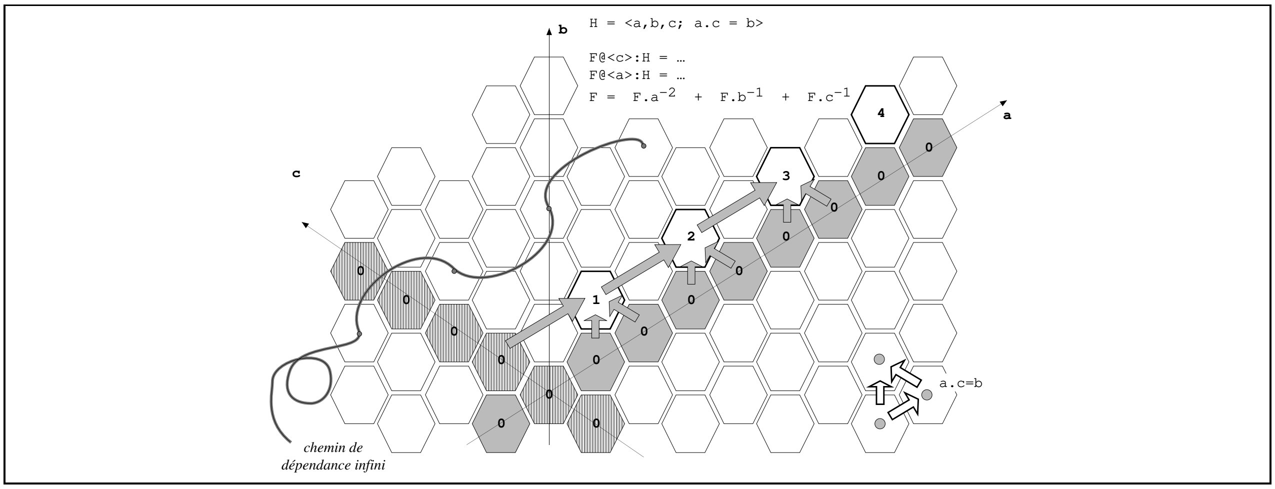


FIG. 17 –: *Cette figure illustre un champ basé sur une grille hexagonale $H = \langle a, b, c\ ;\ b = a.c\rangle$. Le champ F est défini par une expression récursive. Les cosets $\langle a\rangle : H$ et $\langle c\rangle : H$ correspondent aux cas terminaux. L'ensemble de dépendances est $\mathcal{D}F = \{a^{-2}, b^{-1}, c^{-1}\}$. L'entier qui apparaît dans une cellule correspond à l'instant de production de la valeur de la cellule. À l'instant 0, on connaît la valeur des point des cosets. Les grandes flèches correspondent à un mouvement inverse d'un déplacement de $\mathcal{D}F$: ces translations peuvent être utilisées pour calculer de nouveaux points à partir des points connus. Dans cet exemple, à chaque instant on ne peut calculer qu'un seul nouveau point. Les cellules qui ont une valeur différentes de $\perp$ ont une frontière plus épaisse. La courbe infinie qui part d'une cellule en maigre montre le début d'un des chemins infinis de dépendances : ce chemin de dépendances « saute » par dessus le coset et part à l'infini.*

ne cadre pas bien avec une évaluation paresseuse. L'idée est de propager les valeurs à partir des point définis vers les points indéfinis, tel un front de vague. À l'instant 0, la valeur de g est connue pour les points du domaine :

$$Def_0 = C1 \cup C2 \cup \ \ldots \ \cup Cn$$

Alors, à l'instant 1, il est possible de calculer, *en parallèle* les valeurs des point du domaine :

$$L_0 = Def_0.s_1 \cap \ \ldots \ \cap Def_0.s_p$$

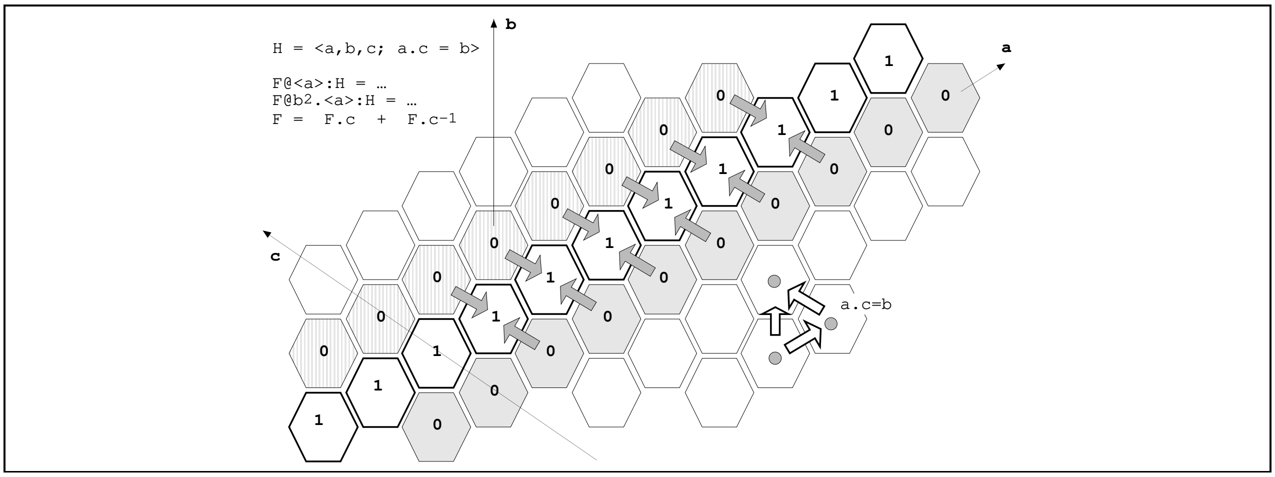


FIG. 18 –: *Ce schéma illustre un autre cas de figure dans le calcul des dépendances : toutes les cellules ayant une valeur différentes de $\perp$ sont valuées à l'instant 1. L'ensemble des dépendances est cette fois $\mathcal{D}F = \{a, a^{-1}\}$. Les cosets $\langle a\rangle : H$ et $c^2.\langle a\rangle : H$ correspondant aux cas terminaux de la récursion sont basés sur le même sous-groupe.*

Ainsi, à l'instant 1, les valeurs de F sont connues pour :

$$Def_1 = Def_0 \cup L_0$$

De façon plus générale, il est possible à l'instant t de connaître les valeurs pour Def_t et de calculer les valeurs de L_t :

$$\begin{array}{ll} Def_t & = Def_{t-1} \cup L_{t-1} \\ L_t & = Def_t.s_1 \cap \ \ldots\ \cap Def_t.s_p \end{array}$$

Les calculs des valeurs des points de L_t sont indépendants et peuvent avoir lieu en parallèle. Les figures 17 et 18 illustrent ces ensembles pour deux exemples de GBF.

Une approche compilée d'une stratégie d'évaluation gouvernée par les données devrait inférer à la compilation l'allure de L_t mais c'est compliqué dans le cas général : par exemple, L_t n'est pas forcément un coset. En fait, cette stratégie engendre plusieurs problèmes. D'abord les cosets définissant les équations ne sont pas forcément distincts. Il faut alors choisir une valeur pour les éléments appartenant à plusieurs cosets. Ensuite, quand il s'agit de calculer la valeur d'un élément précis w, il y a deux nouveaux problèmes :

- Le premier est de savoir si w est calculable. On a vu un procédé itératif permettant de connaître tous les points calculables (en admettant qu'on sache trouver l'intersection des cosets). Cependant, cette méthode n'est pas forcément décidable : si w est calculable alors la méthode répond positivement en temps fini.
- Même si l'élément est calculable, on ne peut pas procéder au calcul sur tous les Def_t puisqu'ils peuvent être infinis. Il faut alors réduire l'ensemble de départ à un sous-ensemble fini, tout en conservant les chances d'atteindre w. Cette opération n'est pas du tout immédiate et se ramène souvent à déterminer les points nécessaires pour atteindre w (mécanisme demand-driven). De plus, les ensembles qu'on manipule ont peu de chances d'avoir des structures de groupes.

Une approche possible serait de trouver une approximation de L_t. Dans le cas de tableaux définis récursivement et qui correspondent à des champs abéliens libres, la méthode d'ordonnancement par hyperplan fait la supposition que l'instant auquel un élément du tableau sera calculé, est donné par une combinaison linéaire des indices du tableau [Tor93]. Autrement dit, les L_t sont approximés par un coset (l'analogue, dans le cas des groupes abélien, d'un hyperplan). Cette voie d'étude reste à explorer.

Mais les raisons discutées ci-dessus (illustrées par la figure 19) nous ont incité à étudier en priorité la stratégie d'évaluation demand-driven.

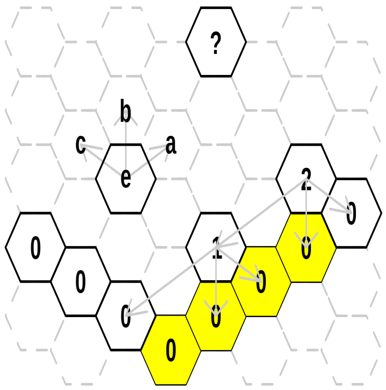


FIG. 19 –: *La figure, qui reprend l'exemple présenté figure 17 visualise un champ basé sur une grille hexagonale $H = \langle a, b, c;\ b = a.c\rangle$. À l'étape 0, nous connaissons la valeur des éléments appartenant aux coset $b^{-3}.\langle a\rangle : H$ et $b^{-2}.\langle c\rangle : H$ (ces éléments sont nommés 0). En supposant que l'équation générale implique $F.a^{-2}$, $F.b^{-1}$ et $F.c^{-1}$, à l'étape 1, nous ne pouvons calculer qu'une seule nouvelle valeur (pour le mot c^{-2}, nommé 1). On peut noter qu'aucune valeur n'est définie pour $a^3.c$, cette cellule appelée "?" n'est jamais atteinte par le processus d'évaluation. Si on réduit l'ensemble d'origine le long de $\langle a\rangle$ à l'ensemble $\{a.b^{-3}, a^2.b^{-3}, a^3.b^{-3}, a^4.b^{-3}\}$ (cellules grisées) alors la valeur de $a^2.c^{-2}$ (marquée 2) ne peut plus être calculée (il manque la valeur de $a^5.b^{-3}$). Cette figure est un graphique* Mathematica *extrait de l'implémentation développée par J.* SOULA *[Sou96].*

Chapitre VI

Implémentation de GBF abéliens

VI.1 Outils théoriques pour l'implémentation des GBF abéliens

À partir des concepts que nous avons développés dans le chapitre précédent, J. SOULA a développé lors d'un stage de DEA [Sou96] les outils permettant l'implémentation des groupes abéliens. Rappelons que l'implémentation de l'algorithme de la section V.7 nécessite la résolution des problèmes suivants :

1. Pouvoir comparer deux mots d'un groupe et déterminer s'ils représentent le même élément dans le groupe est un mécanisme fondamental et nécessaire. Il sert notamment à tester l'appartenance d'un élément à un sous-ensemble.
2. Le choix de l'équation à appliquer implique l'existence d'un test d'appartenance à un coset. Dans le cas des ensembles quelconques, la seule solution possible est d'énumérer l'ensemble pour comparer ces éléments à celui concerné ; malheureusement ce test échoue pour les ensembles infinis. Cependant, bien que les cosets soient infinis, leur structure particulière peut nous amener à trouver des solutions particulières.
3. Comme on l'a dit dans le chapitre précédent, les cosets n'étant pas forcément distincts, il faut un mécanisme de choix en cas d'appartenance à plusieurs cosets.
4. Si l'élément concerné n'appartient à aucun coset, alors il faut appliquer l'équation générale avec tous les problèmes que l'emploi de la récursivité implique.
5. Il est important d'offrir à l'utilisateur la possibilité de construire de nouveaux groupes grâce aux opérations telles que l'intersection, le produit direct, etc. Cela permet une meilleure expressivité des structures. Par exemple, on peut considérer l'espace et le temps comme le groupe produit espace$\times$temps, le groupe quotient permettant de partionner l'espace uniformément. Il apparaît comme construction de forme et comme base du champ résultant de la β-réduction suivant un sous-groupe normal. Le problème est alors de pouvoir manipuler convenablement ces constructions.

L'outil principal permettant d'aborder la résolution de ces problèmes dans le cas abélien est la *forme normale de* SMITH (FNS) qui permet de rendre effectif l'isomorphisme entre un graphe abélien et un produit de $\mathbb{Z}$-modules.

Nous rappelons ci-dessous rapidement quelques résultats puis nous évoquons comment le calcul de la FNS permet de résoudre les problèmes évoqués. Les algorithmes correspondant ont été développés par J. SOULA et implémentés dans le langage Mathematica. On trouvera un exposé détaillé dans [Sou96].

VI.1.1 Un groupe abélien particulier : le $\mathbb{Z}$-module

Un $\mathbb{Z}$-module est un groupe abélien noté $(\mathbb{Z}/n\mathbb{Z}, +)$ avec $n \in \mathbb{N}$. Mathématiquement, c'est le groupe quotient de $\mathbb{Z}$ par le sous-groupe des multiples de n, $n\mathbb{Z}$. Intuitivement c'est un groupe de n éléments dont chacun d'eux correspond à un sous-ensemble de $\mathbb{Z}$ et qui en forme une partition.

Chaque élément est nommé par l'un des entiers qui le constituent : l'élément de $\mathbb{Z}/n\mathbb{Z}$ noté $\tilde{k}$ représente l'ensemble $\{\dots, k-n, k, k+n, k+2n, \dots\}$. La loi d'addition est cohérente avec celle des entiers : $\tilde{p}+\tilde{q} = \widetilde{p+q}$. On peut la représenter graphiquement par un anneau.

On note le cas particulier de $\mathbb{Z}/0\mathbb{Z}$ qui est isomorphe à $\mathbb{Z}$ (et donc infini). Ce $\mathbb{Z}$-module particulier est appelé *module libre*, alors que les $\mathbb{Z}$-modules finis sont appelés *modules de torsion*.

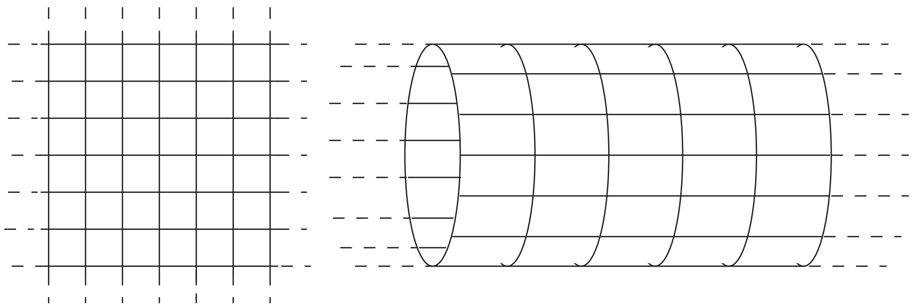


FIG. 1 –: *Deux exemples de $\mathbb{Z}$-modules de dimension 2 illustrant le partage entre module libre et module fini. Le premier, $\mathbb{Z} \times \mathbb{Z}$ représente une grille infinie, le second, $\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$, est un tube infini.*

Les $\mathbb{Z}$-modules de dimension supérieure sont mathématiquement des produits cartésiens de $\mathbb{Z}$-modules de dimension 1 et ils peuvent s'identifier intuitivement à des grilles multi-dimensionnelle particulières (chaque axe correspondant à un module). En effet, les axes de ces grilles sont soit des anneaux (module de torsion), soit infinis (module libre). La figure 1 illustre des exemples de produit de deux $\mathbb{Z}$-modules. Dans la suite, nous étendrons le nom de $\mathbb{Z}$-module à des produits cartésiens de modules de dimension 1.

VI.1.2 Représentation d'un graphe abélien

Soit un groupe abélien G, de générateur $g_1, \dots, g_n$. Toutes les relations $w_1 = w_2$ peuvent se réécrire en $w_1.w_2^{-1} = e$ et donc les relations définies par la présentation finie du groupe G sont équivalentes à une liste de mots de la forme $r_1, \dots, r_m$.

Tout mot $g_{i_1}^{\pm 1}. \ \dots \ .g_{i_p}^{\pm 1}$ peut se réécrire sous la forme $g_1^{\alpha_1}. \ \dots \ .g_n^{\alpha_n}$ en utilisant la commutativité et la règle $g^x.g^y = g^{x+y}$. Soit alors :

$$\begin{array}{lll} \varphi' & : & \{g_1, \dots, g_n\}^* \to \mathbb{Z}^n \\ & & g_{i_1}^{\pm 1}. \ \dots \ .g_{i_p}^{\pm 1} \mapsto (\alpha_1, \dots, \alpha_n) \end{array}$$

grâce à φ', on peut associer à tout r_i un n-uplet de $\mathbb{Z}^n$. On définit alors la matrice des règles R_G par :

$$R_G \quad = \quad [\varphi'(r_1), \dots, \varphi'(r_m)]$$

où $\varphi(r_i)$ représente la i^{e} colonne de R_G qui est une matrice $n \times m$. R_G représente une application linéaire de $\mathbb{Z}^m$ dans $\mathbb{Z}^n$ muni de leur base canonique. Tout vecteur appartenant à l'image de R_G est une combinaison linéaire des r_i et donc représente l'élément neutre.

Par ailleurs, on peut définir :

$$\begin{array}{lll} \varphi & : & (\mathbb{Z}^n, +) \to (G, .) \\ & & (\alpha_1, \dots, \alpha_n) \mapsto g_1^{\alpha_1}. \ \dots \ .g_n^{\alpha_n} \end{array}$$

qui est un morphisme de groupe. Ce morphisme n'est pas forcément injectif à cause des relations sur G. En fait, le noyau[1] de φ est exactement l'image de R_G. En notant $\mathtt{Ker}_\varphi$ le noyau de φ et $\mathtt{Im}_{R_G}$ l'image de R_G, on a $\mathtt{Im}_{R_G} = \mathtt{Ker}_\varphi$.

1. Le noyau d'un morphisme f, noté $\mathtt{Ker}_f$, est l'ensemble des éléments dont l'image est l'élément neutre.

Tester l'égalité des éléments de G représentés par les mots w_1 et w_2 revient à tester que l'élément de G associé à $w_1.w_2^{-1}$ est e, c'est-à-dire que :

$$\varphi'(w_1.w_2^{-1}) \quad \in \quad \varphi^{-1}(e) = \mathtt{Im}_{R_G}$$

Pour décider si un élément w de G appartient au coset $u.(H:G)$, on remarque que u est un représentant de la classe $u.(H:G)$ de G/H et que w est un représentant de $w.(H:G)$. Deux classes de G/H sont soit distinctes soit confondues et par conséquent $w \in u.(H:G)$ si et seulement si $u =_{G/H} w$. L'appartenance à un coset revient donc à tester l'égalité dans un groupe quotient (dont on connait la présentation, cf. section V.3.5).

Le mécanisme fondamental dont nous avons besoin est de savoir tester l'appartenance à l'image de R_G (ou de $R_{G/H}$ dans le cas de l'appartenance à un coset). Or, ce n'est pas chose aisée. On va donc effectuer un changement de base de $\mathbb{Z}^n$ pour obtenir une forme exploitable de $\mathtt{Im}_{R_G}$. C'est le but de la FNS.

VI.1.3 Calcul de la forme normale de Smith

On cherche L et K tel que, étant donné une matrice R :

$$L.R.K \quad = \quad D$$

avec D une matrice diagonale, L et K des matrices à coefficients dans $\mathbb{Z}$ et dont les inverses sont aussi à coefficients dans $\mathbb{Z}$. La matrice D est de la forme

$$\begin{bmatrix} d_1 & & & \\ & \ddots & & \\ & & d_n & \end{bmatrix} \quad \text{ou} \quad \begin{bmatrix} d_1 & & \\ & \ddots & \\ & & d_n \\ & & \end{bmatrix}$$

Les termes de la diagonale sont positifs et vérifient :

$$\begin{aligned} d_i = 0 \quad &\Rightarrow \quad d_{j \geq i} = 0 \\ d_i \neq 0 \quad &\Rightarrow \quad \frac{d_{i+1}}{d_i} \in \mathbb{Z} \end{aligned}$$

Avec ces conditions, la matrice D est unique.

Le calcul de A et K peut théoriquement poser des problèmes de complexité, qui sont cependant bien connus et maîtrisés [KB79, CC82, Ili89, HHR93]. La matrice L étant inversible, on peut l'interpréter comme la matrice d'un changement de base. Dans cette nouvelle base, l'image de R est engendrée par les vecteurs de D. Autrement dit :

$$L.\,\mathtt{Im}_R \quad = \quad (d_1\ \mathbb{Z}, \ldots, d_q\ \mathbb{Z})$$

De ce fait, tester l'égalité de w avec e est équivalent à tester l'appartenance suivante :

$$L.\varphi'(w) \quad \in \quad (d_1\ \mathbb{Z}, \ldots, d_q\ \mathbb{Z})$$

ce qui est simple.

REMARQUE Le produit de $\mathbb{Z}$-module isormorphe à G est $\mathbb{Z}/d_1\mathbb{Z} \times \cdots \times \mathbb{Z}/d_n\mathbb{Z}$.

VI.2 Implémentation en Mathematica

J. SOULA a affiné les outils précédents et les a implémenté à l'aide du langage Mathematica, à partir du package de D. JABON effectuant le calcul de la FNS d'une matrice entière. Il a aussi développé un début d'environnement permettant la définition de GBF et le calcul d'une définition récursive suivant l'algorithme que nous avons défini en section V.7. Par exemple, on peut définir :

$$
\begin{array}{lcl}
G & = & \langle a, b\,;\ a^8 = e, b^8 = e\rangle \\
H1 & = & \langle a\, b\rangle : G \\
H2 & = & \langle a^2\, b^2\rangle : G \\
H3 & = & \langle b\rangle : G \\
H4 & = & \langle\rangle : G \\
C1 & = & a^2.H1 \\
C2 & = & a\, b^{-1}.H2 \\
C3 & = & a^3.H3 \\
C4 & = & a^2.H4 \\
F@C1 & = & 1 \\
F@C2 & = & 2 \\
F@C3 & = & 3 \\
F@C4 & = & 4 \\
F & = & F.a^{-1} + F.b^{-1}
\end{array}
$$

Le calcul de ce GBF peut se visualiser par un objet graphique, et la figure 2, extraite de [Sou96] illustre le résultat.

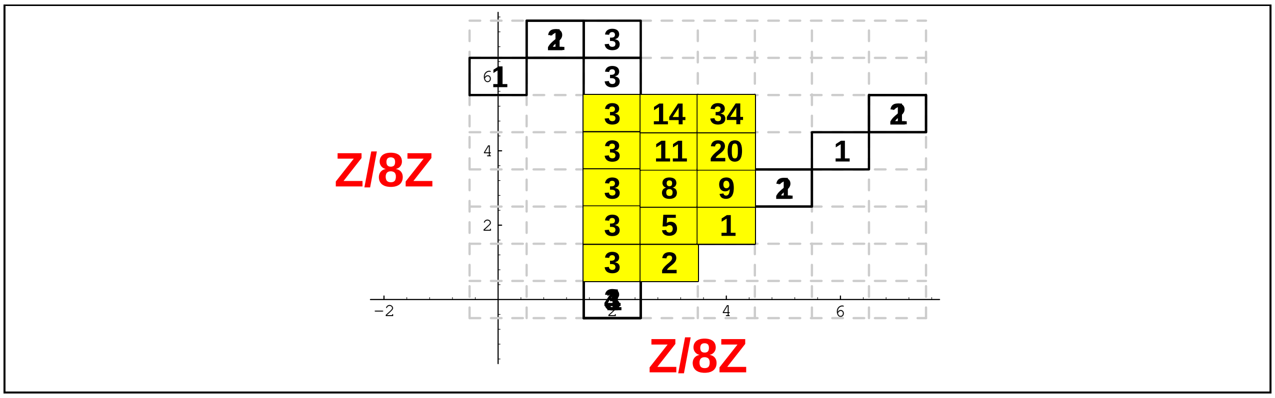


FIG. 2 –: *Cette figure est un graphique* Mathematica *obtenu par la fonction* `display`*. Il représente l'index* `g` *du champ* `f`*. Les valeurs inscrites dans les cellules sont les valeurs déja calculées du champ en ces mêmes points. On voit notamment les cosets* `C1` *à* `C4` *dont les valeurs vont respectivement de 1 à 4. La cellule labélée par 34 représente le mot* a^4b^5 *(on vérifie que 34 est bien la réponse prévue par la sémantique). Les cellules grisées sont celles qui ont été nécessaires lors de calculs récursif de* a^4b^5*.*

Chapitre VII

Exemples de GBF

Dans ce chapitre, nous complétons les exemples énumérés dans [Mic95] afin de montrer l'utilité et la puissance d'expression des concepts que nous avons développés au chapitre V :

- Nous commençons par un exemple artificiel qui illustre la décomposition du calcul par des cosets.
- Nous poursuivons par deux exemples de résolution d'une équation aux dérivées partielles.
- Nous évoquons ensuite un problème de simulation d'un écoulement de fluide par une méthode de gaz sur réseaux.
- Enfin, nous revenons à l'exemple de l'ammonite, pour faire croître une spirale et une ammonite «moins discrétisée», c'est-à-dire un peu moins carrée.

VII.1 Exemple de combinaison de sous-champs

Nous donnons maintenant un exemple arbitraire qui correspond à un champ F sur un cylindre. Nous utilisons le produit libre «$*$» de deux groupes qui est défini comme l'union de leurs présentations respectives et l'identification de leurs éléments identités.

$$\begin{array}{lcl} \mathit{D1}(n) & = & \langle a\ ;\ a^n = e \rangle \\ \mathit{G1} & = & \langle \mathit{gauche}, \mathit{droit}\ ;\ \mathit{gauche} = \mathit{droit}^{-1} \rangle \\ \mathit{Cyl}(n) & = & \mathit{D1}(n) * \mathit{G1} \end{array}$$

$$\begin{array}{lcl} F@\langle\rangle & = & 1 \qquad (1) \\ F@\mathit{D1}(4) & = & 1 \qquad (2) \\ (F@\mathit{G1})\ \texttt{as}\ L & = & L.\mathit{gauche} + 1 \qquad (3) \\ F[\mathit{Cyl}(4)] & = & F.\mathit{gauche} + F.a \qquad (4) \end{array}$$

Cet exemple introduit la notion de forme paramétrée, e.g. $\mathit{D1}(n)$ et utilise un nom local (la construction $\ldots$ `as` L) pour dénoter un sous-champ. Un groupe H dénote implicitement le coset $e.H$ si nécessaire et les éléments d'un produit sont implicitement sous-groupes du produit. Ainsi, $\mathit{D1}(4)$ dans l'équation (2) signifie réellement $e.(\mathit{D1}(4) : \mathit{Cyl}(4))$.

Dans la définition de F, nous avons $e.\mathit{G1} \cap e.\mathit{D1}(4) = e.(\langle\rangle : \mathit{Cyl}(4)) = \{e\}$, les formes sont donc bien définies (et dans l'équation (1), $\langle\rangle$ dénote $e.(\langle\rangle : \mathit{Cyl}(4))$). La figure 1 permet de visualiser la partition correspondante.

L'intérêt de cet exemple est de montrer que la valeur sur un coset peut aussi se définir par une expression récursive (et pas seulement par une constante). Il suffit de modifier légèrement l'algorithme de calcul donné figure 16 : au lieu de retourner c_i si $w \in Ci$, il suffit de relancer le processus de calcul.

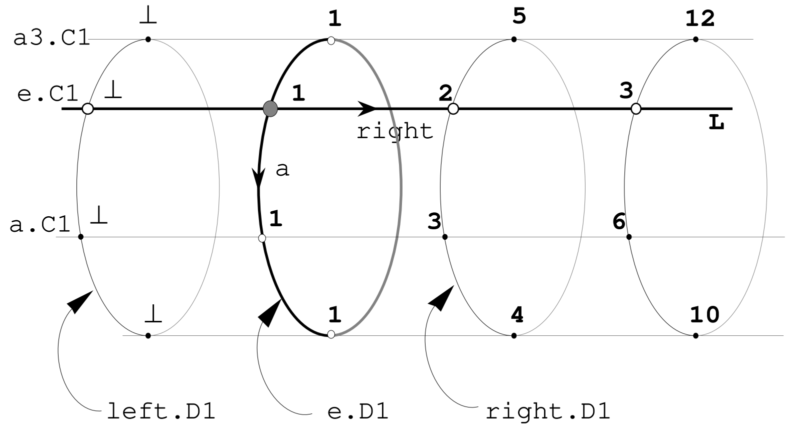


FIG. 1 –: *Définition d'un champ F sur une partition complexe.*

VII.2 Résolution numérique d'une équation aux dérivées partielles parabolique par une méthode explicite

Nous proposons maintenant un exemple pris dans un domaine plus concret : le problème consiste en la simulation de la diffusion de la chaleur dans une fine barre de métal. Nous détaillons brièvement son traitement numérique, puis nous exprimons la solution dans le cadre des champs basés sur les groupes.

Les deux extrémités de la barre, dont on désire connaître l'évolution de la température, sont maintenues à 0^oC. La solution de l'équation parabolique :

$$\frac{\partial chaleur}{\partial t} = k\frac{\partial^2 chaleur}{\partial x^2} \tag{5}$$

spécifie la température $chaleur(x,t)$ à une distance x d'une extrémité de la barre à un instant t. Une méthode explicite de résolution utilise une approximation aux différences finies de l'équation (5) sur une grille $(X_i = ih, T_j = jk)$ qui discrétise l'espace des variables [Smi66]. Une approximation aux différences finies de l'équation (5) est:

$$\frac{chaleur_{i,t+1} - chaleur_{i,t}}{k} = \frac{chaleur_{i+1,t} - 2\ chaleur_{i,t} + chaleur_{i-1,t}}{h^2}$$

qui peut être réécrit :

$$chaleur_{i,j+1} = r\ chaleur_{i-1,j} + (1-2r)chaleur_{i,j} + r\ chaleur_{i+1,j} \tag{6}$$

où $r = k/h^2$. Cette formule définit la température inconnue $chaleur_{i,j+1}$ au $(i, j+1)^{e}$ point de la grille en termes de la température connue le long de la j^{e} ligne temporelle.

Il ne reste plus qu'à calculer les valeurs pivot inconnues de *chaleur* le long de la première ligne temporelle $T = k$, en termes des bornes connues et des valeurs initiales le long de $T = 0$, puis de progresser le long de la seconde ligne temporelle et ainsi de suite.

En fait, dans l'équation (6), $j+1$ est une astuce pour faire référence à l'instant suivant j (qui est le sens de ∂t) et $i+1$, $i-1$ représentent le codage de l'accès aux voisins de gauche et de droite du point i de la barre (en d'autres termes, le groupe additif sur les entiers est utilisé pour coder le groupe des translations sur une ligne uni-dimensionnelle).

Il est très facile d'implémenter cet exemple dans le formalisme des champs basés sur les groupes. Pour définir le domaine espace-temps, nous utilisons le produit libre de domaines plus simples.

$Temps$	$= \langle passé \rangle$	nous ne pouvons que progresser dans le temps, il n'est donc pas nécessaire de spécifier l'inverse du générateur	(7)
$Barre$	$= \langle gauche, droit\ ;\ gauche = droit^{-1} \rangle$	une barre infinie	(8)

$$EspaceTemps \quad = Temps * Barre \tag{9}$$

On définit les cosets correspondants aux conditions intiales et aux bornes (cf. figure 2) :

$$\begin{array}{lll}
BorneInitialeGauche & = e.(\langle\rangle : EspaceTemps) & \\
BorneInitialeDroite & = droit^n.(\langle\rangle : EspaceTemps) & \\
Initiale & = e.(Barre : EspaceTemps) & \text{la barre au temps 0} \\
BorneGauche & = e.(Temps : EspaceTemps) & \text{la borne gauche le long du temps} \\
BorneDroite & = droit^n.(Temps : EspaceTemps) & \text{la borne droite le long du temps}
\end{array}$$

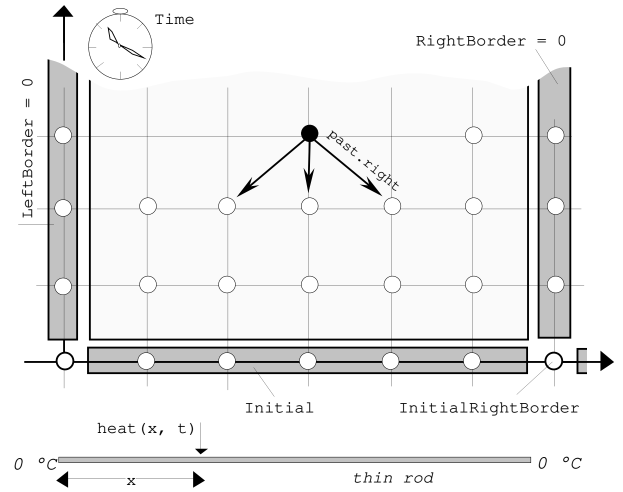


FIG. 2 –: *Résolution numérique explicite de l'équation différentielle partielle parabolique gouvernant la diffusion de la chaleur dans une barre de métal fine et uniforme.*

Nous pouvons maintenant définir le champ *chaleur* :

$$\begin{array}{ll}
chaleur@BorneGaucheInitiale & = 0 \\
chaleur@BorneDroiteInitiale & = 0 \\
chaleur@Initiale & = début \quad \text{une distribution initiale de la température} \\
chaleur@BorneGauche & = 0 \\
chaleur@BorneDroite & = 0 \\
chaleur[EspaceTemps] & = 0.4 * chaleur.passé + 0.3* \\
 & \quad (chaleur.passé.gauche + chaleur.passé.droit)
\end{array}$$

La solution générale de *chaleur* exprime simplement le fait que la valeur courante de la chaleur en un point est égale à une combinaison linéaire de ses voisins à l'instant précédent. Le champ *début* est un paramètre du programme et correspond à une distribution initale de la chaleur. La définition suivante peut être utilisée pour spécifier la distribution symétrique de la chaleur dans la barre :

$$\begin{array}{ll}
rampe@\langle\rangle & = 0 \\
rampe[Barre] & = 1 + rampe.gauche \\
r & = rampe/n \\
début[barre] & = r * (1 - r)
\end{array}$$

On notera l'utilisation d'une forme infinie pour modéliser la discrétisation finie de la barre. En suivant les remarques de la fin de la section précédente, on aurait idéalement des valeurs indéfinies en dehors du domaine implémentant la barre. Ceci peut être obtenu grâce à l'utilisation explicite d'un opérateur de restriction :

$$début[Barre] = r * (1 - r) \;\texttt{on}\; droit^p \;\texttt{where}\; 0 \leq p < n$$

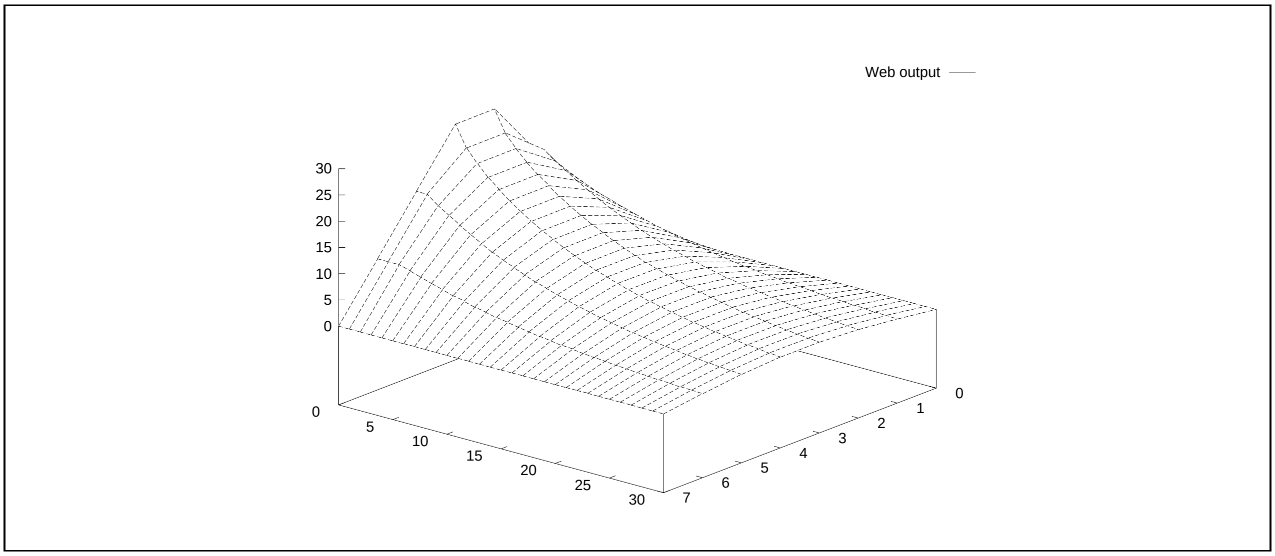


FIG. 3 –: *Résultat de la résolution d'une équation différentielle partielle parabolique.*

VII.3 Relaxation red-black

Dans l'exemple précédent, les topologies plus riches permises par les formes ne sont pas utilisées. Dans cet exemple nous allons utiliser les cosets pour définir et manipuler des sous-domaines de calcul.

La relaxation red-black fonctionne comme toutes les relaxations en calculant une séquence d'approximations de la solution jusqu'à ce que l'erreur soit suffisamment faible. Nous donnons ici la définition du domaine impliqué par la relaxation red-black de GAUSS-SEIDEL. Elle consiste en la mise à jour du champ F. Le domaine de F, une grille à deux dimensions, est divisée en deux ensembles de points (les noirs et les rouges). Nous exprimons cette partition à l'aide de cosets :

$$\begin{array}{ll} Grille & = \langle x, y\ ;\ \rangle \\ SousGrille & = \langle\, x^2, y^2, x.y\,\rangle : Grille \\ Red & = e.SousGrille \\ Black & = x.SousGrille \end{array}$$

L'algorithme commence par le calcul de la valeur des cellules rouges dépendantes des cellules noires voisines ; puis c'est le calcul de la valeur des cellules noires qui est effectué en fonction des cellules rouges précédemment calculées. La formule de la relaxation est par exemple :

$$F_{i,j} \leftarrow F_{i,j} + k((\nabla^2 F)_{i,j} - f_{i,j})$$

où $\nabla^2 F$ désigne le Laplacien discret :

$$(\nabla^2 F)_{i,j} = \frac{F_{i+1,j} + F_{i-1,j} + F_{i,j+1} + F_{i,j-1} - 4F_{i,j}}{h^2}$$

La formule ci-dessus apparaît, par exemple, dans la solution d'une équation aux dérivées partielles de POISSON $\nabla^2 F = f$; la fonction f étant une donnée du problème.

En 81/2, la séquence d'approximation est implémentée comme un stream : F désigne la suite des champs et l'opérateur \$ autorise l'accès à la valeur du stream au top précédent. Le champ F' représente l'étape rouge, le champ F résume les étapes rouge et noir. Le programme correspondant est :

$$\begin{array}{ll} f[Grille] & = \ldots \text{une donnée du problème} \ldots \\ F[Grille] & = \$F' \\ F'[Grille] & = \$F \\ F'@Red & = \$F + k((\$F.x + \$F.x^{-1} + \$F.y + \$F.y^{-1} - 4\$F)/(h^2) - f) \\ F@Black & = \$F' + k((\$F'.x + \$F'.x^{-1} + \$F'.y + \$F'.y^{-1} - 4\$F')/(h^2) - f) \end{array}$$

VII.4 Le modèle des gaz sur réseau

Les modèles des gaz sur réseaux sont une tentative de décrire la cinétique des molécules d'un fluide par un modèle de type automate cellulaire. Afin de ne pas introduire d'invariants parasites, et de respecter les symétries qui apparaissent dans l'expression continue des lois de la mécanique des fluides, on est passé des modèles initiaux de Hardy, Pazzis et Pomeau sur une grille NEWS (dans les années 70), aux modèles à maillage triangulaire (où chaque sommet a 6 voisins) introduit par Frisch, Hasslacher et Pomeau [FHP86]. Nous ne rentrerons pas dans le détail de ces modèles et nous retiendrons que, du point de vue de l'informaticien, il s'agit d'implémenter un « jeu de la vie » sur un réseau hexagonal, avec des règles de transitions un peu particulières (cf. figure 4).

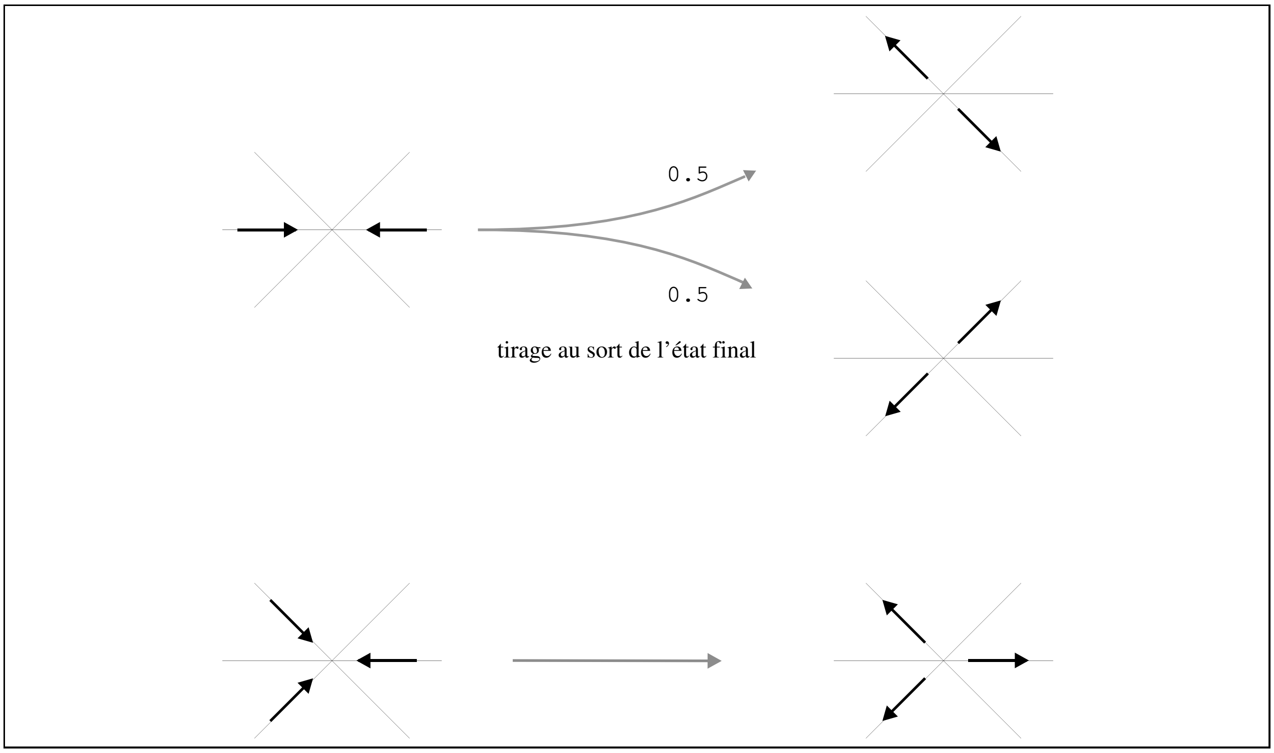
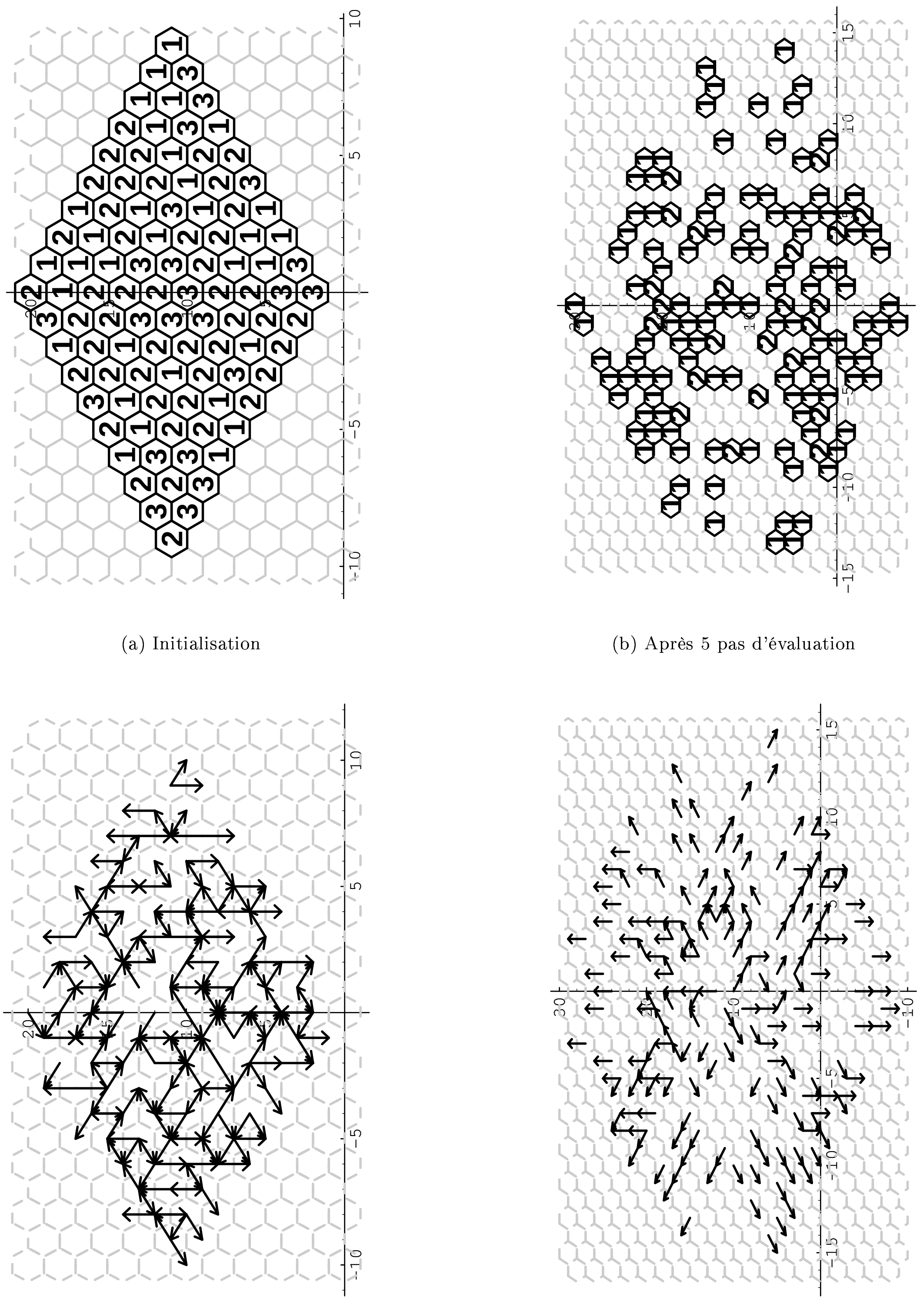


Fig. 4 –: *Un exemple de famille de règles pour une simulation de gaz sur un réseau à maille hexagonale. Les règles sont représentées aux symétries de rotation près et correspondent au modèle FHP.1 [FHP86]. La règle non-déterministe pour les collisions de type 2 (2 particules en jeu) évite de biaiser le comportement physique malgré la topologie discrète du réseau. Toutes les évolutions de type 1 laissent le vecteur vitesse invariant.*

L'intérêt de cette approche c'est qu'elle s'adapte à des problèmes très variés et se décline pour simuler des obstacles, des fluides différents, des réactions chimiques, etc.

Pour calculer les résultats présentés dans la figure 5, nous n'avons pas pu utiliser l'outil développé par J. Soula, car il n'intégre pas l'aspect stream. Mais nous avons pu utiliser les fonctions de visualisation sur grille hexagonale. Nous avons donc construit manuellement et ajouté sous Mathematica les manipulations de champs nécessaires. Cette tâche est simple puisque la forme normale d'un mot de *H2* est facile à calculer : il suffit de mettre chaque mot sous la forme $a^p.c^q$ en remplaçant b par $a.c$. Mathematica permet la manipulation symbolique de mots dans un monoïde commutatif. En utilisant l'égalité précédente comme une règle de réécriture, la normalisation des mots se fait implicitement.

Les règles de la figure 4 ont été codées directement, en maintenant la liste des particules présentes sur un site, plus précisément, en gardant le vecteur vitesse de ces particules. Dans le cadre d'une simulation réelle, il faudrait encoder les configurations possibles afin de ne pas manipuler inutilement ces ensembles de tailles variables (ils ont au plus 6 éléments).



(a) Initialisation

(b) Après 5 pas d'évaluation

(c) Initialisation

(d) Après 5 pas d'évaluation

FIG. 5 –: *Deux exemples d'évolutions avec les mêmes règles mais à partir de deux situations initiales différentes. À l'instant 0, les particules sont concentrées dans une petite « boule ». La pression fait s'expanser la boule de gaz. Les nombres correspondent au nombre de particules qui occupent un site. Les vecteurs correspondent aux vecteurs vitesses des particules. Les lois d'évolution sont celles décrites par la figure 4.*

VII.5 La croissance de l'ammonite hexagonale

Dans cet exemple, nous nous intéressons à nouveau à la croissance de l'ammonite (cf. section III.3, page 22), mais dans le monde *H2*. Comme pour l'exemple précédent, nous avons besoin d'un stream de GBF et donc nous construisons de façon ad hoc les outils nécessaires à la manipulations de champs. L'implémentation de *H2* sous Mathematica est celle esquissé dans l'exemple précédent.

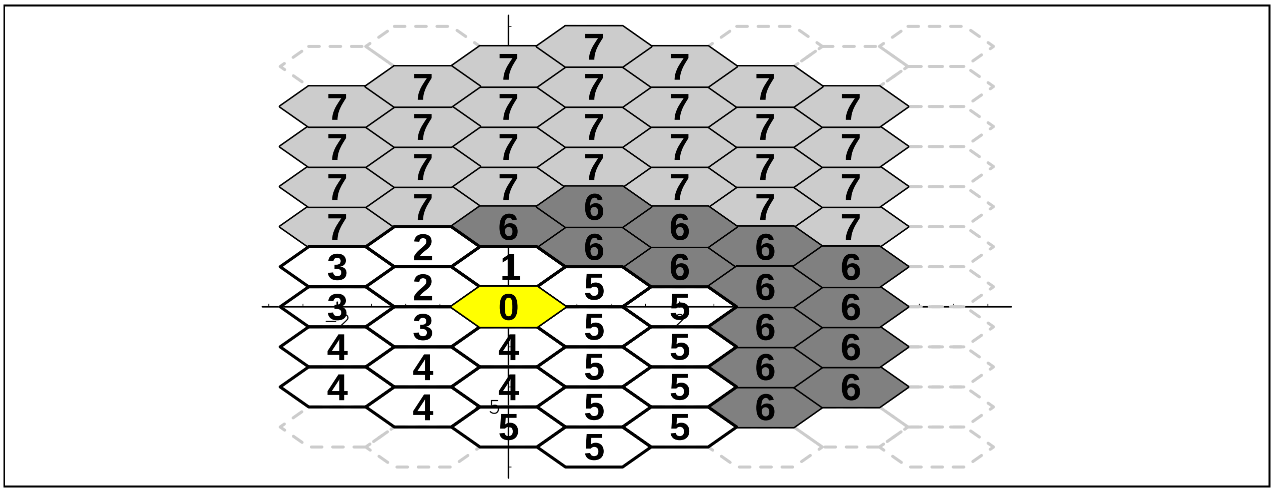


FIG. 6 –: *La croissance de l'ammonite sur un réseau hexagonal. Le numéro d'une cellule correspond au numéro de génération de la cellule. On a figuré en différents niveaux de gris différentes générations.*

Une ammonite dans un monde hexagonal [Tho94, pp 190–192] suit le même principe de croissance, mais au lieu d'être discrétisée par un carré dans une grille NEWS, elle est discrétisée par une region de *H2*. Le programme 81/2 de la modélisation est le suivant :

$$
\begin{array}{lcl}
H2 & = & \langle a,\, b,\, c;\; b = a.c\rangle \\
 & & \\
A@\langle\rangle@0 & = & 0 \\
A[H2] & = & \$A \,\#\, B.d \\
B & = & n\,\texttt{where}\,\$A \\
d & = & \texttt{if}\, dir == 0\, \texttt{then}\, a^{\$la} \\
 & & \texttt{elseif}\, dir == 1\ \texttt{then}\ b^{\$lb} \\
 & & \texttt{elseif}\, dir == 2\ \texttt{then}\ c^{\$lc} \\
 & & \texttt{elseif}\, dir == 3\ \texttt{then}\ a^{-\$la} \\
 & & \texttt{elseif}\, dir == 4\ \texttt{then}\ b^{-\$lb} \\
 & & \texttt{elseif}\, dir == 5\ \texttt{then}\ c^{-\$lc} \\
 & & \\
cpt@0 & = & 0 \\
cpt & = & \$cpt + 1\, \texttt{when}\, \texttt{Clock} \\
dir & = & cpt\, \texttt{mod}\, 6 \\
 & & \\
la@0 & = & 1 \\
lb@0 & = & 1 \\
lc@0 & = & 1 \\
la & = & \texttt{if}\, dir == 0 \,||\, dir == 3\, \texttt{then}\, \$lb + \$lb\, \texttt{else}\, \$lb \\
lb & = & \texttt{if}\, dir == 1 \,||\, dir == 4\, \texttt{then}\, \$lb + \$lb\, \texttt{else}\, \$lb \\
lc & = & \texttt{if}\, dir == 2 \,||\, dir == 5\, \texttt{then}\, \$lb + \$lb\, \texttt{else}\, \$lb
\end{array}
$$

Les variables l_x représentent la largeur de l'ammonite suivant la direction $x \in \{a, b, c\}$. Il y a une double quantification pour A : sur le temps et sur l'espace. La garde $@\langle\rangle@0$ indique que l'équation est valide pour le premier top et pour les éléments de $\langle\rangle : H2$. Le résultat est illustré par la figure 6.

L'écriture du calcul de la direction est complexe à exprimer et fastidieux à rédiger : on pourrait penser à l'abréger en introduisant un générateur correspondant à la rotation désirée :

$$H' = \langle a,\, b,\, c,\, r\,;\; a.c = b,\, r.a = b,\, r.b = c,\, r.c = a^{-1},\, r.a^{-1} = b^{-1},\, r.b^{-1} = c^{-1},\, r.c^{-1} = a\rangle$$

Mais ceci n'est pas correct : de $r.c^{-1} = a$ on obtient $r = a.c = b$ qui n'est pas le résultat désiré. On ne peut pas faire cohabiter la rotation voulue avec les mouvements de translation de *H2* dans le même groupe. Par contre, on pourrait envisager l'action du groupe :

$$\langle r;\; r^6 = e\rangle$$

sur le groupe *H2*, ce qui serait une extension intéressante.

VII.6 La construction d'une spirale dans *H2*

Nous voudrions à présent construire une spirale dans *H2*. Cela se fait en emboîtant, à une position calculée, un segment dont la longueur grandit. On considère trois segments, suivant chacune des directions. La position où on insère ce segment est calculée dans *pos* :

$$
\begin{array}{lcl}
H2 & = & \langle a,\, b,\, c;\; b = a.c\rangle \\
 & & \\
aSeg@0 & = & 0 \\
aSeg[H2] & = & n\,\texttt{where}(\$aSeg)\,\#\,(\$aSeg.a) \\
bSeg@0 & = & 0 \\
bSeg[H2] & = & n\,\texttt{where}(\$bSeg)\,\#\,(\$bSeg.a) \\
cSeg@0 & = & 0 \\
cSeg[H2] & = & n\,\texttt{where}(\$cSeg)\,\#\,(\$cSeg.a) \\
 & & \\
spi@<>@0 & = & 0 \\
spi & = & \$spi\,\#\,seg \\
 & & \\
seg & = & \texttt{if}\; dir == 0\; \texttt{then}\; aSeg.pos \\
 & & \texttt{elseif}\; dir == 1\; \texttt{then}\; bSeg.pos \\
 & & \texttt{elseif}\; dir == 2\; \texttt{then}\; cSeg.pos \\
 & & \texttt{elseif}\; dir == 3\; \texttt{then}\; aSeg.(pos.a^{-n}) \\
 & & \texttt{elseif}\; dir == 4\; \texttt{then}\; bSeg.(pos.b^{-n}) \\
 & & \texttt{elseif}\; dir == 5\; \texttt{then}\; cSeg.(pos.c^{-n}) \\
 & & \\
pos & = & e \\
pos & = & \texttt{if}\; dir == 0\; \texttt{then}\; pos.a^{n} \\
 & & \texttt{elseif}\; dir == 1\; \texttt{then}\; pos.b^{n} \\
 & & \texttt{elseif}\; dir == 2\; \texttt{then}\; pos.c^{n} \\
 & & \texttt{elseif}\; dir == 3\; \texttt{then}\; pos.a^{-n} \\
 & & \texttt{elseif}\; dir == 4\; \texttt{then}\; pos.b^{-n} \\
 & & \texttt{elseif}\; dir == 5\; \texttt{then}\; pos.c^{-n} \\
 & & \\
cpt@0 & = & 0 \\
cpt & = & \$cpt + 1\; \texttt{when}\,\texttt{Clock} \\
dir & = & cpt\,\texttt{mod}\,6
\end{array}
$$

Remarquons que la valeur de *pos* est un élément de *H2*. La croissance de la spirale n'est pas faite de manière locale : ce n'est pas l'activité d'une cellule, dépendant d'un état interne, qui dirige la croissance, comme dans l'automate de LANGTON. *On n'est donc pas du tout dans un modèle de programmation de type automate cellulaire ou programmation plane [Roz90], mais dans un modèle géométrique où l'on manipule des segments de droite, des positions, etc.* Le résultat est illustré par la figure 7.

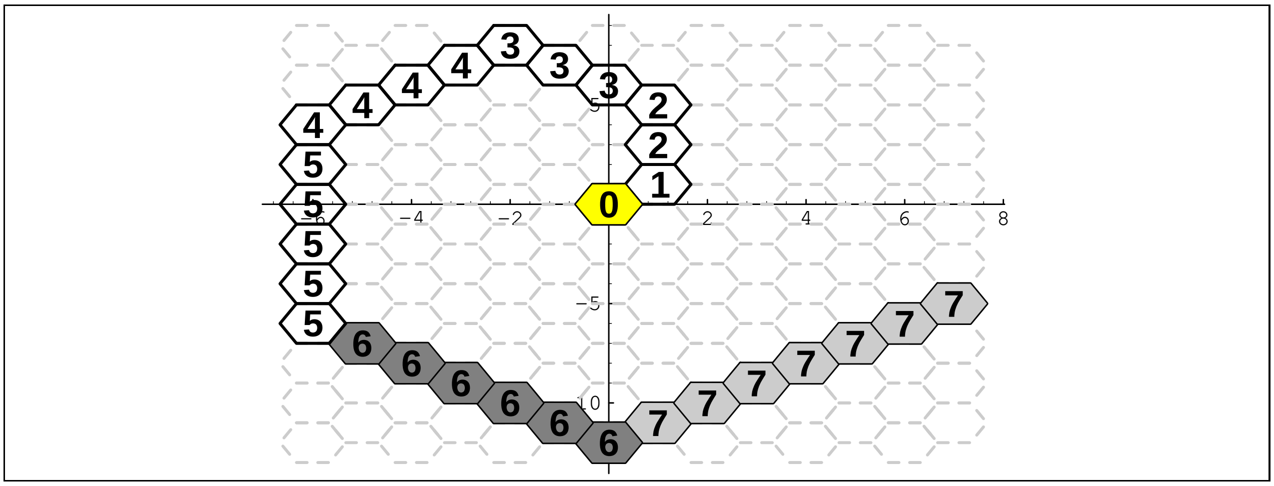


FIG. 7 –: *La construction d'une spirale sur un réseau hexagonal.*

Troisième partie

Amalgames

Chapitre VIII

Représentation des espaces hétérogènes

Nous avons vu dans la partie précédente qu'un espace correspond à un ensemble muni d'une structure de voisinage. On peut donc représenter un espace par un graphe où les sommets sont les points de l'espace, et où un arc relie des points voisins. Une structure de groupe décrit de manière remarquablement compacte le graphe associé à certains espaces : les *espaces homogènes*.

Nous allons à présent nous intéresser à une autre classe d'espaces, les *espaces hétérogènes*. Pour ces espaces, la structure de voisinage ne peut être décrite autrement qu'en énumérant explicitement les voisins de chaque point. Ce sont donc des structures finies et **notre problème est de construire des espaces hétérogènes**.

Nous allons voir que *ces espaces interviennent naturellement pour décrire la structure et les calculs d'un programme*. En fait, *un système* 81/2 *est un espace hétérogène*. Construire un espace hétérogène revient donc à construire un système 81/2 qui est une collection d'équations, comme résultat d'un calcul. Nous donnons le nom d'**amalgame** à ce calcul qui combine des systèmes pour construire d'autre systèmes.

L'intérêt des amalgames pour la simulation des SD est multiple : ils permettent de structurer les programmes de simulation, ils permettent de factoriser et de réutiliser des fragments de simulation, et ils permettent la construction dynamique d'un programme de simulation comme le résultat d'un calcul. Ce dernier point est par exemple absolument nécessaire pour les simulations dont l'espace des états doit être calculé conjointement à l'évolution du système.

Structure de la partie. Cette partie est structurée en quatre chapitres. Ce chapitre-ci expose le lien entre la notion d'espace et la notion de graphe data-flow d'un programme. Un système 81/2 constitue une représentation textuelle d'un graphe data-flow et nous nous posons le problème de la construction de ces graphes. Une solution est esquissée, qui repose sur trois opérateurs : l'opérateur d'amalgamation « { . . . } », l'opérateur de sélection « **.** » et l'opérateur de concaténation « # ». Nous illustrons ces constructions par la structuration et la construction de simulations de SD.

Le chapitre IX compare les amalgames avec d'autres formalismes et mécanismes de programmation. La notion de *nom* est centrale dans le calcul des amalgames et on retrouve une préoccupation similaire dans de nombreux domaines : par exemple la modélisation de l'*héritage* dans les langages à objets, la *délégation* dans les langages à prototypes, les *pages jaunes* dans un environnement distribué, les *applets* dans les programmes construits dynamiquement, les mécanismes de *call-back*... Nous comparons l'approche des amalgames avec diverses approches qui ont été proposées dans ces domaines.

L'objet du chapitre X est de formaliser la notion d'amalgame, ce qui est fait à travers une sémantique opérationnelle du processus de calcul. Le calcul que nous présentons possède deux propriétés importantes : il est déterministe et il préserve une certaine notion de correction.

Enfin, le dernier chapitre de la partie, le chapitre XI, évoque une implémentation de la sémantique en Mathematica et illustre le concept d'amalgame par des exemples. Les exemples donnés ne concernent que les « amalgames purs », c'est-à-dire que ces exemples n'impliquent ni les GBF, ni les streams, ni d'autres constructions 81/2. L'intégration des amalgames (et des GBF) au sein du langage 81/2 est discuté dans la quatrième partie de ce document.

Structure du chapitre. Ce chapitre est structuré en six sections. La première section de ce chapitre fait le lien entre les formes, telles qu'elles ont été définies dans la partie précédente, les graphes des dépendances et les systèmes.

Dans la deuxième section nous faisons le lien entre les systèmes et les graphes data-flow.

La troisième section aborde le problème de la construction des systèmes vus comme des graphes data-flow. Deux solutions sont classiquement utilisées : la construction explicite par combinateur, qui repose sur la notion de fonction, et la construction implicite qui repose sur la notion de nom. Cette deuxième solution est la plus adaptée à un langage déclaratif mais son aspect implicite empêche d'exprimer un système comme le résultat d'un calcul.

Dans la quatrième section, nous proposons une nouvelle approche, les *amalgames*, qui permet la construction dynamique et incrémentielle de systèmes dans le cadre d'un langage déclaratif. Pour cela, nous introduisons trois opérateurs qui permettent de combiner des systèmes : l'opérateur d'amalgamation « $\{\ldots\}$ », l'opérateur de sélection « **.** » et l'opérateur de concaténation « # ». Ces opérateurs étendent aux amalgames les opérations 8½ de même nom qui s'appliquaient aux systèmes.

L'avant-dernière section illustre les bénéfices apportés par les amalgames aux applications de simulation des SD.

Dans la dernière section de ce chapitre nous revenons sur le processus de réduction d'un amalgame et nous montrons certaines difficultés de l'évaluation d'une expression d'amalgame : bien qu'impliquant peu d'opérateurs, les règles de réduction doivent être soigneusement conçues sous peine d'obtenir des réductions indésirables. Cette section est principalement un catalogue des divers phénomènes qu'on peut rencontrer lors de la réduction d'un amalgame : capture de variables, liaisons à l'extérieur d'un terme, réduction tardive d'un terme, etc.

VIII.1 Forme, graphe des dépendances et système

La notion d'espace intervient dans les programmes de simulation des SD, comme un objet naturel de l'application. Mais nous avons évoqué au début du chapitre IV, que cette notion intervient aussi naturellement quand on veut représenter les données et les calculs d'un programme (indépendamment du domaine d'application de ce programme). C'est la notion de *graphe des dépendances*. Or la notion de graphe des dépendances est *représentée* directement en 8½ par la notion de *système*. Un système permet donc de décrire un *espace hétérogène*.

GBF $\qquad$ système

$\downarrow \qquad \downarrow$

espace ↔ *forme* ↔ *graphe des dépendances*

FIG. 1 –: *Relation entre forme, espace, graphe des dépendances, GBF et systèmes.*

Ce point de vue, illustré par la figure 1, va s'éclaicir en prenant l'exemple de *iota* (voir la section V.6.2, page 62). Le graphe des dépendances de *iota* en un point, disons le point $gauche^{-2}$, est un graphe dont les sommets sont les points de *G1*. Il y a un arc allant d'un sommet s_1 à un sommet s_2, si on a besoin de la valeur de s_2 pour calculer la valeur de s_1. Par exemple, pour $gauche^{-2}$, le graphe des dépendances est figuré par le schéma :

$$(e) \longleftarrow (gauche^{-1}) \longleftarrow (gauche^{-2})$$

Puisque *iota* est spécifié par une définition récursive de GBF, *iota* «respecte le voisinage», et donc, *le graphe des dépendances* du calcul de la valeur d'un point de *iota* est un *sous-graphe de G1* .

Si on fait l'union de tous les graphes des dépendances pour tous les points de de *G1*, on retrouve *G1* tout entier. Mais pour un autre champ que *iota*, on pourra ne retrouver qu'un sous-graphe. La figure 2 en donne un exemple.

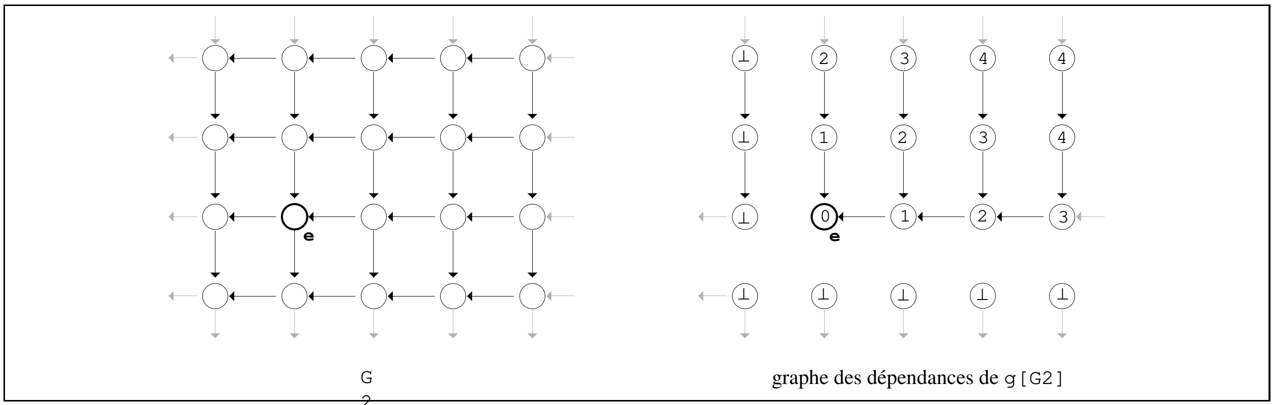


FIG. 2 –: *Un exemple de forme et de graphe des dépendances associé à un champ g sur cette forme. Le champ g est défini par : $g@\langle\rangle = 0,\ g@\langle x\rangle = 1 + g.x^{-1},\ g = 1 + g.y^{-1}$. L'union des graphe des dépendances pour tous les points de la forme ne constitue qu'un sous-graphe de celle-ci.*

Examinons à présent ce qui se passe pour l'autre concept de collection en 81/2 : les systèmes. Rappelons (cf. section II.3.4, page 16) qu'un système est un paquet d'équations rassemblées logiquement. Par exemple :

$$\{a = 1, b = a + 1, c = a + b, d = a + b + c\}$$

Cette collection est une expression qui va être calculée et se réduire en :

$$\{a = 1, b = 2, c = 3, d = 6\}$$

Le calcul d'un système consiste à calculer chaque partie droite des définitions. Puisqu'il y a calcul, il y a donc un graphe des dépendances qui correspond simplement au graphe des dépendances entre les définitions :

– chaque définition correspond à un sommet ;
– il y a un arc de s_1 vers s_2 si la valeur associée à s_2 est requise pour calculer la valeur de s_1.

Le graphe des dépendances associé au système précédent est schématisé par :

$$\{a = 1, b = a + 1, c = a + b, d = a + b + c\} \qquad (a) \longleftarrow (b) \longleftarrow (c) \longleftarrow (d)$$

On voit que cette fois le graphe associé ne peut être décrit de manière compacte : il est spécifié à travers les équations d'un système.

Le graphe des dépendances des éléments d'un système représente exactement la même chose que le graphe des dépendances des points d'un GBF. Il est donc naturel de parler d'espace à son propos. Cependant, contrairement aux GBF, l'espace sous-jacent à un système est qualifié d'*hétérogène* afin de bien marquer qu'il correspond à un graphe quelconque.

VIII.2 Système et graphe data-flow

La forme d'un GBF, qui représente un espace homogène, est une entité que nous avons définie pour elle-même, car elle peut se factoriser entre plusieurs spécifications de GBF. Par contre, un espace hétérogène est une structure trop ad hoc pour qu'il soit utile de la factoriser entre plusieurs spécifications de systèmes. Autrement dit, alors qu'on définit les GBF comme un calcul sur une forme donnée *a priori*, le graphe des dépendances d'un système est donné a posteriori comme sous-produit du calcul.

En effet, un système correspond à un objet « plus riche » qu'un graphe des dépendances : il décrit aussi des calculs[1]. Par exemple, le système :

$\{a = 1,\ b = 2,\ c = a + b\}$ et le système : $\{a = 3,\ b = 4,\ c = a * b\}$

1. les calculs décrits par un système respectent le graphe des dépendances, de la même façon que la définition d'un GBF respecte la forme du GBF.

présentent le même graphe des dépendances :

$$(a) \longrightarrow (c) \longleftarrow (b)$$

mais correspondent à des calculs différents[1]. Le graphe qui représente sous une forme abstraite à la fois les calculs et les dépendances, est le *graphe data-flow* du système ou DFG (acronyme de l'anglais **D**ata **F**low **G**raph). Les sommets du DFG sont les opérateurs des expressions et sous-expressions qui apparaissent en partie droite des définitions du système. Il y a un arc du sommet s_1 vers le sommet s_2 si :

- soit s_2 est une sous-expression de s_1 ;
- soit il y a une définition $x = s_2(\dots)$ et x est argument de s_1.

La figure 3 illustre un exemple : on reconnaît bien un graphe data-flow tel qu'il est défini par exemple dans [Klo87] et on peut retrouver simplement le graphe des dépendances associé[2].

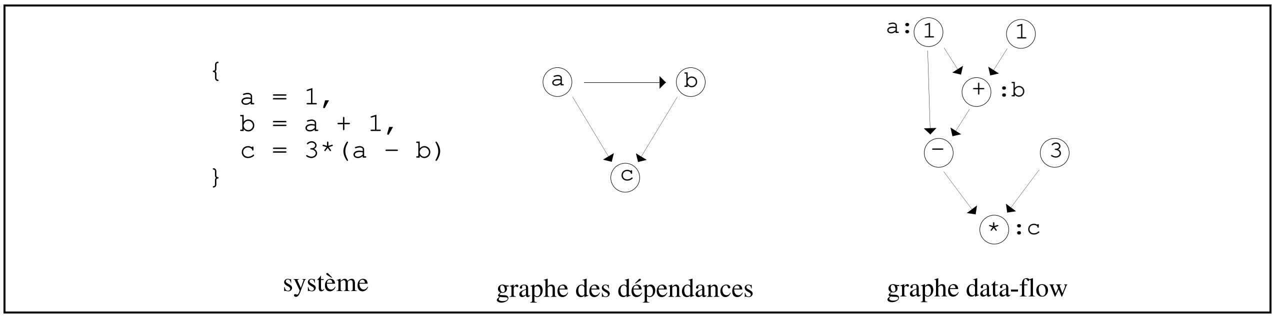


FIG. 3 –: *Graphe des dépendances et graphe data-flow d'un système.*

VIII.3 Construction des DFG

Notre problème est de construire des systèmes par combinaison de systèmes existants, de la même façon qu'on construit des GBF par combinaison de GBF. Et de la même manière que l'on s'intéresse à la définition récursive de GBF par des équations, nous serons intéressés par la définition récursive de systèmes par des équations. Notre approche ne repose pas sur la notion classique de combinateurs [Klo87, Mee89], mais s'appuie sur la notion de *nom*, ce qui permet de rester dans un cadre purement déclaratif. Cela nous conduit à la notion d'*amalgame* qui généralise la notion de système.

Regardons les systèmes comme la représentation textuelle d'un DFG. Alors un système $8_{1/2}$ est un DFG « complet » qui décrit un calcul à effectuer, sans possibilité de paramétrer ce calcul. Pour construire des nouveaux DFG, il faut disposer de DFG «incomplets» qu'on peut combiner pour construire plusieurs DFG différents. Par exemple, l'expression :

$$1 + (2 + 3)$$

représente un calcul « complet » : on ne peux pas combiner cette expression pour construire des calculs différents (on peut tout au plus la juxtaposer avec d'autres calculs indépendants). Par contre, l'expression :

$$1 + (2 + x)$$

1. Identifier deux calculs s'ils conduisent aux même graphe des dépendances, a été formalisé dans le cas des fonctions par [Fre95]. Grossièrement, deux fonctions sont *intensionnellement équivalentes* si les arbres de calculs (i.e. le graphe des dépendances) coïncident.

2. Pour retrouver le graphe des dépendances à partir du DFG, il faut supprimer dans le graphe data-flow les sommets qui ne correspondent pas à un identificateur (ce sont les opérateurs qui ne sont pas partie droite d'une expression, mais en sous-expression). Puis il faut ajouter les arcs nécessaires pour maintenir la connexité initiale. Par exemple, si on supprime s_2 dans $s_1 \to s_2 \to s_3$, il faut ajouter un arc pour maintenir $s_1 \to s_3$. Enfin, il faut renommer les sommets restant avec le nom de l'identificateur associé.

où on peut remplacer x par un entier par un mécanisme à définir, est une entité qui n'est pas un DFG mais qui peut servir à construire beaucoup de DFG différents en remplaçant x par des valeurs différentes.

On voit qu'il y a donc en fait deux problèmes à résoudre pour construire des DFG :

- il faut disposer d'entités « composables »,
- il faut disposer d'un mécanisme de composition.

Deux approches sont classiquement utilisées pour répondre à ces problèmes [Klo87] :

- *l'approche fonctionnelle :* l'entité composable est la *fonction* et le mécanisme de composition, les *combinateurs* ;
- *l'approche déclarative :* l'entité composable est l'*expression ouverte*, i.e. un DFG avec des arcs pendants identifiés par un nom, et le mécanisme de composition est un mécanisme de *liaison des noms.*

VIII.3.1 Construction fonctionnelle de DFG

Par abstraction de certaines sous-expressions d'un DFG, on transforme le DFG qui représente le calcul d'une expression en une fonction. Ces fonctions peuvent se combiner pour construire d'autre fonctions. On obtient de nouveau un DFG à partir d'une fonction par application.

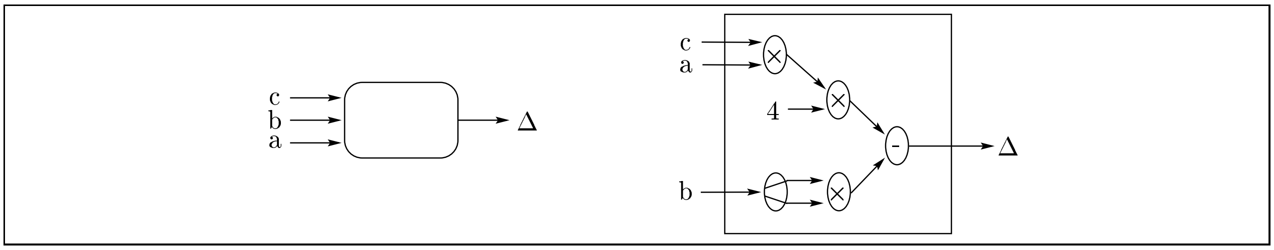
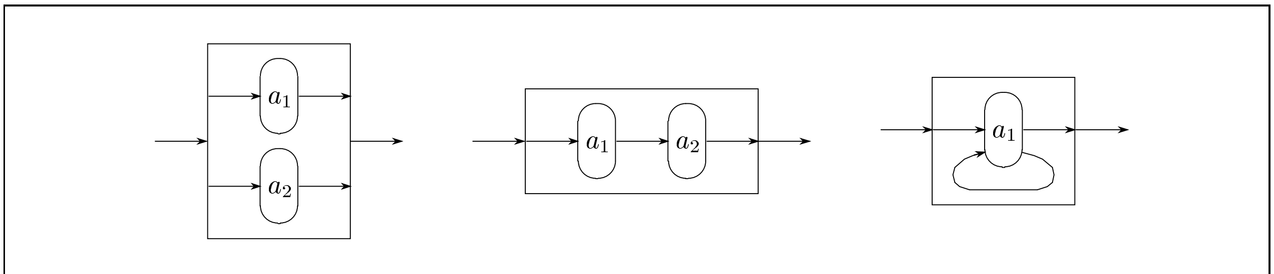


FIG. 4 –: *La « boîte » correspondant au calcul du déterminant d'un polynôme du second degré.*

En termes de graphe, ce mécanisme de construction admet une interprétation simple : une fonction correspond à un DFG mais avec des *entrées*, c'est-à-dire des arcs entrant sur un sommet mais qui n'ont pas de sommet source ; voir l'illustration sur la figure 4 : c, a et b sont des entrées. On considère aussi des *sorties* qui correspondent à des arcs sortant d'un sommet, mais sans sommet but. Un combinateur est un mécanisme qui associe les sorties d'un premier DFG aux entrées d'un second [Mee89].



(a) Composition parallèle : $a_1 \Downarrow a_2$. (b) Composition série : $a_1 \Rightarrow a_2$. (c) Rebouclage : $\circlearrowleft a_1$.

FIG. 5 –: *Les opérateurs de composition fonctionnelles de DFG : la mise en parallèle, la mise en série et le rebouclage.*

On identifie trois combinateurs principaux [Klo87] (cf. figure 5) :

1. la *mise en série* : les sorties d'un premier DFG sont connectées aux entrées d'un second DFG ;

2. la *mise en parallèle* : deux DFG sont groupés en une entité où les entrées correspondent aux entrées des deux DFG et les sorties correspondent aux sorties des deux DFG ;

3. le *rebouclage* : une sortie d'un DFG est connectée sur une entrée du même DFG pour obtenir un mécanisme de «feed-back».

Ces combinateurs reposent sur la *position* des entrées et des sorties les unes par rapport aux autres. Si on a nommé les entrées/sorties dans les schémas des figures 4 et 5, c'est uniquement pour la commodité du lecteur. D'ailleurs il est bien connu qu'on peut changer le nom des paramètres d'une fonction, sans changer sa sémantique.

VIII.3.2 Construction déclarative de DFG

Dans l'approche déclarative, l'objet de base qui peut être connecté, est aussi un graphe avec des entrées : comme précédemment, ce sont des arcs pendants. Mais ces arcs sont nommés et le mécanisme de composition consiste à relier les arcs pendants avec les sommets de même nom.

Si on veut faire le parallèle entre le point de vue textuel (les systèmes) et le point de vue graphique (les DFG), l'occurrence d'un identificateur id dans une expression e correspond à un arc. Si cet identificateur est lié à une définition, c'est un arc qui relie l'expression de définition de id à e. Un arc pendant représente un identificateur qui ne réfère à aucune définition. Autrement dit, la composition de graphes se fait simplement en (re)liant les occurrences d'un identificateur avec l'expression qui le définit.

Introduisons un peu de vocabulaire. Un *identificateur* est un élément d'un ensemble de noms. Une *référence* est l'occurrence d'un identificateur dans une expression. On dit qu'une référence est *libre* si l'identificateur ne réfère à aucune définition, sinon on dit qu'il est *lié*. Le même identificateur peut avoir des occurrences libres et liées suivant l'expression où il apparaît. Une expression qui contient une référence libre (ou qui contient une sous-expression qui contient une référence libre) est dite *ouverte*. Sinon, l'expression est dite *close* : toutes les références d'une expression close sont liées. Une expression close correspond à un DFG.

La composition des expressions ouvertes repose donc sur la notion de nom : un nom sert de *référence commune* entre les définitions et les expressions. On peut comparer une expression ouverte et une fonction : une expression ouverte peut être complétée en fournissant les définitions manquantes, et une fonction attend des arguments pour calculer un résultat. Il existe cependant deux grandes différences entre les fonctions et les expressions ouvertes :

– Les arguments sont explicitement abstraits dans une fonction, alors que les références libres d'une expression ouverte le sont implicitement. Par suite, l'application de fonction doit être explicite, alors que ce n'est pas le cas pour la liaison des références. Le mécanisme de *curryfication* du λ-calcul et des langages fonctionnels permet de fournir une partie seulement des arguments attendus, mais elle implique un ordre strict suivant lequel ces arguments doivent être fournis. Ce n'est que très récemment que des extensions du λ-calcul ont été proposées [GAK94, Dam94c] pour s'affranchir de cette contrainte, et cela au prix d'une théorie assez complexe. À l'opposé, la fermeture d'une expression est naturellement incrémentielle et non ordonnée.

– Les arguments d'une fonction sont repérés par une position : c'est la notion de *variable* opposée à celle de *nom*. Dans les langages fonctionnels et leur modèle, le λ-calcul en particulier, la notion de variable implique des opérations d'α-renommage pour éviter le phénomène de *capture*, c'est-à-dire qu'une variable abstraite soit identifiée avec une variable en position d'argument. Par exemple, $(\lambda x.(\lambda y.x + y))y$ doit être converti avant évaluation en $(\lambda x.(\lambda z.x + z))y$ afin d'éviter la capture de la variable libre y par l'abstraction y (cette expression doit s'évaluer en $\lambda z.y + z$ alors que sans renommage on obtiendrait $\lambda y.y + y$). Au contraire, les noms ne sont pas des variables muettes et deux noms réfèrent toujours au même objet.

La construction de DFG par l'approche déclarative est *implicite*, ce qui nous pose un problème. En effet, l'utilisation d'une méthode de construction implicite dans le cadre de 81/2 permet de programmer de façon simple et naturelle. Mais cette approche tend à séparer nettement la phase de construction des systèmes (qui se fait «à la main» et qui correspond alors à la phase de programmation) de la phase de calcul (qui consiste à «faire fonctionner» le DFG décrit). Or on aimerait calculer un DFG, et non seulement le construire à la main.

Pour cela, il faut introduire des opérateurs qui peuvent combiner des expressions ouvertes : on veut calculer avec des expressions ouvertes comme on calcule avec des fonctions. On pourrait penser à introduire des combinateurs analogues à ceux de l'approche fonctionnelle. Mais ces opérateurs restent de bas niveau et coexistent mal avec le schéma de nommage des systèmes (ils reposent sur une notion de position des entrées et non sur une notion de nom). Il est plus simple, plus économique, et plus expressif d'étendre la sémantique des opérateurs de concaténation et de sélection des systèmes 81/2, aux systèmes ouverts. En effet :

- Une expression ouverte représente un DFG avec des entrées non connectées (i.e. des nœuds pendants), qu'on peut utiliser pour la connection avec d'autres expressions ouvertes.
- L'opération de création d'un système $\{\dots\}$ permet de construire un DFG arbitraire ou une expression ouverte.
- L'opérateur de concaténation permet l'union des DFG arguments et la connection des entrées avec les sorties correspondantes en se basant sur une l'égalité des identificateurs de même nom.
- L'opération de sélection permet l'extraction d'un sous-graphe dans un graphe donné (le sous-graphe qui calcule une sortie donnée).

On appelle **amalgame** les expressions obtenues combinant des expressions ouvertes avec ces opérateurs.

VIII.4 Les amalgames

Dans cette section, nous allons préciser le concept d'amalgame. La définition formelle des amalgames se fera dans le chapitre X. Les amalgames sont les expressions qu'on peut construire en combinant des termes de base (les entiers et les identificateurs par exemples) avec les trois opérateurs :

1. l'opérateur n-aire d'*amalgamation* «$\{\dots,\dots\}$»,
2. l'opérateur binaire de *concaténation* « $\#$ »,
3. l'opérateur binaire de *sélection* « **.** ».

VIII.4.1 Les systèmes

Les termes qui sont obtenus par une opération d'amalgamation sont appelés *systèmes*. Un système représente un ensemble de définitions. Une définition est un couple :

identificateur = *expression*

Par exemple, l'expression suivante spécifie un système qui regroupe deux définitions

$\{a = 1,\, b = 2 + 3\}$

On suppose que toutes les parties gauches des définitions d'un système sont différentes. Par exemple, on interdit $\{a = 1,\, a = 2\}$. La partie droite d'une définition est une expression quelconque. Par exemple, on peut imbriquer les systèmes :

$\{a = 1,\ b = \{a = 2,\, c = 3\}\}$

On remarquera que le système englobé redéfinit l'identificateur a : ce n'est pas en contradiction avec la règle précédente car les systèmes sont différents.

Nous utilisons le terme de *référence* pour désigner un identificateur qui apparaît dans une expression en partie droite d'une définition. Ces références peuvent être *liées* ou *libres* suivant qu'elles réfèrent à une définition d'un système ou non. Un système constitue une «région de visibilité» pour ses définitions et nous allons détailler les particularités du mécanisme de liaison.

VIII.4.2 La liaison des références

Le mécanisme de liaison est le mécanisme qui permet d'associer une expression à une référence id. Cette expression est l'expression e en partie droite d'une définition $id = e$. Par exemple :

$$\{a = b,\ b = 2\} \tag{1}$$

définit un système contenant deux définitions ; l'identificateur b en partie droite de la première définition réfère la seconde définition. Par souci de clarté, on a indiqué par des flèches les définitions qui sont référées. L'ordre des définitions d'un système n'a pas d'importance quand à la liaison. On aurait par exemple la même liaison si on avait inversé les définitions de l'exemple précédent :

$$\{b = 2,\ a = b\} \tag{2}$$

et on peut même avoir des références mutuellement récursives :

$$\{x = y,\ y = x\} \tag{3}$$

Les définitions d'un système ne sont pas visibles en dehors du système. Par exemple dans :

$$\{a = x,\ b = \{x = 1,\, y = 2\}\}$$

la référence x en partie droite de la définition de a ne peut pas se lier avec la définition de x qui apparaît dans le système en partie gauche de b. Un système constitue une unité de visibilité. Le *scope* d'une définition est le sous-ensemble de l'arbre syntaxique dans lequel une référence peut se lier à cette définition. Les règles de visibilité sont les règles usuelles qui suivent la structure en bloc de l'imbrication des systèmes.

L'imbrication des système permet les redéfinitions. Il se pose alors le problème de l'accès aux différentes définitions d'un même identificateur. Le traitement classique de la redéfinition des identificateurs consiste à masquer toutes les définitions précédentes et de ne permettre l'accès qu'à la dernière définition (par exemple, dans les langages fonctionnels ou impératifs comme Pascal, la redéfinition masque la définition précédente). Mais il est intéressant de pouvoir accéder à une définition masquée. Par exemple, dans un langage orienté objet comme C++, il est possible de référer à une méthode de la classe mère, même si cette méthode a été redéfinie. C'est pourquoi nous introduisons un *opérateur d'échappement de scope* : x^n est une référence qui va chercher sa liaison n niveau au-dessus du système directement englobant. Par exemple, dans le terme

$$\{a = 1,\ b = \{a = 2, x = a^1\}\} \tag{4}$$

l'occurrence de a en partie droite de la définition de x réfère à la définition $a = 1$ grâce à l'échappement « ^{1} ».

VIII.4.3 Les références libres

Une référence qui n'est pas liée à une définition est une référence *libre*. Par exemple dans le système :

$$\{a = x\}$$

la référence x est libre. De la même manière qu'il est utile de pouvoir « sauter des niveaux » dans une liaison, il est utile de pouvoir « sauter par dessus des définitions ». On utilise donc aussi l'opérateur d'échappement pour une référence libre. Par exemple dans le système :

$$\{a = \{x = 1, y = x^1\}\}$$

la référence à x en partie droite de la définition de y n'est pas liée à la définition $x = 1$ car l'opérateur d'échappement « ^{1} » spécifie qu'il faut chercher une liaison 1 niveau au-dessus du scope courant.

VIII.4.4 Conventions pour la notation des références

Comme l'échappement x^0 se comporte de la même manière que la référence x, on peut systématiser les échappements et convenir qu'une occurrence x dénote par commodité x^0. Ainsi, toutes les références sont de la forme id^n où id est un identificateur et n un entier.

Cette notation ne suffit pas à empêcher certaines ambiguïtés qui apparaissent lorsqu'on manipule les amalgames. Sans entrer dans les détails, qui seront précisés à la section VIII.6, on a besoin de distinguer les références liées et les références libres. Pour cela, on notera l'échappement d'une référence liée par un indice : id_n, et l'échappement d'une référence libre par un exposant : id^n.

L'exemple suivant (cf. figure 6) illustre un exemple d'utilisation de références libres et liées :

$$\{a = 1, b = \{a = 2, c = a_0, d = a_1, e = z^0, f = a^2\}\} \tag{5}$$

Au premier niveau de l'expression (5), il apparaît deux définitions a et b. Le système lié à b spécifie cinq définitions, pour les identificateurs a, c, d, e et f. La référence a^2 de f attend une définition de a dans un environnement englobant qui n'existe pas. La référence z^0, reste libre en étant candidate à une liaison future à partir du niveau courant.

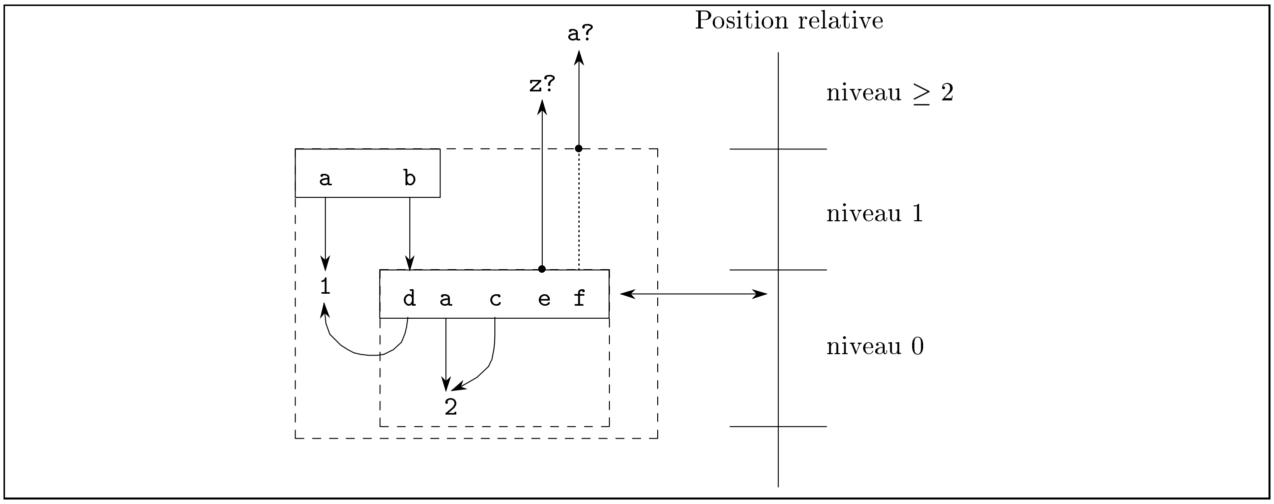


FIG. 6 –: *Un exemple d'accès à de multiples redéfinitions d'identificateurs. La représentation graphique correspond à l'expression (5). La définition des systèmes peut se représenter sous la forme d'un arbre, où les nœuds correspondent à des systèmes (le premier système défini étant la racine), les sous-systèmes définissent des nœuds et les définitions terminales des feuilles. Les flèches en traits continus indiquent les liaisons, les flèches en traits pointillés représentent les systèmes traversés par les références pendantes, les boîtes en traits ininterrompus représentent des définitions de systèmes, les boites en pointillés représentent les inclusions de systèmes.*

Les notations id_n et id^n sont une «généralisation» de la notation de de BRUIJN [dB72] définie pour résoudre les problèmes d'α-conversion du λ-calcul. Dans son utilisation conventionnelle, la notation est utilisée pour éviter la capture de variables en supprimant les noms et en ne laissant que les indices qui réfèrent au niveau d'abstraction (le nombre de λ traversés) d'une variable définie. Dans les amalgames, on utilise *en plus* les noms qui permettent de réferer à un identificateur se trouvant à une place précise d'une expression (on trouvera des constructions similaires pour le λ-calcul dans [BF82, Dam94c, LF96]).

Évidement, cette notation peut sembler contraignante pour le programmeur, de la même manière que l'utilisation de la notation de de BRUIJN est contraignante en λ-calcul. Aussi, pour alléger les écritures, on définit du *sucre syntaxique* qui permet au programmeur d'écrire à la place d'une référence correcte :

1. *un identificateur id*. Dans ce cas, cet identificateur est implicitement converti en une référence liée id_p si la plus proche définition visible de id se trouve dans le p^e niveau englobant. S'il n'y a pas de définitions visible de id, cet identificateur est implicitement converti en une référence libre id^0.

2. *une référence libre* id^n. Dans ce cas, cette référence libre reste libre s'il n'y a pas de définition visible de id dans les scopes englobant de niveau supérieur ou égal à n. Sinon, la référence libre est convertie en une référence liée id_p avec $p \geq n$, p étant le plus petit niveau supérieur à n qui contient une définition de id.

3. *une référence liée* id_n. Dans ce cas, une définition de id doit apparaître dans le n^{e} scope englobant.

Par exemple, l'expression :

$$\{a = 1, b = a, c = d\}$$

se traduit implicitement en :

$$\{a = 1, b = a_0, c = d^0\}$$

car il existe une définition de a mais aucune définition pour d. Dans la suite, tous les termes que nous écrivons sont soumis à ces conventions.

VIII.4.5 La concaténation et la sélection

On peut combiner des expressions par concaténation ou bien par sélection. Nous verrons ci-dessous la sémantique des ces opérateurs. Pour le moment nous voulons juste préciser deux points.

Les arguments d'une concaténation ou bien d'une sélection sont *a priori* quelconque. Par exemple, on peut avoir $\{a = x \# y,\ b = x \mathbin{.} (1 + u),\ x = \{u = 1,\ v = 2\},\ y = z^0\}$. Par ailleurs, le premier argument d'une sélection constitue un scope pour le second argument. Par exemple,

$$\{a = 1,\ b = \{a = 2\} \mathbin{.} (a_1 + a_0)\}$$

(notez les indices nécessaires à exprimer les liaisons décrites par les flèches).

VIII.4.6 Évaluation des amalgames

Nous avons défini les termes d'un langage que nous appelons *amalgames*. Certains de ces termes sont atomiques et correspondent à des constantes (par exemple, l'expression $\{a = 1, b = 2\}$). Par contre, un terme comme $\{a = 1\} \# \{b = 2\}$ n'est pas un terme atomique car il peut se *réduire* (on utilise aussi le terme *évaluer*) en $\{a = 1, b = 2\}$. Nous disons que $\{a = 1, b = 2\}$ est la valeur de $\{a = 1\} \# \{b = 2\}$ (la situation est analogue au langage des expressions arithmétiques : si 1 est une constante, le terme $1 + 2$ peut se réduire en la constante 3 qui est la valeur de $1 + 2$).

Le processus de réduction d'un amalgame est formalisé dans le chapitre X. Informellement, le calcul de la valeur d'un amalgame implique trois processus :

1. Les références liées dans une expression sont remplacées par leurs définition. Par exemple, l'expression $\{a = 1, b = a_0\}$ se réduit en la constante $\{a = 1, b = 1\}$.

2. La concaténation de deux systèmes se réduit en un seul système et les liaisons possibles sont effectuées. Par exemple, l'expression $\{a = 1\} \# \{b = a^0\}$ se réduit en $\{a = 1, b = a_0\}$ qui lui même se réduit en la constante $\{a = 1, b = 1\}$. On remarquera dans cet exemple que l'évaluation de deux systèmes a transformé une référence libre a^0 en une référence liée a_0 qui a ensuite été réduite par le premier processus.

3. L'évaluation d'une opération de sélection correspond à l'évaluation de l'expression à droite d'un point dans un environnement augmenté par les définitions du système en partie gauche du point. Par exemple, l'expression $\{a = 1\} \mathbin{.} a$ se réduit en 1 et l'expression $\{a = 1\} \mathbin{.} \{b = a_1\}$ se réduit en $\{b = 1\}$.

Ce processus de réduction pose de nombreux problèmes. En particulier :

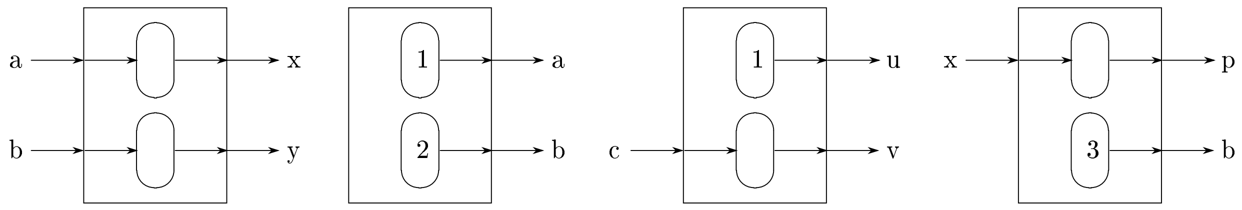
- quels sont les termes constants ?
- quand faut-il lier des références libres ?
- quelle est la stratégie de liaison à adopter ?
- quelle est la stratégie d'évaluation, c'est-à-dire, comment combiner les trois processus de réduction décrits précédemment ?

Ces problèmes ne sont pas simples et sont illustrés par des exemples dans la section VIII.6 à la fin de ce chapitre. Nous avons rejeté cette section à la fin de ce chapitre car elle consiste essentiellement en un catalogue des configurations que l'on peut rencontrer lors de l'évaluation des amalgames.

VIII.4.7 Quelques exemples de réductions

La figure 7 illustre quelques exemples de composition de DFG dans le style fonctionnel et leurs équivalents en amalgames. Comme on peut le voir, si l'opération de concaténation réalise toutes les opérations de composition, la composition effectivement réalisée dépend de l'identification des expressions.

La figure 9, à la fin du chapitre, illustre une construction plus complexe où un système est construit comme le résultat d'un calcul. Si on essaie de représenter ce DFG, il faut que les valeurs qui transitent sur un arc soient des systèmes, à savoir des DFG. On a donc « élargi » les arcs pour laisser apparaître la structure des valeurs manipulées. L'objet présenté est un graphe de graphes.

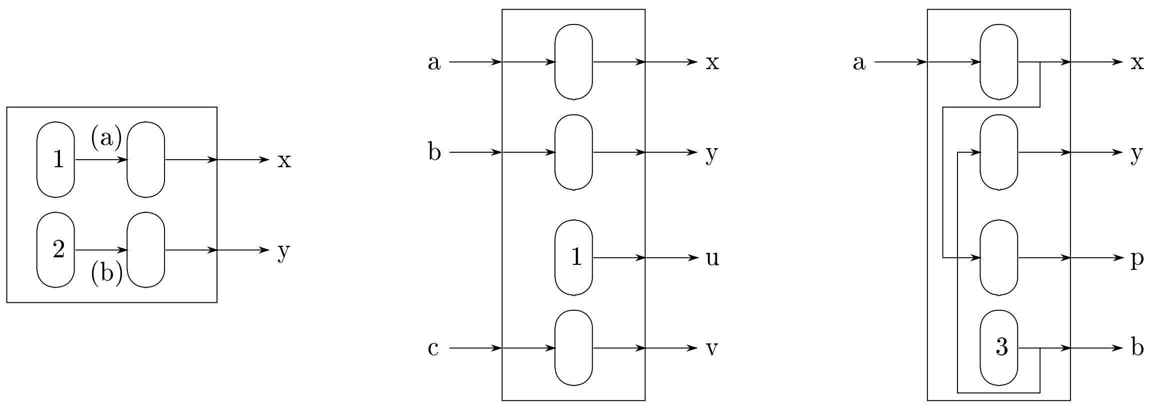


(a) DFG de l'expression $s_1 \equiv \{x = a, y = b\}$.

(b) DFG de l'expression $s_2 \equiv \{a = 1, b = 2\}$.

(c) DFG de l'expression $s_3 \equiv \{u = 1, v = c\}$.

(d) DFG de l'expression $s_4 \equiv \{p = x, b = 3\}$.



(e) DFG de l'expression $s_1 \Rightarrow s_2$ qui correspond à l'expression $8_{1/2}\ s_1 \cdot s_2$.

(f) DFG de l'expression $s_1 \Downarrow s_3$ qui correspond à l'expression $8_{1/2}\ s_1 \# s_3$.

(g) DFG de l'expression $s_1 \# s_4$. Ce DFG ne peut pas s'obtenir comme le résultat d'une seule opération de mise en série, de mise en parallèle ou de rebouclage.

FIG. 7 –: *Exemple de composition de système dans un style fonctionnel et avec les amalgames.*

VIII.4.8 Discussion sur le choix des opérateurs de composition

Nous avons choisi seulement deux opérateurs pour composer des amalgames : la concaténation et la sélection. Nous avons vu ci-dessous que cela nous permettait néanmoins de faire le même type de combinaisons que celles permises par les trois opérateurs de composition fonctionnelle que nous avons décrits plus haut.

Par ailleurs, la concaténation et la sélection sont déjà présent dans le langage 81/2 : nous avons juste étendu leur sémantique (et permis les expressions ouvertes).

Enfin, ces opérateurs admettent une interprétation simple en terme de mécanisme de programmation.

La concaténation correspond à un mécanisme d'enrichissement, comme celui de l'*héritage* dans les langages à objets. On remarquera que tous les systèmes peuvent être concaténés. Il n'est pas nécessaire qu'ils soient d'une forme particulière, à l'inverse des langages orientés objets, pour lesquels seules les classes peuvent être complétées par héritage et une instance de classe ne peut pas être étendue.

Dans 81/2 classique, l'opérateur de sélection est l'opérateur d'accès à la valeur d'une définition : il est en cela semblable à l'accès à un item[1] dans un enregistrement. Afin d'alléger l'écriture des expressions comportant un grand nombre d'accès à un enregistrement, le langage Pascal introduit une construction qui permet de ne pas spécifier à chaque fois l'enregistrement accédé : la construction `with`. Ainsi, l'expression utilisant les items `a`, `b` et `c` d'un enregistrement `r`, que l'on note normalement `r.a + r.b + r.c` devient avec la construction `with` :

```
with r
  a+b+c
```

Cette construction, outre la concision qu'elle apporte, permet le mélange de façon implicite des références à un champ et à une variable. En effet, l'expression `r.a + x` où `x` est une variable, peut s'écrire :

```
with r
  a+x
```

le compilateur se chargeant de retrouver quelles sont les variables qui correspondent à l'accès aux items d'un enregistrement et quelles sont les variables qui correspondent à des variables Pascal « réelles ». Nous avons décidé de reprendre ce point de vue pour la sémantique de l'opérateur de sélection. Ainsi, l'expression :

$$\{a = 1, b = 2\} \;.\; ((a + b) + x)$$

correspond à :

```
with {a=1, b=2}
  (a+b) + x
```

et s'évalue en :

$$(\{a = 1, b = 2\} \;.\; a + \{a = 1, b = 2\} \;.\; b) + x$$

c'est-à-dire en $3 + x$. Le terme $3 + x$ est une valeur qui correspond à une expression ouverte du langage.

Une autre analogie peut être faite avec la construction `let rec ...` des langages fonctionnels. Par exemple, l'expression :

$$\{a = 1, b = 2\} \;.\; (a + b + x)$$

correspond à l'expression ML :

$$\texttt{let rec } a = 1 \texttt{ and } b = 2 \texttt{ in } a + b + x$$

(on utilise `let rec` plutôt que `let` car les définitions de a et b pourraient êtres récursives comme dans $\{a = 1, b = a_0 + 1\}$). Cependant, les définitions dans un `let rec` ML, contrairement à un `with` Pascal ou à une sélection, ne sont pas des objets ML. En effet, dans l'expression : `let rec` $\diamond$ `in` $\star$, les définitions $\diamond$ et l'expression $\star$ ont une nature différente (une définition n'est pas une expression) et sont rassemblées par le programmeur. Au contraire, dans l'expression $\diamond$. $\star$, les définitions du système $\diamond$ sont de même nature que $\star$: elles peuvent être calculées. On a ainsi « internalisé dans le langage », c'est-à-dire rendu accessible et modifiable par le calcul, la notion d'*environnement*.

1. On parle traditionnellement de « champ » pour désigner les éléments d'un enregistrement. Afin de ne pas introduire de confusion avec les GBF, nous utiliserons plutôt le terme d'*item*.

VIII.5 Simulation et construction de DFG

Les amalgames permettent de construire de nouveaux systèmes. Par exemple la réduction du système :

$$\{\quad \begin{array}{rcl} s &=& \{a = 1,\ b = c^0\}, \\ t &=& \{c = a^0,\ d = a^0 + b^0\}, \\ u &=& (s \# t) \,.\, (b + d) \quad \} \end{array}$$

entraînera la construction d'un nouveau système $\{a = 1, b = c^0, c = a^0, d = a^0 + b^0\}$ comme résultat de la réduction de $s \# t$ et ce nouveau système sera utilisé comme environnement d'évaluation de $(b + d)$.

On voit donc bien que les amalgames permettent de construire de nouveaux systèmes et des les utiliser. Dans cette section nous allons examiner l'apport de la construction dynamique des systèmes à la simulation. Les amalgames sont nécessaires dans un langage de haut-niveau pour la simulation de SD, car ils permettent :

- de représenter des espaces hétérogènes ;
- de structurer les programmes ;
- de construire dynamiquement des simulations comme résultat d'un calcul.

En particulier, le dernier point apparaît comme absolument nécessaire dans les simulations de morphogénèse ou plus généralement pour les simulations dont l'espace des états doit être calculé conjointement à l'évolution du système.

VIII.5.1 Les systèmes comme éléments de structuration

La structuration des programmes de grande taille est devenue une nécessité indiscutable. La modélisation et la simulation de grandes applications implique la définition d'un grande nombre d'équations. Il est souvent préférable que la définition des expressions respecte la hiérarchie présente dans le système modélisé, et que l'on puisse réutiliser des expressions déjà existantes.

Il existe un besoin important de constructions modulaires et incrémentielle des simulations. Par exemple, on peut noter que la plupart des logiciels commerciaux de simulation, mettent en avant leurs nombreuses bibliothèques, avant même leur moteur (SES Workbench, Extend, Interactive Physics...). En effet, les phénomènes à simuler sont de plus en plus complexes à décrire, ce qui nécessite absolument de factoriser et de capitaliser les développements déjà effectués.

Ces besoins de *modularité* ne trouvent pas de répondant en 81/2. Les systèmes esquissés dans [Gia91a] ne représentent qu'un *sucre syntaxique* qui ne fournit aucun mécanisme de paramétrisation des programmes. Nous avons vu lors de l'exemple du wlumf, dans le chapitre (cf. section III.1, page 19), que le programme pouvait se structurer en séparant la définition de l'environnement de la définition du comportement. Mais cette séparation est purement logique et correspond uniquement à une notion de visibilité des identificateurs. Cela n'est pas suffisant. Reprenons cet exemple, en le sophistiquant légèrement, pour illustrer le problème.

On veut *paramétrer* le comportement du wlumf pour le faire dépendre de son environnement et de son espèce. Le programme devient :

$$\begin{array}{lcl} env_1 = \{ && \\ \quad cpt &=& \$cpt + 1\, \texttt{when}\, \texttt{Clock}\, -2;\ cpt@0 = 0; \\ \quad \texttt{boolean}\, nourriture &=& 0 == (cpt\%2); \\ \}; && \\ && \\ espece_1 = \{ && \\ \quad limg_{inf} &=& 3; \\ \quad limg_{sup} &=& 10; \\ \quad dec_G &=& 1; \\ \quad init_G &=& 6; \\ \}; && \end{array}$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{wlumf = \{} \\
\quad affame & = & (glucose < espece_1 \,.\, limg_{inf}); \\
\quad affame@0 & = & \texttt{false}; \\
\quad glucose & = & \texttt{if}\, mange \\
 & & \texttt{then}\, espece_1 \,.\, limg_{sup} \\
 & & \texttt{else}\, \max(0, \$glucose - espece_1 \,.\, dec_G)\, \texttt{when}\, \texttt{Clock}; \\
\quad glucose@0 & = & espece_1 \,.\, init_G; \\
\quad mange & = & \$affame \,\&\&\, env_1 \,.\, nourriture; \\
\quad mange@0 & = & false; \\
\}; & &
\end{array}
$$

Le système env_1 définit deux tissus : le tissu entier *cpt* et le tissu booléen *nourriture*. Le système $espece_1$ définit les tissus constants $limg_{inf}$, $limg_{sup}$, dec_G et $init_G$ qui paramétrisent les comportements d'un wlumf suivant son « espèce ». Le système $wlumf$ définit trois tissus : le tissu booléen $affame$, le tissu entier *glucose* et enfin le tissu booléen *mange*. Le comportement d'un wlumf dépend alors de son espèce, et de son environnement.

On aimerait pouvoir simuler l'évolution de plusieurs espèces de wlumf dans plusieurs environnements différents. La figure 8 représente deux tracés des variables décrivant le wlumf dans l'environnement env_1 que nous avons précédemment décrit et un wlumf d'une espèce $espece_2$ dans l'environnement env_2 suivant :

$$env_2 = env_1;$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{espece_2 = \{} \\
\quad limg_{inf} & = & 3; \\
\quad limg_{sup} & = & 7; \\
\quad dec_G & = & 1; \\
\quad init_G & = & 3; \\
\}; & &
\end{array}
$$

La définition de $wlumf$, pour la deuxième simulation a consisté à changer toutes les occurrences de env_1 en env_2 et de $espece_1$ en $espece_2$. La définition d'un wlumf fait *explicitement* référence aux définitions de *env* et de *espece* pour trouver les définitions dont il a besoin.

Il est possible, grâce aux expressions ouvertes, de laisser les identificateurs de $wlumf$ qui doivent être résolus *plus tard* (c'est-à-dire dans un autre contexte) sous la forme d'identificateurs sans définitions. Deux problèmes se posent alors :

1. Quel est le statut d'un système qui contient des références libres (c'est-à-dire des identificateurs qui ne réfèrent à aucune définition) ?

2. Par quel mécanisme pourra-t-on fournir un jour des définitions à ces identificateurs ?

À la première question, beaucoup de langages répondent en maintenant une distinction très forte entre les entités représentant des calculs, ou des données, qui doivent être complétés, et les valeurs « ordinaires » manipulées par le langage. Par exemple, le langage C++ maintient une distinction entre les notions de classe, et d'instance d'une classe. Un *package* paramétré en Ada n'est pas une valeur comme une autre (par exemple, l'instanciation d'un package paramétré doit donner un package non paramétré : on ne peut pas utiliser les packages $\forall\alpha.List(\alpha)$ et $\forall\alpha.Stack(\alpha)$ pour créer un package $\forall\alpha.List(Stack(\alpha))$). Même dans un langage fonctionnel comme SML, le langage de manipulation des modules est clairement séparé du noyau du langage fonctionnel [HMM90, Apo94] (il n'est pas encore possible, par exemple, de définir une fonction récursive distribuée sur plusieurs modules, même si des solutions commencent à apparaître [DS96]).

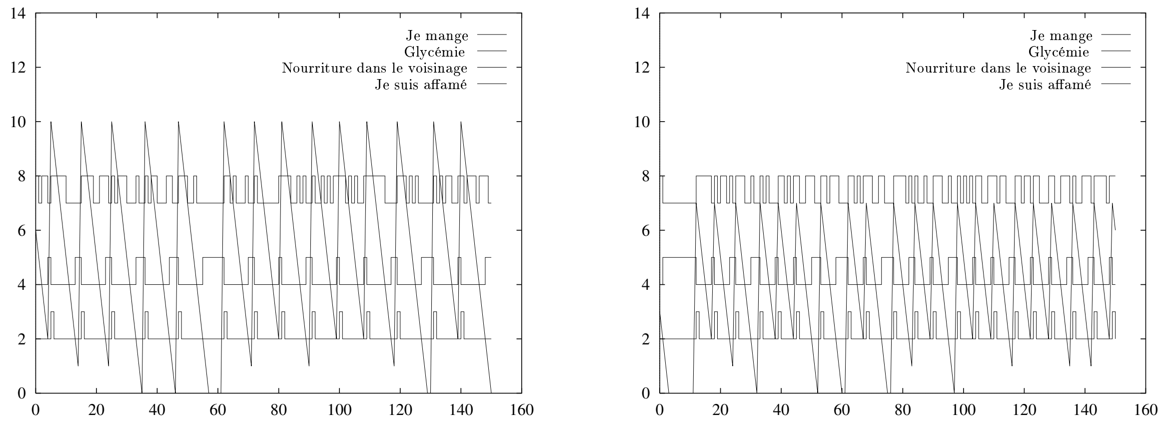
Avec les amalgames, nous avons décidé au contraire de donner le statut de valeur de première classe aux *systèmes ouverts*.

La seconde question pose le problème de la « fermeture » d'une expression ouverte, c'est-à-dire des mécanismes permettant de fournir une définition à chaque référence libre. Nous avons esquissé une proposition dans la section précédente : les opérateurs $\centerdot$ et $\#$. L'utilisation de $\#$ est évidente pour compléter et enrichir des programmes, et l'opérateur $\centerdot$ permet de paramétrer les programmes.

Avec les amalgames, il devient très simple de structurer et de paramétrer la simulation du comportement d'un wlumf. Par exemple, si on avait précédemment écrit :

$$\begin{array}{lcl} \multicolumn{3}{l}{wlumf = \{} \\ \quad affame & = & (glucose < limg_{inf}); \\ \quad affame@0 & = & \texttt{false}; \\ \quad \ldots & = & \ldots\} \end{array}$$

$wlumf$ serait alors un système ouvert que l'on peut compléter en $env_1 \centerdot (espece_2 \centerdot wlumf)$ ou bien en $env_2 \centerdot (espece_2 \centerdot wlumf)$, etc.



(a) 1er environnement et 1re espèce

(b) 2e environnement et 2e espèce

FIG. 8 –: *Comportement d'un système dynamique hybride. Évaluation du comportement du « wlumf » de deux espèces différentes.*

Le mécanisme que nous avons mis en œuvre ci-dessus grâce aux amalgames, est un mécanisme de paramétrisation de programme. Nous développons intégralement à la page 169 un exemple correspondant à un autre type de structuration : le paradigme objet. La mise en œuvre à travers les amalgames fait alors intervenir l'opérateur de concaténation.

VIII.5.2 Construction incrémentielle de programmes et simulation de SD

Un résultat informatique est classiquement le résultat de deux phases distinctes :

1. une phase de définition du problème et sa résolution sous la forme d'un programme ;
2. une phase d'obtention des résultats à travers l'exécution du programme précédemment défini.

Les mécanismes de liaisons que nous avons introduit (opérateurs $\centerdot$ et $\#$) permettent la construction dynamique d'un programme comme résultat de l'exécution d'un programme. On obtient donc un entrelacement des deux phases ci-dessus. Les deux exemples suivants illustrent cette possibilité et son intérêt.

Certaines méthodes adaptatives de résolutions numériques d'une équation aux dérivées partielles adaptent au cours du calcul la discrétisation de l'espace modélisé. Par exemple, dans un problème de dynamique des fluides, on prend un maillage plus lâche dans les régions où l'écoulement est laminaire et on resserre le maillage dans les domaines où l'écoulement a un régime turbulent. Le calcul est donc constitué de deux phases qui s'entrelacent, la phase de calcul *du* maillage, et une phase de calcul *sur* le maillage. Cette dernière phase a une structure qui dépend complètement du maillage. Certaines méthodes avancées génèrent un code spécifique à partir du maillage.

Notre deuxième exemple de programme de simulation dynamiquement calculé consiste à modéliser une « sélection darwinienne » sur un ensemble de wlumfs. Nous avons montré dans la section précédente qu'un wlumf pouvait être calculé comme le résultat de la complétion d'un wlumf prototypique $wlumf$, qui est un système ouvert, par une espèce :

$$creature \quad = \quad espece \,.\, wlumf$$

Cette équation peut parfaitement être étendue à une population de 100 wlumfs, par la définition de l'expression :

$$creature[100] \quad = \quad espece \,.\, wlumf$$

ce qui définit une collection de 100 éléments, chaque élément de la collection étant la réplique d'un wlumf.

Nous supposons à présent que la paramétrisation d'un wlumf, décrit par le système *espece*, est obtenue par la concaténation de deux demi-paramétrisations :

$$espece \quad = \quad \$haplo_1 \,\#\, \$haplo_2 \,\#\, \{CHR_X = \$haplo_1, CHR_Y = \$haplo_2\}$$

où *haplo* représente le nom du demi matériel génétique que chaque animal reçoit d'un de ses deux parents. Le système contenant les équations CHR_X et CHR_Y que l'on concatène à *espece* a simplement pour but de permettre de retrouver les haplotypes correspondant à une espèce donnée, grâce aux expressions $espece \,.\, CHR_X$ et $espece \,.\, CHR_Y$.

La simulation du comportement d'un wlumf consiste à faire évoluer cette créature dans son environnement. À chaque wlumf est associé une quantité qui correspond à ses «performances» dans l'environnement. Cette note est déterminée par le wlumf lui-même (on rajoute une équation $performance = \ldots$ dans la description du $wlumf$) :

$$evaluation \quad = \quad (env \,.\, creature) \,.\, performance$$

Il est possible de trier les *evaluation* obtenues et de ne retenir que les 10 meilleures performances (on suppose que l'on dispose d'une fonction `sort()` qui retourne la permutation réalisant le tri de son vecteur argument) :

$$\begin{array}{lcl} tri & = & \mathtt{sort}(evaluation) \\ meilleurs & = & creature(tri : [10]) \end{array}$$

Les dix meilleurs haplotypes mâles et les dix meilleurs haplotypes femelles sont alors répliqués de manière à engendre les 100 nouvelles espèces de wlumf pour la génération suivante :

$$\begin{array}{lcl} haplo_1 & = & (meilleurs \,.\, CHR_X) : [100] \\ haplo_2 & = & (meilleurs \,.\, CHR_Y) : [100] \end{array}$$

On peut évidemment introduire du bruit, faire varier les appariements entre haplotypes, considérer des environnements variés, etc.

Cet exemple[1] correspond à la création *totalement dynamique* d'un DFG. En effet, on ne connaît pas le système *espece* qui viendra compléter $wlumf$ pour en faire une *creature*, ce système étant le résultat d'un calcul complexe.

VIII.5.3 La composition de programmes comme paradigme de programmation

Nous venons de voir que les amalgames permettent un schéma de composition de programmes par complétion des expressions ouvertes à travers leurs références libres. L'opération de concaténation des amalgames permet de fournir des définitions manquantes et la sélection permet une *restriction* des définitions. Lors de ces deux opérations de concaténation et de sélection, une opération de *liaison* est nécessaire. Cette liaison, fondée sur des *identificateurs* permet une résolution *à l'exécution* des références libres.

La liaison dynamique des références est un mécanisme de programmation fondamental, comme le montre les deux exemples suivants : la liaison dynamique de programmes et la programmation distribuée incrémentielle.

1. Cet exemple à été choisi en raison de sa nature didactique et non pas pour sa plausibilité biologique.

VIII.5.3.1 La liaison dynamique de programme

Les langages de programmation compilés classiques reposent sur une stratégie de compilation qui force à définir, *avant* la compilation finale d'un programme, l'ensemble des routines qui seront incorporées dans le programme. Par exemple, il est nécessaire de fournir au moment de la compilation d'un programme C l'ensemble des librairies qu'il utilise afin de pouvoir générer une unité autonome *statique*. En fait, ce n'est pas tout à fait vrai. Bien qu'il soit *toujours* nécessaire, de fournir au moment de la compilation les librairies utilisées, l'opération finale d'*édition de liens* n'inclut pas toujours les définitions des fonctions, dans le code exécutable. Un mécanisme d'édition dynamique de liens permet de *factoriser* l'utilisation de certaines librairies ; seules des *références* vers ces librairies sont générées dans le code exécutable, et ces librairies seront alors automatiquement chargées en mémoire si nécessaire par le *loader* de l'exécutable [HO91, KR93, Car97]. Cette opération de liaison dynamique des librairies, permet une certaine *évolution* des programmes, après leur écriture.

Par exemple, le remplacement de la librairie `X11` qui définit les «widgets athena» (la librairie `libXaw`) par une librairie équivalente définissant les mêmes widgets en « 3D » (la librairie `libXaw3d`) permet d'exécuter les programmes, qui faisaient appel à cette première librairie, mais avec une définition différente des widgets. L'utilisation des références libres, permet la définition similaire d'expressions *flexibles*, dont les définitions peuvent être changées au gré des évolutions et des contextes d'évaluation. Évidemment, il serait utile de fournir un système de typage, qui assure la compatibilité entre les différentes versions. Bien qu'un système de types soit toujours intéressant, on remarquera que dans le cas de l'édition de liens dynamique, aucun système de types n'intervient ; si au chargement de l'application, une erreur apparaît (comme l'absence de la librairie), celle-ci conduit à une sortie violente du programme par « *core dump* ».

VIII.5.3.2 La programmation distribuée incrémentielle

Nous avons récemment assisté à l'émergence d'une nouvelle classe d'application : les applications distribuées, dont les composants ne se trouvent pas nécessairement sur la machine qui a lancé le programme. Un exemple paradigmatique est donné par la notion d'« applet » introduite par les navigateurs [Rou96, Sun95a, Sun95b, Mee96]. Un applet est une portion de code qui est chargée dynamiquement, *à la demande*, lors de l'acquisition de données faisant référence à cet applet. Le chargement dynamique permet l'ajout de fonctionnalités, la modification du comportement d'un navigateur... Cette capacité des environnements et des langages à évoluer dynamiquement, permet d'améliorer l'expressivité : on peut toujours disposer des dernières versions des librairies qui existent et étendre les services offerts. Une implémentation naïve du mécanisme de transfert des applets serait de transférer l'ensemble du code du lieu d'où provient le document y faisant référence. Cependant, dans un réseau aussi diffus qu'est l'*Internet*, cela serait immanquablement synonyme d'inefficacité. Il est très largement préférable et plus flexible de rapatrier les seules «originalités [1]» de l'applet considéré, et d'utiliser les applets locaux ou de récupérer sur des sites proches de l'hôte les applets manquants, pour les références manquantes.

Ce mécanisme trouve une traduction immédiate avec les amalgames. Un applet est modélisé sous la forme d'un terme ouvert. Seules les contributions originales sont incorporées sous la forme d'équations, les fragments de code accessibles sur d'autres machines étant définis sous la forme de références libres. Une fois l'expression ouverte récupérée (par une quelconque opération de communication, par l'utilisation de `socket` par exemple), il suffit de la compléter en fournissant divers environnements à l'expression, jusqu'à ce que celle-ci ne comporte plus de références libres.

Par exemple, un algorithme original de tri est diffusé sous la forme d'un système 8$_{1/2}$ de nom *tri*. Le système *tri* utilise trois références libres : une référence pour permettre à l'utilisateur de fournir les éléments à trier (une collection par exemple), une référence pour que l'utilisateur fournisse une fonction de comparaison de deux éléments et enfin une fonction de hachage. Cette fonction de hachage est la seule fonction qui manque à l'utilisateur pour pouvoir utiliser l'applet. Si l'utilisateur sait que la fonction est disponible sur le site `lri.fr` par exemple, il lui suffit d'écrire l'expression (en supposant que la fonction de *tri* se trouve sur

1. On peut définir comme partie «originale» d'un applet la portion de code qui n'existait pas auparavant et qui consiste en la véritable contribution du programmeur de l'applet.

le site `evry.fr`, et que l'appel d'un applet a sur une machine m s'effectue par $a@@m$) :

$$r \quad = \quad (\texttt{lri.fr}\,\centerdot\,(\underbrace{\{data = \{1,3,4,5,2\}, cmp(x,y) = LriHashFct\}}_{A}\,\centerdot\,tri@@\texttt{evry.fr}))\,\centerdot\,result$$

La fonction de tri est explicitement référencée par l'expression tri@@`evry.fr`. Les définitions manquantes de la fonction de tri sont prises sur la machine `lri.fr` par une opération de sélection : on ne fournit aucune fonction en particulier pour spécifier que toutes les fonctions définies par `lri.fr` sont accessibles et utilisables. Enfin, on évalue l'expression A dans l'expression où la fonction de tri est définie en fournissant une équation $data = \{\ldots\}$ (qui est une référence libre de la fonction de tri), ainsi qu'une fonction de comparaison de deux éléments (qui est supposés être définie dans `lri.fr`). Le résultat du tri étant défini par $result$, il suffit d'effectuer une sélection avec cet argument.

VIII.6 Un aperçu des problèmes posés par l'évaluation des amalgames

Le processus de réduction d'un amalgame, bien qu'impliquant peu d'opérateurs, présente certaines difficultés : le règles de réduction doivent être soigneusement conçues afin d'éviter des réductions indésirables. Aussi, nous décrivons dans cette section plusieurs phénomènes qui se passent lors de la réduction d'un amalgame.

VIII.6.1 Création de scopes et références

Nous décrivons les problèmes qui surgissent lors de l'utilisation de la sélection et de la concaténation.

VIII.6.1.1 Création d'un scope avec l'opérateur de sélection.

Nous avons vu que l'évaluation d'une opération de sélection correspondait à l'évaluation de l'opérande droit dans l'environnement défini par l'opérande gauche. Mais les opérateurs explicites d'échappement de scope doivent permettre à une référence de référer une définition qui se trouve hors de l'environnement défini par l'opérande gauche. Par exemple, dans l'expression :

$$\{\underbrace{a = 1}_{A}, b = \underbrace{\{a = 2\}}_{B}\,\centerdot\,\underbrace{\{a = 3, u = a, v = a, w = a\}}_{C}\}$$

on aimerait, dans l'expression C pouvoir référer aussi bien la définition de a dans l'expression A, que dans l'expression B ou C. C'est pourquoi nous avons considéré qu'une opération de sélection *crée un scope.* L'expression que nous venons de définir, dont les définitions u, v et w font référence respectivement à la première, la seconde et la troisième définition de a s'écrit :

$$\{a = 1, b = \{a = 2\}\,\centerdot\,\{a = 3, u = a_2, v = a_1, w = a_0\}\}$$

et s'interprète comme :

$$\{a = 1, b = \{a = 2\}\,\centerdot\,\{a = 3, u = a_2, v = a_1, w = a_0\}\}$$

VIII.6.1.2 Référence dans une opération de concaténation

Un problème similaire à celui correspondant à la sélection apparaît lors de la concaténation. Il n'y a cependant pas de création de scope. Lors d'une concaténation de deux expressions, il est possible d'adopter deux stratégie de résolution de la concaténation. Soit l'expression :

$$\{\underbrace{a = 2}_{A}, b = \underbrace{\{c = a\}}_{B} \# \underbrace{\{a = 1\}}_{C}\}$$

On peut choisir un mécanisme de liaison des références qui soit *lexical* ou *tardif*. Une liaison lexicale des références conduit à lier le definiens a de B avec le definiendum a de A sans attendre la réduction de la concaténation. En effet, la définition étant présente, la liaison s'effectue et l'évaluation de l'expression est :

$$\{a = 2, b = \{c = a_1\} \# \{a = 1\}\} \quad \Rightarrow \quad \{a = 2, b = \{c = 2, a = 1\}\}$$

Une liaison tardive des références nécessite de résoudre la concaténation avant de lier la référence a de B. Par conséquent, l'expression s'évaluerait en :

$$\{a = 2, b = \{c = a^0\} \# \{a = 1\}\} \quad \Rightarrow \quad \{a = 2, b = \{c = 1, a = 1\}\}$$

Nous adoptons le mode lexical de liaison des références.

VIII.6.2 Évaluation dans un terme faisant intervenir l'opérateur de concaténation ou de sélection

Les expressions utilisant les opérateurs de concaténation et de sélection posent le problème de l'évaluation des opérandes de ces opérateurs. En effet, dans une expression de la forme $e \# e'$ ou $e \centerdot e'$, quelle est la stratégie d'évaluation à adopter pour l'évaluation de e et e' ? Il est nécessaire de tenir compte que, pour se réduire :

- dans le cas d'une concaténation, e et e' soient des systèmes,
- dans le cas d'une sélection, e soit un système.

Les systèmes, ainsi que l'opérateur de sélection, définissant des scopes, et les opérateurs de référence permettant d'échapper du scope de définition, il apparaît les notions de *localité* (le scope de définition d'une expression) et de *globalité* (les scopes éventuellement définis par un opérateur). Deux stratégies d'évaluations des sous-expressions d'une expression sont alors possibles :

1. évaluer une sous-expression complètement dans son environnement de définition puis compléter par les définitions apportées par un opérateur : c'est la stratégie global puis local,
2. évaluer localement, puis globalement.

Par exemple, l'expression suivante :

$$E \quad \equiv \quad \{a = 1, b = \{c = a\}, d = \{a = 2\}, e = b \# d\} \tag{6}$$

aura un résultat différent si on évalue l'expression liée à b dans l'environnement où a a pour valeur 1, avant de calculer e (stratégie locale, puis globale), et dans ce cas la valeur de l'expression est :

$$\begin{array}{lll} E & \to & \{a = 1, b = \{c = a_1\}, d = \{a = 2\}, e = b \# d\} \\ & \to & \{a = 1, b = \{c = 1\}, d = \{a = 2\}, e = b \# d\} \\ & \to & \{a = 1, b = \{c = 1\}, d = \{a = 2\}, e = \{c = 1\} \# \{a = 2\}\} \\ & \to & \{a = 1, b = \{c = 1\}, d = \{a = 2\}, e = \{c = 1, a = 2\}\} \end{array} \tag{7}$$

ou bien si on privilégie pour le système la définition venant de d (stratégie globale, puis locale), et où le résultat serait :

$$\begin{array}{lll} E & \to & \{a = 1, b = \{c = a\}, d = \{a = 2\}, e = \{c = a\} \# \{a = 2\}\} \\ & \to & \{a = 1, b = \{c = a_1\}, d = \{a = 2\}, e = \{c = 2, a = 2\}\} \\ & \to & \{a = 1, b = \{c = 1\}, d = \{a = 2\}, e = \{c = 2, a = 2\}\} \end{array} \tag{8}$$

La stratégie adoptée consiste à *réduire autant que possible une expression dans son environnement de définition*. Le résultat est une expression (possiblement) incomplète qui sera utilisée ailleurs et éventuellement complétée. Mais la complétion ne porte que sur ce qui était manquant dans l'environnement de départ (les références libres).

Dans l'exemple (6) précédent, l'environnement de définition fournit a à b, et en conséquence, l'environnement d'utilisation ne doit pas fournir sa propre définition pour a quand il utilise b. On choisit comme stratégie d'évaluation une stratégie locale, puis globale (soit le processus de réduction détaillé en (7)), c'est-à-dire, les expressions sont évaluées dans leur environnement de définition jusqu'à ce que l'expression ne puisse plus être évaluée, avant d'être propagée aux références.

VIII.6.3 Les problèmes de liaisons

Les amalgames permettent la définition d'expressions ouvertes contenant des références libres. La présence de telle références dans l'évaluation d'expressions et la présence dans des sous-expressions de références liées, mais qui n'ont pas encore été substituées à leur définition, va induire un certain nombre de problèmes pour conserver une liaison correcte des expressions. Nous détaillons les problèmes qui concernent plus particulièrement les *liaisons*.

VIII.6.3.1 La substitution de références liées

Une expression peut référencer une définition contenant une sous-expression impliquant une référence liée comme par exemple dans l'expression suivante :

$$\{y = 2, x = y_0, a = \{b = x_1, y = 1\}\}$$

où la définition de x implique la référence liée y ayant pour valeur 2. L'évaluation de cette expression va amener la substitution de la référence liée x_1 de b à sa définition. Cependant, il n'est pas correct d'effectuer la seule substitution littérale, c'est-à-dire sans modifier le terme. En effet, l'expression suivante (après substitution littérale) :

$$\{y = 2, x = y_0, a = \{b = y_0, y = 1\}\}$$

n'est pas correcte : les références liées, après substitution, doivent toujours être liées à leur définition initiale. La substitution correcte est bien sûr :

$$\{y = 2, x = y_0, a = \{b = y_1, y = 1\}\}$$

afin d'avoir pour résultat $\{y = 2, x = 2, a = \{b = 2, y = 1\}\}$. Nous appelons *lifting* cette opération qui incrémente l'indice de liaison d'une référence liée. Il est important de remarquer que les expressions, où apparaissent des références liées à des sous-expressions, ne doivent pas être liftées. Dans l'expression :

$$\{x = \{g = \{e = a_1\}, a = 2\}, y = \{a = 9, e = x\}\}$$

lors de la substitution du definiens de x, la référence liée a_1 ne doit pas être liftée en a_2 car celle-ci est liée à la sous-expression de x où a a pour valeur 2. Le lifting de l'expression amènerait une liaison incorrecte avec la définition $a = 9$, ce qui est évidemment erroné (ici, le lifting erroné conduirait toujours à une expression où les références liées sont liées à une définition, car a est défini dans y, ce qui est un pur hasard). La valeur voulue de l'expression est :

$$\{x = \{g = \{e = a_1\}, a = 2\}, y = \{a = 9, e = \{g = \{e = a_1\}, a = 2\}\}\}$$

Pour ce qui concerne les références libres, nous décidons de ne pas lifter les indices lors de la substitution. Par exemple, l'expression :

$$\{a = x^1, b = \{x = 1, c = \{x = 2, y = a_2\}\}\}$$

va se réduire en :

$$\{a = x^1, b = \{x = 1, c = \{x = 2, y = x_1\}\}\}$$

car la référence libre à x est injectée dans l'expression y, où une définition de x existe. La référence devient une référence liée à cette définition. L'expression se réduit ensuite en :

$$\{a = x^1, b = \{x = 1, c = \{x = 2, y = 1\}\}\}$$

alors que le lifting de la référence libre injectée aurait conduit à l'expression :

$$\{a = x^1, b = \{x = 1, c = \{x = 2, y = x^3\}\}\}$$

qui n'aurait pu se réduire plus. Cette stratégie qui ne lifte pas les références libres dans une substitution, est absolument nécessaire si l'on désire implémenter un mécanisme de *capture* de définition dans un nouveau scope. Dans l'exemple précédent, la référence x_1 capture la définition $x = 1$ dans le premier scope englobant le terme où elle est injectée par substitution. Cette stratégie est différente de la stratégie de DAMI (cf. section IX.3.3.1, page 116), qui lifte les références libres. En effet, dans cette dernière approche, la liaison de références libres ne se fait pas par capture mais à l'aide d'une forme dédiée d'application.

VIII.6.3.2 Liaisons dans un terme substitué à une référence libre

Nous avons vu précédemment que les références permettaient d'atteindre une expression qui se situait en dehors de leurs scopes. Nous avons aussi vu que l'évaluation de la sélection nécessitait d'avoir comme opérande gauche un système. Néanmoins, il se peut que l'évaluation d'une sélection conduise à substituer à une référence son definiens comme dans l'expression suivante :

$$\{a = x^0, b = \{x = 1\} \centerdot (y^0 \centerdot a_2)\}$$

La substitution de la référence liée a_2 par sa définition est souhaitable car ce n'est pas la référence libre y^0 ni le système $\{x = 1\}$ qui peuvent influencer la référence liée. En effet, celle-ci est liée «par delà» l'expression de sélection. Il n'est cependant pas possible de lier la définition substituée x^0 en x_1 en utilisant la définition du système. En effet, dans l'expression :

$$\{a = x^0, b = \{x = 1\} \centerdot (y^0 \centerdot x^0)\}$$

il se peut que la référence libre y^0, lors de sa future liaison, définisse une valeur pour x, capturant ainsi la référence libre. Par contre, dans l'exemple suivant (qui est une modification du précédent exemple où la référence libre y^0 est remplacée par un système quelconque ne définissant pas de x) :

$$\{a = x^0, b = \{x = 1\} \centerdot (\{\dots\} \centerdot x^0)\}$$

alors la référence libre x^0 peut être liée en x_1 car nous sommes assurés qu'aucune autre définition de x ne peut survenir pour capturer x^0. Il apparaît que les liaisons qui doivent survenir au moment d'une substitution d'une référence par son definiens, en particulier dans une opération de sélection, sont différentes selon que les termes *nécessaires* sont ou non présents.

VIII.6.3.3 Liaisons dans un terme substitué à une référence liée

Un problème similaire à celui que nous venons d'évoquer survient quand une référence liée se trouve impliquée dans une sélection. Suivant le moment où la substitution de la référence liée par sa définition a lieu, des résultats différents peuvent être obtenus, pour une même expression. Par exemple, dans l'expression suivante :

$$\{pb = x_0 \centerdot \{c = 1\} \centerdot a^0 \centerdot b^0, x = \{u = \{\} \centerdot v_1, v = w_0, w = 1, a = \{b = c^0\}\}\}$$

on peut décider de contracter la sous-expression $(x_0 \centerdot \{c = 1\}) \centerdot (a^0 \centerdot b^0)$ avant d'avoir propagé le definiens de la référence liée x_0. Si on effectue l'évaluation de la sélection $\{c = 1\} \centerdot (a^0 \centerdot c^0)$, on obtient l'expression :

$$x_0 \centerdot (a^0 \centerdot c^0)$$

Cette évaluation est *localement* correcte car on est sûr qu'aucune définition pour les références libres a^0 et c^0 ne peut advenir, car l'expression à gauche du point, le système $\{c = 1\}$, ne fournira pas de définitions. Cette simplification a cependant eu lieu *trop tôt* : la référence liée x_0 n'a pas été substituée par le definiens de sa référence alors que le definiens allait justement apporter une définition pour a. Il est donc nécessaire de *retarder* l'évaluation afin d'obtenir les propagations correctes des definiens aux références liées pour obtenir le résultat souhaité, qui est dans ce cas l'expression :

$$\{pb = 1, x = \{u = 1, v = 1, w = 1, a = \{b = c^0\}\}\}$$

VIII.6.3.4 Liaisons induites par une substitution

Nous venons de voir que la substitution d'une référence à sa définition pouvait nécessiter la liaison de l'expression substituée. Il se peut aussi que la substitution d'une référence conduise à la liaison d'une expression *en dehors* de l'expression substituée. Par exemple, dans l'expression :

$$\{a = \{x = 1\}, b = \{\dots\} \centerdot (a_1 \centerdot x^0)\}$$

lors de la substitution de a_1 par sa définition, il sera nécessaire de transformer la référence libre x^0 en référence liée x_0 car la substitution fournit une définition de x accessible pour la référence libre.

Le même phénomène se produit lors de l'évaluation de deux systèmes. Par exemple, dans l'expression :

$$(\{x = 1\} \# \{y = 2\}) \mathbin{.} x^0 \tag{9}$$

au moment où la concaténation des deux systèmes va se réduire en un seul système, il sera nécessaire de lier x^0 afin que celui ci réfère la définition $x = 1$. De façon plus générale, il est nécessaire de lier une expression à chaque fois qu'un système est créé. Par exemple, dans les deux expressions suivantes :

$$\{a = 1\} \mathbin{.} (\{b = a^0\} \# \{\ldots\}) \tag{10}$$
$$\{a = 1, c = \{b = a^0\} \# \{\ldots\}\} \tag{11}$$

il est nécessaire, après évaluation des systèmes, de lier à chaque fois a^0 en a_1.

L'expression (9) induit une liaison à l'extérieur du terme qui est substitué alors que dans les expressions (10) et (11) ce sont des substitution extérieures au terme qui nécessitent des liaisons.

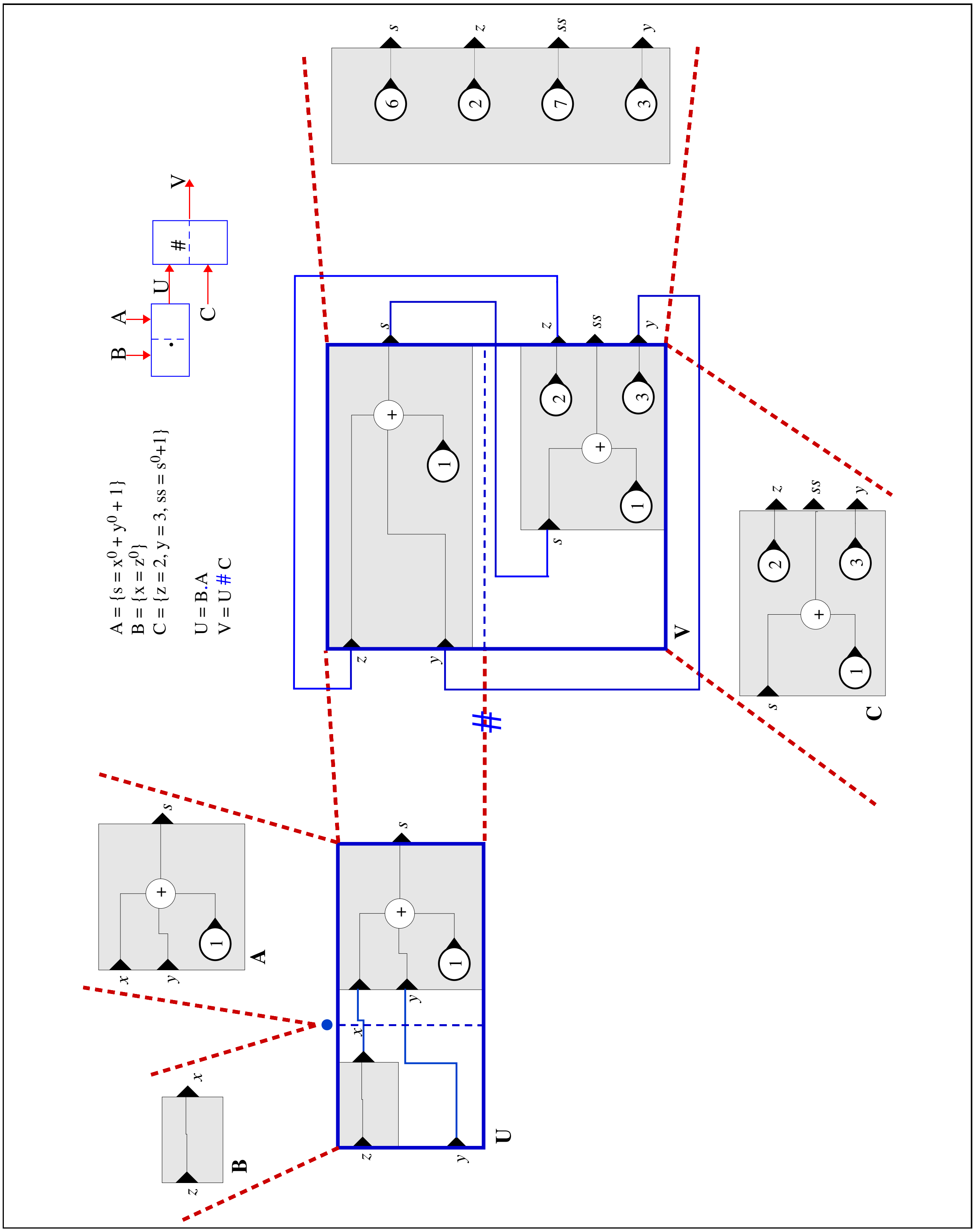


FIG. 9 –: *Construction d'une expression comme le calcul de sous-expressions en utilisant les amalgames. Il faut que les valeurs qui transitent sur un arc soient des systèmes, à savoir des DFG. On a donc « élargi » les arcs pour laisser apparaître la structure des valeurs manipulées. L'objet présenté est un graphe de graphes.*

Chapitre IX

Les amalgames et les formalismes existants

Dans ce chapitre, nous allons comparer les amalgames à d'autres formalismes et mécanismes de programmation. Du point de vue des mécanismes de programmation, les amalgames permettent :

- la définition d'une notion d'agrégat de données accessibles par un nom ;
- une extension de l'agrégat (rajouter des données) ;
- une restriction de l'agrégat (extraire un sous-ensemble de données) ;
- des définitions récursives ;
- des redéfinitions de noms et un accès aux multiples définitions d'un même nom ;
- une portée lexicale et dynamique des noms ;
- des expressions ouvertes, c'est-à-dire comportant des références libres ;
- enfin, des mécanismes de liaison des références libres.

On retrouve des préoccupations similaires à celles des amalgames dans de nombreux domaines :

- dans les *langages orientés objets* et la modélisation des notions de classes, d'instances, d'héritage... à travers la notion d'enregistrement (cf. section IX.2) ;
- dans les *théories de noms* développées pour modéliser des schémas dynamiques de liaison, la communication entre processus, la curryfication des arguments dans un ordre quelconque... (cf. section IX.3) ;
- dans la *programmation incrémentielle* qui définit la notion d'*enrichissement* de programmes par ajouts successifs (cf. section IX.4) ;
- dans les *langages réflexifs* qui essayent de représenter à travers des *structures de données* des mécanismes d'évaluation du langage, comme par exemple, la notion d'*environnement* qui permet la liaison variable $\leftrightarrow$ valeur (cf. section IX.5) ;

Nous allons comparer l'approche des amalgames avec divers formalismes et mécanismes de programmation qui ont été proposées dans ces domaines. Mais auparavant, nous allons effectuer un rappel sur les modes de liaisons des identificateurs dans divers langages de programmation.

IX.1 Les modes de liaison des identificateurs

Tous les langages ayant une notion de variable définissent des règles de portée des identificateurs. Les langages classiques ne permettant généralement que la définition d'expressions closes (toutes les variables réfèrent à une définition), la spécification des règles de portée suffisent à définir les règles de liaisons.

Dans les langages récent comme Pascal, Scheme [RCe+92], ML, Ada... on a privilégié une gestion lexicale de la portée des variables, aux dépends d'une gestion dynamique. Cela implique que toute variable utilisée réfère à une définition localisée syntaxiquement dans le texte du programme. Il n'y a donc pas d'occurrence de variable non définie dans une expression. Par exemple, l'expression `let f x = x+a` en ML, où `a` n'est pas défini auparavant, provoquera une erreur.

Ce n'est pas le cas pour les langages à liaison dynamique (par exemple Frantz Lisp) ou bien pour les langages de manipulation symbolique (comme Mathematica [Wol88] ou Mapple [CGG+91] par exemple) pour lesquels cette expression est correcte et où `a` prendra sa valeur au moment de son utilisation. Par exemple, dans un langage comme Frantz Lisp, l'expression `(let ((a 5)) (f 10))` aura pour valeur `15` car l'évaluation de `(f 10)` a lieu dans un environnement où `a` est défini.

Cependant, la liaison des variables dans ce type de langage est complètement dynamique : ainsi, en Mathematica, si on a :

```
y = 1;
Foo[x_] := x + y + a
```

alors :

```
a = 1; y = 666; Foo[10]
```

s'évaluera en `677`. Ce comportement est à opposer au comportement de Common Lisp [Ste82], où une définition analogue provoquera la liaison lexicale de `y` et une liaison *dynamique* de `a`. Ainsi, en Common Lisp, la définition :

```
(let ((y 1))  (defun Foo (x) (+ x (+ y a))))
```

suivi de l'évaluation :

```
(progn (setq a 1) (setq y 666) (Foo 10))
```

aura pour valeur `12` (la variable `y` apparaissant dans l'expression de `Foo` étant liée syntaxiquement, la redéfinition de `y` par `(setq y 666)` n'a pas d'effet sur l'évaluation). Remarquons que la construction `let ... in ...` utilisée pour la liaison syntaxique n'est pas la même que celle utilisée pour la liaison dynamique : `setq`. La structure du lieur n'est d'ailleurs pas la même car la liaison syntaxique reflète la portée par bloc dans le texte du programme, alors que la liaison effectuée par un `setq` est valide dans le temps jusqu'au prochain `setq`. Une autre différence est qu'un identificateur dans un lieur à portée syntaxique est une *variable muette*, c'est à dire sujette à α-conversion, alors que dans un lieur dynamique le nom est pertinent, ce qui permet la liaison « plus tard », indépendamment de toute construction explicite reliant l'endroit de définition et l'endroit d'utilisation de l'identificateur. Si ce mode de gestion des noms admet une sémantique opérationnelle relativement simple à décrire, il a fallu attendre des travaux récents pour obtenir une théorie équationelle d'un mécanisme de capture de noms [Dam94c, Dam94a, Gar95, LF96].

En Common Lisp, l'évaluation de `(Foo 10)` provoquera une erreur si `a` n'est pas liée au moment de son évaluation. Les langages de manipulation symbolique ne provoquent pas d'erreur dans ce cas, et retournent comme résultat de l'évaluation, l'expression symbolique où les valeurs connues ont été substituées aux variables liées (dynamiquement) avant une éventuelle simplification. Par exemple, en Mathematica, si on définit `Bar[x_] := x+a`, alors l'expression :

```
Bar[10]
```

dans un contexte où `a` a pour valeur `5` s'évaluera en `15` (simplification de `10+5`) alors que `bar[10]` s'évaluera en `10+a` s'il n'existe aucune définition pour `a`. Dans ce langage, `10+a` est une valeur au même titre que `15`. La table 1 dresse une rapide classification des modes de liaison.

Nous avons adopté pour les amalgames une stratégie analogue à celle de Common Lisp pour la résolution des références. Cependant, nous ne voulons pas distinguer références liées syntaxiquement et références liées

Moment de la liaison	Type de liaison	Langage
à la définition	lexicale	ML
	lexicale retardée	méthode virtuelle C++
à l'évaluation	dynamique	Franz Lisp (erreur si pas de définition associée)
	symbolique	Mathematica, Mapple, (toujours une valeur)

TAB. 1 –: *Les différents modes de liaison des variables*

dynamiquement : en effet, nous ne voulons pas redéfinir de liaisons au cours du temps, mais simplement retarder le moment de cette liaison. Il n'est donc pas nécessaire de distinguer deux lieurs, mais simplement de permettre des expressions non closes. Ainsi le langage garde la propriété d'assignation unique (modulo la visibilité des identificateurs). De plus, une expression ouverte est une valeur admissible et donc, comme dans le cas des langages de manipulations symboliques, l'évaluation d'une expression ouverte ne provoque pas d'erreur.

IX.2 Enregistrements et langages orientés objets

Le modèle de programmation des langages orientés objets est caractérisé principalement par le traitement uniforme des programmes et des données sous la forme d'*objets* qui maintiennent *localement* les informations sur leur état. Les programmes sont conçus comme des ensembles d'objets qui interagissent entre eux par passage de messages.

La définition d'un cadre formel pour la programmation orientée objet est l'origine de nombreux travaux : cela fait de nombreuses années que des langages orientés objets existent et sont utilisés sans disposer de cadre formel, en particulier pour le typage (simula67 [DMN68], Smalltalk [GR83], C++ [cpl95, SE94, Str94] par exemple). Citons en particulier [GG94] :

« Contrairement à la programmation fonctionnelle, fondée sur le λ-calcul, ou la programmation logique, fondée sur la logique, il manque un modèle simple à la programmation orientée objet que l'on puisse utiliser comme base pour [une] définition et [des] discussions. »

Aucun des langages ou formalismes théoriquements bien fondés ne disposent de constructions permettant de définir des fonctionnalités objets. La modélisation des fonctionnalités des programmes orientés objets (encapsulation, héritage, polymorphisme...) a amené la définition de nouveaux opérateurs pour des langages existants et l'introduction de nouvelles fonctionnalités dans des formalismes existants.

IX.2.1 Le point de vue « objets comme enregistrements » de Cardelli

Les langages orientés objets possèdent tous certaines propriétés, mais aucune propriété particulière ne semble nécessaire à la qualification d'« orienté objet » pour un langage (on y trouve les notions d'*encapsulation*, d'*héritage*, de *messages*, de *classes*... on trouvera une classification dans [Weg87]). Dans [Car84], CARDELLI suggère que la notion d'héritage est la seule notion qui permet de définir un langage comme un langage orienté objet. Il considère la structure de données « enregistrement » des langages fonctionnels comme un objet des langages orientés objets. Un enregistrement est un ensemble fini d'associations label $\leftrightarrow$ valeur [1]. L'opérateur de sélection « `.` » permet d'accéder à la valeur d'un item d'un enregistrement. Une notion de *soi* (`self` dans les langages orientés objets) est définie à l'aide de la construction `rec` utilisée dans la définition d'un enregistrement. Cette notion de soi permet à un objet de modifier la valeur de certains de ses propres items.

Désirant capturer la notion d'objet et d'héritage des langages orientés objets, CARDELLI définit un type et une relation de sous-typage : si un enregistrement a est de type τ et $\tau \leq \tau'$ alors a est aussi de type τ' [2]).

1. Dans la définition de CARDELLI, on note les enregistrements entre accolades ; afin de ne pas établir de confusion avec la notation des systèmes, on ré-utilisera la notation de la précédente section : l'enregistrement avec deux items a et b de valeur 1 et `true` se notera $\{\!|\, a = 1, b = \texttt{true} \,|\!\}$.

2. Intuitivement, un enregistrement d'un type τ est un sous-type d'un enregistrement de type τ' si τ a au moins tous les items de τ' (et peut-être des items supplémentaires), et les items commun de τ et τ' sont en relation de sous-typage [Car84, pp

Le type d'un enregistrement est l'ensemble des types de ses éléments, ensemble où chaque type est précédé du label de l'item dont il est le type. Deux opérateurs permettent l'enrichissement et l'appauvrissement de type : `and` et `ignoring`. CARDELLI définit un système de typage pour son système et montre qu'il est sûr. Ce système de typage a été implémenté dans le langage Amber [Car85].

IX.2.1.1 Extension et restriction sur les enregistrements.

Les opérateurs `and` et `ignoring` permettent de contraindre des types. Dans [CM91], CARDELLI et MITCHELL définissent un ensemble d'opérations supplémentaires sur les enregistrements, l'*extension* (notée $\{\!|\, r|x = a \,|\!\}$ pour un enregistrement ajoutant un item labellé x et de valeur a, pourvu que x ne soit pas déjà défini dans r) et la *restriction* (notée $r\backslash x$ pour l'enregistrement r privé de l'item labellé x), en s'intéressant toujours au typage des opérations. Un système de types du second ordre est défini [1] (qui n'a pas la propriété de *type principal*). Ces deux opérateurs permettent de modéliser la surcharge (par la restriction d'un item et la définition d'un item avec le même label mais une nouvelle valeur) et le renommage (par la restriction d'un item et la définition d'un item avec un label différent mais une valeur identique). La définition d'enregistrements se fait à partir de l'enregistrement vide, puis par concaténation. Ce système de types permet la vérification mais pas l'inférence de types.

Les systèmes de CARDELLI et MITCHELL ne permettent pas la concaténation d'enregistrements car, dans leur système de types, la concaténation est en conflit avec la règle de subsomption :

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

En effet, lors de la concaténation de deux enregistrements a et b définissant tous deux un item c de type différent dans a (`Int` par exemple) et b (`Bool` par exemple), quel est le type de $a \parallel b$ (l'expression $a \parallel b$ représente la concaténation des enregistrements a et b) ? Le traitement classique de ce problème consiste à écraser les définitions dans un sens particulier (gauche-droite par exemple), cette solution n'est pas satisfaisante. Le lecteur intéressé par cette problématique se reportera à [CM91, pp 46] : la solution choisie consiste à ne permettre aucune redéfinition d'items dans la concaténation d'enregistrements, mais elle implique le changement du système de typage.

IX.2.2 Les enregistrements de Rémy

RÉMY [Rém92]) montre, et implémente dans une version de ML, qu'on obtient la concaténation d'enregistrements à partir du moment où on dispose de l'extension d'enregistrements. Le but recherché est toujours de fournir un système de types pour les opérations sur les enregistrements afin de mimer les constructions des langages orientés objets. D'autres propositions [Wan87, Wan93] ont été formulées, se concentrant toujours sur le typage des expressions.

IX.2.3 Les enregistrements dans les langages SML et Haskell

Nous choisissons d'étudier la façon dont deux représentants des langages fonctionnels (un strict et l'autre non-strict) permettent la manipulation des enregistrements. Nous étudions d'abord le langage strict, puis nous verrons le langage non-strict.

IX.2.3.1 Dans SML

Les enregistrements définis en SML [Pau92, Har93] sont *flexibles* : il n'est pas nécessaire de définir le type de l'enregistrement préalablement à sa définition (contrairement au langage CAML par exemple, où les enregistrements doivent avoir leur type préalablement déclaré). La définition de l'objet définit en même

4].

1. WAND [Wan87] propose une alternative à ces deux opérateurs sous la forme d'un seul opérateur qui met à jour l'item d'un enregistrement s'il existe déjà, ou bien ajoute un nouvel item s'il n'existe pas.

temps son type (pour décrire des fragments de programme, on utilise la syntaxe du langage décrit : il ne faut pas confondre ces constructions avec des opérateur 81/2) :

`val x = { a = 1, b = 2 }` ⇒ `val x = {a=1,b=2} : {a:int, b:int}`

On accède aux items par le label en position fonctionnelle (par exemple, pour accéder à l'item `a` de l'enregistrement `x` défini ci-dessus : `#a x` ⇒ `val it = 1 : int`). Les items peuvent aussi être étiquetés par un indice *numérique* (par exemple, `#1 {1=10, 2=20}` ⇒ `val it = 10 : int`). L'indice numérique étant considéré comme une constante, il est possible d'utiliser librement toutes les combinaisons de labels alphabétiques et numériques. Il n'est pas possible de définir une fonction «polymorphe-sur-les-enregistrements» (par exemple une fonction qui s'applique à tous les enregistrements qui ont un item étiqueté «a») ni de spécifier comme label d'un item le résultat d'un calcul (par exemple, l'expression `#(1+1) x` n'est pas autorisée).

IX.2.3.2 Dans Haskell

En Haskell [HF92], la notion de produit cartésien avec label n'est pas définie en tant que telle, mais il est possible de nommer par un label symbolique un argument d'un constructeur de type algébrique. La présence de ces noms symboliques crée implicitement des fonctions qui permettent l'accès à cet argument ce qui permet d'émuler un mécanisme de gestion des enregistrements. Par exemple, l'expression `data C = F {f :: Int, g :: Bool}` définit un type de données `C`, avec un unique constructeur `F` qui utilise deux arguments labellés par `f` et `g`. Une valeur de type `C` peut se construire à l'aide du constructeur `F`, de façon classique : `F 5 true`, ou bien en faisant explicitement référence aux labels : `F {g = True, f = 5}` (on remarquera que l'ordre de spécification des items est indifférent). Comme les constructeurs en Haskell ne sont pas stricts, il est possible de construire des enregistrements ayant une valeur indéfinie, en omettant la définition d'un item au moment de la construction : par exemple, `F 5`, construit l'enregistrement `F {f = 5, g = Undefined}`.

IX.2.4 Comparaison entre la notion d'enregistrement et la notion d'amalgame en 81/2

Il est tentant d'assimiler les systèmes 81/2 à des enregistrements. Chaque définition $id = exp$ d'un système correspond à la spécification d'un item de nom a de l'enregistrement. L'imbrication possible des systèmes correspond au fait que la valeur d'un item peut être un enregistrement.

Cependant, il n'est pas possible de définir un enregistrement *récursif*, en SML, telle que la valeur de l'item soit égale à « soi-même » : `val x =` {| `f = x` |}. L'utilisation de traits impératifs (les valeurs « mutables ») permet de lever cette restriction mais de façon ad hoc c'est-à-dire en sortant du modèle de calcul. Le langage Haskell, par son traitement paresseux des structures de données, permet une telle définition. Par exemple, l'expression

```
> data F = C { g :: F};
g :: F -> F
data F = C { g :: F }
```

spécifie un type algébrique `F` avec un seul constructeur `C` dont l'argument est labellé par `g`. Un observateur qui a le même nom que le label est implicitement créé et la valeur de l'item `g` est de type `F` :

```
> let x = C { g = x};
x :: F
```

La variable `x` est de type `F` et l'item `g` de `x` a pour valeur `x` lui-même. Une telle expression est aussi définissable à l'aide des amalgames : $x = \{g = x\}$.

Une autre différence essentielle entre un système et un enregistrement est que les labels d'un enregistrement n'agissent jamais comme une variable alors que les labels d'un système agissent comme des références. Par exemple, dans l'expression $\{a = 1, b = a\}$, le a de $b = a$ fait référence à la constante 1, alors que dans l'enregistrement $\{\!| a = 1, b = a |\!\}$, a fait référence à une variable a définie dans un contexte englobant. On peut obtenir un comportement similaire en liant l'enregistrement à une variable X et en remplaçant toutes les références à l'item a d'un système X par l'expression $X \,.\, a$. Dans ce cas, il est impératif de se placer dans un modèle de calcul similaire à celui de Haskell, à cause des références récursives.

On peut comparer les mécanismes d'extensions d'enregistrements et la concaténation de systèmes : les deux mécanismes étendent les items de l'enregistrement et du système et permettent l'accès à de nouvelles définitions.

On peut aussi comparer l'accès à un item. La sélection de la valeur d'un item peut être décrite par l'opérateur de sélection des amalgames. Le comportement est cependant différent car le point considère que l'opérande de gauche d'une sélection est un environnement, permettant l'évaluation d'une *expression* en partie droite. Ce comportement peut tout de même être rendu avec des enregistrements en transformant une expression $\{\ldots, id = e, \ldots\} \,.\, (e' + e'')$ par une *distribution* du système sur les termes de l'opérande droit du point : $(\{\!| \ldots, id = e, \ldots |\!\} \,.\, e') + (\{\!| \ldots, id = e, \ldots |\!\} \,.\, e'')$. Si la règle de distribution est évidente pour la distribution de l'évaluation par rapport à une opération arithmétique, cette distribution est plus délicate quand l'opération porte sur l'évaluation elle-même (comme c'est le cas avec les opérateurs de concaténation et de sélection). Par exemple, faut-il admettre une règle de la forme :

$$s \,.\, (e \# e') \quad \equiv \quad (s \,.\, e) \# (s \,.\, e')$$

sachant qu'une telle règle ne permettra de liaisons locales entre les éléments de e et e' qu'après les liaisons impliquées par s ?

Les travaux qui concernent les enregistrements sont essentiellement dirigés par la préoccupation de *typer* statiquement les expressions. Le typage des enregistrements où le label de sélection est le résultat d'un calcul interdit un typage statique des expressions (par exemple, on ne sait pas typer actuellement une expression de la forme $\lambda x\, y.(x \,.\, y)$ où le y de sélection est lié avec le paramètre formel y). Notre préoccupation actuelle n'est pas tant de typer les termes des amalgames que de définir leur réduction. De ce point de vue, les mécanismes proposés par les langages objets posent peu de problèmes : pas de définition récursives (sauf pour Haskell qui répond par une stratégie d'évaluation paresseuse), pas d'assimilation label $\leftrightarrow$ variable, pas d'extension de valeurs (sauf pour les systèmes proposés par CARDELLI, RÉMY ou WAND pour lesquels les autres opérations sont restreintes afin que le type reste inférable), pas de sélection calculée. Or ce sont justement ces mécanismes de calcul hautement dynamiques qui nous intéressent dans les amalgames.

IX.3 Les théories de noms

La gestion des espaces de noms est un problème crucial pour les nouveaux langages de programmation. Par exemple les problèmes de modélisation de l'*héritage* dans les langages à objets [Rém94], de la *délégation* dans les langages à prototypes [US87], des *pages jaunes* ou autres mécanismes de nommage dans un environnement distribué [Que96], des *applets* dans les programmes construits dynamiquement [Rou96], des mécanismes de *call-back*, etc., sont tous clairement reliés à la gestion des noms. La seule opération que l'on peut faire sur un nom est de tester son égalité avec un autre nom [Ode93, Ode94].

La notion de variable, par exemple dans le λ-calcul, est bâtit sur la notion de nom. Mais une variable du λ-calcul doit être muette : la réduction des termes du λ-calcul est compliquée par cette problématique. En effet, deux variables de même nom ne doivent pas entrer en collision sous risque de voir apparaître le phénomène de *capture de variables* tant redouté.

La notion de référence dans les amalgames est aussi bâtit sur la notion de nom. Les éléments d'un système sont identifiés par des noms et deux références de même nom référencent la même expression, si elles sont dans le même scope et si elles ont le même échappement.

Nous présentons dans cette sections des travaux qui ont été mené autour de cette notion de nom et de son éventuelle interaction avec la notion de variable. Nous détaillons la notion de nom dans la formalisation des processus communicants de MILNER(cf. section IX.3.1) ainsi que deux introductions dans le λ-calcul de la notion de nom : le λN-calcul (cf. section IX.3.3.1) et le λ-calcul avec labels (cf. section IX.3.3.2).

IX.3.1 Le π-calcul de Milner

Le π-calcul de Milner est un modèle de calcul concurrent [Mil93, MPW92]. Aucune distinction n'est faite entre les notions de constante et de variable : il ne subsiste que la notion de *nom*[1]. Cette notion est indispensable à la modélisation des communications concurrentes : les liens de communication sont identifiés par des noms et le calcul est représenté comme la circulation de noms le long de canaux. Le π-calcul introduit une notion de *migration* qui facilite la création et la visibilité des noms. À l'origine, le π-calcul était un calcul du premier ordre (seuls des noms circulent sur les canaux). Dans [Mil91], une extension du π-calcul est proposée, où des tuples de noms peuvent circuler sur les canaux et non plus uniquement les noms seuls. Dans [San93], est présentée une extension d'ordre supérieur du π-calcul qui permet la circulation de processus sur les canaux. Nierstrasz [Nie91, Nie93] étend le π-calcul de Milner en ajoutant la possibilité de 1) communiquer des tuples, 2) d'effectuer des applications fonctionnelle, 3) de communiquer des expressions.

Le π-calcul est une extension de CCS [Mil80]. Les principales différences sont 1) les noms de canaux sont des valeurs transmissibles, 2) il est possible de générer de nouveaux canaux. L'élément de base du π-calcul est le nom. Il représente un *canal*, canal par lequel il sera possible de faire transiter des *messages*. Les messages du π-calcul sont des noms de canaux qui remplissent principalement les deux rôles suivants : 1) servir de nom de canal sur lequel pourra être envoyé un nouveau message, 2) servir à déclencher l'émission d'un message. Une grammaire restreinte des termes du π-calcul est (cf. [Ama96]) :

$$P ::= \mathbf{0} \mid \overline{a}b.P \mid a(b).P \mid \nu a.P \mid (P_1 \parallel P_2) \mid \gamma_1.P_1 + \gamma_2 P_2 \mid !P$$

avec l'interprétation suivante :

- $\mathbf{0}$ est le processus qui est terminé. On omet $\mathbf{0}$ dans $\overline{a}b.\mathbf{0}$ que l'on simplifie en $\overline{a}b$;
- $\overline{a}b.P$ est le processus qui émet b sur le canal a puis devient P ;
- $a(b).P$ est le processus qui lit sur le canal a un nom et qui devient P dans lequel b à pour valeur le nom lu sur le canal a ;
- $\nu a.P$ est le processus qui crée un nom unique puis devient P dans lequel a réfère ce nom unique. Ce mécanisme permet la restriction de la portée d'un nom par la création d'un nouveau nom « frais » local à une expression : $\nu c.(\dots)$ crée un nom c « frais » qui ne sera définis que dans la portée du $(\dots)$ et où une référence à c en dehors de cette portée réfère a un autre nom ;
- $(P_1 \parallel P_2)$ est la composition parallèle des processus P_1 et P_2 ;
- $\gamma_1.P_1 + \gamma_2.P_2$ est la « somme gardée » où seul un des deux processus P_1 et P_2 s'exécute même si les deux gardes sont valides (les gardes γ_i représentent une émission ou une réception de messages et sont donc de la forme $\overline{a}b$ ou $(a)b)$;
- $!P$ définit la réplication du processus P autant de fois que cela est désiré ; $!P$ est équivalent à $P \parallel \cdots \parallel P \parallel !P$.

Le π-calcul veut capturer la notion de communication inter-processus. Un processus est une entité qui est en attente sur un canal d'une information et qui, en fonction de l'information reçue sur un canal va émettre vers un autre processus une information. Un exemple classique du π-calcul est la communication entre une base et des agents mobiles. Les agents mobiles, ne connaissant que la base peuvent communiquer entres eux via celle-ci ou bien ils peuvent communiquer directement entre eux via la création dynamique d'un canal.

Soient 3 processus B, P_1 et P_2 avec un canal de communication entre B et P_1 connu de ces deux processus et de nom $C_{P_1 \leftrightarrow B}$ et entre B et P_2 de nom $C_{P_2 \leftrightarrow B}$. Les processus P_1 et P_2 qui n'ont pas connaissance l'un de l'autre, ni de canal de communication entre eux peuvent néanmoins s'échanger un message :

$$\nu CANAL.(\underbrace{\overline{C_{P_2 \leftrightarrow B}}\ CANAL.\overline{CANAL}.X}_{P_1}) \parallel \underbrace{C_{P_2 \leftrightarrow B}(c).\overline{C_{P_1 \leftrightarrow B}}\ Z}_{B} \parallel \underbrace{C_{P_1 \leftrightarrow B}(c).c(m).Y}_{P_2}$$

1. La notion de nom est essentielle pour Milner, citons « *The pervasive notion we seize upon is* naming. » dans [Mil91, pp 2]

Le nom du canal entre P_1 et P_2 est local à l'exemple et est échangé entre le processus P_2 et la base B puis entre la base B et le processus P_1. Une fois le nom du canal émis par P_2, celui ci émet l'information, ici X sur le nouveau canal, que reçoit P_1 une fois le nom du canal reçu via la base.

Nous pouvons voir le π-calcul comme l'importation de noms (grâce à la réception) et l'exportation de noms (grâce à l'émission). Un objet peut être vu comme un terme qui attend des définitions sur les canaux correspondant à ses variables libres et qui émet des définitions sur les canaux nommés par ses variables liées. Par ailleurs, Milner a montré la possibilité d'un codage du λ-calcul avec une stratégie de réduction paresseuse en π-calcul [Mil91, Mil92].

IX.3.2 Comparaison avec 81/2

On peut voir l'occurrence d'une référence liée comme un canal de communication qui attend une valeur provenant de la définition référée. C'est le point de vue de Boudol [Bou89] dans son γ-calcul : une extension du λ-calcul basée sur le principe que la β-réduction est une *communication* entre une λ-abstraction réceptrice et un opérande émis le long d'un canal unique nommé λ.

D'un point de vue analogue, un système $\{a = e, b = e'\}$ est un ensemble de processus parallèles. Un processus élémentaire $a = e$ correspond au calcul de la valeur de e et à sa diffusion sur le canal de nom a.

Est-ce que les amalgames peuvent se réduire au π-calcul ? On peut essayer de traduire un terme 81/2 contenant une référence libre comme un processus en attente de la valeur de cette référence. Par exemple, une traduction de $(a^0 + 1)$ est :

$$a(r_a).r_a + 1$$

avec a le nom du canal associé à a^0 dans la traduction. Une expression 81/2 contenant plusieurs références libres peut lier les références dans un ordre quelconque, ce qui induit l'utilisation d'une somme gardée. Par exemple, une expression e qui dépend des références libres a^0 et b^0 se traduit en :

$$(a(r_a).b(r_b).T[e] + b(r_b).a(r_a).T[e])$$

où $T[e]$ correspond à la traduction de e en π-calcul.

Pour la traduction de la concaténation, on peut s'appuyer sur le fait que l'expression :

$$\{\dots, a = e, \dots\} \# \{\dots, b = f, \dots\}$$

s'évalue en $\{\dots, a = e, \dots, b = f, \dots\}$ qui est une composition parallèle. Donc, l'expression $a \# b$ devrait se traduire par une expression de la forme $a \parallel b$.

La sélection, $e \,.\, e'$ constitue en fait une évaluation de e' dans les liaisons fournies par e, les liaisons de e étant oubliées dans le résultat. Une traduction pourrait être :

$$\nu\, dom(e).(T[e] \parallel T[e'])$$

où *dom* représente la liste des liaisons de e.

Cette esquisse de traduction passe pudiquement sous silence le problème de portée des définitions, en particulier la gestion des indices des références libres et liées. Par ailleurs, le codage des amalgames en π-calcul nécessite la possibilité de transmettre des expressions sur un canal, car la valeur d'une définition peut être un terme complexe et ne se réduit pas à un référence libre. Par suite, le codage des amalgames en π-calcul nécessite les extensions d'ordre supérieur du π-calcul. Il n'est pas clair du tout que la stratégie de liaison des noms de canaux permette le codage de la capture d'une référence libre et en particulier, en présence de la définition de noms locaux par l'opérateur ν. Cependant, cette direction de recherche mériterait certainement d'être approfondie.

IX.3.3 Les extension du λ-calcul

IX.3.3.1 L'ajout d'indices et de noms dans le λ-calcul : le λN-calcul

Dami [Dam94c] désire capturer les traits caractéristiques des langages objets. Il propose une extension du λ-calcul avec indices et noms. Il est possible d'abstraire sur plusieurs arguments au même niveau d'abstraction (un seul λ), les arguments sont alors distingués par leur nom, l'ordre n'important pas. Les indices sont une introduction dans le λ-calcul de la notation de de BRUIJN [dB72] utilisée originellement pour résoudre les problèmes d'α-conversion dans la réduction d'un terme. DAMI utilise des indices pour permettre l'accès a des noms qui auraient été redéfinis dans une abstraction : une variable est un couple (nom, index), l'index indiquant à quelle abstraction (quel « λ ») une variable fait référence (0 étant l'abstraction la plus proche).

À partir du λN-calcul, DAMI [Dam94a] propose des constructions de haut niveaux qui permettent, entres autres, la liaison dynamique et les enregistrements extensibles. À l'abstraction $\lambda x.e$ du λ-calcul, DAMI ajoute la construction $\lambda(x\,y).e$ qui est un objet qui attend deux arguments qui peuvent être fournis dans un ordre quelconque. Ainsi, les expressions $\lambda(x\,y).x + y$ et $\lambda(y\,x).x + y$ sont équivalentes. On généralise à un nombre quelconque d'arguments (y compris nul). Un argument est fourni à ce nouveau type d'abstraction grâce à un nouveau type d'application :

$$(\lambda(x\,y).x + y)(y \rightarrow 2)$$

qui s'évalue en

$$(\lambda(x).x + 2) \tag{1}$$

Les arguments d'une telle abstraction ne sont pas α-convertibles (ce sont des *noms* et pas des *variables* du λ-calcul). Les dernières parenthèses qui vont subsister après la consommation du nom x de l'expression (1) sont retirées par une opération explicite de fermeture (« *close* » chez DAMI) :

$$(\lambda().5)!$$

qui s'évalue en 5.

Un enregistrement est codé suivant la méthode classique en λ-calcul (cf. [Dam94b, pp 10] et [Bar84, pp 129]) mais en utilisant le nouveau mécanisme d'abstraction :

$$\begin{array}{ll} T[\{x_1 = a_1 \ldots x_n = a_n\}] & = \lambda(sel) \centerdot sel(x_1 \rightarrow\uparrow_0 [a_1]) \ldots (x_n \rightarrow\uparrow_0 [a_n])! \\ T[a \centerdot x] & = a(sel \rightarrow \lambda(x)x)! \end{array}$$

où T représente la fonction de traduction des enregistrements vers les termes du λN-calcul. Ce codage a une propriété intéressante : un enregistrement placé en position fonctionnelle agit comme un environnement. Par exemple, l'expression :

$$r \centerdot (\lambda(x\,y \ldots).e)$$

a pour valeur l'expression e avec les variables abstraites par le λ (c'est-à-dire x, y, ...) fournies par l'enregistrement r. Par exemple :

$$\begin{array}{l} r = \{x = 1, y = 2, f = \lambda(x).x + 1\} \\ r.(\lambda(f\ x\ y).(f\ x) * (f\ y)) \Rightarrow 6 \end{array}$$

où le symbole « $\Rightarrow$ » signifie « se réduit en ». Cependant, l'enregistrement r doit fournir toutes les variables requises par l'abstraction sous peine d'erreur :

$$\{\} \centerdot (\lambda(x).x) \Rightarrow \texttt{error}$$

car la clôture (qui n'est pas représentée dans l'expression) d'un terme où des noms sont abstraits, génère une erreur. Par contre, l'enregistrement peut fournir plus de liaisons que nécessaires, car :

$$(\lambda(x).e)(y \rightarrow e') \Rightarrow \lambda(x).e$$

Il est donc crucial de disposer d'un mécanisme permettant l'abstraction d'une expression sur l'ensemble de ses variable libres. Cette opération est fournie par l'opérateur « quote », dénoté « $'$ » tel que par exemple :

$$\begin{array}{lll} '(y + x) & = & \lambda(x\,y).x + y \\ '(e\ r) & = & \lambda r.r \centerdot {}'e \end{array}$$

si r est un enregistrement. Ainsi, l'expression $q_e = '(y+x)$ peut être évaluée dans divers environnements :

$$q_e\{x=1, y=2\} \Rightarrow 3$$
$$q_e\{x=2, y=3\} \Rightarrow 5$$

L'utilisation des indices du λN-calcul permet à une expression de référer un nom hors de son environnement de définition. Par exemple, dans l'expression $\lambda(xy) \rightarrow' (x + \hat{~}y)$, le nom x est abstrait, mais grâce au quote, il ne réfère pas à celui de l'abstraction, alors que le $\hat{~}y$ s'échappe du quote et réfère a celui de l'abstraction. Il est ainsi possible de lier lexicalement et dynamiquement à l'intérieur d'un quote. Par contre, il n'est pas permis d'avoir un terme qui ne soit pas clos à l'issue du quote. De plus, il n'est pas permis d'avoir un indice, dans une abstraction (comme dans $\lambda(a)x(x).'x + \hat{~}x + \hat{~}\hat{~}x + \hat{~}\hat{~}\hat{~}x)$, qui réfère un terme hors de l'abstraction. Ici, les deux premières occurrences de x réfèrent le dernier x abstrait, le troisième x réfère le premier x abstrait et le dernier x est une erreur (car à trois niveaux d'abstractions, aucun x n'est défini). Dans ce cas, le terme est erroné et génère une erreur (DAMI rapproche le terme erroné d'un « method not found » des langages orientés objets).

Les enregistrements sont différenciés, selon qu'ils sont *ouverts* ou *fermés* (suivant la notation de RÉMY [Rém93]). Un enregistrement *fermé* est un enregistrement dont on connaît tous les items et dont il est possible de dire si un item est absent ou présent (c'est à la base du système de typage de RÉMY) ; un enregistrement *ouvert* est un enregistrement dont on connaît les éléments présents mais dont on ne peut rien dire sur les éléments qui viendront plus tard. Les enregistrements ouverts permettent l'*extension* d'enregistrements[1] et par là même une certaine forme de concaténation, avec la restriction que deux enregistrements quelconques ne peuvent pas être concaténés entres-eux[2]. Par exemple, l'extension de r_1 par r_2 (notée $r_1 << r_2$) nécessite un enregistrement de type quelconque à gauche mais un enregistrement ouvert à droite (de la forme $\{+\ l_1 = e_1, \ldots\}$).

L'extensibilité ayant forcé DAMI à gérer du sous-typage, un système de typage fondé sur les types récursifs contraints [AW93, EST95, Tha94] est proposé, système qui permet de garantir la propriété de type principal.

IX.3.3.2 La curryfication dans un ordre quelconque : le λ-calcul avec labels

La motivation des travaux de GARRIGUE [Gar94, Gar95] sur le λ-calcul avec labels est de permettre, comme pour la nouvelle abstraction et la nouvelle application de DAMI, l'application dans un ordre quelconque de la fonction curryfiée à ses arguments.

Le moyen employé ici est différent : au lieu d'introduire une nouvelle sorte de variable et un nouveau type d'application, GARRIGUE étiquette les arguments d'une fonction par un label. Par exemple, la fonction

$$f(x,y,z) \quad = \quad x + y + z \tag{2}$$

peut s'étiqueter de la façon suivante, en utilisant le label id_x pour la variable x, id_y pour y et id_z pour z :

$$R \quad = \quad \lambda id_x \approx x\ id_y \approx y\ id_z \approx z.x + y + z$$

où le symbole « ≈ » représente la *liaison* entre les labels et les variables du λ-terme[3]. L'abstraction se comporte comme une curryfication, mais permettant une sélection des variables lors de l'application, dans un ordre quelconque. Par exemple, l'expression R définie dessus peut être réduite en une seule application :

$$R\ id_z \approx 3\ id_y \approx 2\ id_x \approx 1 \quad \Rightarrow \quad 6$$

On remarquera que l'ordre de spécification des variables n'est plus important grâce à l'utilisation de la liaison label ≈ variable. Il est aussi possible de réduire R en plusieurs étapes, comme c'est le cas des fonctions curryfiées, par « consommations » successives des variables :

$$R' = R\ id_z \approx 3 \qquad \Rightarrow \quad \lambda id_x \approx x\ id_y \approx y.x + y + 3$$

1. On parle dans ce cas de « concaténation asymétrique » car ce n'est pas une opération commutative.
2. Cette contrainte permet de définir un type principal pour la concaténation d'enregistrement [Dam95].
3. GARRIGUE utilise le symbole « ⇒ » ou « -> » pour spécifier la liaison entre le label et la variable. Pour ne pas confondre avec le symbole d'évaluation que nous utilisons dans ce document nous utilisons à la place le symbole « ≈ ».

$$R'\ id_y \approx 2\ id_x \approx 1 \quad \Rightarrow \quad 6$$

Les couples label $\approx$ variable commutent implicitement, dans le λ-calcul avec labels, pour permettre ce mécanisme de curryfication.

Nous venons de voir que GARRIGUE définit des labels *symboliques* pour la «localisation» des variables : il est ainsi possible de référer à un argument par un nom et non plus par un rang dans l'ordre des applications. Mais il permet aussi la spécification par indices numériques de *positions*. En l'absence de labels, un argument numérique est automatiquement associé aux variables abstraites d'un λ-terme. Par exemple, l'expression (2) se réécrit :

$$\lambda 1 \approx x\ 2 \approx y\ 3 \approx z.x + y + z$$

et la réduction de ce terme sera faite par référence explicite à l'indice numérique comme c'était le cas avec des labels symboliques. D'une façon similaire aux labels, un λ-calcul avec positions relatives est défini, très proche du précédent, où cette fois ci les labels sont uniquement des indices numériques.

GARRIGUE considère que les notions de label symbolique et d'indice numérique sont orthogonales et il permet l'utilisation conjointe de labels avec indices. En absence de label, un *label invisible*, noté « ι », est utilisé ; en l'absence d'indice numérique, c'est la constante 1 qui est implicitement utilisée. Les labels sont donc des couples $(nom, indice)$. On remarquera que le λ-calcul est un cas particulier de ce calcul où les arguments sont de la forme $(\iota, 1)$. Les deux extensions décrites ci-dessus sont unifiées dans le *λ-calcul sélectif* qui utilise conjointement les labels symboliques et les indices numériques comme labels.

IX.3.4 Comparaison avec 81/2

Les deux extensions présentées enrichissent le λ-calcul avec un nouveau type de variable, les noms, qui ne sont pas sujets à α-conversion, et qui doivent être liés par un nouveau type d'application. L'objet qui permet de lier plusieurs de ces noms en même temps est naturellement l'enregistrement.

L'intérêt pour ce qui nous concerne est que ces approches étendent la sélection à une notion d'évaluation dans un environnement particulier. Mais, dans ces extensions, l'environnement est contraint de fournir toutes les références libres du terme que l'on désire évaluer (les termes ouverts ne sont pas permis). De plus, la fermeture de l'application doit se faire explicitement. Ce dernier point est problématique, dans la mesure où il est possible de déterminer syntaxiquement si un terme est clos, c'est-à-dire qu'il ne contient plus de références libres.

L'extension d'enregistrements permet d'obtenir un comportement similaire à la concaténation des amalgames. Néanmoins, la concaténation des enregistrements est asymétrique dans le λN-calcul alors qu'elle est symétrique dans les amalgames. De plus, une notion d'erreur est définie qui correspond par exemple à la sélection d'un nom dans un enregistrement, alors que cette notion est absente dans les amalgames. Au contraire, l'absence de définition pour une référence permet de retarder la liaison de la référence. On trouve cependant une définition, avec l'introduction des indices sur les noms, et une gestion des noms qui ressemble à celle des amalgames : il est possible de redéfinir un nom et d'accéder à une redéfinition. Cependant, l'accès à un nom qui n'est pas abstrait génère une erreur, alors que la référence, dans les amalgames, reste « symbolique ». Enfin, le λN-calcul dispose d'un système de typage, ce qui n'est pas le cas pour les amalgames.

Notons que la distinction entre nom et variable peut s'interpréter comme la distinction entre référence libre et liée des amalgames, dans le sens où une référence liée est destinée à disparaître alors que les références libres sont des constantes et que de ce fait leur nom importe.

L'avantage de l'approche de GARRIGUE est d'unifier les deux types d'applications, en considérant un mécanisme de position par nom indexé. Une interprétation de ce mécanisme est de comprendre un nom indexé comme un *stream* : le nom représente le canal et les index les jetons qui circulent sur ce canal. On a ainsi un pont entre les notions de communication, de passage d'argument par position et de stream de valeurs. Cette analogie, pour l'instant formelle, doit faire l'objet de recherches supplémentaires car elle permettrait d'unifier l'aspect stream de 81/2 avec les amalgames.

IX.4 La programmation incrémentielle

La programmation incrémentielle est un paradigme de programmation dans lequel on essaye de définir des opérateurs qui permettent la définition de programmes pouvant être facilement étendus (par de nouvelles fonctionnalités par exemple) sans nécessiter une re-compilation complète de ceux-ci. Nous discutons les modèles proposés par LAMPING et sa proposition de systèmes paramétrés unifiés (cf. section IX.4.1), les variables et procédures quasi-statiques de LEE & FRIEDMAN (cf. section IX.4.3) et enfin le $\lambda\mathbf{C}$-calcul de LEE & FRIEDMAN (cf. section IX.4.5) et le calcul de contextes typé de HASHIMOTO & OHORI (cf. section IX.4.6), qui sont tous deux des extensions du λ-calcul avec une notion de capture de variables.

IX.4.1 La paramétrisation de Lamping

LAMPING [Lam88] est à l'origine des travaux sur la *paramétrisation*. Un système est *paramétré* quand, une, ou plusieurs de ses entrées déterminent la valeur de sa sortie. Par exemple, dans le λ-calcul, on trouve la notion de paramétrisation dans les arguments de la fonction et dans les variables libres d'une expression (c'est la variation des arguments ou des variables qui influence le résultat d'un terme du λ-calcul). La paramétrisation est à l'origine des concepts de module, encapsulation, accès explicites à des environnements... LAMPING présente un système formel qui a une puissance d'expression égale au λ-calcul.

Dans les langages fonctionnels, il existe une distinction entre les variables d'une fonction et les arguments d'une fonction : les variables permettent la «communication» d'informations dans une expression alors que les arguments communiquent les informations entre les différentes parties du programme. Cette distinction est faite par LAMPING qui propose, en plus de la liaison lexicale classique des identificateurs (nommés *paramètres lexicaux*) et de la construction `let ... in` des langages fonctionnels, une liaison par *environnement* (nommés *paramètres de données*). Par exemple :

$$\mathbf{let\ f \leftarrow\ data\ x:\ (x * x)\ in\ (supply\ x \leftarrow\ 3\ to\ f) + (supply\ x \leftarrow\ 4\ to\ f)} \tag{3}$$

s'évalue en 25. La construction `data x:` permet de définir `x` comme un paramètre de donnée qui sera résolu non pas lexicalement comme les paramètres du `let` mais explicitement à l'aide de `supply ... to`. La construction `supply ... to` est le pendant de la construction `let ... in` du λ-calcul. La sémantique de ce formalisme, exprimée dénotationnellement, nécessite deux environnements $\mathcal{L}$ et $\mathcal{U}$ représentant l'*environnement lexical* et l'*environnement d'utilisation* des expressions. Une opération de «transmission» des définitions est définie par du sucre syntaxique :

`transmit` *def* `to` *expr*

Il est possible de composer des environnements par une opération de composition : `b1 ∘ b2` permet de chercher les liaisons dans `b2` puis `b1`. Il est donc possible d'étendre un ensemble de liaisons, mais il n'est pas possible par contre de chercher sélectivement une liaison redéfinie.

Une limitation (levée mais de manière peu élégante) est la nette séparation entre paramètre lexical et paramètre de donnée. Il est possible de compléter une expression impliquant des paramètres de données par des liaisons mais il n'est pas possible d'affecter une expression de paramètres lexicaux par des définitions. En fait, en interchangeant les deux environnements $\mathcal{L}$ et $\mathcal{U}$ dans une construction `with ... in`, il est possible de résoudre le problème.

IX.4.2 Comparaison avec 81/2

LAMPING fait une séparation syntaxique très forte, essentielle même, entre les paramètres lexicaux et dynamiques, alors que les amalgames unifient les deux modes de liaisons, toutes les références «aspirent» à la liaison, les références libres restant sous une forme symbolique. Grossièrement, la construction (3) peut se traduire dans le formalisme des amalgames par :

$$\{f = x * x\}\ .\ (\{x = 3\}\ .\ f + \{x = 4\}\ .\ f)$$

Il apparaît que les amalgames ne font pas la distinction entre les deux types d'environnement de LAMPING. Cela est dû à deux facteurs essentiels:

1. les amalgames ne sont pas importés dans un autre modèle de calcul qui nécessite une notion d'environnement propre (la liaison des arguments des fonctions) ;
2. nous désirons un mécanisme de liaison qui soit *implicite*.

IX.4.3 Importation et exportation de définitions : les variables quasi-statiques

LEE & FRIEDMAN [LF93] proposent une nouvelle abstraction, les *variables quasi-statiques* qui associent à un identificateur une variable. Le mécanisme des variables quasi-statiques définit des fonctions spécifiées par un nom externe (non susceptible d'α-renommage) et un nom interne (susceptible d'α-renommage). La définition de variables quasi-statiques se fait grâce à l'utilisation d'abstractions :

$$(\textbf{qs–lambda0}\ ((in_1\ ext_1)\ldots(in_n\ ext_n))\ expression)$$

Les noms $in_1 \ldots in_n$ sont les noms internes qui sont liés dans *expression* alors que $ext_1 \ldots ext_n$ sont les noms externes qui sont visibles des autres fonctions. Afin d'éviter les ambiguïtés, on évite d'utiliser les mêmes noms pour les paramètres internes et externes. Il est ainsi possible de définir deux fonctions *pair*? et *impair*? :

$$(\textbf{define}\ impair?\ (\textbf{qs–lambda0}\ ((pairFct\ F))\ (x)\ (\textbf{if}\ (zero?\ x)\ \#f\ (pairFct\ (-\ x\ 1))))))$$
$$(\textbf{define}\ pair?\ (\textbf{qs–lambda0}\ ((impairFct\ G))\ (x)\ (\textbf{if}\ (zero?\ x)\ \#t\ (impairFct\ (-\ x\ 1))))))$$

qui utilisent les fonctions $pairFct$ et $impairFct$, ces fonctions étant liées à F et G. Afin de pouvoir réellement utiliser ces fonctions, il est nécessaire de résoudre les variables quasi-statiques par un mécanisme proche de l'instanciation des langages orientés objets. La résolution des variables quasi-statiques s'effectue grâce à la fonction (**resolve** *n fonction externe qs_proc*) qui associe dans la procédure quasi-statique *qs_proc* à la variable quasi-statique *externe* la liaison avec *fonction*, l'argument *n* spécifiant le nombre de variables quasi-statiques à résoudre. Ce mécanisme de résolution *dynamique* permet, à l'exécution, la résolution des variables quasi-statiques. Ainsi, on peut résoudre les variables quasi-statiques de *impair*? et *pair*? :

$$\begin{array}{l}(\textbf{letrec}\ (\quad (mon\text{–}impair?\ (\textbf{resolve1}\ mon\text{–}pair?\ F\ impair?))\\ \qquad\qquad (mon\text{–}pair?\ (\textbf{resolve1}\ mon\text{–}impair?\ G\ pair?)))\\ \qquad (mon\text{–}odd)\ 10))\end{array}$$

qui définissent deux fonctions (qui ne sont plus quasi-statiques) *mon–impair*? et *mon–pair*? convenablement liées. Par ce mécanisme, des identificateurs dans des environnements différents peuvent partager des objets via le nom externe associé à une variable quasi-statique. L'introduction de cette construction dans le langage Scheme permet une cohabitation maîtrisée de variables statiques et dynamiques. Ce mécanisme permet l'exportation de définitions grâce au nom externe et l'importation de définitions par un mécanisme dynamique de liaison nom-externe, nom-interne. On notera une approche différente dans [QD96] pour obtenir un résultat similaire, cette approche nécessitant aussi des opérateurs *explicites*. Le partage de variables par delà la limite d'environnements lexicaux est rendu possible et on peut, à l'aide de **resolve1**, spécifier des liaisons dans des scopes différents.

IX.4.4 Comparaison avec 8 1/2

Lors de l'utilisation de variables quasi-statiques, la notion d'environnement n'est pas présente, ni les notions d'extension ou de restriction des définitions. De plus, si on utilise une procédure utilisant une variable quasi-statique sans avoir résolu toutes ses variables quasi-statiques, alors une erreur est déclenchée à l'exécution. Les variables quasi-statiques représentent une première tentative pour introduire la notion de substitution avec capture (à l'aide de la fonction `resolve`), mécanisme formalisé dans la proposition suivante du même auteur, le λ**C**-calcul.

IX.4.5 Liaison dynamique et λ-calcul : le λC-calcul

Le $\lambda\mathbf{C}$-calcul de LEE & FRIEDMAN [LF96] est une formalisation de la liaison statique et dynamique en λ-calcul. LEE & FRIEDMAN ajoutent à la β-réduction classique par substitution des variables libres classique (modulo α-renommage) une substitution de « contexte » où cette fois ci il y a capture des variables (pas de renommage).

En λ-calcul [Bar84], un terme e est une *variable* x, une *abstraction* $\lambda x.e$ ou une *application* e e. LEE & FRIEDMAN ajoutent aux termes du λ-calcul, la notion de contexte $\mathbf{C}$, qui est un *identificateur* $\mathbf{x}$, une abstraction $\lambda\mathbf{x}.\mathbf{C}$, une *application* $\mathbf{C}$ $\mathbf{C}$ ou un *trou* $\mathbf{h}$. Le mécanisme de substitution classique du λ-calcul se nomme la *β-substitution* qui consiste à remplacer dans un terme toutes les variables libres de même nom en renommant les variables liées pour éviter le phénomène de *capture* de variables. La substitution de e en x dans un terme e' (noté $[e/x]\ e'$) est un mécanisme qui préserve les variables liées. Par exemple, dans l'expression :

$$[y/x]\ \lambda y.x \quad \Rightarrow \quad \lambda y'.y \tag{4}$$

la variable liée y est renommé en y' lors de la substitution de x en y. L'opération correspondante pour ce qui concerne les contextes est le *remplissage de trous* qui au contraire *capture* les variables. Par exemple, la syntaxe de la substitution restant la même, dans l'expression :

$$[\mathbf{y}/\mathbf{h}]\ \lambda\mathbf{y}.\mathbf{h} \quad \Rightarrow \quad \lambda\mathbf{y}.\mathbf{y} \tag{5}$$

l'identificateur $\mathbf{y}$ n'est pas renommé en un nouvel $\mathbf{y}'$ « frais ». Ces deux substitutions exhibent un comportement différent comme nous le montre l'application des λ-termes des équations (4) et (5) :

$$\begin{array}{lcl} (\lambda y'.y)\ 10 & \Rightarrow & y \\ (\lambda\mathbf{y}.\mathbf{y})\ 10 & \Rightarrow & 10 \end{array}$$

Dans la première expression, la substitution n'a pas lié la variable abstraite alors que le « remplissage de trou » de la seconde expression a lié l'identificateur abstrait résultant en la fonction identité. LEE & FRIEDMAN étendent la β-substitution définie sur les contextes aux termes.

Une fois cette extension définie dans le λ-calcul, LEE & FRIEDMAN définissent une notion d'abstraction sur les identificateurs libres d'un terme. Cette abstraction est notée $\Phi\{\mathbf{x_1} : x_1, \ldots, \mathbf{x_n} : x_n\}.e$ où tous les $\mathbf{x_i}$ sont différents et réfèrent des variables libres de e. Cette construction est une introduction dans le λ-calcul d'un mécanisme similaire aux variables quasi-statiques introduites dans Scheme, avec les $\mathbf{x_i}$ qui représentent les *noms externes* et les x_i qui représentent les paramètres internes.

L'introduction de contextes et d'un mécanisme de substitution sur ces contextes permet un gain en expressivité tel que le codage du λN-calcul (cf. section IX.3.3.1), du λ-calcul avec labels (cf. section IX.3.3.2), le système de paramétrisation de LAMPING (cf. section IX.4.1) ainsi que bien sur les variables quasi-statiques (cf. section IX.4.3).

IX.4.6 Liaison dynamique, typage et λ-calcul : le calcul avec contexte typé

Le calcul avec contexte typé [HO96] de HASHIMOTO & OHORI représente une tentative similaire à celle de LEE & FRIEDMAN d'introduction d'un mécanisme de capture de variables par l'utilisation de contextes. Contrairement au $\lambda\mathbf{C}$-calcul où deux espaces de noms étaient utilisés pour faire la différence entre les variables et les identificateurs, HASHIMOTO & OHORI définissent un système de typage qui joue un rôle analogue. Ce système de typage spécifie pour les contextes, en plus de leur type, des propriétés de capture de variables comme : le type du contexte, le type du terme qui sera produit par substitution du contexte et enfin, l'ensemble des variables, avec leur type, qui seront capturées par substitution du contexte. Le système de typage est explicite et un système d'inférence de type reste à définir. On notera la référence aux travaux de [Kah92, Tal91, Tal93].

IX.4.7 Comparaison avec 8$_{1/2}$

Le $\lambda\mathbf{C}$-calcul est une généralisation, dans le cadre du λ-calcul de la liaison des variables par β-réduction et par un mécanisme de liaison tardive : le remplacement de contexte dans un λ-terme. LEE & FRIEDMAN rapprochent cela de la compilation où des variables sont liées dynamiquement. La nouvelle définition permet l'écriture de termes ouverts où les variables libres sont abstraites par une opération Φ qui permet la liaison dynamique à l'exécution. La substitution avec capture du $\lambda\mathbf{C}$-calcul modélise exactement le premier processus de calcul de la réduction des amalgames (cf. section VIII.4.6). Lee & Friedman définissent les constructions :

1. définition d'abstractions de code $\Phi\rho.e$ pour simuler la notion de programme (code) compilé ;
2. une opération unaire `exec` qui permet l'exécution de code compilé $\Phi\rho.e$ en e ;
3. une opération unaire $\mathtt{lam}_\mathtt{x}$ pour permettre la construction de λ-abstractions de code $\Phi\rho.\lambda x.e$;
4. une opération binaire `app` pour permettre la construction d'applications de code compilé $\Phi\rho.(e_1\ e_2)$;

et montrent qu'elles vont lui permettre de spécifier les comportements suivants :

- définition de code compilé avec des variables libres qui représentent des possibilités de liaisons dynamiques ;
- compilation incrémentielle par liaison dynamique de code partiellement lié ;
- exécution de code compilé.

La formalisation de ces mécanismes repose fondamentalement sur le concept de substitution avec capture et Lee & Friedman le développent dans le λ-calcul. Ils développent ainsi une théorie équationnelle rendant compte en même temps des notions de fonction et de capture.

Les amalgames veulent embarquer la notion de capture dans le contexte des systèmes, de leur extension et de leur évaluation relativement à un système. Cela nous amène à des développements spécifiques. En effet, si l'opération de sélection se rapproche de l'application de fonction (on peut envisager s . e comme l'application curryfiée de e dont on a abstrait certaines variables, aux arguments fournis par s), l'extension échappe à une interprétation simple en termes de fonctions. En fait, la réduction de la concaténation est un mécanisme non-local qui s'oppose à une formalisation de type équationnelle.

On remarquera que Lee & Friedman font une distinction entre les notions de *variable* et d'*identificateur* d'où la nécessité de définir de nouveaux types d'abstractions et d'applications. Afin de résoudre ce problème sans séparer l'espace des noms, Hashimoto & Ohori introduisent un typage des expressions. Les amalgames définissent un objet unique, les identificateurs, ce qui permet de manipuler les expression avec une seule sorte d'opérateur ; de plus, les expressions ne nécessitent aucun typage.

IX.5 Les langages réflexifs : l'accès à la représentation des données et à l'environnement

Un langage réflexif est un langage qui a la notion de *soi*, c'est-à-dire qu'il est capable de se modifier. Plusieurs mécanismes ont été proposés et les constructions réflexives ont souvent été modélisées par le modèle de la « tour réflexive » [Smi84]. Une tour d'interprètes est définie où chaque niveau de la tour évalue le niveau qui lui est supérieur. La tour est exécutée par une *machine ultime*. Deux opérations, donnant accès au programmeur à l'interprète, permettent de changer le niveau courant de l'interprète : la *réification* fait augmenter le niveau et rend l'état courant du calcul disponible (les expressions interprétées, les environnements...), la *réflexion* fait descendre le niveau donnant l'état dans lequel le calcul commence (cf. figure 1). Ce mécanisme permet au programmeur d'avoir accès aux structures implicites qu'il manipule, et de les modifier explicitement. Par exemple, la réification permet de changer un environnement (une clôture par exemple) en une structure de données que l'on peut manipuler explicitement, la réflexion permettant de changer une structure de données (un enregistrement par exemple) en un environnement.

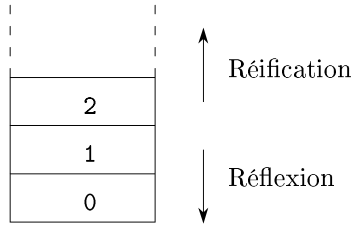


FIG. 1 –: *La tour d'interprètes dans un langage réflexif*

IX.5.1 Programme ≡ données

Nous nous intéressons ici à deux mécanismes particuliers : le premier est la capacité à représenter un programme sous la forme de données manipulables par le programmeur ; le deuxième à gérer explicitement la notion d'environnement d'évaluation. Le premier n'appelle qu'un commentaire de notre part (cf. section VIII.3.2) : on peut, en Lisp par exemple, représenter par une liste un fragment de programme Lisp. Les opérations sur les listes permettent de construire n'importe quelle liste et donc de construire n'importe quel programme. Par exemple, la fonction `H` suivante en Common Lisp [Ste82, Ste84], prend deux s-expressions, `(f x y)` et `(g u v)`, correspondant à l'application de fonctions binaires, et construit puis évalue le programme `(f (g x v) (g u y))` :

```
(defmacro H ((f x y) (g u v)) `(,f (,g ,x ,v) (,g ,u ,y)))
```

Ce genre de traitement n'est pas possible avec les amalgames : il n'y a pas de représentation explicite d'un amalgame. Quand nous parlons de *valeur symbolique*, il ne s'agit pas d'un programme représenté sous forme de données du premier ordre et que l'on peut transformer en « vrai » programme par une réification dans la tour des interprètes. Il s'agit d'une abstraction de code similaire en cela à une fonction. Cette abstraction est « semi-opaque » dans le sens suivant : on ne peut pas, par exemple, accéder au premier argument de l'expression $a+b$, comme c'est le cas pour une s-expression Lisp. Un amalgame n'est donc pas « transparent ». Mais il n'est pas opaque non plus dans le sens où il est possible de remplacer certains « trous », les références libres d'un terme, par une expression arbitraire. La situation est analogue pour le λ-calcul : on ne peut pas trouver de fonction f telle que $f(P\ Q) \Rightarrow P$ car on peut toujours réduire $(P\ Q)$ avant d'appliquer f. Un opérateur comme `H` ne peut donc pas être défini. Par contre, en λ-calcul comme pour les amalgames, les programmes sont aussi des données que l'on peut combiner pour former d'autres programmes (à défaut de les « casser » pour en extraire des parties).

IX.5.2 La notion d'environnement

La notion d'environnement est classiquement séparée, dans les langages de programmation, de la notion de structure de données : un ensemble de liaisons sont définies dans un environnement (par exemple la *clôture* des langages fonctionnels) alors que les instructions, les opérateurs du langage permettent la manipulation des structures de données (les enregistrements, les produits cartésiens...). Ces deux notions n'ont jamais d'intersection (sauf dans les langages réflexifs) : il n'est pas permis de manipuler le contenu d'une clôture ou d'augmenter *explicitement* les définitions d'un `let` ; un environnement ne peut pas être transformé en structure de données... Elles ne jouent donc pas les mêmes rôles.

Le formalisme des amalgames unifie néanmoins la notion de structure de données et d'environnement à travers les systèmes et les mécanismes de sélection et d'extension. Nous présentons le langage 3-Lisp (cf. section IX.5.3) un des premiers langages réflexifs qui a initié la notion de tour réflexive, Pebble (cf. section IX.5.4) un langage qui manipule les environnements explicitement, puis Symmetric Lisp, un langage parallèle manipulant les environnements (cf. section IX.5.5) et enfin l'introduction dans le langage Scheme d'opérateurs permettant une manipulation explicite de la notion d'environnement (cf. section IX.5.6). D'autres langages permettent la manipulation de la notion d'environnement, citons par exemple Plasma II [LS89].

IX.5.3 Le langage réflexif 3-Lisp et ses variantes

Un des premiers langages réflexifs est le langage 3-Lisp [Smi82, Smi84, dRS84] : des primitives permettant de se déplacer dans la tour réflexive sont ajoutées à un dialecte Lisp basé sur une tour d'interprètes. Un programme d'un niveau n est exécuté par un autre programme d'un niveau $n-1$ (typiquement l'interprète du programme). Par un mécanisme de déplacement dans cette tour, les primitives permettent à un programme d'avoir accès aux structures définissant son propre programme. Les opérateurs préfixes ↑ et ↓ permettent le déplacement dans la tour d'interprètes afin d'avoir accès à une structure et ce qu'elle désigne. Par exemple, `↑(+ 2 2)` s'évalue en `'4`. Le programmeur peut, par ce mécanisme, ajouter des fonctionnalités à l'interprète exécutant son programme.

Les auteurs de 3-Lisp n'utilisent pas les possibilités de changement de niveau d'interprète pour définir des notions de programmation incrémentielle. Néanmoins, les bases sont définies pour permettre la définition de langages qui conduiront à traiter les environnements comme objets de première classe et à les utiliser pour structurer des programmes.

Deux autres langages, Brown et Blond sont des variations sur le thème des langages réflexifs :

- Brown [FW84, WF88] est un interprète Scheme [RCe+92] dans lequel a été rajouté des constructions permettant la construction de fonctions à partir d'expressions (grâce à la fonction `denotation`) et inversement (grâce à la fonction `meaning`). Dans le cas de Brown les environnements sont réifiés en *fonctions* et non pas en structures de données (contrairement à 3-LISP de SMITH).
- Le langage Blond[1] [DM88], est un langage réflexif, et correspond à une évolution de Brown. La réification et la réflexion sont de nouveau implémentés comme des déplacements dans la tour réflexive des interprètes. Contrairement à Brown où plusieurs interprètes cohabitent, en Blond, un interprète unique est présent. Seul son niveau d'exécution change, ainsi chaque niveau dispose de son propre environnement.

IX.5.4 Le langage Pebble

Le langage Pebble [LB88, BL84a] de BURSTALL et LAMPSON est un langage fonctionnel basé sur le λ-calcul avec types qui explore les notions de type de données, de type de données abstrait et de module. La notion de programmation incrémentielle est centrale dans l'approche de Pebble : un programme est vu comme la *liaison* de modules séparés. Un module produit une *implémentation*, c'est-à-dire un ensemble de déclarations de types et des procédures. Les relations entre diverses implémentation sont assurées par les *interfaces*, qui définit le type d'implémentation qui est produit ou requis par un module (c'est la *signature* du module qui est seule visible des autres modules). La notion de *module* est une fonction des implémentation vers des implémentations. La correspondance suivante :

$$\begin{array}{rcl} \text{implémentation} & \leftrightarrow & \text{valeur} \\ \text{interface} & \leftrightarrow & \text{type} \\ \text{module} & \leftrightarrow & \text{fonction} \end{array}$$

est donc établie. Elle met en rapport la notion de module, interface et implémentation avec les notions classiques de valeur, de type et de fonction des langages fonctionnels.

Le langage Pebble a de nombreuses particularités (traiter les types comme des valeurs, permettre le polymorphisme ad hoc, ne pas différencier les phases de compilation et d'exécution[2]...) mais nous ne nous intéresserons qu'à une en particulier : la capacité de définir des liaisons comme valeurs et la manipulation de ces liaisons. En Pebble, un ensemble de liaisons va permettre de définir un *environnement*. Les environnements vont être à la base d'un système de module. Les modules ne sont qu'un point de vue, en Pebble, sur les liaisons, et non pas une introduction de mécanismes particuliers, comme ce peut être le cas en SML.

La notion de liaison est présente en Pebble où elle est une association d'un nom et d'une valeur (notée $a \sim 3$), la liaison elle même étant une valeur. La portée des définitions est limitée par les constructions classiques `LET` $x : int \sim y + z$ `IN` $x + abs\ x$ ou bien `WHERE`. Le type d'une liaison est une «déclaration». Les

1. Le nom du langage Blond est un jeu de mot avec le langage Brown et marque son origine scandinave.
2. Une certaine évaluation, qui peut être de type symbolique, peut apparaître au moment de la compilation [BL84b, pp 290].

liaisons étant des valeurs, une liaison peut être un argument d'une fonction et une fonction peut retourner une liaison. La valeur d'une liaison est l'évaluation de sa partie droite et l'« attachement » au label (sa partie gauche). Par exemple, la valeur de $x : int \sim y + 3$ dans un environnement où y vaut 3 a pour valeur $x \sim 7$ (on omettra dans la suite le type de la définition). Un tuple de liaisons est encore une liaison : `LET` $x \sim 0$ `IN LET` [$x \sim 3$, $y \sim x$] `IN [x, y]` a pour valeur `[3, 0]` car les deux liaisons dans la paire sont évalué dans l'environnement extérieur. Il est possible d'effectuer des liaisons en série : `B1; B2` est équivalent à `[B1, LET B1 IN B2]`. Ce sont les seules opérations sur les liaisons.

Un ensemble de liaisons peut être utilisé comme *environnement* d'évaluation d'une expression, comme dans l'exemple : `LET` $b \sim (x \sim 3)$ `IN LET` b `IN x` qui s'évalue en 3. Les liaisons pouvant être des arguments de fonctions, l'expression suivante `LET` $f(b : (x : int \times y : int) \rightarrow int) \sim$ `LET` b `IN` $x + y$ `IN` f[$x \sim 1$, $y \sim 2$] évalue `x+y` dans un environnement de liaisons où x vaut `1` et y vaut `2`. Le résultat est bien sûr `3`.

IX.5.5 Le langage Symmetric Lisp

Jagannathan [GJL87a, GJL87b, Jag89] a défini le langage Symmetric Lisp permettant la définition d'expressions qui s'évalueront en environnements (et d'utiliser ces environnements pour évaluer d'autres expressions) ainsi que la définition d'environnements. La première conséquence est que le rôle joué par de nombreuses structures différentes dans les langages (module, enregistrement, classe, type abstrait...) est assuré ici par une structure unique. De plus, Symmetric Lisp est un langage *parallèle* (de type concurrent) où tous les éléments d'un environnement s'évaluent en même temps.

La gestion des environnements se fait à l'aide de trois constructions : `NAME`, `PRIVATE` et `ALPHA`. La création d'un environnement se fait à l'aide des constructions `ALPHA` et `NAME`. Par exemple :

```
(ALPHA
   (NAME x (+ 1 1))
   (NAME y (* x 2)))
```

définit un environnement qui lie `x` à `2` et `y` à `4`. La règle de liaison est identique à celle utilisée pour les amalgames : les variables des expressions d'un environnement utilisent les définitions de l'environnement si elles existent, sinon elles sont cherchées dans l'environnement englobant. Un nom qui n'est pas défini reste sous la forme d'un label (offrant ainsi la liaison dynamique des variables). L'utilisation de la construction `PRIVATE` permet de définir une variable *locale*, qui ne sera pas visible. L'environnement peut aussi être accédé par une *position* : la construction `A-NTH` permet d'accéder à la n^{e} définition d'un `ALPHA`. L'équivalent des fonctions `car` et `cdr` Lisp se nomme `ACAR` et `ACDR` et s'applique sur un `ALPHA`. Une fois un environnement créé, on peut évaluer une expression dans cet environnement à l'aide de la construction `WITH`. Par exemple, dans l'environnement :

```
(NAME pays
  (ALPHA
     (NAME france 'paris)
     (NAME portugal 'lisbonne)))
```

l'expression `(WITH pays france)` s'évalue en `'paris`. Il est possible d'altérer la valeur définie dans un environnement par un `WITH!` (toutes les constructions se terminant par un point d'exclamation effectuent un effet de bord). L'extension d'environnement est possible, suivant un mécanisme d'extension d'enregistrement : un environnement est déclaré *ouvert* (par la construction `(NAME environ (OPEN-ALPHA))` qui s'évalue en `(NAME environ (ALPHA *)))` et pourra être candidat au rajout de définitions. Le rajout de définitions se fait grâce à `ATTACH!` qui ajoute à une `ALPHA` ouverte une nouvelle définition. Le rajout de définitions ne permet cependant pas la concaténation de deux environnements quelconques.

La liaison par `NAME` permet d'avoir une expression quelconque (dont les fonctions) et ainsi définir la notion de module. Par exemple, une librairie `lib` définit une fonction `sin` :

```
(NAME lib (ALPHA (NAME sin (LAMBDA (x) < définition de sin >))))
```

que l'on peut ensuite utiliser : `(WITH lib (sin 2.0))`.

IX.5.6 Les constructions explicites `reflect` et `reify`

Suite à la définition du langage Symmetric Lisp définissant la notion d'environnement comme valeur de première classe à partir de constructions explicites `NAME` et `ALPHA`, JAGANNATHAN [JA92, Jag94b, Jag94a] ne retient que les deux opérateurs de transformation *explicites* qui permettent de transformer une structures de données en environnement et inversement (une implémentation de ces opérateurs apparaît dans le langage Rascal). Ces opérateurs utilisent la terminologie des langages réflexifs en n'en conservant que l'esprit (il n'est plus question de tour réflexive) et vont lui permettre de modéliser des fonctionnalités des langages objets (héritage, modules). L'opérateur `reflect` permet la transformation d'une structure de données qui associe à un label une valeur en un environnement qui pourra être utilisé pour l'évaluation d'expressions. La structure de données choisie est l'enregistrement classique des langages fonctionnels. Par exemple, `reflect {a=2, b=3} in (a+b)` s'évalue en 7. Inversement, `reify` permet de transformer un environnement en un enregistrement qui peut être manipulé par le programmeur. Par exemple, l'expression `let a=1 and b=2 in (reify)` a pour résultat `{a=1, b=2}`. La valeur d'un enregistrement est accédée classiquement par un opérateur de sélection, et une opération de concaténation d'enregistrement est définie : par exemple, `(• r1 r2)` concatène les enregistrements `r1` et `r2`.

IX.5.7 Comparaison avec $8_{1/2}$

Le langage Pebble introduit un nouvel objet, la liaison, qui est typé. Il est possible de combiner deux ensembles de liaisons (par une composition en série `B1 ; B2`) mais pas de définir d'expression récursive telle que :

$$\{a = 1, b = c\} \# \{c = a, d = b\}$$

où chaque environnement va chercher des liaisons dans l'autre environnement. La notion de liaison permet la définition de `let` extensible puisqu'il est possible de compléter les variables libres d'un `let` par les définitions d'un environnement. Il n'est cependant pas possible de référer la re-définition d'une liaison, la première liaison masquant toute les précédentes.

La nécessité des opérateurs spécifiques `reflect` et `reify` provient de distinction faite entre structure de données et environnement d'évaluation (une clôture). L'environnement d'évaluation est opaque pour le programmeur et ce n'est que grâce à l'usage d'un opérateur explicite que l'on peut pénétrer cette opacité. Les amalgames prennent le point de vue opposé : la structure de données *est* l'environnement d'évaluation : il y a unification des deux concepts. On notera aussi que ces deux opérateurs ne permettent pas l'accès à une redéfinition.

Il n'est pas possible, en Symmetric Lisp de combiner deux ensembles de liaisons quelconques et d'augmenter un ensemble par de nouvelles définitions (contrairement à LAMPING ou aux amalgames). Les restrictions sont les mêmes que pour les enregistrements extensibles. Par contre, le mécanisme qui consiste à considérer les enregistrements comme des environnement est identique à notre approche, ainsi que les règles de portée des variables.

Le tableau en page suivante tente de résumer la comparaison entre les amalgames et les divers langages et formalismes que nous avons étudiés.

1. Il n'est pas possible de définir de structures de données récursives en SML. On notera cependant que cela est possible en CAML mais que ce mécanisme n'est pas réellement offert à l'utilisateur.

2. Il est possible en Haskell de définir plus que de simples fonctions récursives. Le mécanisme d'évaluation paresseux des expressions permet la définition de structures de données récursives.

3. Le λ**C**-calcul permettant « l'émulation » du λ-calcul avec labels et du λN-calcul, les deux réponses sont valables.

langage	notion d'agrégat	restriction	extension	extension de type	définition récursive	accès aux redéfinitions	accès ≡ éval dans un env^{t}	variable	nom	portée lexicale et dynamique	capture de variables	typage	formalisme d'accueil
C	`struct`	non	non	non	par pointeur	non	non	oui	non	non	non	oui	impératif
C++	`struct`	non	non	héritage	par pointeur	oui	non	oui	non	non	non	oui	objet impératif
SML	`record`	non	non	non	`let rec` [1]	non	non	oui	non	non	non!	oui	λ-calcul
Haskell	`record`	non	non	héritage	`let rec` [2]	non	non	oui	non	non	non	oui	λ-calcul
λ-calcul avec labels	tuple	non	oui	non	non	non	non	oui	oui	oui	oui	oui	λ-calcul
λN-calcul	tuple	oui	oui	non	non	oui	oui	oui	oui	oui	non	oui	λ-calcul
π-calcul	tuple	non	non	non		non	non	oui	oui	oui	oui	oui	algèbre à la CCS
Lamping	bindings	non	oui	non	non	non	non	oui	oui	oui	oui	non	λ-calcul
variables quasi-statiques	liste	oui	oui	non	oui	non	non	oui	oui	oui	oui	non	Scheme
$\lambda\mathbf{C}$-calcul	tuple	oui	oui	non	non	oui	oui/non [3]	oui	oui	oui	oui	non	λ-calcul
λ-calcul+ `C` + types	tuple	non	oui	oui	non	non	non	oui	oui	oui	oui	oui	λ-calcul
Pebble	bindings	non	oui	oui	oui	non	non	oui	oui	oui	oui	oui	λ-calcul + types
3-Lisp	liste	oui	oui	non	non	oui	oui	oui	non	oui	oui	non	Lisp
reflect, reify	record	oui	oui	non	non	non	oui	oui	oui	oui	oui	non	Scheme
Symmetric Lisp	Alpha	non	oui	non	non	non	oui	oui	oui	oui	oui	oui	Lisp
8½	collection	non	oui	non	non	non	oui	oui	non	oui	non	oui	déclaratif
amalgames	{...}	.	#	non	oui	oui	oui	non	oui	oui	oui	non	déclaratif

TAB. 2 –: *Comparaison des divers formalismes étudiés (les notes en exposant sont reportées à la page précédente en notes de bas de page).*

Chapitre X

Une sémantique des amalgames

X.1 Le langage de base des amalgames

X.1.1 La définition du langage Σ

DÉFINITION X.1 **(Le langage Σ)**
Nous nous intéressons à un sous-ensemble des amalgames, le langage Σ. La syntaxe abstraite des termes de Σ est la suivante (donnée dans le style BNF) :

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textsc{Ref}} \\
p & ::= & id^n \mid id_n \\
\multicolumn{3}{l}{\textsc{Sys}} \\
s & ::= & \{\ldots, id = e, \ldots\} \\
\multicolumn{3}{l}{\Sigma} \\
e & ::= & p \mid s \mid e \,\#\, e' \mid e \,\centerdot\, e'
\end{array}
$$

où n appartient à $\mathbb{N}$, id à $\textsc{Id}$, e à Σ, p à $\textsc{Ref}$ et s à $\textsc{Sys}$. La construction $id = e$, qui n'est pas un élément de Σ s'appelle une *définition* : id est le definiendum et e le definiens. Tous les definiendum d'un système sont distincts. On utilise $\equiv$ comme signe d'égalité dans Σ.

REMARQUE Les opérateurs $\centerdot$ et $\#$, en l'absence de parenthésage associent à gauche, i.e. $a \diamond b \diamond c \equiv (a \diamond b) \diamond c$ où $\diamond \in \{\centerdot, \#\}$. L'opérateur $\centerdot$ à une précédence supérieure a $\#$, i.e. $a \centerdot b \# c \equiv (a \centerdot b) \# c$.

REMARQUE Le langage Σ ne comporte que les constructions de définitions de systèmes, les opérateurs de concaténation et de sélection ainsi que les références libres et liées. Les identificateurs sont volontairement exclus de Σ (leur utilisation sera décrite dans la section X.2) car ils ne sont pas nécessaires à la définition des amalgames.

REMARQUE En utilisant la grammaire de Σ, il est possible d'écrire l'ensemble des termes de Σ. Malheureusement, tous les termes qui sont syntaxiquement corrects ne sont pas sémantiquement justes :

- il se peut que des références soient spécifiées libres alors qu'elles doivent être liées car une définition existe dans un scope englobant à la référence. Par exemple, dans l'expression $\{a = 1, b = a^0\}$, b définit une référence libre alors que la référence doit être liée car une définition de a apparaît dans le scope courant.

- il se peut que des références liées soient incorrectement liées car aucune définition n'apparaît dans le scope référé. Par exemple, dans l'expression $\{a = 1, b = c_0\}$, b définit une référence liée à c alors qu'aucune définition de c n'apparaît dans le scope courant ni dans aucune scope englobant.

Nous allons donc définir un sous-langage, le langage $\Sigma_{\mathcal{C}}$ tel que tous les éléments de $\Sigma_{\mathcal{C}}$ soient correctement liés.

X.1.2 Le langage $\Sigma_{\mathcal{C}}$

Avant de définir ce qu'est un terme correct, il nous est nécessaire de définir un certain nombre de fonctions et de prédicats auxiliaires. Nous commençons par la définition de ces fonctions auxiliaires puis nous définirons ensuite la propriété de correction.

Rappel des conventions sur les listes. Nous utilisons une notation proche de celle adoptée par le langage CAML pour la spécification et la manipulation des listes : $\texttt{List}(X)$ représente l'ensemble des listes d'éléments de X ; la liste vide est notée $<>$; la concaténation d'un atome h et d'une liste t se note $h{::}t$; l'accès au n^{e} élément d'une liste l se note $\texttt{nth}(n, l)$ la liste privée de son premier élément se note $\texttt{tl}(l)$ et enfin, le nombre d'éléments d'une liste l se note $\texttt{lg}(l)$. Par convention, $\texttt{tl}(<>) = <>$ et les itérés de $\texttt{tl}()$ se définissent par : $\texttt{tl}^0(l) = l$, $\texttt{tl}^1(l) = \texttt{tl}(l)$ et $\texttt{tl}^n(l) = \texttt{tl}^{(n-1)}(\texttt{tl}(l))$.

L'évaluation d'amalgames nécessite parfois d'effectuer des liaisons (la transformation de références libres en références liées) lors de l'évaluation d'une expression (cf. section VIII.6.3.4). Il va donc être nécessaire de spécifier une fonction de liaison des références libres des expressions. Pour permettre cette liaison, il est nécessaire de connaître, pour une expression qui se trouve dans un certain scope, l'ensemble des definiendum au niveau de l'expression. L'ensemble des definiendum est appelé l'*environnement de liaison*.

DÉFINITION X.2 **(Scope et environnement de liaison)**
Un *scope* est un élément de $\textsc{Scope} = \{\star^n, \star_n\} \cup \mathcal{P}(\textsc{Id})$. Un *environnement de liaison* est un élément de $\mathcal{E}_{\mathrm{L}} = \texttt{List}(\textsc{Scope})$.

REMARQUE La constante $\star_n$ est utilisée quand on ne connaît pas les liaisons, des réductions devant encore avoir lieu ; la constante $\star^n$ est utilisée quand on ne connaît pas les liaisons, ce qui correspond à un « trou » dans l'expression, c'est-à-dire une référence libre.

On ne distingue généralement pas les constantes $\star_n$et $\star^n$. On note alors ces constantes par $\star$ qui désigne indifféremment un de ces deux éléments. Par exemple, intuitivement, l'environnement de liaison de $\bullet$ dans l'expression :

$$\{a = \{f = e, \underbrace{b = \{c = 1, d = f\}}_{A}\}, e = 1, h = a_1 \,\textbf{.}\, (z^1 \,\textbf{.}\, \bullet)\} \tag{1}$$

est $< \star^n, \star_n, \{a, e, h\} >$; au niveau de la définition A il est égal à : $< \{f, b\}, \{a, e, h\} >$.

DÉFINITION X.3 **(La fonction** $\texttt{Dom}()$**)**
Nous avons besoin d'une fonction permettant de connaître les liaisons induites par une expression. La fonction totale $\texttt{Dom}()$ de Σ dans $\textsc{Scope}$ est définie par :

$$\begin{array}{lcl}
\texttt{Dom}(l, id^n) & = & \star^n \\
\texttt{Dom}(l, id_n) & = & \star_n \\
\texttt{Dom}(l, \{\dots, id = e, \dots\}) & = & \{\dots, id, \dots\} \\
\texttt{Dom}(l, e \,\#\, e') & = & \texttt{Comb}(\texttt{Dom}(e), \texttt{Dom}(e')) \\
\texttt{Dom}(l, e \,\textbf{.}\, e') & = & \star_n
\end{array}$$

où la fonction $\texttt{Comb}() : \textsc{Scope} \mapsto \{\star^n, \star_n\}$ est la fonction qui définit la liaison d'une concaténation :

$$\begin{array}{lcl}
\texttt{Comb}(\star_n, p) & = & \star_n \\
\texttt{Comb}(p, \star_n) & = & \star_n \\
\texttt{Comb}(\{\dots\}, \{\dots\}) & = & \star_n \\
\texttt{Comb}(p, p') & = & \star^n
\end{array}$$

REMARQUE Par exemple, le domaine du système $P \equiv \{a = 1, b = a_0, c = b^1\}$ est :

$$\texttt{Dom}(P) = \{a, b, c\}$$

Nous devons définir une fonction qui cherche récursivement dans un environnement de liaison un élément id. Si l'élément n'existe pas ou si une constante $\star$ est rencontrée alors on renvoie $\star$, sinon on renvoie la position dans la liste de l'ensemble qui contient id.

DÉFINITION X.4 **(La fonction index())**
La fonction $\mathtt{index}() : \mathcal{E}_{\mathrm{L}} \times \mathrm{ID} \times \mathbb{N} \mapsto \{\star\} \cup \mathbb{N}$ est définie :

$$\begin{array}{lcl}
\mathtt{index}(<>, id, n) & = & \star \\
\mathtt{index}(\star^n :: t, id, n) & = & \star \\
\mathtt{index}(\star_n :: t, id, n) & = & \star \\
\mathtt{index}(h :: t, id, n) & = & \mathtt{if}\, id \in h\, \mathtt{then}\, n\, \mathtt{else}\, \mathtt{index}(t, id, n+1)
\end{array}$$

DÉFINITION X.5 **(Le prédicat okRefLibre())**
Le prédicat $\mathtt{okRefLibre}(l, a)$ est la fonction totale de $\mathcal{E}_{\mathrm{L}} \times \mathrm{ID} \mapsto \mathrm{BOOL}$ qui vérifie qu'il n'y a pas de définition de a dans la liste l. Il est défini comme :

$$\mathtt{okRefLibre}(l, a) = (\mathtt{index}(l, a, 0) == \star)$$

DÉFINITION X.6 **(Le prédicat okRefLiée())**
Contrairement à une référence libre, une référence liée doit apparaître dans le scope qu'elle réfère, et non pas dans un quelconque scope englobant. Le prédicat $\mathtt{okRefLiée}(l, e)$, fonction totale de $\mathcal{E}_{\mathrm{L}} \times \mathrm{ID} \mapsto \mathrm{BOOL}$, est donc vrai si la définition de e apparaît dans le scope définit par le premier élément de e :

$$\begin{array}{lcl}
\mathtt{okRefLiée}(<>, a) & = & \mathtt{false} \\
\mathtt{okRefLiée}(\star :: l', a) & = & \mathtt{false} \\
\mathtt{okRefLiée}(s :: l', a) & = & (a \in s)
\end{array}$$

DÉFINITION X.7 **(Le prédicat $\mathtt{correct_B}()$)**
Le prédicat $\mathtt{correct_B}(e) : \mathcal{E}_{\mathrm{L}} \times \Sigma \mapsto \mathrm{BOOL}$ est une fonction totale qui vérifie que toutes les références liées correspondent à un definiens :

$$\begin{array}{lcl}
\mathtt{correct_B}(l, a^p) & = & \mathtt{true} \\
\mathtt{correct_B}(l, a_p) & = & \mathtt{okRefLiée}(\mathtt{tl}^p(l), a) \\
\mathtt{correct_B}(l, \{\ldots, id = e, \ldots\}) & = & \bigwedge \mathtt{correct_B}(\{\ldots, id, \ldots\} :: l, e) \\
\mathtt{correct_B}(l, e \,\#\, e') & = & \mathtt{correct_B}(l, e) \wedge \mathtt{correct_B}(l, e') \\
\mathtt{correct_B}(l, e \,.\, e') & = & \mathtt{correct_B}(l, e) \wedge \mathtt{correct_B}(\mathtt{Dom}(e) :: l, e')
\end{array}$$

et $\mathtt{correct_B}(e)$ abrège $\mathtt{correct_B}(<>, e)$. L'expression $\bigwedge \ldots$ doit s'interpréter comme la conjonction sur les definiens e du système.

DÉFINITION X.8 **(Le prédicat $\mathtt{correct_F}()$)**
Le prédicat $\mathtt{correct_F}(e) : \mathcal{E}_{\mathrm{L}} \times \Sigma \mapsto \mathrm{BOOL}$ est une fonction totale qui vérifie que toutes les références libres ne correspondent à aucun definiens :

$$\begin{array}{lcl}
\mathtt{correct_F}(l, a^p) & = & \mathtt{okRefLibre}(\mathtt{tl}^p(l), a) \\
\mathtt{correct_F}(l, a_p) & = & \mathtt{true} \\
\mathtt{correct_F}(l, \{\ldots, id = e, \ldots\}) & = & \bigwedge \mathtt{correct_F}(\{\ldots, id, \ldots\} :: l, e) \\
\mathtt{correct_F}(l, e \,\#\, e') & = & \mathtt{correct_F}(l, e) \wedge \mathtt{correct_F}(l, e') \\
\mathtt{correct_F}(l, e \,.\, e') & = & \mathtt{correct_F}(l, e) \wedge \mathtt{correct_F}(\mathtt{Dom}(e) :: l, e')
\end{array}$$

L'expression $\mathtt{correct_F}(e)$ abrège $\mathtt{correct_F}(<>, e)$.

On peut définir à présent les termes corrects : ce sont les éléments de Σ où les références libres et liées sont correctes : c'est-à-dire où il existe une définition pour toute référence liée et où une référence libre à une variable x n'est pas dans la portée d'une définition de x.

DÉFINITION X.9 **(Le prédicat correct())**
Le prédicat $\mathtt{correct}(e) : \Sigma \mapsto \mathrm{BOOL}$, fonction totale, est la conjonction des prédicats de correction des termes libres et liés :

$$\begin{array}{lcl}
\mathtt{correct}(e) & = & \mathtt{correct}(<>, e) \\
\mathtt{correct}(l, e) & = & \mathtt{correct_B}(l, e) \wedge \mathtt{correct_F}(l, e)
\end{array}$$

REMARQUE Pour justifier ces définitions, rappelons nous que nous avons vu dans la section VIII.6.3.3 que les opérations de sélection et de concaténation ne devaient pas permettre de liaison quand l'expression n'était pas complète. Reprenons l'exemple de la section VIII.6.3.3 après substitution de a_2 en x^0 :

$$\{a = x^0, b = \{x = 1\} \centerdot (y^0 \centerdot a_2)\} \quad \to \quad \underbrace{\{a = x^0, b = \{x = 1\} \centerdot (y^0 \centerdot x^0)\}}_{A}$$

Il n'est pas possible de changer x^0 en x_1 car il se peut que y^0 devienne plus tard un système et définisse une valeur pour x. Nous avons aussi vu que la définition d'une sélection créait un scope (cf. section VIII.6.1). L'expression A ci-dessus est correcte, malgré la définition de x dans l'expression $\{x = 1\}$ car elle est inatteignable par x^0 en l'état. Comme nous l'avons vu, deux constantes spéciales permettent de spécifier un scope qui n'est pas encore connu : $\star^n$ et $\star_n$. Une référence est donc libre si elle n'apparaît pas dans un scope englobant ou bien si une « barrière » matérialisée par $\star$ apparaît dans un scope englobant.

DÉFINITION X.10 **(Le langage $\Sigma_{\mathcal{C}}$)**
Le langage $\Sigma_{\mathcal{C}}$, sous-langage de Σ est tel que chaque élément de $\Sigma_{\mathcal{C}}$ est correct :

$$\Sigma_{\mathcal{C}} = \{x \mid x \in \Sigma \wedge \texttt{correct}(x)\}$$

X.2 Le langage $\Sigma_{\mathcal{I}}$: Σ avec du sucre syntaxique

Le langage Σ (et plus particulièrement son sous-ensemble $\Sigma_{\mathcal{C}}$) est extrêmement contraignant : on ne peut pas écrire d'expression utilisant un identificateur sans référence explicite à sa définition. Par exemple, l'expression suivante :

$$\{a = 1, b = a\} \tag{2}$$

ne peut être définie. Seule sa version dans Σ :

$$\{a = 1, b = a_0\}$$

peut être définie. Cette particularité de Σ ne facilite pas la définition d'expressions complexes dans lesquelles certaines liaisons sont implicites (telles que celle que nous venons de définir) et on veut pouvoir les écrire sans nécessairement connaître dans quel scope la définition se trouve, celui-ci étant spécifié implicitement et sans ambiguïté par le contexte.

À cette fin, nous définissons un sur-ensemble de Σ : le langage $\Sigma_{\mathcal{I}}$ où l'utilisation des identificateurs est possible. Maintenant, l'expression (2) peut être définie. La figure 1 montre le schéma d'inclusion des langages Σ, $\Sigma_{\mathcal{C}}$ et $\Sigma_{\mathcal{I}}$.

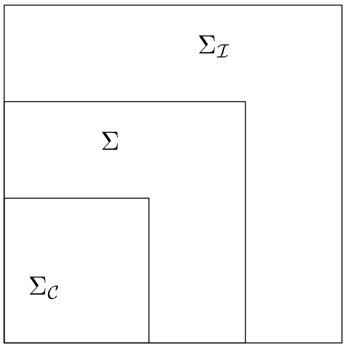


FIG. 1 –: *Le schéma d'inclusion des langages $\Sigma_{\mathcal{I}}$, Σ, $\Sigma_{\mathcal{C}}$.*

Le langage $\Sigma_{\mathcal{I}}$ correspond en fait à l'ajout de « sucre syntaxique » au langage Σ, avec un identificateur x se traduisant par :

- soit une référence liée, i.e. désignant la définition la plus proche,

– soit une référence libre (s'il n'existe pas de définition accessible) dont l'indice est 0 par convention.

Par exemple, l'expression :

$$\{a = 1, b = a, c = d\}$$

qui est un terme de $\Sigma_{\mathcal{I}}$ se réécrit en le terme de Σ :

$$\{a = 1, b = a_0, c = d^0\}$$

car il existe une définition de a mais aucune définition pour d.

Définition X.11 **(Le langage $\Sigma_{\mathcal{I}}$)**
Formellement, le langage $\Sigma_{\mathcal{I}}$ est défini par la grammaire :

$$\begin{array}{lll}
\textsc{Ref} & & \\
p & ::= & id^n \mid id_n \mid id \\
\textsc{Sys} & & \\
s & ::= & \{\ldots, id = e, \ldots\} \\
\Sigma_{\mathcal{I}} & & \\
e & ::= & id \mid p \mid s \mid e \,\#\, e' \mid e \,.\, e'
\end{array}$$

X.2.1 La liaison des expressions

La fonction de liaison $\mathtt{bind}(e)$ effectue la transformation de certaines références libres en références liées.

Définition X.12 **(La fonction de liaison $\mathtt{bind}()$)**
La fonction $\mathtt{bind}()$ de $\mathcal{E}_{\mathrm{L}} \times \Sigma$ dans Σ est une fonction totale spécifiée par :

$$\begin{array}{lll}
\mathtt{bind}(l, id_n) & \equiv & id_n \\
\mathtt{bind}(l, id^n) & \equiv & \mathtt{if}(n > \mathtt{lg}(l))\,\mathtt{then}\, id^n \,\mathtt{else}\, \mathtt{bind_F}(l, id, n) \\
\mathtt{bind}(l, \{\ldots, id = v, \ldots\}) & \equiv & \{\ldots, id = \mathtt{bind}(\{\ldots, id, \ldots\} :: l, v), \ldots\} \\
\mathtt{bind}(l, e \,\#\, e') & \equiv & \mathtt{bind}(l, e) \,\#\, \mathtt{bind}(l, e') \\
\mathtt{bind}(l, e \,.\, e') & \equiv & \mathtt{bind}(l, e) \,.\, \mathtt{bind}(\mathtt{Dom}(e) :: l, e')
\end{array}$$

La fonction $\mathtt{bind_F}()$ de Σ dans Σ est définie comme :

$$\mathtt{bind_F}(l, id, n) = \mathtt{if}\, \iota == \star \,\mathtt{then}\, id^n \,\mathtt{else}\, id_\iota \qquad \text{avec } \iota = \mathtt{index}(\mathtt{tl}^n(l), id, n)$$

Remarque Il apparaît que la stratégie de liaison privilégie les liaisons *locales* puis *globales*, en accord avec le choix décrit en section VIII.6.2.

La liaison d'une référence libre id^n s'effectue en remontant *au moins* n scopes englobants puis la liaison est effectuée avec la première définition ayant pour nom id. Si la constante $\star$ est rencontrée, alors c'est qu'une liaison future reste possible et alors la référence doit rester libre. La liaison d'une sélection respecte ce schéma : la liaison de $e \,.\, e'$ effectue la liaison de e' dans les définitions de $\mathtt{Dom}(e)$.

X.2.2 Une fonction de traduction d'un terme de $\Sigma_{\mathcal{I}}$ en un terme de $\Sigma_{\mathcal{C}}$

La fonction $\mathtt{trad}()$ permet de traduire un terme de $\Sigma_{\mathcal{I}}$ en un terme de $\Sigma_{\mathcal{C}}$. Les termes de $\Sigma_{\mathcal{I}}$ peuvent se trouver dans trois cas différents :

1. Le terme est correctement défini et correctement lié. Dans ce cas, la fonction de traduction est la fonction identité.

2. Le terme est correctement défini, mais il est insuffisamment lié. Dans ce cas, les liaisons doivent être établies entre les références libres et les définitions qui peuvent apparaître. Par exemple, dans l'expression :

$$\{a = 1, b = \{a = 2, c = a^0, d = a^1\}\}$$

la première référence libre à a^0, doit être une référence liée, de même que pour la seconde référence libre a^1. La traduction de l'expression précédente en un terme de $\Sigma_{\mathcal{C}}$ est :

$$\{a = 1, b = \{a = 2, c = a_0, d = a_1\}\}$$

3. Le terme est incorrectement défini. Dans ce cas, la traduction du terme n'est pas possible car on ne peut déterminer quelles sont les liaisons correctes. Par exemple, dans le terme :

$$\{a = 1, b = c_0\}$$

où la référence liée c_0 ne correspond à aucune définition. Il n'est pas souhaitable de traduire une telle référence erronée en une référence libre car un tel terme signifie sûrement une erreur de définition. Dans ce cas, l'opération de traduction du terme échoue.

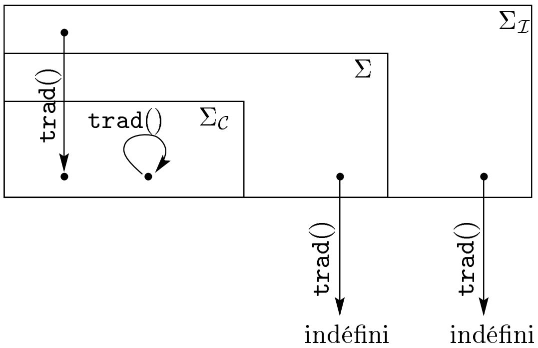


FIG. 2 –: *La traduction d'un terme de $\Sigma_{\mathcal{I}}$: si la fonction de traduction est indéfinie, alors la traduction du terme est une erreur, sinon le terme traduit est un élément de $\Sigma_{\mathcal{C}}$.*

Si l'on reprend le schéma d'inclusion de la section précédente, nous pouvons donner une représentation graphique de la fonction de traduction qui associe à un terme de $\Sigma_{\mathcal{I}}$ un terme de $\Sigma_{\mathcal{C}}$ ou bien une erreur (cf. figure 2). Un terme de Σ qui n'est pas un élément de $\Sigma_{\mathcal{C}}$ est un terme incorrect par définition et donc sa traduction doit échouer. Un terme correct doit se traduire en lui même, donc la fonction de traduction est la fonction identité dans $\Sigma_{\mathcal{C}}$. Un terme de $\Sigma_{\mathcal{I}}$ est soit traduisible correctement où alors il est incorrect. L'erreur de la traduction ne correspond pas à un terme de $\Sigma_{\mathcal{C}}$ mais à un échec de la fonction. En effet, on ne désire pas évaluer les termes erronés ni les propager lors de l'évaluation (comme c'est le cas dans le traitement de l'évaluation du λN-calcul). Nous modélisons cela par une fonction *partielle*. Seuls les termes traduisibles en un terme correct ont une image par cette fonction.

DÉFINITION X.13 **(La fonction `trad()`)**

$$\texttt{trad}(e) \quad \equiv \quad \texttt{bind}(\texttt{IdToRef}(e)) \qquad \text{si } \texttt{correct}(\texttt{bind}(\texttt{IdToRef}(e))) = \texttt{true}$$
$$\text{et indéfini sinon.}$$

où la fonction $\texttt{IdToRef}() : \Sigma_{\mathcal{I}} \mapsto \Sigma$ effectue la traduction des identificateurs id en références libres id^0.

DÉFINITION X.14 **(La fonction `IdToRef()`)**

$$\begin{array}{lcl} \texttt{IdToRef}(id) & \equiv & id^0 \\ \texttt{IdToRef}(id^p) & \equiv & id^p \\ \texttt{IdToRef}(id_p) & \equiv & id_p \end{array}$$

$$\begin{array}{lcl}
\texttt{IdToRef}(\{\ldots, id = e, \ldots\}) & \equiv & \{\ldots, id = \texttt{IdToRef}(e), \ldots\} \\
\texttt{IdToRef}(e \# e') & \equiv & \texttt{IdToRef}(e) \# \texttt{IdToRef}(e') \\
\texttt{IdToRef}(e \;\texttt{.}\; e') & \equiv & \texttt{IdToRef}(e) \;\texttt{.}\; \texttt{IdToRef}(e')
\end{array}$$

La fonction `IdToRef()` est une fonction totale de $\Sigma_{\mathcal{I}}$ dans Σ et `bind()` est une fonction totale de Σ dans Σ. La fonction `trad()` est donc la restriction d'une fonction totale aux éléments de $\Sigma_{\mathcal{I}}$ dont l'image est correcte. Par suite, la fonction `trad()` est une fonction partielle de $\Sigma_{\mathcal{I}}$ dans Σ.

Par construction, `trad()` traduit bien un terme de $\Sigma_{\mathcal{I}}$ en un terme de $\Sigma_{\mathcal{C}}$. Si le terme *n'est pas traduisible*, alors la traduction n'est pas définie. On aimerait néanmoins s'assurer que la traduction d'un terme de $\Sigma_{\mathcal{C}}$ laisse ce terme inchangé.

LEMME X.1 **(La traduction est conservatrice)**
La propriété suivante est vraie :

$$\forall e \in \Sigma_{\mathcal{C}},\ \texttt{trad}(e) \quad \equiv \quad e$$

La preuve de ce lemme est faite page 144.

X.3 Les règles de réduction des termes de Σ

X.3.1 Les fonctions et prédicats auxiliaires de la sémantique opérationnelle

Dans la section suivante, nous décrivons les fonctions nécessaire à la définition de l'évaluateur des termes des amalgames. Puis nous spécifierons les règles et établirons les principales propriétés de la réduction.

X.3.1.1 L'environnement d'évaluation

La relation de réduction sur les termes de $\Sigma_{\mathcal{C}}$ procède par substitution des definiens aux références liées et par simplification des opération de concaténation et de sélection. Il est nécessaire, quand on réduit une sous-expression, de connaître les définitions qui apparaissent dans un scope englobant. Pour disposer de cette information, la relation de réduction utilise un *environnement d'évaluation*. Comme les expressions peuvent être imbriquées, nous utilisons une structure de liste pour l'environnement ; chaque élément de cette liste est une fonction partielle qui associe à un identificateur son definiens.

DÉFINITION X.15 **(Environnement et environnement d'évaluation)**
Un *environnement* est fonction partielle σ de signature $\textsc{Id} \to \Sigma$. On note $[id \mapsto e]$ la fonction qui associe l'expression e à l'identificateur id et qui est indéfinie pour tous les autres identificateurs. Si σ est un environnement, alors $\sigma' = \sigma \uplus [id' \mapsto e']$ est l'environnement tel que :

$$\sigma'(id) = \begin{cases} e' & \text{si } id = id' \\ \sigma(id) & \text{sinon} \end{cases}$$

L'écriture $[id \mapsto e, \ldots, id' \mapsto e']$ est une abréviation de $[id \mapsto e] \uplus \cdots \uplus [id' \mapsto e']$.

Un *environnement d'évaluation*, noté ρ, est un élément de $\mathcal{E}_{\mathrm{E}} = \texttt{List}((\textsc{Id} \to \Sigma) \cup \{\star_n, \star^n\})$.

L'opération de mise à jour des références liées hors de leur expression est effectuée par la fonction `Lift()`. Rappelons que l'opération de *lift* des environnements permet que les définitions d'un système à un niveau n soient toujours accessibles à un niveau supérieur (cf. section VIII.6.3.1). L'opération de lift d'une expression doit changer les références dans un terme qui réfèrent à l'extérieur d'un terme, mais pas celles qui sont liées dans le terme.

DÉFINITION X.16 **(La fonction `Lift()`)**
On définit la fonction totale `Lift()` de Σ dans Σ :

$$\texttt{Lift}(e) = \texttt{Lift}(0, e)$$

où la fonction $\texttt{Lift}(n, e)$ propage le lift sur l'expression en incrémentant la valeur de n qui correspond au nombre de scopes rencontrés, chaque fois que cela est nécessaire. La fonction $\texttt{Lift}()$ de $\mathbb{N} \times \Sigma$ dans Σ :

$$\begin{array}{lll}
\texttt{Lift}(n, id_m) & \equiv & id_p \quad \text{avec } p = (\texttt{if}\, m \geq n\, \texttt{then}\, m+1\, \texttt{else}\, m) \\
\texttt{Lift}(n, id^n) & \equiv & id^n \\
\texttt{Lift}(n, \{\ldots, id = v, \ldots\}) & \equiv & \{\ldots, id = \texttt{Lift}(n+1, v), \ldots\} \\
\texttt{Lift}(n, e \# e') & \equiv & \texttt{Lift}(n, e) \# \texttt{Lift}(n, e') \\
\texttt{Lift}(n, e \,.\, e') & \equiv & \texttt{Lift}(n, e) \,.\, \texttt{Lift}(n+1, e')
\end{array}$$

Une référence n'est incrémentée que si elle réfère à un scope qui est supérieur au nombre de scopes traversés. On est ainsi assuré que seules les liaisons qui réfèrent une définition hors de leur expression, seront mise à jour.

La fonction $\texttt{Nenv}(\rho, e)$ de construction d'un nouvel environnement de réduction à partir de l'environnement existant ρ et de l'expression e est définie de la façon suivante :

Définition X.17 **(La fonction $\texttt{Nenv}()$)**
La fonction $\texttt{Nenv}()$ de $\mathcal{E}_{\mathrm{E}} \times \Sigma$ dans $\mathcal{E}_{\mathrm{E}}$ est définie :

$$\begin{array}{lll}
\texttt{Nenv}(\rho, \{\ldots, id = e, \ldots\}) & = & [\ldots, id \mapsto e, \ldots] :: \texttt{Lift}_{\texttt{E}}(\rho) \\
\texttt{Nenv}(\rho, v) & = & \texttt{Dom}(v) :: \texttt{Lift}_{\texttt{E}}(\rho) \\
\texttt{Lift}_{\texttt{E}}(<>) & = & <> \\
\texttt{Lift}_{\texttt{E}}(\star :: t) & = & \star :: \texttt{Lift}_{\texttt{E}}(t) \\
\texttt{Lift}_{\texttt{E}}(h :: t) & = & (\lambda x.\, \texttt{Lift}(h(x))) :: \texttt{Lift}_{\texttt{E}}(t)
\end{array}$$

La liaison d'une expression nécessitera de connaître les liaisons qui apparaissent dans un environnement d'évaluation. La fonction $\texttt{Dom}_{\texttt{E}}()$ permet de récupérer les identificateurs qui définissent des expressions.

Définition X.18 **(Les fonctions $\texttt{Dom}_{\texttt{E}}()$ et $\texttt{Def}()$)**
On définit la fonction $\texttt{Dom}_{\texttt{E}}()$ de $\mathcal{E}_{\mathrm{E}}$ dans $\mathcal{E}_{\mathrm{L}}$ et $\texttt{Def}()$ de $(\textsc{Id} \mapsto \Sigma) \mapsto \mathcal{P}(\textsc{Id})$ par :

$$\begin{array}{lll}
\texttt{Dom}_{\texttt{E}}(<>) & = & <> \\
\texttt{Dom}_{\texttt{E}}(\star :: t) & = & \star :: \texttt{Dom}_{\texttt{E}}(t) \\
\texttt{Dom}_{\texttt{E}}(\sigma :: t) & = & \texttt{Def}(\sigma) :: \texttt{Dom}_{\texttt{E}}(t) \\
\texttt{Def}([id \mapsto \ldots, \ldots, id' \mapsto \ldots]) & = & \{id, \ldots, id'\}
\end{array}$$

X.3.1.2 La garde de l'évaluation

Définition X.19 **(Le prédicat $\texttt{C}()$)**
Le prédicat $\texttt{C}() : \mathcal{E}_{\mathrm{L}} \times \Sigma \mapsto \textsc{Bool}$ est défini par induction structurelle sur la structure des termes :

$$\begin{array}{lll}
\texttt{C}(l, id_n) & = & \texttt{false} \\
\texttt{C}(l, id^n) & = & \star_n \notin \texttt{tl}^n(l) \\
\texttt{C}(l, \{\ldots, id = v, \ldots\}) & = & \bigwedge \texttt{C}(\texttt{Dom}(\{\ldots, id = v, \ldots\}) :: l, v) \\
\texttt{C}(l, e \# e') & = & ((e \notin \textsc{Sys}) \vee (e' \notin \textsc{Sys})) \wedge \texttt{C}(\texttt{Dom}(e) :: l, e') \\
\texttt{C}(l, e \,.\, e') & = & (e \notin \textsc{Sys}) \wedge \texttt{C}(l, e) \wedge \texttt{C}(l, e')
\end{array}$$

Remarque Le prédicat $\texttt{C}(\texttt{Dom}_{\texttt{E}}(\rho), v)$, qui est utilisé pour la spécification de la sémantique est vrai quand le terme v ne peut plus être réduit dans un environnement ρ. Ce résultat est démontré par le lemme (X.2).

X.3.2 Spécification de l'évaluateur

La relation ternaire $\epsilon \vdash e \rightarrow e'$ de $\mathcal{E}_{\mathrm{E}} \times \Sigma \times \Sigma$ exprime que e se réduit en un pas de réduction en e' dans un environnement d'évaluation ρ. On note ϵ l'environnement d'évaluation vide : $<>_{\mathcal{E}_{\mathrm{E}}}$. Si $\epsilon \vdash e \rightarrow e'$, alors on note plus simplement $e \rightarrow e'$. Nous adoptons les conventions suivantes :

- $\rho \vdash e \not\to e'$: représente *l'impossibilité* de réduire e en e'.
- $e \downarrow \rho$: est un prédicat qui est vrai si $\nexists e' \in \Sigma, \rho \vdash e \to e'$. On dit que e est un terme en *ρ-forme normale* car il n'est plus possible de le réduire dans ρ. Un terme est en *forme normale* s'il est en ϵ-forme normale. On abrège $e \downarrow \epsilon$ par $e \downarrow$.
- $\rho \vdash e \to_n e'$: représente la *composition* de n réductions. Autrement dit, $\rho \vdash e \to_n e'$ veut dire qu'il existe $e_0, \ldots, e_n \in \Sigma$ où $e_0 \equiv e$ et $e_n \equiv e'$ avec $\rho \vdash e_0 \to e_1, \ldots, \rho \vdash e_{n-1} \to e_n$. Par convention, $\rho \vdash e \to_0 e'$ veut dire que $e \downarrow \rho$ et $e \equiv e'$.
- $\rho \vdash e \twoheadrightarrow e'$: si et seulement si $\rho \vdash e \to_n e'$ et $\rho \downarrow e'$. On abrège $\epsilon \vdash e \twoheadrightarrow e'$ par $e \twoheadrightarrow e'$.

Les règles (1) définissent une sémantique opérationnelle des amalgames dans le style SOS [Plo81, Ast91]. L'évaluation se fait par induction sur la structure d'un terme. La sémantique que nous proposons est partiellement stricte : un programme qui termine correspond à un terme qui possède une forme normale. Certains termes peuvent posséder une forme normale alors qu'un de leurs sous-termes n'en possèdent pas ; un tel sous-terme donne lieu à une série infinie de réductions. Par exemple, cela peut être le cas si le sous-terme se trouve à gauche d'une opération de sélection.

$$Ref^n : \quad \frac{}{\rho \vdash id^n \to id^n} \quad \star_n \in \mathtt{tl}^n(\rho)$$

$$Ref_C : \quad \frac{u = \mathtt{nth}(n+1,\rho)(id) \quad u' = \mathtt{bind}(\mathtt{Dom}_{\mathtt{E}}(\rho), u)}{\rho \vdash id_n \to u'} \quad \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), u)$$

$$Ref_{\neg C} : \quad \frac{u = \mathtt{nth}(n+1,\rho)(id)}{\rho \vdash id_n \to id_n} \quad \neg\, \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), u)$$

$$\{\ldots\} : \quad \frac{\rho' = \mathtt{Nenv}(\rho, \{\ldots, id = u, \ldots\}) \quad \overset{\cdots}{\rho' \vdash u \to_n u'}}{\rho \vdash \{\ldots, id = u, \ldots\} \to \{\ldots, id = u', \ldots\}} \quad n \in \{0,1\} \wedge \sum n \geq 1$$

$$\#_1 : \quad \frac{s = \mathtt{bind}(\mathtt{Dom}_{\mathtt{E}}(\rho), \{\ldots, id = u, \ldots, id' = u', \ldots\})}{\rho \vdash \{\ldots, id = u, \ldots\} \# \{\ldots, id' = u', \ldots\} \to s} \quad \begin{cases} id \neq id' \\ \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), \{\ldots, id = u, \ldots\}) \\ \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), \{\ldots, id' = u', \ldots\}) \end{cases}$$

$$\#_2 : \quad \frac{\rho \vdash u \to_n u' \quad \rho \vdash v \to_m v'}{\rho \vdash u \# v \to u' \# v'} \quad n, m \in \{0,1\} \wedge n + m \geq 1$$

$$\centerdot_{S_1} : \quad \frac{}{\rho \vdash \{\ldots, id = u, \ldots\} \centerdot v \to v} \quad \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\mathtt{Nenv}(\rho, \{\ldots, id = u, \ldots\})), v)$$

$$\centerdot_{S_2} : \quad \frac{\rho \vdash u \to_n u' \quad \mathtt{Nenv}(\rho, u') \vdash v \to v'}{\rho \vdash u \centerdot v \to u' \centerdot v'} \quad n \in \{0,1\} \wedge u \in \textsc{Sys}$$

$$\centerdot_{\neg S_1} : \quad \frac{\rho \vdash u \to u' \quad v' = \mathtt{bind}(\mathtt{Dom}_{\mathtt{E}}(\mathtt{Nenv}(\rho, u')), v)}{\rho \vdash u \centerdot v \to u' \centerdot v'} \quad u \notin \textsc{Sys} \wedge u' \in \textsc{Sys}$$

$$\centerdot_{\neg S_2} : \quad \frac{\rho \vdash u \to_m u' \quad \mathtt{Nenv}(\rho, u') \vdash v \to_n v'}{\rho \vdash u \centerdot v \to u' \centerdot v'} \quad \begin{cases} u \notin \textsc{Sys} \wedge u' \notin \textsc{Sys} \\ n, m \in \{0,1\} \wedge n + m \geq 1 \end{cases}$$

RÈG. 1: *La sémantique opérationnelle de l'évaluateur partiellement strict des amalgames.*

X.4 Trois propriétés importantes de l'évaluation

La spécification des règles de réduction et des prédicats utilisés nous permet la démonstration de trois propriétés sur l'évaluation : les règles de réduction sont déterministes, le prédicat $\mathtt{C}(e)$ implique que e est en forme normale et les règles de réduction préservent la correction.

X.4.1 Déterminisme de l'évaluation

Nous aimerions savoir si les règles de réduction que nous venons de décrire réduisent toujours un même terme en un terme identique lors de deux réductions de ce terme et ceci dans le cas où la réduction de l'expression ne diverge pas. Ce problème est celui du *déterminisme* de la réduction, qui assure que la forme normale d'un terme est unique.

THÉORÈME X.1 **(Déterminisme de l'évaluation)**
La réduction d'un terme de Σ est déterministe :

$$\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \rho \vdash e \twoheadrightarrow e' \wedge \rho \vdash e \twoheadrightarrow e'' \Rightarrow e' \equiv e''$$

En particulier, si $e \twoheadrightarrow e'$, alors e' est unique. La preuve de ce théorème est faite page 145.

X.4.2 Le prédicat $\mathtt{C}()$ et la notion de forme normale

Le lemme suivant est important car il permet d'évaluer efficacement $e \downarrow \rho$ ou bien de déterminer si $p \neq 0$ dans $\rho \vdash e \rightarrow_p e'$ à l'aide d'un critère purement syntaxique : $\mathtt{C}()$. En effet, le calcul du prédicat $e \downarrow \rho$ demande *a priori* la vérification que $\forall e' \in \Sigma, \rho \vdash e \not\twoheadrightarrow e'$. Évidemment, une vérification exhaustive n'est pas envisageable. L'intérêt de ce lemme est donc de transformer une condition difficilement calculable en un prédicat portant sur la structure d'un terme, pour les termes qui sont $\mathtt{correct}()$ relativement à ρ. Le théorème de la section suivante montre que quand on part d'un terme de $\Sigma_{\mathcal{C}}$, cette propriété est vérifiée pour tous les termes rencontrés lors de l'évaluation.

LEMME X.2 **($\mathtt{C}()$ et forme normale)**

$$\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \mathtt{correct}(\mathtt{Dom}_{\mathtt{E}}(\rho), e) \Rightarrow \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), e) = e \downarrow \rho$$

La preuve de ce lemme est faite page 146.

REMARQUE Voici un exemple tel que $e \downarrow \rho \neq \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), e)$: l'expression $id_1 \downarrow \epsilon$ est vraie. En effet, la fonction $\mathtt{nth}(2, \epsilon)$ n'est pas définie et donc aucune règle ne peut s'appliquer. De plus, par définition, $\mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), id_1)$ est faux. On vérifie bien que $\mathtt{correct}(\mathtt{Dom}_{\mathtt{E}}(\epsilon), id_1)$ est faux.

X.4.3 L'évaluation préserve la correction

L'évaluation d'un terme de $\Sigma_{\mathcal{C}}$ a pour résultat un terme de $\Sigma_{\mathcal{C}}$, soit :

THÉORÈME X.2 **(L'évaluation préserve la correction)**

$$\forall e \in \Sigma_{\mathcal{C}}, \quad e \twoheadrightarrow e' \Rightarrow e' \in \Sigma_{\mathcal{C}}$$

La preuve de ce théorème[1] est faite pages 149–157.

1. Cette propriété est analogue à la *subject reduction* d'un type qui demande à ce que le type d'une expression se conserve lors des réductions de l'expression. On peut d'ailleurs voir les éléments de $\Sigma_{\mathcal{C}}$ comme les habitants d'un type et le prédicat $\mathtt{correct}()$ comme l'inférence de ce type.

X.5 Discussion sur les règles de réduction

Les règles de réduction spécifiées en (1) sont de trois sortes :

1. le traitement des références : l'opération de conservation d'une référence libre (règle (3)), les opérations à effectuer sur les identificateurs liés (règles (4) et (5)), la définition de liaisons des identificateurs avec le système (règle (6)),
2. les opérations de réduction des termes impliquant l'opérateur de concaténation (règles (8) et (9)),
3. les opérateurs de réduction des termes impliquant l'opérateur de sélection (règles (10), (11), (12) et (13)).

Afin de mieux saisir les règles de réduction spécifiées pour la sémantique opérationnelle, nous les rappelons et donnons à chaque fois un commentaire quant à leur définition.

X.5.1 Règle de conservation d'une référence libre

$$Ref^n \quad : \quad \frac{}{\rho \vdash id^n \longrightarrow id^n} \quad \star_n \in \mathtt{tl}^n(\rho) \tag{3}$$

RÈG. 2: *Règle de conservation d'une référence libre.*

Il est nécessaire de définir une règle pour une référence libre quand il existe dans l'environnement de la référence une constante $\star$ qui notifie qu'une référence liée apparaît dans une expression englobante. Dans ce cas, le résultat de l'évaluation d'une référence libre est la même référence libre pour ne pas bloquer le processus d'évaluation. Par contre, si aucune constante star n'apparaît, alors on est sûr que toutes les expressions englobantes sont complètes, on ne doit en conséquence fournir aucune règle de réduction pour une référence libre.

X.5.2 Règle de substitution d'un definiens

$$Ref_C \quad : \quad \frac{u = \mathtt{nth}(n+1,\rho)(id) \quad u' = \mathtt{bind}(\mathtt{Dom_E}(\rho), u)}{\rho \vdash id_n \longrightarrow u'} \quad \mathtt{C}(\mathtt{Dom_E}(\rho), u) \tag{4}$$

$$Ref_{\neg C} \quad : \quad \frac{u = \mathtt{nth}(n+1,\rho)(id)}{\rho \vdash id_n \longrightarrow id_n} \quad \neg\, \mathtt{C}(\mathtt{Dom_E}(\rho), u) \tag{5}$$

RÈG. 3: *Règle de substitution d'un definiens.*

La valeur d'une référence liée dans un environnement ρ est définie dans le n^{e} élément de ρ. Nous sommes assurés que la référence liée se trouve effectivement dans ρ grâce aux phases de traduction (cf. section X.2.2) et de liaison (cf. section X.2.1) et par le théorème (X.2) qui nous assurent qu'une référence libre est transformée en une référence liée, seulement si une définition existe. Cette règle est la *règle de substitution* de la sémantique des amalgames. Une fois que la valeur est substituée, il est nécessaire de lier le terme substitué car ce peut être un terme qui comprend une référence libre ayant une définition dans l'environnement où elle est injectée.

La règle (5) définit la valeur d'une référence liée quand la définition de la référence ne peut encore être substituée. Dans ce cas, c'est que le definiens correspondant doit encore être réduit avant d'être substitué sans risque de capture indue d'identificateurs. L'expression reste alors une référence liée.

Les deux règles de substitution des références liées que nous venons de décrire impliquent, pour que la substitution de la référence liée par son definiens puisse avoir lieu, que la définition soit `C()`. Cette contrainte provient du fait qu'un terme ne peut être substitué que si celui-ci a capturé toutes les références qui étaient dans sa portée. Par exemple, si on omet la condition `C()`, alors la réduction de l'exemple suivant :

$$\{a = \{y = x^0\} \mathbin{.} y_0, b = \{x = y\} \mathbin{.} a_1\}$$

conduit, après substitution de la référence liée a_1 par son definiens (convenablement lié) $\{y = x^1\} \mathbin{.} y_0$ à l'expression suivante :

$$\{a = \{y = x^0\} \mathbin{.} x^0, b = \{x = y^0\} \mathbin{.} \{y = x_1\} \mathbin{.} y_0\}$$

Le problème apparaît une fois cette substitution effectuée : y_0 étant liée à la définition $y = x_1$, lors de la prochaine substitution, la référence liée x_1 se liera avec la définition $x = y^0$. Nous arrivons à un cycle où les substitutions ne font que changer x_1 en y_0 et inversement :

$$\begin{array}{ll} \to & \{a = x^0, b = \{x = y^0\} \mathbin{.} \{y = x_1\} \mathbin{.} x_1\} \\ \to & \{a = x^0, b = \{x = y^0\} \mathbin{.} \{y = x_1\} \mathbin{.} y_0\} \\ \to & \{a = x^0, b = \{x = y^0\} \mathbin{.} \{y = x_1\} \mathbin{.} x_1\} \end{array}$$

La définition de la référence liée $\{y = x^0\} \mathbin{.} y_0$ n'aurait pas dû être substituée aussi tôt. En effet, ce n'est que lors de la prochaine réduction que la valeur correcte de a est connue : x^0. L'utilisation du prédicat `C()` nous assure qu'une substitution n'a lieu qu'une fois que le terme ne peut plus être réduit, afin de ne pas donner lieu à des captures non désirées de références.

X.5.3 Règle de réduction d'un système

La règle (6) définit la valeur d'un système. La valeur d'un système s dans un environnement ρ implique la réduction de chaque partie droite des définitions du système. Chaque terme doit être réduit dans un environnement tel que toutes les définitions qui apparaissent en dehors de s soient visibles ainsi que les propres définitions de s. Le lift de l'environnement ρ permet de rendre accessible ces définitions dans s (à l'aide de la fonction `Lift()`, cf. section X.3.1.1).

$$\{\ldots\} \quad : \quad \frac{\rho' = \texttt{Nenv}(\rho, \{\ldots, id = u, \ldots\}) \qquad \overset{\ldots}{\rho' \vdash u \to_n u'}}{\rho \vdash \{\ldots, id = u, \ldots\} \to \{\ldots, id = u', \ldots\}} \quad n \in \{0,1\} \wedge \sum n \geq 1 \tag{6}$$

RÈG. 4: *Règle de réduction d'un système.*

Un formalisme de *macro-expansion* est utilisé pour exprimer que la règle doit s'appliquer s'il est possible qu'au moins une réduction soit applicable sur une définition. En effet, la règle suivante (qui est une modification de la règle correcte où on remplace la relation de réduction « $\to_n$ » par la relation « $\to$ ») :

$$\frac{\rho' = \texttt{Nenv}(\rho, \{\ldots, id = v, \ldots\}) \quad \rho' \vdash v \to v'}{\rho \vdash \{\ldots, id = v, \ldots\} \to \{\ldots, id = v', \ldots\}} \tag{7}$$

nécessite, pour pouvoir s'appliquer, que chacune des prémisses puisse être prouvée, c'est-à-dire donner une preuve qui termine par un axiome. Or, dans la réduction d'un système, comme celui de l'exemple suivant :

$$\{a = c^0, b = a_0\}$$

la première sous-expression, $a = 1$, ne peut pas être réduite car aucune règle n'est définie sur une référence libre, alors que la seconde sous-expression peut (et doit) être réduite. La règle (7) ne permet pas la réduction d'un tel terme. L'utilisation d'une relation de réduction qui réduit un terme en 0 en 1 opération permet de réduire cette expression :

- la première expression est réduite en 0 opération, i.e. elle ne se réduit plus,

– la seconde expression est réduite en 1 opération.

La quantité $\sum n$ s'instancie donc en $0 + 1 = 1$ qui est supérieur ou égal à 1 : la règle peut donc s'appliquer. Une fois cette expression réduite, en un pas, par la règle $\{\ldots\}$, le système a pour valeur :

$$\{a = c^0, b = c^0\}$$

la règle ne peut plus s'appliquer car tous les definiens sont irréductibles. Ce formalisme évite la multiplication des règles et la spécification d'un ordre de réduction : les réductions sont effectuées *en parallèle*, tant que l'expression n'est pas en forme normale.

X.5.4 Règle de réduction d'une concaténation

$$\#_1 \quad : \quad \frac{s = \mathtt{bind}(\mathtt{Dom}_{\mathtt{E}}(\rho), \{\ldots, id = u, \ldots, id' = u', \ldots\})}{\rho \vdash \{\ldots, id = u, \ldots\} \# \{\ldots, id' = u', \ldots\} \to s} \quad \begin{cases} id \neq id' \\ \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), \{\ldots, id = u, \ldots\}) \\ \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), \{\ldots, id' = u', \ldots\}) \end{cases} \tag{8}$$

RÈG. 5: *Règle de réduction d'une concaténation.*

La règle (8) définit la valeur de la concaténation de deux systèmes. La réduction de deux systèmes ne peut avoir lieu que quand ces deux systèmes sont en forme normale. La valeur de la réduction de deux systèmes est l'ensemble des définitions de chaque système. On supposera que deux identificateurs de même nom n'apparaissent pas dans deux systèmes qui vont être concaténés ; en cas de collisions d'identificateurs, il serait nécessaire de numéroter les différentes occurrences d'un même identificateur et d'établir une règle de précédence pour la liaison. Une fois les définitions réunies, il est nécessaire de lier le système résultant pour permettre aux éventuelles références libres de chaque système qui auraient une définition dans l'autre système de se lier effectivement (cf. section VIII.6.3.4).

La concaténation de deux systèmes ne peut avoir lieu que quand les deux termes sont en forme normale. La raison de cette stratégie est de ne combiner que des termes réduits, comme dans l'expression $(1 + 2) + 3$ où il faut d'abord faire le calcul de $1 + 2$ avant de pouvoir ajouter 3.

On peut se poser la question de savoir si une stratégie d'évaluation qui combine les systèmes, mène au même résultat, même quand ceux-ci ne sont pas en forme normale. Voici un exemple qui montre que ce n'est pas le cas ; cependant cet exemple est un exemple « étendu » car on permet plusieurs occurrences du même identificateur dans la définition d'un système. Considérons le terme :

$$A \quad \equiv \quad \{x = y^0, p = \{a = x_1, y = 1\} \# \{y = 2\}\}$$

Avec la stratégie d'évaluation standard, nous obtenons les réductions :

$$\begin{array}{lll} A & \to & \{x = y^0, p = \{a = y_0, y = 1\} \# \{y = 2\}\} \\ & \to & \{x = y^0, p = \{a = 1, y = 1, y = 2\}\} \end{array}$$

alors qu'avec la stratégie alternative, on obtient :

$$A \quad \to \quad \{x = y^0, p = \{a = y_?, y = 1, y = 2\}\}$$

Le terme $y_?$ est une référence ambiguë : quelle définition de y faut-il choisir ? Cette ambiguïté n'apparaît pas avec la stratégie de réduction contrainte. Voilà pourquoi nous demandons à ce que les systèmes d'une concaténation soient en forme normale avant d'êtres réduits.

X.5.5 Règle de réduction de deux expressions concaténées

La règle (9) définit la valeur de la concaténation de deux termes qui ne sont pas tous les deux en forme normale. Si les deux opérandes sont des systèmes, alors la règle s'applique jusqu'à ce que les termes aient

$$\#_2 \quad : \quad \frac{\rho \vdash u \rightarrow_n u' \quad \rho \vdash v \rightarrow_m v'}{\rho \vdash u \# v \rightarrow u' \# v'} \quad n, m \in \{0,1\} \wedge n + m \geq 1 \tag{9}$$

RÈG. 6: *Règle de réduction de deux expressions concaténées.*

atteint tous deux une forme normale, afin que la règle précédemment décrite puisse s'appliquer et contracter les deux systèmes en un système unique. Nous réutilisons ici le système de macro-expansion déjà utilisé pour la spécification de la règle de réduction du système.

X.5.6 Règle de réduction d'une sélection

$$\centerdot_{S_1} \quad : \quad \frac{}{\rho \vdash \{\ldots, id = u, \ldots\} \centerdot v \rightarrow v} \quad \mathtt{C}(\mathtt{Dom_E}(\mathtt{Nenv}(\rho, \{\ldots, id = u, \ldots\})), v) \tag{10}$$

RÈG. 7: *Règle de réduction d'une sélection.*

La règle (10) définit la valeur d'une opération de sélection quand l'opérande droit est en forme normale et que l'opérande gauche est un système. Si le terme de droite est en forme normale, alors c'est que plus aucune réduction ne peut avoir lieu. Dans ce cas, la valeur d'une opération de sélection est, par définition, égale au terme droit.

On remarquera que si la contrainte de la forme normale était levée sur le terme de droite, alors l'expression suivante :

$$\{a = b_0 + c_0, b = 1, c = 2\} \centerdot \{b = 10, c = 20, k = a_1\}$$

pourrait être réduite en l'expression :

$$\{b = 10, c = 20, k = \phi\}$$

avec $\phi \in \{a_1, b_1 + c_1, 3\}$ suivant l'état de réduction du terme gauche avant l'évaluation de la sélection. Ces termes ne sont pas corrects, la valeur désirée étant bien sûr :

$$\{b = 10, c = 20, k = 3\}$$

Si on lève la contrainte que le terme gauche d'une sélection soit un système, alors l'expression suivante :

$$\{a = \{b = 1\}, c = a_0 \centerdot b^0\}$$

peut se réduire en :

$$\{a = \{b = 1\}, c = b^0\}$$

car l'expression $c = a_0 \centerdot b^0$ peut être réduite sans attendre la substitution de la référence liée avec son definiens. La solution attendue, dans ce cas, est bien sûr l'expression :

$$\{a = \{b = 1\}, c = 1\}$$

où le definiens de la référence liée est susbtitué puis l'opérande droit de la sélection lié à la définition $b = 1$.

X.5.7 Règle de réduction de la sélection d'un système et d'une expression

La règle (11) définit la valeur d'une opération de sélection quand l'opérande gauche est un système et que le terme de droite n'est pas en forme normale. Dans ce cas, les deux opérandes sont évalués : le système u dans ρ si c'est possible et l'expression v dans ρ augmenté des définitions de u. La valeur de $s \centerdot e$ est alors la sélection des évaluations de u et de v.

$$\bullet_{S_2} \quad : \quad \frac{\rho \vdash u \to_n u' \quad \mathtt{Nenv}(\rho, u') \vdash v \to v'}{\rho \vdash u \;\mathbf{.}\; v \to u' \;\mathbf{.}\; v'} \quad n \in \{0,1\} \land u \in \textsc{Sys} \tag{11}$$

RÈG. 8: *Règle de réduction de la sélection d'un système et d'une expression.*

X.5.8 Règle de réduction d'une expression qui devient un système, et d'une expression

$$\bullet_{\neg S_1} \quad : \quad \frac{\rho \vdash u \to u' \quad v' = \mathtt{bind}(\mathtt{Dom_E}(\mathtt{Nenv}(\rho, u')), v)}{\rho \vdash u \;\mathbf{.}\; v \to u' \;\mathbf{.}\; v'} \quad u \notin \textsc{Sys} \land u' \in \textsc{Sys} \tag{12}$$

RÈG. 9: *Règle de réduction d'une expression qui devient un système et d'une expression.*

La règle (12) définit la valeur d'une opération de sélection quand l'opérande gauche n'est pas un système, mais qui devient un système après une opération de réduction. Dans ce cas, l'opérande droit est lié avec les définitions du système nouvellement défini. La sélection étant maintenant de la forme $s \;\mathbf{.}\; e$ avec $s \in \textsc{Sys}$, seules les règles $\bullet_{S_1}$ ou $\bullet_{S_2}$ s'appliqueront sur cette expression.

X.5.9 Règle de réduction de la sélection de deux expressions quelconques

$$\bullet_{\neg S_2} \quad : \quad \frac{\rho \vdash u \to_m u' \quad \mathtt{Nenv}(\rho, u') \vdash v \to_n v'}{\rho \vdash u \;\mathbf{.}\; v \to u' \;\mathbf{.}\; v'} \quad \begin{cases} u \notin \textsc{Sys} \land u' \notin \textsc{Sys} \\ n, m \in \{0,1\} \land n + m \geq 1 \end{cases} \tag{13}$$

RÈG. 10: *Règle de réduction de la sélection de deux expressions quelconques.*

Enfin, la règle (13) définit la valeur d'une opération de sélection quand l'opérande gauche n'est pas un système et ne le deviendra pas après un pas de réduction. Dans ce cas, les deux opérandes sont évalués, si c'est possible, l'opérande gauche dans l'environnement ρ, et l'opérande droit dans l'environnement ρ convenablement lifté pour tenir compte de la future transformation de l'opérande gauche en un système.

X.6 Preuves du chapitre

Afin d'alléger les écritures, nous convenons que :

DÉFINITION X.20 $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma,$

$$\begin{array}{l}
\mathtt{C}(\rho, e) = \mathtt{C}(\mathtt{Dom_E}(\rho), e) \\
\mathtt{correct}(\rho, e) = \mathtt{correct}(\mathtt{Dom_E}(\rho), e) \\
\mathtt{correct_B}(\rho, e) = \mathtt{correct_B}(\mathtt{Dom_E}(\rho), e) \\
\mathtt{correct_F}(\rho, e) = \mathtt{correct_F}(\mathtt{Dom_E}(\rho), e)
\end{array}$$

DÉFINITION X.21 $(+)$ $\forall x, y \in \alpha, \forall l, l' \in \mathtt{List}(\alpha),$

$$\begin{array}{lcl}
x + y & = & < x, y > \\
x + l & = & x :: l
\end{array}$$

$$\begin{array}{lcl} l + x & = & l\,@ < x > \\ l + l' & = & l\,@\,l' \end{array}$$

Ces définitions n'introduisent aucune ambiguïté vu les domaines des arguments des prédicats et opérateurs. Remarquons que + est associatif, nous omettrons donc les parenthèses.

Rappelons que si $\{id = u, \dots\}$ abrège le terme $\{id = u, id' = u', id'' = u''\}$ alors $\{id, \dots\}$ abrège l'ensemble $\{id, id', id''\}$ et $\bigwedge f(l, u)$, où f est un prédicat à deux arguments l et u, représente $f(l, u) \wedge f(l, u') \wedge f(l, u'')$.

La notation $(s \in S)$ dénote le prédicat prenant la valeur vraie si s est un élément de l'ensemble S et faux sinon. Ce prédicat est aussi utilisé pour les listes : $(s \in l)$ est vrai si s est un élément de la liste l. L'égalité dans Σ se note avec $\equiv$; le prédicat $==$ est le prédicat d'égalité sur SCOPE.

Beaucoup de preuves de ce chapitre se font par induction sur la structure d'un terme de Σ. Afin de simplifier la lecture, nous détaillerons les cas toujours de la même façon : les références libres (cas 1), puis les références liées (cas 2), les systèmes (cas 3), la concaténation (cas 4) et enfin la sélection (cas 5). Les sous-cas éventuels sont distingués par une lettre (a, b ...) et ensuite par un chiffre romain (i, ii ...).

X.6.1 Correction de la traduction

LEMME X.1 **(La traduction est conservatrice)**
La propriété suivante est vraie :

$$\forall e \in \Sigma_{\mathcal{C}},\ \mathtt{trad}(e) \quad \equiv \quad e$$

Preuve.
Nous allons montrer le résultat suivant :

$$\forall e \in \Sigma_{\mathcal{C}}, \quad \mathtt{bind}(\mathtt{IdToRef}(e)) \quad \equiv \quad e$$

et par suite, $\mathtt{trad}(e)$ est définit car on a bien $\mathtt{correct}(e)$ par hypothèse, et cela montrera que $\mathtt{trad}(e) \equiv e$. Pour montrer ce résultat, on va d'abord montrer que :

$$\forall e \in \Sigma_{\mathcal{C}}, \quad \mathtt{IdToRef}(e) \quad \equiv \quad e$$

Ce résultat est trivial au vu de la définition de $\mathtt{IdToRef}()$. Il suffit donc de montrer que :

$$\forall e \in \Sigma_{\mathcal{C}}, \quad \mathtt{bind}(e) \quad \equiv \quad e \tag{14}$$

Pour montrer ce résultat, on va montrer plus généralement que :

$$\forall e \in \Sigma, \quad \mathtt{correct_F}(l, e) \Rightarrow \mathtt{bind}(l, e) \equiv e \tag{15}$$

Le résultat (14) se retrouve en prenant $l =<>$ et en remarquant que si $e \in \Sigma_{\mathcal{C}}$, alors on a $\mathtt{correct}(e)$ et par conséquence $\mathtt{correct_F}(<>, e)$.

La preuve de (15) se fait par induction sur la structure de e en supposant $\mathtt{correct_F}(l, e)$ vrai.

1. si $e \equiv id^n$, alors on distingue deux cas :
   (a) si $n > \mathtt{lg}(l)$, alors $\mathtt{bind}(l, id^n) \equiv id^n$.
   (b) si $n \leq \mathtt{lg}(l)$, alors $\mathtt{bind}(l, id^n) \equiv \mathtt{bind_F}(\mathtt{tl}^n(l), id, n)$. Par hypothèse, on a $\mathtt{correct_F}(l, id^n)$ et donc $\mathtt{okRefLibre}(\mathtt{tl}^n(l), id)$ est vrai, c'est-à-dire $\star == \mathtt{index}(\mathtt{tl}^n(l), id, 0)$. Or $(\star == \mathtt{index}(l', id, p)) \Leftrightarrow (\star == \mathtt{index}(l', id, q))$ pour tout couple (p, q). Donc $\mathtt{bind_F}(\mathtt{tl}^n(l), id, n) \equiv id^n$.
2. si $e \equiv id_n$, alors $\mathtt{bind}(l, id_n) \equiv id_n$ par définition.
3. si $e \equiv \{id = u, \dots\}$, alors on a $\mathtt{correct_F}(l, \{id = u, \dots\}) = \bigwedge \mathtt{correct_F}(\{id, \dots\} + l, u)$ et par suite on peut appliquer l'hypothèse de récurrence à chaque u partie droite d'une définition du système ; il vient : $\mathtt{bind}(\{id, \dots\} + l, u) \equiv u$ et donc : $\mathtt{bind}(l, \{id = u, \dots\}) \equiv \{id = \mathtt{bind}(\{id, \dots\} + l, u), \dots\} \equiv \{id = u, \dots\}$.

4. si $e \equiv u \# v$, alors $\texttt{correct}_{\texttt{F}}(l, u \# v) = \texttt{correct}_{\texttt{F}}(l, u) \wedge \texttt{correct}_{\texttt{F}}(l, v)$. On peut donc appliquer l'hypothèse de récurrence à u et v et on en déduit que $\texttt{bind}(l, u) \equiv u$ et $\texttt{bind}(l, v) \equiv v$. En conséquence, on a : $\texttt{bind}(l, u \# v) \equiv \texttt{bind}(l, u) \# \texttt{bind}(l, v) \equiv u \# v$.

5. si $e \equiv u \,\centerdot\, v$, alors de l'hypothèse $\texttt{correct}_{\texttt{F}}(l, u \,\centerdot\, v)$, on peut déduire $\texttt{correct}_{\texttt{F}}(l, u)$ et $\texttt{correct}_{\texttt{F}}(\texttt{Dom}(u) + l, v)$. On peut donc appliquer l'hypothèse de récurrence à u avec l'environnement de liaison l et à v avec l'environnement de liaison $\texttt{Dom}(u) + l$. Il vient : $\texttt{bind}(l, u) \equiv u$ et $\texttt{bind}(\texttt{Dom}(u) + l, v) \equiv v$ et donc : $\texttt{bind}(l, u \,\centerdot\, v) \equiv \texttt{bind}(l, u) \,\centerdot\, \texttt{bind}(\texttt{Dom}(u) + l, v) \equiv u \,\centerdot\, v$.

□

X.6.2 Déterminisme de l'évaluation

Pour montrer le déterminisme de l'évaluation, nous avons besoin du résultat suivant :

LEMME X.3 $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \texttt{C}(\rho, e) \Rightarrow e \downarrow \rho$

Preuve.
La preuve du lemme se fait par induction sur la structure de e en supposant $\texttt{C}(\rho, e)$ et en montrant $e \downarrow \rho$.

1. Si $e \equiv id^n$, alors seule la règle Ref^n est susceptible de s'appliquer. Mais on a $\texttt{C}(\rho, e)$ et donc on a $\star_n \notin \texttt{tl}^n(\rho)$. Par suite, cette règle ne peut s'appliquer et on a bien $e \downarrow \rho$.

2. Si $e \equiv id_n$, alors $\texttt{C}(\rho, e)$ est faux par définition et donc la conjonction est forcément vraie.

3. Si $e \equiv \{id = u, \dots\}$, alors seule la règle $\{\dots\}$ est susceptible de s'appliquer. Par hypothèse on a $\texttt{C}(\rho, e)$ et donc $\texttt{C}(\{id, \dots\} + \texttt{Dom}_{\texttt{E}}(\rho), u)$ pour tout u partie droite d'une définition de e. On peut donc appliquer l'hypothèse de récurrence aux u et il vient $u \downarrow \texttt{Nenv}(\rho, e)$. La règle $\{\dots\}$ ne peut donc s'appliquer puisque $\sum n = 0$ et donc $e \downarrow \rho$.

4. Si $e \equiv u \# v$, seules les règles $\#_1$ et $\#_2$ sont susceptibles de s'appliquer. De l'hypothèse $\texttt{C}(\rho, e)$ on tire $\texttt{C}(\rho, u)$ et $\texttt{C}(\rho, v)$. On peut donc appliquer l'hypothèse de récurrence à u et v et il vient $u \downarrow \rho$ et $v \downarrow \rho$. Par suite la règle $\#_2$ ne peut pas s'appliquer car $n + m = 0$. Puisqu'on a $\texttt{C}(\rho, e)$, alors on a aussi $(u \notin \textsc{Sys}) \vee (v \notin \textsc{Sys})$ et donc on ne peut pas appliquer $\#_1$ qui requiert $(u \in \textsc{Sys}) \wedge (v \in \textsc{Sys})$. Par suite, aucune règle ne peut s'appliquer et donc $e \downarrow \rho$.

5. Si $e \equiv u \,\centerdot\, v$, seules les quatre règles $\centerdot_{S_1}, \centerdot_{S_2}, \centerdot_{\neg S_1}$ et $\centerdot_{\neg S_2}$ sont susceptibles de s'appliquer. Par définition, $\texttt{C}(\rho, e) = (u \notin \textsc{Sys}) \wedge \texttt{C}(\rho, u) \wedge \texttt{C}(\texttt{Dom}(u) + \texttt{Dom}_{\texttt{E}}(\rho), v)$ et donc seules $\centerdot_{\neg S_1}$ et $\centerdot_{\neg S_2}$ sont susceptible, de s'appliquer, les autres règles demandant $(u \in \textsc{Sys})$ pour pouvoir s'appliquer. Mais comme on a $\texttt{C}(\rho, u)$, on peut appliquer l'hypothèse de récurrence et donc $u \downarrow \rho$ et donc la règle $\centerdot_{\neg S_1}$ ne peut s'appliquer. Il en va de même pour $\centerdot_{\neg S_2}$. En effet, $\texttt{C}(\texttt{Dom}(u) + \texttt{Dom}_{\texttt{E}}(\rho), v) = \texttt{C}(\texttt{Nenv}(\rho, u), v)$. En appliquant l'hypothèse de récurrence, on a $v \downarrow \texttt{Nenv}(\rho, u)$ et donc la quantité $n + m$ dans cette règle s'instancie en 0. Donc aucune règle ne s'applique et on a bien $e \downarrow \rho$.

□

Nous pouvons à présent passer à la démonstration du théorème :

THÉORÈME X.1 **(Déterminisme de l'évaluation)**
La réduction d'un terme de Σ est déterministe :

$$\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \rho \vdash e \twoheadrightarrow e' \wedge \rho \vdash e \twoheadrightarrow e'' \Rightarrow e' \equiv e''$$

Preuve.
Nous montrerons ce résultat en montrant qu'il est vrai avec un pas de réduction.

$$\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \rho \vdash e \rightarrow e' \wedge \rho \vdash e \rightarrow e'' \Rightarrow e \equiv e''$$

et pour montrer cela, nous montrons qu'au plus une seule règle s'applique à un terme donné. La preuve est faite par étude des termes.

1. Si $e \equiv id^n$, alors seule la règle Ref^n est susceptible de s'appliquer.
2. Si $e \equiv id_n$, alors seules les deux règles Ref_C et $Ref_{\neg C}$ sont susceptibles de s'appliquer et elles sont mutuellement exclusives car gardées respectivement par les prédicats $\mathtt{C}()$ et $\neg\,\mathtt{C}()$.
3. Pour $e \in \textsc{Sys}$: alors seule la règle $\{\dots\}$ peut s'appliquer.
4. Pour $e \equiv u \,\#\, v$, alors seules les règles $\#_1$ et $\#_2$ peuvent s'appliquer et elles sont mutuellement exclusives :
   (a) Si la règle $\#_1$ s'applique, alors on a $\mathtt{C}(\rho, u)$ et $\mathtt{C}(\rho, v)$ et par le lemme (X.3) on a $u \downarrow \rho$ et $v \downarrow \rho$. Donc la règle $\#_2$ ne peut s'appliquer car la quantité $n + m$ s'instancie en 0.
   (b) Inversement, si la règle $\#_2$ s'applique, alors on on a soit $\rho \vdash u \to u'$ soit $\rho \vdash v \to v'$ (et éventuellement les deux). Supposons qu'on ait $\rho \vdash u \to u'$, alors on peut appliquer la contraposée du lemme (X.3) et on en déduit que $\neg\,\mathtt{C}(\rho, u)$ et donc la règle $\#_1$ ne peut s'appliquer.
5. Pour $e \equiv u \,\centerdot\, v$, on distingue deux cas mutuellement exclusifs :
   (a) Si $u \notin \textsc{Sys}$, alors seules les règles $\centerdot_{\neg S_1}$ et $\centerdot_{\neg S_2}$ peuvent s'appliquer et elles sont mutuellement exclusives par le fait que u' soit ou non un élément de $\textsc{Sys}$.
   (b) Si $u \in \textsc{Sys}$, alors seules les règles $\centerdot_1$ et $\centerdot_2$ peuvent s'appliquer mais elles sont mutuellement exclusives :
      i. Si la règle $\centerdot_{S_1}$ s'applique, alors on a $\mathtt{C}(\mathtt{Nenv}(\rho, u), v)$ et donc avec le lemme (X.3) $v \downarrow \mathtt{Nenv}(\rho, u)$. Par suite, $\forall v' \in \Sigma, \mathtt{Nenv}(\rho, u) \vdash v \not\to v'$ et donc la règle $\centerdot_{S_2}$ ne peut s'appliquer.
      ii. Inversement, si la règle $\centerdot_{S_2}$ s'applique, alors on a $\mathtt{Nenv}(\rho, u) \vdash v \to v'$ pour un certain v' et donc on n'a pas $v \downarrow \mathtt{Nenv}(\rho, u)$. En appliquant la contraposée du lemme (X.3), on en déduit $\neg\,\mathtt{C}(\mathtt{Nenv}(\rho, u), v)$. La règle $\centerdot_{S_1}$ ne peut donc s'appliquer.

□

X.6.3 Le prédicat $\mathtt{C}()$ et la notion de forme normale

Pour montrer qu'un terme de $\Sigma_{\mathcal{C}}$ est en forme normale quand il est $\mathtt{C}()$ (lemme (X.2)) nous avons besoin des trois résultats suivants.

LEMME X.4 $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall id \in \textsc{Id}, \forall n \in \mathbb{N}, \quad \mathtt{correct}(\rho, id_n) \Rightarrow \mathtt{nth}(n+1, \rho)(id)$ est défini.

Preuve.
Supposons que $\mathtt{correct}(\rho, id_n)$ est vrai. Alors, $\mathtt{correct}(\rho, id_n) = \mathtt{okRefLiée}(\mathtt{tl}^n(\mathtt{Dom_E}(\rho), id))$ est vrai. Par suite $\mathtt{tl}^n(\mathtt{Dom_E}(\rho)) \neq <>$: supposons $\mathtt{tl}^n(\mathtt{Dom_E}(\rho)) = s + l$ avec l un certain environnement de liaison. Alors : $\mathtt{okRefLiée}(\mathtt{tl}^n(\mathtt{Dom_E}(l)), id) = (id \in s)$ doit être vrai (donc $s \neq \star$). En conséquence, $(id \in \mathtt{nth}(n+1, \mathtt{Dom_E}(\rho)))$ est un prédicat bien défini. Or $\mathtt{nth}(n+1, \mathtt{Dom_E}(\rho)) = \mathtt{Def}(\mathtt{nth}(n+1, \rho))$, donc $id \in \mathtt{Def}(\mathtt{nth}(n+1, \rho))$ ce que l'on voulait démontrer. □

LEMME X.5 $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \quad \mathtt{Dom_E}(\rho) = \mathtt{Dom_E}(\mathtt{Lift_E}(\rho))$

Preuve.
On prouve le précédent lemme par récurrence sur la longueur de ρ.

1. Si $\rho = <>$, alors $\mathtt{Dom_E}(\rho) = <>$ et $\mathtt{Dom_E}(\mathtt{Lift_E}(<>)) = \mathtt{Dom_E}(<>) = <>$.
2. Sinon, supposons le lemme vrai pour les listes de longueur inférieure ou égale à n et considérons la liste $s + \rho'$ de longueur $n + 1$.
   (a) Si $s = \star$,
   alors $\mathtt{Dom_E}(\rho) = \mathtt{Dom_E}(\star + \rho') = \star + \mathtt{Dom_E}(\rho')$ et $\mathtt{Dom_E}(\mathtt{Lift_E}(\star + \rho')) = \star + \mathtt{Dom_E}(\mathtt{Lift_E}(\rho'))$. Il suffit donc d'appliquer l'hypothèse de récurrence à ρ' pour égaler les deux termes.

(b) Si $s = \sigma \in \textsc{Id} \to \Sigma$
alors $\mathtt{Dom}_\mathtt{E}(\sigma + \rho') = \mathtt{Def}(\sigma) + \mathtt{Dom}_\mathtt{E}(\rho')$ et $\mathtt{Dom}_\mathtt{E}(\mathtt{Lift}_\mathtt{E}(\sigma + \rho')) = \mathtt{Def}(\mathtt{Lift}(\sigma)) + \mathtt{Dom}_\mathtt{E}(\mathtt{Lift}_\mathtt{E}(\rho'))$
Les queues de listes sont égales par hypothèse de récurrence. Il suffit donc de montrer que : $\mathtt{Def}(\sigma) = \mathtt{Def}(\mathtt{Lift}(\sigma))$ ce qui est le cas puisque $\mathtt{Lift}()$ est une fonction totale.

□

LEMME X.6 $\forall e \in \Sigma, \forall \rho \in \mathcal{E}_\mathrm{E}, \quad \mathtt{Dom}(e) + \mathtt{Dom}_\mathtt{E}(\rho) = \mathtt{Dom}_\mathtt{E}(\mathtt{Nenv}(\rho, e))$

Preuve.
On considère deux cas :

1. si $e \not\equiv \{\dots\}$, alors $\mathtt{Dom}_\mathtt{E}(\mathtt{Nenv}(\rho, e)) = \mathtt{Dom}_\mathtt{E}(\star + \mathtt{Lift}_\mathtt{E}(\rho)) = \star + \mathtt{Dom}_\mathtt{E}(\mathtt{Lift}_\mathtt{E}(\rho))$ et on applique le lemme (X.5) à la queue de la liste.
2. si $e \equiv \{id = u, \dots\}$, alors $\mathtt{Dom}(e) = \{id, \dots\}$ et $\mathtt{Dom}_\mathtt{E}(\mathtt{Nenv}(\rho, e)) = \mathtt{Dom}_\mathtt{E}([id \mapsto u, \dots] + \mathtt{Lift}_\mathtt{E}(\rho))$
$= \mathtt{Def}([id \mapsto u, \dots]) + \mathtt{Dom}_\mathtt{E}(\mathtt{Lift}_\mathtt{E}(\rho))$
$= \mathtt{Dom}(e) + \mathtt{Dom}_\mathtt{E}(\mathtt{Lift}_\mathtt{E}(\rho))$ et on applique le lemme (X.5) à la queue de la liste.

□

À présent, nous pouvons passer à la preuve du lemme (X.2) :

LEMME X.2 **($\mathtt{C}()$ et forme normale)**

$$\forall \rho \in \mathcal{E}_\mathrm{E}, \forall e \in \Sigma, \quad \mathtt{correct}(\mathtt{Dom}_\mathtt{E}(\rho), e) \Rightarrow \mathtt{C}(\mathtt{Dom}_\mathtt{E}(\rho), e) = e \downarrow \rho$$

REMARQUE Ce lemme précise le résultat du lemme (X.3) dans le cas où on a $\mathtt{correct}(\rho, e)$.

REMARQUE Le théorème (X.2) nous montre que si on part d'un terme correct, alors on obtient un terme correct. De plus, on peut s'assurer par inspection des règles de réduction que si on part d'un terme correct, alors les réductions induites ont lieu sur des termes corrects. Par suite, on peut appliquer (X.2) pour savoir si une prémisse $\rho \vdash e \rightarrow_n e'$ dans une règle s'instancie avec $n = 0$ ou non. Ceci est à la base de l'implémentation de la réduction des amalgames sous Mathematica (cf. chapitre XI, page 159).

Preuve.
La preuve se fait par induction sur la structure d'un terme $e \in \Sigma$ en supposant $\mathtt{correct}(\rho, e)$.

1. Si $e \equiv id^n$,
alors seule Ref^n est susceptible de s'appliquer. Elle s'applique si et seulement si $(\star_n \in \mathtt{tl}^n(\rho))$ mais $(\star_n \in \mathtt{tl}^n(\rho)) = (\star_n \in \mathtt{tl}^n(\mathtt{Dom}_\mathtt{E}(\rho))) = \neg\, \mathtt{C}(\rho, id^n)$. Par suite, si la règle s'applique on a $\mathtt{C}(\rho, e)$ et $e \downarrow \rho$ qui sont tous les deux faux (puisqu'on a pu appliquer une règle). Et si elle ne s'applique pas, alors on a $\mathtt{C}(\rho, e)$ vrai ainsi que $e \downarrow \rho$ (puisqu'aucune règle ne s'applique). On a donc bien $\mathtt{C}(\rho, e) = e \downarrow \rho$.
2. Si $e \equiv id_n$,
alors $\exists e' \in \Sigma$ tel que $\rho \vdash id_n \rightarrow e'$. En effet, on a par hypothèse $\mathtt{correct}(\rho, id_n)$ et en appliquant le lemme (X.4), il vient que $\mathtt{nth}(n+1, \rho)(x)$ est bien défini. Part suite, on peut appliquer soit la règle Ref_C soit la règle $Ref_{\neg C}$ suivant la valeur de $\mathtt{C}(e')$ et donc $id_n \downarrow \rho$ est faux, ce qui est aussi le cas pour $\mathtt{C}(id_n)$ par définition.
3. Si $e \equiv \{id = u, \dots\}$,
alors la règle $\{\dots\}$ est la seule à pouvoir s'appliquer et s'applique si une au moins des prémisses s'instancie avec $n = 1$. Elle ne s'applique donc pas si toutes les prémisses s'instancient avec $n = 0$, c'est-à-dire si $\bigwedge u \downarrow \rho'$ avec $\rho' = \mathtt{Nenv}(\rho, \{id = u, \dots\})$. Mais on a par hypothèse $\mathtt{correct}(\rho, e)$ c'est-à-dire $\mathtt{correct}(\rho', u)$ pour tout u en partie droite d'une définition de e. On peut donc appliquer l'hypothèse de récurrence et on en déduit que : $\mathtt{C}(\{id, \dots\} + \mathtt{Dom}_\mathtt{E}(\rho), u) = u \downarrow \{id, \dots\} + \mathtt{Dom}_\mathtt{E}(\rho)$. Or, $\{id, \dots\} + \mathtt{Dom}_\mathtt{E}(\rho) = \mathtt{Dom}_\mathtt{E}(\mathtt{Nenv}(\rho, \{id = u, \dots\}))$ par le lemme (X.6) et donc on a aussi $\mathtt{C}(\rho', u) = u \downarrow \rho'$.
Par suite :

$$\{id = u, \dots\} \downarrow \rho \quad = \quad \bigwedge u \downarrow \mathtt{Nenv}(\rho, \{id = u, \dots\})$$

$$\begin{aligned} &= \bigwedge \mathtt{C}(\mathtt{Nenv}(\rho, \{id = u, \dots\}), u) \\ &= \mathtt{C}(\rho, \{id = u, \dots\}) \end{aligned}$$

ce que l'on voulait montrer.

4. Si $e \equiv u \# v$,
alors les règles $\#_1$ et $\#_2$ sont seules susceptibles de s'appliquer. On distingue deux cas.

   (a) Si $(u \in \text{Sys}) \wedge (v \in \text{Sys}) \wedge \mathtt{C}(\rho, u) \wedge \mathtt{C}(\rho, v)$,
   alors la règle $\#_1$ peut s'appliquer car `bind()` a une valeur définie (c'est une fonction totale). Par suite, $u \# v \downarrow \rho = \mathtt{false}$. Mais $\mathtt{C}(\rho, u \# v) = ((u \notin \text{Sys}) \vee (v \notin \text{Sys})) \wedge \mathtt{C}(\rho, u) \wedge \mathtt{C}(\rho, v) = \mathtt{false}$. On a donc bien $e \downarrow \rho = \mathtt{C}(\rho, e)$.

   (b) Sinon, on est dans le cas où $\neg((u \in \text{Sys}) \wedge (v \in \text{Sys}) \wedge \mathtt{C}(\rho, u) \wedge \mathtt{C}(\rho, v))$
   et la règle $\#_2$ est la seule susceptible de s'appliquer. Par hypothèse, on a $\mathtt{correct}(\rho, u \# v)$ et donc on a aussi $\mathtt{correct}(\rho, u)$ et $\mathtt{correct}(\rho, v)$. On peut donc appliquer l'hypothèse de récurrence et il vient que $u \downarrow \rho = \mathtt{C}(\rho, u)$ et $v \downarrow \rho = \mathtt{C}(\rho, v)$. Donc, $\mathtt{C}(\rho, u \# v) = ((u \notin \text{Sys}) \vee (v \notin \text{Sys})) \wedge u \downarrow \rho \wedge v \downarrow \rho$. On envisage deux cas :

      i. Si on suppose que $(u \downarrow \rho = \mathtt{false}) \vee (v \downarrow \rho = \mathtt{false})$ alors la règle $\#_2$ s'applique et donc $u \# v \downarrow \rho = \mathtt{false}$ mais $\mathtt{C}(\rho, u \# v) = \mathtt{false}$ aussi.

      ii. Sinon, on est dans le cas $(u \downarrow \rho = \mathtt{true}) \wedge (v \downarrow \rho = \mathtt{true})$. La règle $\#_2$ ne peut s'appliquer et donc $u \# v \downarrow \rho = \mathtt{true}$ dans ce cas. Comme nous sommes dans le cas où par hypothèse on a : $\neg((u \in \text{Sys}) \wedge (v \in \text{Sys}) \wedge u \downarrow \rho \wedge v \downarrow \rho)$ et que $\neg(u \downarrow \rho = \mathtt{false}) \vee (v \downarrow \rho = \mathtt{false})$ il vient que l'on a $\neg((u \in \text{Sys}) \wedge (v \in \text{Sys}))$ soit $(u \notin \text{Sys}) \vee (v \notin \text{Sys})$. Par suite, nous avons $\mathtt{C}(\rho, u \# v) = \mathtt{true}$.

5. Si $e \equiv u \,.\, v$,
on considère deux sous-cas suivant l'appartenance ou non de u à Sys.

   (a) Si $u \in \text{Sys}$,
   alors par définition, $\mathtt{C}(\rho, u \,.\, v) = \mathtt{false}$. Seules les règles $._{S_1}$ et $._{S_2}$ sont susceptibles de s'appliquer. Mais une des deux règles doit s'appliquer obligatoirement.

   En effet, de l'hypothèse $\mathtt{correct}(\rho, e)$, on tire $\mathtt{correct}(\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho), v)$. Or $\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho) = \mathtt{Dom_E}(\mathtt{Nenv}(\rho, u))$ par le lemme (X.6). On a donc $\mathtt{correct}(\mathtt{Nenv}(\rho, u), v)$ et on peut appliquer l'hypothèse de récurrence : $\mathtt{C}(\mathtt{Nenv}(\rho, u), v) = v \downarrow \mathtt{Nenv}(\rho, u)$. Si la prémisse de $._{S_2}$ ne s'instancie pas, alors $v \downarrow \mathtt{Nenv}(\rho, u)$ est vraie, mais alors on a $\mathtt{C}(\mathtt{Nenv}(\rho, u), v)$ d'après ce qui précède et donc la règle $._{S_1}$ s'applique. Et donc on a $e \downarrow \rho$ faux puisqu'une règle a pu s'appliquer. Si la prémisse de $._{S_2}$ s'instancie, alors la règle s'applique et on a donc encore $e \downarrow \rho$ à faux.

   (b) si $u \notin \text{Sys}$,
   alors seules les règles $._{\neg S_1}$ et $._{\neg S_2}$ sont susceptibles de s'appliquer. On considère deux cas suivant la valeur de $\mathtt{C}(\rho, u)$:

      i. $\mathtt{C}(\rho, u) = \mathtt{true}$.
      On a par hypothèse $\mathtt{correct}(\rho, u \,.\, v)$ et donc en particulier, on a $\mathtt{correct}(\rho, u)$. On peut donc appliquer l'hypothèse de récurrence et il vient que $u \downarrow \rho$. Par suite, seule la règle $._{\neg S_2}$ est susceptible de s'appliquer avec $u' \equiv u$ (la première prémisse s'instantiant avec $\rho \vdash u \to_0 u$). Par ailleurs, de $\mathtt{correct}(\rho, u \,.\, v)$ on en déduit $\mathtt{correct}(\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho), v)$. Mais, par le lemme (X.6), on a $\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho) = \mathtt{Dom_E}(\mathtt{Nenv}(\rho, u))$. On peut donc appliquer l'hypothèse de récurrence à v avec $\mathtt{Nenv}(\rho, u)$ et on distingue deux cas :

         A. si $\mathtt{C}(\mathtt{Nenv}(\rho, u), v) = \mathtt{false}$,
         alors il existe v' tel que $\mathtt{Nenv}(\rho, u) \vdash v \to v'$ et donc $u \,.\, v \downarrow \rho = \mathtt{false}$ car la règle $._{\neg S_2}$ peut s'appliquer. Mais $\mathtt{C}(\rho, u \,.\, v) = (u \notin \text{Sys}) \wedge \mathtt{C}(\rho, u) \wedge \mathtt{C}(\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho), v) = \mathtt{false}$. On a donc ce qu'il faut.

         B. si $\mathtt{C}(\mathtt{Nenv}(\rho, u), v) = \mathtt{true}$,
         alors $v \downarrow \mathtt{Nenv}(\rho, u) = \mathtt{true}$ et donc la règle $._{\neg S_2}$ ne peut s'appliquer car la condition $m + n \geq 1$ n'est pas satisfaite. Par suite, nous avons bien $u \,.\, v \downarrow \rho = \mathtt{true} = \mathtt{C}(\rho, u \,.\, v)$

ii. $\mathtt{C}(\rho, u) = \mathtt{false}$

Alors on a $\mathtt{C}(\rho, u \mathbin{.} v) = \mathtt{false}$. Par ailleurs, on a par hypothèse $\mathtt{correct}(\rho, u \mathbin{.} v)$ et donc en particulier, on a $\mathtt{correct}(\rho, u)$. On peut donc appliquer l'hypothèse de récurrence et il vient $u \downarrow \rho = \mathtt{false}$ et par suite, il existe un u' tel que $\rho \vdash u \rightarrow u'$. Les deux règles $\centerdot_{\neg S_1}$ et $\centerdot_{\neg S_2}$ sont susceptibles de s'appliquer. Deux cas sont possibles.

A. Si $u' \in$ SYS,
alors seule la règle $\centerdot_{\neg S_1}$ peut s'appliquer. La première prémisse étant satisfaite, il suffit que $\mathtt{bind}(\mathtt{Dom}(u') + \mathtt{Dom}_{\mathtt{E}}(\rho), v)$ soit défini, ce qui est le cas car $\mathtt{bind}()$ est une fonction totale. Donc la règle $\centerdot_{\neg S_1}$ s'applique et par suite $u \mathbin{.} v \downarrow \rho = \mathtt{false}$, ce qui était voulu.

B. Si $u' \notin$ SYS,
alors seule la règle $\centerdot_{\neg S_2}$ peut s'appliquer et elle s'applique car la deuxième prémisse est toujours satisfiable (elle est instanciée avec $m = 1$). En conséquence, $u \mathbin{.} v \downarrow \rho = \mathtt{false}$.

□

X.6.4 L'évaluation préserve la correction

X.6.4.1 Plan de la preuve

Pour montrer que la correction se préserve le long de la réduction, il faudra montrer entre autre que quand on substitue une référence liée, l'expression introduite, puis liée est une expression correcte. La liaison ne transformant que des références libres, il doit suffire qu'un terme soit $\mathtt{correct}_{\mathtt{B}}()$ pour que sa liaison soit $\mathtt{correct}()$. Ce résultat est formalisé par le lemme :

$(X.12)$: $\mathtt{correct}_{\mathtt{B}}(l, e) \Rightarrow \mathtt{correct}(l, \mathtt{bind}(l, e))$

Pour montrer ce lemme, il nous faudra un certain nombre de résultats auxiliaires comme :

$(X.8)$: $\mathtt{Dom}(e) =_\star \mathtt{Dom}(\mathtt{bind}(l, e))$
$(X.9)$: $\mathtt{correct}_{\mathtt{F}}(l, \mathtt{bind}(l, e))$
$(X.11)$: $\mathtt{correct}_{\mathtt{B}}(l, e) \Rightarrow \mathtt{correct}_{\mathtt{B}}(l, \mathtt{bind}(l, e))$
...

Le résultat (X.12) étant établi, il nous faudra définir une notion de correction de l'environnement d'évaluation afin de prouver que les expressions qu'on peut en extraire sont $\mathtt{correct}_{\mathtt{B}}()$. C'est le propos de la définition de $\mathtt{correct}_{\mathtt{E}}()$ et du lemme :

$(X.16)$: $\mathtt{correct}_{\mathtt{B}}(\rho, e) \wedge \mathtt{correct}_{\mathtt{E}}(\rho) \Rightarrow \mathtt{correct}_{\mathtt{E}}(\mathtt{Nenv}(\rho, e))$

Là aussi, il nous faudra un certain nombre de résultats intermédiaires :

$(X.14)$: $\mathtt{correct}_{\mathtt{B}}(l + l', e) \Rightarrow \mathtt{correct}_{\mathtt{B}}(l + s + l', \mathtt{Lift}(\mathtt{lg}(l), e))$
$(X.15)$: $\mathtt{correct}_{\mathtt{E}}(l, \rho) \Rightarrow \mathtt{correct}_{\mathtt{E}}(s + l, \mathtt{Lift}_{\mathtt{E}}(\rho))$
$(X.13)$: $\mathtt{Dom}_{\mathtt{E}}(\mathtt{Lift}_{\mathtt{E}}(\rho)) = \mathtt{Dom}_{\mathtt{E}}(\rho)$
...

Ces résultats étant acquis (le schéma des dépendances entre les lemmes est donné figure 3), on pourra enfin passer à la démonstration du théorème : $\forall e \in \Sigma_{\mathcal{C}}, \quad e \twoheadrightarrow e' \Rightarrow e' \in \Sigma_{\mathcal{C}}$.

X.6.4.2 Résultats auxilliaires

DÉFINITION X.22 ($=_\star$)
On définit $=_\star$ relation d'équivalence sur SCOPE par

$\star^n =_\star \star_n$
$s = s' \Rightarrow s =_\star s'$

Rappelons que $\star$ désigne indifférement $\star^n$ ou $\star_n$. La notation $s = \star$ se comprend alors comme $s =_\star \star^n$ (ou bien $s =_\star \star_n$).

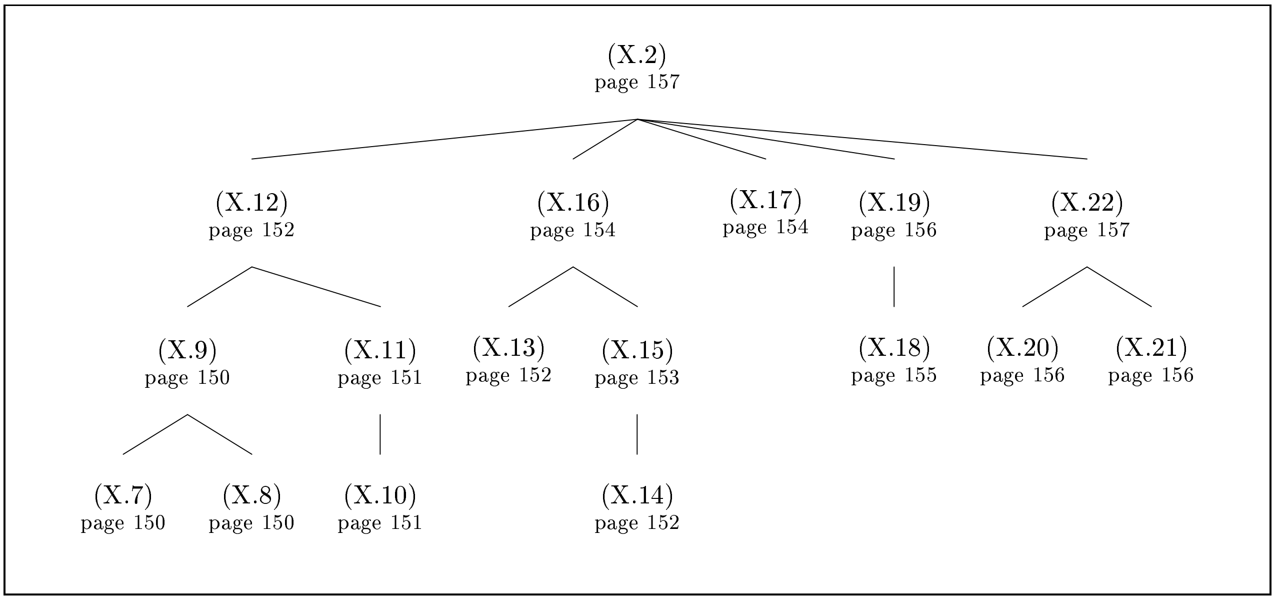


FIG. 3 –: *Le schéma des dépendances entre les résultats nécessaires à la démonstration du théorème (X.2).*

LEMME X.7 $\forall l \in \mathcal{E}_{\mathrm{L}}, \forall e \in \Sigma, \quad \mathtt{Dom}(e) =_\star \mathtt{Dom}(\mathtt{bind}(l,e))$

Preuve.
On envisage deux cas :

1. Si $e \notin$ SYS, alors $\mathtt{bind}(l,e) \notin$ SYS et donc $\mathtt{Dom}(e) =_\star \star =_\star \mathtt{bind}(l,e)$.
2. Sinon, $e \in$ SYS, disons $e \equiv \{id = u, \ldots\}$. Alors $\mathtt{Dom}(e) = \{id, \ldots\}$ et $\mathtt{Dom}(\mathtt{bind}(l,e)) = \mathtt{Dom}(\{id = \mathtt{bind}(\{id, \ldots\} + l, u), \ldots\}) = \{id, \ldots\}$

□

LEMME X.8 $\forall e \in \Sigma, \forall l \in \mathcal{E}_{\mathrm{L}}, \forall s, s' \in$ SCOPE, $s =_\star s'$, alors :

$$\begin{array}{l}\mathtt{bind}(s+l,e) = \mathtt{bind}(s'+l,e)\\ \mathtt{correct}(s+l,e) = \mathtt{correct}(s'+l,e)\\ \mathtt{correct_B}(s+l,e) = \mathtt{correct_B}(s'+l,e)\\ \mathtt{correct_F}(s+l,e) = \mathtt{correct_F}(s'+l,e)\end{array}$$

Preuve. Evidente à partir des définitions. □

LEMME X.9 $\forall l \in \mathcal{E}_{\mathrm{L}}, \forall e \in \Sigma, \quad \mathtt{correct_F}(l, \mathtt{bind}(l,e))$

Preuve.
La preuve se fait par induction sur la structure de e.

1. $e \equiv id^n$
   On distingue deux cas suivant la valeur de n :
   (a) Si $n > \mathtt{lg}(l)$,
   alors $\mathtt{bind}(l, id^n) \equiv id^n$ et $\mathtt{correct_F}(l, id^n) = \mathtt{okRefLibre}(<>, id)$ qui est vrai par définition.
   (b) Si $n \leq \mathtt{lg}(l)$,
   alors on distingue deux cas suivant la valeur de $\mathtt{bind}(l, id^n)$.
   i. Si $\mathtt{bind}(l, id^n) = id^n$, alors $\mathtt{index}(\mathtt{tl}^n(l), id, n) = \star$ et donc :
   $$\mathtt{correct_F}(l, id^n) = \mathtt{okRefLibre}(\mathtt{tl}^n(l), id) = (\star == \mathtt{index}(\mathtt{tl}^n(l), id, 0))$$
   est vrai.

ii. Sinon, $\texttt{bind}(l, id^n) = id_p$ et $\texttt{correct}_\texttt{F}(l, id_p)$ est vrai par définition.

2. $e \equiv id_n$
On a $\texttt{bind}(l, id_n) \equiv id_n$ par définition et $\texttt{correct}_\texttt{F}(l, id_n)$ qui est vrai par définition.

3. $e \equiv \{id = u, \dots\}$
alors $\texttt{correct}_\texttt{F}(l, \texttt{bind}(l, \{id = u, \dots\})) = \bigwedge \texttt{correct}_\texttt{F}(\{id, \dots\} + l, \texttt{bind}(\{id, \dots\} + l, u))$. On applique l'hypothèse de récurrence à chaque u en partie droite d'une définition de e avec l'environnement de liaison $\{id, \dots\} + l$ ce qui prouve la conjonction.

4. $e \equiv u \# v$
alors $\texttt{correct}_\texttt{F}(l, \texttt{bind}(l, u \# v)) = \texttt{correct}_\texttt{F}(l, \texttt{bind}(l, u)) \wedge \texttt{correct}_\texttt{F}(l, \texttt{bind}(l, v))$. On applique l'hypothèse de récurrence à u et v avec l'environnement de liaison l et on a ce qu'il faut.

5. $e \equiv u \,\texttt{.}\, v$
Posons $N \equiv \texttt{bind}(l, u \,\texttt{.}\, v) \equiv \texttt{bind}(l, u) \,\texttt{.}\, \texttt{bind}(\texttt{Dom}(u) + l, v)$. Alors :

$$\texttt{correct}_\texttt{F}(l, N) = \texttt{correct}_\texttt{F}(l, \texttt{bind}(l, u)) \wedge \texttt{correct}_\texttt{F}(\texttt{Dom}(\texttt{bind}(l, u)) + l, \texttt{bind}(\texttt{Dom}(u) + l, v))$$

Pour le premier terme de la conjonction, on applique l'hypothèse de récurrence à u.

Pour le deuxième terme, on applique les lemmes (X.7) et (X.8) ce qui donne :

$$\texttt{correct}_\texttt{F}(\texttt{Dom}(u) + l, \texttt{bind}(\texttt{Dom}(u) + l, v))$$

On applique alors l'hypothèse de récurrence à v avec l'environnement de liaison $\texttt{Dom}(u) + l$.

□

DÉFINITION X.23 **($l \leq l'$)**
Soit $l, l' \in \mathcal{E}_\mathrm{L}$, le prédicat binaire $l \leq l'$ est défini par induction sur la structure de l et l' :

$$\begin{array}{rcccl} <> & \leq & l & = & \texttt{true} \\ s + l' & \leq & s' + l & = & (s \subseteq s') \wedge (l' \leq l) \\ \star + l' & \leq & s + l & = & (l' \leq l) \\ \star + l' & \leq & \star + l & = & (l' \leq l) \end{array}$$

$\leq$ est une relation d'ordre partielle et $l \leq l$.

LEMME X.10 $\forall q \in \mathbb{N}, \forall l, l' \in \mathcal{E}_\mathrm{L}, l \leq l' \quad (x \in \texttt{nth}(q, l)) \Rightarrow (x \in \texttt{nth}(q, l'))$

Preuve.
Si $(x \in \texttt{nth}(q, l))$ est vraie, alors la liste l a un q-ème élément et cet élément est différent de $\star$. Mais alors c'est aussi le cas pour l' car $l \leq l'$. □

LEMME X.11 $\forall l, l' \in \mathcal{E}_\mathrm{L}, \forall e \in \Sigma, \quad \texttt{correct}_\texttt{B}(l, e) \wedge (l' \leq l) \Rightarrow \texttt{correct}_\texttt{B}(l, \texttt{bind}(l', e))$

Corollaire : $\texttt{correct}_\texttt{B}(l, e) \Rightarrow \texttt{correct}_\texttt{B}(l, \texttt{bind}(l, e))$
(il suffit de prendre le lemme avec $l' = l$).

Preuve.
La preuve se fait par induction sur la structure de e en supposant $l' \leq l$ et $\texttt{correct}_\texttt{B}(l, e)$.

1. $e \equiv id^n$
On distingue deux cas suivant la valeur de $\texttt{bind}(l', id^n)$.
   (a) Si $\texttt{bind}(l', id^n) \equiv id^n$, alors $\texttt{correct}_\texttt{B}(l, id^n)$ est vrai par définition.
   (b) Sinon, $\texttt{bind}(l', id^n) \equiv id_p$ et $id \in \texttt{nth}(p+1, l')$. En appliquant le lemme (X.10), $id \in \texttt{nth}(p+1, l)$ et donc $\texttt{correct}_\texttt{B}(l, id_p) = \texttt{okRefLiée}(\texttt{tl}^p(l), id) = (id \in \texttt{nth}(p+1, l))$ est vrai.

2. $e \equiv id_n$
alors $\texttt{correct}_\texttt{B}(l, \texttt{bind}(l', id_n)) = \texttt{correct}_\texttt{B}(l, id_n)$ qui est vrai par hypothèse.

3. $e \equiv \{id = u, \dots\}$
Soit $P \equiv \mathtt{bind}(l', \{id = u, \dots\})$. Alors, par définition, $P \equiv \{id = \mathtt{bind}(\{id, \dots\} + l', u), \dots\}$ et $\mathtt{correct_B}(l, P)$ est égal à $\bigwedge \mathtt{correct_B}(\{id, \dots\} + l, \mathtt{bind}(\{id, \dots\} + l', u))$. De l'hypothèse on tire : $\mathtt{correct_B}(\{id, \dots\} + l, u)$ et puisque $\{id, \dots\} + l' \leq \{id, \dots\} + l$, on peut appliquer l'hypothèse de récurrence à chaque u en partie droite d'une définition de e avec les environnements de liaison $\{id, \dots\} + l'$ et $setid + l$, ce qui permet de conclure à la vérité de la conjonction.

4. $e \equiv u \# v$
$\mathtt{correct_B}(l, \mathtt{bind}(l', u \# v)) = \mathtt{correct_B}(l, \mathtt{bind}(l', u)) \wedge \mathtt{correct_B}(l, \mathtt{bind}(l', v))$. De l'hypothèse on tire $\mathtt{correct_B}(l, u)$ et idem pour v, ce qui permet d'appliquer l'hypothèse de récurrence et de conclure.

5. $e \equiv u \;\mathtt{.}\; v$
Soit $P \equiv \mathtt{bind}(l', u \;\mathtt{.}\; v)$, alors par définition, $P \equiv \mathtt{bind}(l', u) \;\mathtt{.}\; \mathtt{bind}(\mathtt{Dom}(u) + l', v)$ et donc $C = \mathtt{correct_B}(l, P) = \mathtt{correct_B}(l, \mathtt{bind}(l', u)) \wedge \mathtt{correct_B}(\mathtt{Dom}(\mathtt{bind}(l', u)) + l, \mathtt{bind}(\mathtt{Dom}(u) + l', v))$. En appliquant les lemmes (X.7) et (X.8), il vient : $C = \mathtt{correct_B}(l, \mathtt{bind}(l', u)) \wedge \mathtt{correct_B}(\mathtt{Dom}(u) + l, \mathtt{bind}(\mathtt{Dom}(u) + l', v))$. Or, de l'hypothèse $\mathtt{correct_B}(l, e)$ on tire $\mathtt{correct_B}(l, u)$ et $\mathtt{correct_B}(\mathtt{Dom}(u) + l, v)$. On peut donc appliquer l'hypothèse de récurrence à u avec les environnements de liaison l et l' et à v avec les environnements de liaison $\mathtt{Dom}(u) + l$ et $\mathtt{Dom}(u) + l'$ (on vérifie bien que si $l' \leq l$, alors $\mathtt{Dom}(u) + l' \leq \mathtt{Dom}(u) + l$).

□

Remarque On aurait pu prouver directement le corollaire, car on ne se sert jamais de $l' \leq l$ et $l' \neq l$. Cependant, cette preuve est plus générale et la relation d'ordre entre environnements de liaison intervient si on essaie de définir une notion de correction qui s'accommode d'une stratégie de liaison tardive des références libres dans le traitement des expressions impliquant l'opérateur de concaténation.

Lemme X.12 $\forall e \in \Sigma, \forall l \in \mathcal{E}_{\mathrm{L}}, \quad \mathtt{correct_B}(l, e) \Rightarrow \mathtt{correct}(l, \mathtt{bind}(l, e))$

Preuve. Il suffit d'appliquer les lemmes (X.9) et (X.11). □

Lemme X.13 $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \quad \mathtt{Dom_E}(\mathtt{Lift_E}(\rho)) = \mathtt{Dom_E}(\rho)$

Preuve.
La preuve se fait par induction sur la structure de ρ.

1. Si $\rho = <>$,
alors $\mathtt{Dom_E}(\mathtt{Lift_E}(<>)) = \mathtt{Dom_E}(<>) = \mathtt{Dom_E}(\rho)$.

2. Si $\rho = \star + \rho'$
alors $\mathtt{Dom_E}(\mathtt{Lift_E}(\star + \rho')) = \mathtt{Dom_E}(\star + \mathtt{Lift_E}(\rho')) = \star + \mathtt{Dom_E}(\mathtt{Lift_E}(\rho'))$. On applique l'hypothèse de récurrence à la queue de liste et le terme précédent devient $\star + \mathtt{Dom_E}(\rho') = \mathtt{Dom_E}(\rho)$.

3. Si $\rho = \sigma + \rho'$
alors $D = \mathtt{Dom_E}(\mathtt{Lift_E}(\sigma + \rho')) = \mathtt{Def}(\mathtt{Lift}(\sigma)) + \mathtt{Dom_E}(\mathtt{Lift_E}(\rho'))$. Or, $\mathtt{Def}(\mathtt{Lift}(\sigma)) = \mathtt{Def}(\sigma)$ car $\mathtt{Lift}()$ est une fonction totale, et pour la queue de la liste, on peut appliquer l'hypothèse de récurrence, d'où $D = \mathtt{Def}(\sigma) + \mathtt{Dom_E}(\rho') = \mathtt{Dom_E}(\rho)$.

□

Lemme X.14 $\forall e \in \Sigma, \forall l, l' \in \mathcal{E}_{\mathrm{L}}, \forall s \in \textsc{Scope}, \quad \mathtt{correct_B}(l' + l, e) \Rightarrow \mathtt{correct_B}(l' + s + l, \mathtt{Lift}(\mathtt{lg}(l'), e))$

Corollaire : $\forall e \in \Sigma, \forall l \in \mathcal{E}_{\mathrm{L}}, \forall s \in \textsc{Scope}, \quad \mathtt{correct_B}(l, e) \Rightarrow \mathtt{correct_B}(s + l, \mathtt{Lift}(e))$
(il suffit de prendre le lemme avec $l' = <>$).

Preuve.
Posons $p = \mathtt{lg}(l')$ et $l'' = l' + s + l$. La démonstration se fait par induction sur e en supposant $\mathtt{correct_B}(l' + l, e)$.

1. $e \equiv id^n$
alors $\mathtt{Lift}(p, id^n) \equiv id^n$ et $\mathtt{correct_B}(l'', id^n)$ est vrai par définition.

2. $e \equiv id_n$
   On distingue deux cas suivant la valeur de p.
   (a) Si $p \leq n$,
   alors $\texttt{Lift}(p, id_n) \equiv id_{n+1}$ et $\texttt{correct}_{\texttt{B}}(l'', id_{n+1}) = \texttt{okRefLiée}(\texttt{tl}^{n+1}(l''), id) = (id \in \texttt{nth}(n+2, l'')) = (id \in \texttt{nth}(n+2, l'+s+l))$. Or, par hypothèse on a $id \in \texttt{nth}(l'+l, n+1)$ donc on a bien $id \in \texttt{nth}(n+2, l'+s+l)$ puisque $\texttt{lg}(l'+s) = p+1 \leq n+2$.
   (b) Si $p > n$
   alors $\texttt{Lift}(p, id_n) \equiv id_n$ et $\texttt{correct}_{\texttt{B}}(l'', id_n) = (id \in \texttt{nth}(n+1, l'+s+l)) = (id \in \texttt{nth}(n+1, l'+l))$ car $\texttt{lg}(l') = p \geq n+1$. Or le dernier terme est la définition de $\texttt{correct}_{\texttt{B}}(l'+l, id_n)$ qui est vrai par hypothèse.
3. $e \equiv \{id = u, \ldots\}$
   On a $\texttt{correct}_{\texttt{B}}(l'', \texttt{Lift}(n, \{id = u, \ldots\})) = \bigwedge \texttt{correct}_{\texttt{B}}(\{id, \ldots\} + l'', \texttt{Lift}(n+1, u))$. Or de l'hypothèse on tire que pour tout u en partie droite d'une définition de e on a $\texttt{correct}_{\texttt{B}}(\{id, \ldots\} + l' + l, u)$. On peut donc appliquer l'hypothèse de récurrence aux u avec l'environnement de liaison $\{id, \ldots\} + l'$ de longueur $n+1$, ce qui montre la conjonction.
4. $e \equiv u \# v$
   De l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(l'+l, u)$ et on peut donc appliquer l'hypothèse de récurrence. Il en va de même pour v, ce qui montre qu'on a $\texttt{correct}_{\texttt{B}}(l'', \texttt{Lift}(n, u)) \wedge \texttt{correct}_{\texttt{B}}(l'', \texttt{Lift}(n, v))$ or cette dernière expression est la définition de $\texttt{correct}_{\texttt{B}}(l'', \texttt{Lift}(n, u \# v))$.
5. $e \equiv u \,.\, v$
   Par définition, $\texttt{correct}_{\texttt{B}}(l'', \texttt{Lift}(n, u \,.\, v)) = \texttt{correct}_{\texttt{B}}(l'', \texttt{Lift}(n, u)) \wedge \texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l'', \texttt{Lift}(n+1, v))$.

   Pour le premier terme, on peut appliquer l'hypothèse de récurrence, puisqu'à partir de l'hypothèse $\texttt{correct}_{\texttt{B}}(l'+l, u \,.\, v)$ on tire $\texttt{correct}_{\texttt{B}}(l'+l, u)$.

   On en tire aussi $\texttt{correct}(\texttt{Dom}(u) + l' + l, v)$ ce qui permet d'appliquer l'hypothèse de récurrence à v avec les environnements de liaison $\texttt{Dom}(u) + l'$ de longueur $p+1$ et l, ce qui montre le deuxième terme de la conjonction.

□

DÉFINITION X.24 **(correction d'un environnement d'évaluation)**
Le prédicat $\texttt{correct}_{\texttt{E}}() : \mathcal{E}_{\text{E}} \rightarrow \text{BOOL}$ est défini par :

$$\texttt{correct}_{\texttt{E}}(\rho) = \texttt{correct}_{\texttt{E}}(\texttt{Dom}_{\texttt{E}}(\rho), \rho)$$

et $\texttt{correct}_{\texttt{E}}() : \mathcal{E}_{\text{L}} \times \mathcal{E}_{\text{E}} \rightarrow \text{BOOL}$ est défini par :

$$\begin{array}{rcl} \texttt{correct}_{\texttt{E}}(l, <>) & = & \texttt{true} \\ \texttt{correct}_{\texttt{E}}(l, \star + \rho') & = & \texttt{correct}_{\texttt{E}}(l, \rho') \\ \texttt{correct}_{\texttt{E}}(l, \sigma + \rho') & = & \texttt{correct}_{\texttt{E}}(l, \sigma) \wedge \texttt{correct}_{\texttt{E}}(l, \rho') \end{array}$$

avec $\texttt{correct}_{\texttt{E}}() : \mathcal{E}_{\text{L}} \times (\text{ID} \rightarrow \Sigma) \rightarrow \text{BOOL}$ défini par :

$$\texttt{correct}_{\texttt{E}}(l, \sigma) = \bigwedge_{id \in \texttt{Def}(\sigma)} \texttt{correct}_{\texttt{B}}(l, \sigma(id))$$

REMARQUE Un environnement d'évaluation correspond aussi à un environnement de liaison qu'on obtient en lui appliquant $\texttt{Dom}_{\texttt{E}}()$. La définition de la correction d'un environnement d'évaluation exprime qu'un ρ est correct si et seulement si toutes les expressions « stockées » dans ρ sont correctement liées par rapport à l'environnement de liaison que représente ρ.

LEMME X.15 $\forall l \in \mathcal{E}_{\text{L}}, \forall \rho \in \mathcal{E}_{\text{E}}, \forall s \in \text{SCOPE}, \quad \texttt{correct}_{\texttt{E}}(l, \rho) \Rightarrow \texttt{correct}_{\texttt{E}}(s + l, \texttt{Lift}_{\texttt{E}}(\rho))$

Corollaire : $\texttt{correct}_{\texttt{E}}(\rho) \Rightarrow \texttt{correct}_{\texttt{E}}(s + \texttt{Dom}_{\texttt{E}}(\rho), \texttt{Lift}_{\texttt{E}}(\rho))$
(il suffit de prendre $l = \texttt{Dom}_{\texttt{E}}(\rho)$ dans le lemme).

Preuve.
La preuve du lemme se fait par récurrence sur la structure de ρ en supposant $\mathtt{correct_E}(l,\rho)$. Posons $C = \mathtt{correct_E}(s+l, \mathtt{Lift_E}(\rho))$.

1. Si $\rho = <>$
   alors $C = \mathtt{correct_E}(s+l, \mathtt{Lift_E}(<>)) = \mathtt{correct_E}(s+l, <>)$ qui est vrai par définition.

2. Si $\rho = \star + \rho'$
   $C = \mathtt{correct_E}(s+l, \star + \mathtt{Lift_E}(\rho')) = \mathtt{correct_E}(s+l, \mathtt{Lift_E}(\rho'))$. Or on a $\mathtt{correct_E}(s+l,\rho)$ par hypothèse et donc $\mathtt{correct_E}(s+l,\rho')$ ce qui permet d'appliquer l'hypothèse de récurrence et de conclure.

3. Si $\rho = \sigma + \rho'$
   alors $C = \mathtt{correct_E}(s+l, \mathtt{Lift}(\sigma)) \wedge \mathtt{correct_E}(s+l, \mathtt{Lift_E}(\rho'))$.

   De l'hypothèse $\mathtt{correct_E}(l,\rho)$ on tire $\mathtt{correct_E}(l,\rho')$ ce qui permet d'appliquer l'hypothèse de récurrence et de conclure à la vérité du deuxième terme de la conjonction.

   Pour le premier terme, il faut montrer $\displaystyle\bigwedge_{id \in \mathtt{Def}(\mathtt{Lift}(\sigma))} \mathtt{correct_B}(s+l, \mathtt{Lift}(\sigma(id)))$. Or, de l'hypothèse on tire $\mathtt{correct_E}(l,\sigma)$, et donc par définition : $\displaystyle\bigwedge_{id \in \mathtt{Def}(\sigma)} \mathtt{correct_B}(l, \sigma(id))$. On applique le corollaire du lemme (X.14) et il vient que : $\displaystyle\bigwedge_{id \in \mathtt{Def}(\sigma)} \mathtt{correct_B}(s+l, \mathtt{Lift}(\sigma(id)))$. On en déduit la conjonction voulue car, $\mathtt{Lift}()$ étant une fonction totale, $\mathtt{Def}(\mathtt{Lift}(\sigma)) = \mathtt{Def}(\sigma)$.

□

LEMME X.16 $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \mathtt{correct_B}(\rho, e) \wedge \mathtt{correct_E}(\rho) \Rightarrow \mathtt{correct_E}(\mathtt{Nenv}(\rho, e))$

Preuve.
On suppose la prémisse de l'implication et on montre sa conclusion. Distinguons deux cas suivant que e appartient ou non à SYS.

1. Si $e \notin$ SYS,
   alors $\mathtt{Nenv}(\rho,e) = \star + \mathtt{Lift_E}(\rho)$ et $\mathtt{correct_E}(\star + \mathtt{Lift_E}(\rho)) = \mathtt{correct_E}(\star + \mathtt{Dom_E}(\rho), \mathtt{Lift_E}(\rho))$. On voit alors qu'il suffit d'appliquer le corollaire du lemme (X.15).

2. Si $e \in$ SYS,
   alors $\mathtt{Nenv}(\rho,e) = \sigma + \mathtt{Lift_E}(\rho)$ et $C = \mathtt{correct_E}(\sigma + \mathtt{Lift_E}(\rho)) = \mathtt{correct_E}(\mathtt{Def}(\sigma) + \mathtt{Dom_E}(\mathtt{Lift_E}(\rho)), \sigma + \mathtt{Lift_E}(\rho))$. On peut appliquer le lemme (X.13) et de plus, $\mathtt{Def}(\sigma) = \mathtt{Dom}(e)$. Il vient alors que

$$\begin{aligned} C &= \mathtt{correct_E}(\mathtt{Dom}(e) + \mathtt{Dom_E}(\rho), \sigma + \mathtt{Lift_E}(\rho)) \\ &= \mathtt{correct_E}(\mathtt{Dom}(e) + \mathtt{Dom_E}(\rho), \sigma) \wedge \mathtt{correct_E}(dome + \mathtt{Dom_E}(\rho), \mathtt{Lift_E}(\rho)) \end{aligned}$$

   Le corollaire du résultat (X.15) permet de conclure pour le deuxième terme de la conjonction, à partir de l'hypothèse $\mathtt{correct_E}(\rho)$.

   Pour le premier terme, on doit calculer $\mathtt{correct_E}(\mathtt{Dom}(e) + \mathtt{Dom_E}(\rho), \sigma) = \displaystyle\bigwedge_{id \in \mathtt{Def}(\sigma)} \mathtt{correct_B}(\mathtt{Dom}(e) + \mathtt{Dom_E}(\rho), \sigma(id))$. Or, $e \in$ SYS, disons $e \equiv \{id = u, \ldots\}$ et donc $\mathtt{Def}(\sigma) = \mathtt{Dom}(e) = \{id, \ldots\}$ et $\sigma = [id \mapsto u, \ldots]$. Le prédicat précédent devient donc : $\bigwedge \mathtt{correct_B}(\mathtt{Dom}(e) + \mathtt{Dom_E}(\rho), u)$ où la conjonction itère sur les u parties droites des définitions de e comme à l'accoutumé. Mais, par hypothèse, on a : $\mathtt{correct_B}(\mathtt{Dom_E}(\rho), e) = \bigwedge \mathtt{correct_B}(\mathtt{Dom}(e) + \mathtt{Dom_E}(\rho), u)$ et on peut donc conclure.

□

LEMME X.17 $\forall s \subseteq \mathrm{ID}, \forall l, l' \in \mathcal{E}_{\mathrm{L}}, \forall e \in \Sigma, \quad \mathtt{correct_B}(l + \star + l', e) \Rightarrow \mathtt{correct_B}(l + s + l', e)$

Corollaire : $\forall s \subseteq \mathrm{ID}, \forall l \in \mathcal{E}_{\mathrm{L}}, \forall e \in \Sigma, \quad \mathtt{correct_B}(\star + l, e) \Rightarrow \mathtt{correct_B}(s + l, e)$.

Preuve.
La preuve se fait par induction sur e en supposant $\texttt{correct}_{\texttt{B}}(l + \star + l', e)$.

1. $e \equiv id^n$
   $\texttt{correct}_{\texttt{B}}(l + s + l', id^n)$ est vrai par définition.

2. $e \equiv id_n$
   De l'hypothèse $\texttt{correct}_{\texttt{B}}(l{+}\star{+}l', id_n)$ on tire $n \neq \texttt{lg}(l)$ et donc $\texttt{nth}(n{+}1, l{+}\star{+}l') = \texttt{nth}(n{+}1, l{+}s{+}l')$ et par conséquent $\texttt{correct}_{\texttt{B}}(l + \star + l', id_n) = \texttt{correct}_{\texttt{B}}(l + s + l', id_n)$.

3. $e \equiv \{id = u, \ldots\}$
   $\texttt{correct}_{\texttt{B}}(l{+}s{+}l', e) = \bigwedge \texttt{correct}_{\texttt{B}}(\{id, \ldots\}{+}l{+}s{+}l', u)$. Mais de l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(\{id, \ldots\}+ l + \star + l', u)$ ce qui permet d'appliquer l'hypothèse de récurrence avec les environnements de liaison $\{id, \ldots\} + l$ et l' et de conclure.

4. $e \equiv u \,\#\, v$
   $\texttt{correct}_{\texttt{B}}(l + s + l', e) = \texttt{correct}_{\texttt{B}}(l + s + l', u) \wedge \texttt{correct}_{\texttt{B}}(l + s + l', v)$. Mais de l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(l + \star + l', u)$ et $\texttt{correct}_{\texttt{B}}(l + \star + l', v)$ ce qui permet d'utiliser l'hypothèse de récurrence et de conclure.

5. $e \equiv u \;\textbf{.}\; v$
   $\texttt{correct}_{\texttt{B}}(l + s + l', e) = \texttt{correct}_{\texttt{B}}(l + s + l', u) \wedge \texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l + s + l', v)$. Mais de l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(l + \star + l', u)$ et $\texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l + \star + l', v)$ ce qui permet d'utiliser l'hypothèse de récurrence sur u avec les environnements de liaison l et l' et sur v avec les environnements de liaison $\texttt{Dom}(u) + l$ et l'. La conclusion suit.

$\square$

Lemme X.18 $\forall s, s' \subseteq \textrm{Id}, \forall l, l' \in \mathcal{E}_{\textrm{L}}, \forall e \in \Sigma, \quad \texttt{correct}_{\texttt{B}}(l + s + l', e) \Rightarrow \texttt{correct}_{\texttt{B}}(l + s \cup s' + l', e)$

Preuve.
La preuve se fait par induction sur e en supposant $\texttt{correct}_{\texttt{B}}(l + s + l', e)$.

1. $e \equiv id^n$
   $\texttt{correct}_{\texttt{B}}(l + s \cup s' + l', id^n)$ est vrai par définition.

2. $e \equiv id_n$
   On distingue deux cas suivant la valeur de n.
   (a) Si $n = \texttt{lg}(l)$,
   alors de l'hypothèse $\texttt{correct}_{\texttt{B}}(l + s + l', e)$ on tire $(id \in s)$ et par suite $(id \in s \cup s')$ et donc $\texttt{correct}_{\texttt{B}}(l + s \cup s' + l', id_n)$.
   (b) Sinon,
   $\texttt{nth}(n, l{+}s{\cup}s'{+}l') = \texttt{nth}(n, l{+}s{+}l')$ et donc $\texttt{correct}_{\texttt{B}}(l{+}s{\cup}s'{+}l', id_n) = \texttt{correct}_{\texttt{B}}(l{+}s{+}l', id_n)$ qui est vrai par hypothèse.

3. $e \equiv \{id = u, \ldots\}$
   $\texttt{correct}_{\texttt{B}}(l + s \cup s' + l', e) = \bigwedge \texttt{correct}_{\texttt{B}}(\{id, \ldots\} + l + s \cup s' + l', u)$. Mais, de l'hypothèse, on tire $\texttt{correct}_{\texttt{B}}(\{id, \ldots\} + l + s + l', u)$ ce qui permet d'appliquer l'hypothèse de récurrence avec les environnements de liaison $\{id, \ldots\} + l$ et l' et de conclure.

4. $e \equiv u \,\#\, v$
   $\texttt{correct}_{\texttt{B}}(l + s \cup s' + l', e) = \texttt{correct}_{\texttt{B}}(l + s \cup s' + l', u) \wedge \texttt{correct}_{\texttt{B}}(l + s \cup s' + l', v)$. Mais de l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(l + s + l', u)$ et $\texttt{correct}_{\texttt{B}}(l + s + l', v)$ ce qui permet d'utiliser l'hypothèse de récurrence et de conclure.

5. $e \equiv u \;\textbf{.}\; v$
   $\texttt{correct}_{\texttt{B}}(l + s \cup s' + l', e) = \texttt{correct}_{\texttt{B}}(l + s \cup s' + l', u) \wedge \texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l + s \cup s' + l', v)$. Mais de l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(l + s + l', u)$ et $\texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l + s + l', v)$ ce qui permet d'utiliser

l'hypothèse de récurrence sur u avec les environnements de liaison l et l' et sur v avec les environnements de liaison $\texttt{Dom}(u) + l$ et l'.

$\square$

LEMME X.19 $\forall l \in \mathcal{E}_{\mathrm{L}}, \forall u, \ldots, u', \cdots \in \Sigma,$

$$\texttt{correct}_{\texttt{B}}(l, \{id = u, \ldots\}) \wedge \texttt{correct}_{\texttt{B}}(l, \{id' = u', \ldots\}) \Rightarrow \texttt{correct}_{\texttt{B}}(l, \texttt{bind}(l, \{id = u, \ldots, id' = u', \ldots\}))$$

Preuve.
On pose $s = \{id, \ldots\}$ et $s' = \{id', \ldots\}$. Alors, $P \equiv \texttt{bind}(l, \{id = u, \ldots, id' = u', \ldots\}) \equiv \{id = \texttt{bind}(s \cup s' + l, u), \ldots, id' = \texttt{bind}(s \cup s' + l, u'), \ldots\}$ d'où $\texttt{correct}_{\texttt{B}}(l, P) = \bigwedge \texttt{correct}_{\texttt{B}}(s \cup s' + l, \texttt{bind}(s \cup s' + l, u)) \wedge \bigwedge \texttt{correct}_{\texttt{B}}(s \cup s' + l, \texttt{bind}(s \cup s' + l, u'))$. Or de l'hypothèse on tire $\texttt{correct}_{\texttt{B}}(s + l, u)$ et $\texttt{correct}_{\texttt{B}}(s' + l, u')$. On peut donc appliquer le lemme (X.18) aux u et u' pour conclure. $\square$

LEMME X.20 $\forall e \in \Sigma, \forall l, l' \in \mathcal{E}_{\mathrm{L}}, \forall s \subseteq \mathrm{ID}, \quad \texttt{correct}_{\texttt{F}}(l + s + l', e) \Rightarrow \texttt{correct}_{\texttt{F}}(l + l', e)$

Corollaire : $\forall e \in \Sigma, \forall l \in \mathcal{E}_{\mathrm{L}}, \forall s \subseteq \mathrm{ID}, \quad \texttt{correct}_{\texttt{F}}(s + l, e) \Rightarrow \texttt{correct}_{\texttt{F}}(l, e)$
(c'est le lemme avec les environnements de liaison $<>$ et l).

Preuve.
La démonstration se fait par induction sur la structure de e en supposant $\texttt{correct}_{\texttt{F}}(l + l', e)$.

1. $e \equiv id^n$
   À partir de l'hypothèse on a $(\star == \texttt{index}(l + s + l', id, 0))$ est vrai, alors $(\star == \texttt{index}(l + l', id, 0))$ est vrai aussi car $s \neq \star$ et donc on a $\texttt{correct}_{\texttt{F}}(l + l', id^n)$.

2. $e \equiv id_n$
   $\texttt{correct}_{\texttt{F}}(l + l', id^n)$ est vrai par définition.

3. $e \equiv \{id = u, \ldots\}$
   On a $\bigwedge \texttt{correct}_{\texttt{F}}(\{id, \ldots\} + l + s + l', u)$ et donc on peut appliquer l'hypothèse de récurrence avec les listes $\{id, \ldots\} + l$ et l'. Il vient $\bigwedge \texttt{correct}_{\texttt{F}}(\{id, \ldots\} + l + l', u)$ et donc $\texttt{correct}_{\texttt{F}}(l + l', \{id = u, \ldots\})$.

4. $e \equiv u \,\#\, v$
   On tire de l'hypothèse $\texttt{correct}_{\texttt{F}}(l + s + l', u)$ et donc en appliquant l'hypothèse de récurrence, on a $\texttt{correct}_{\texttt{F}}(l + l', u)$. Il en va de même pour v d'où $\texttt{correct}_{\texttt{F}}(l + l', u \,\#\, v)$.

5. $e \equiv u \,\texttt{.}\, v$
   On tire de l'hypothèse $\texttt{correct}_{\texttt{F}}(l + s + l', u)$ et $\texttt{correct}_{\texttt{F}}(\texttt{Dom}(u) + l + s + l', v)$. On peut donc appliquer l'hypothèse de récurrence à u avec l et l' et à v avec les environnements $\texttt{Dom}(u) + l$ et l'. On a donc $\texttt{correct}_{\texttt{F}}(l + l', u)$ et $\texttt{correct}_{\texttt{F}}(\texttt{Dom}(u) + l + l', v)$ et donc $\texttt{correct}_{\texttt{F}}(l + l', u \,\texttt{.}\, v)$.

$\square$

REMARQUE Le lemme est faux pour $s \in \mathrm{SCOPE}$, par exemple si on prend $s = \star^n$.

LEMME X.21 $\forall e \in \Sigma, \forall s \in \mathrm{SCOPE}, \forall l, l' \in \mathcal{E}_{\mathrm{L}}, \quad \texttt{correct}_{\texttt{B}}(l + s + l', e) \wedge \texttt{C}(l + s + l', e) \Rightarrow \texttt{correct}_{\texttt{B}}(l + l', e)$

Corollaire : $\texttt{correct}_{\texttt{B}}(s + l, e) \wedge \texttt{C}(s + l, e) \Rightarrow \texttt{correct}_{\texttt{B}}(l, e)$
(c'est le lemme avec les environnements de liaison $<>$ et l).

Preuve. La démonstration se fait par induction sur la structure de e en supposant $\texttt{correct}_{\texttt{B}}(l + s + l', e)$ et $\texttt{C}(l + s + l', e)$.

1. $e \equiv id^n$
   $\texttt{correct}_{\texttt{B}}(l + l', id^n)$ est vrai par définition.

2. $e \equiv id_n$
   $\texttt{C}(l + s + l', e)$ est faux par définition et donc l'implication est vraie.

3. $e \equiv \{id = u, \dots\}$
   De $\texttt{correct}_{\texttt{B}}(l+s+l', \{id = u, \dots\})$ on déduit $\texttt{correct}_{\texttt{B}}(\{id, \dots\} + l + s + l', u)$ et de $\texttt{C}(l+s+l', \{id = u, \dots\})$ on déduit $\texttt{C}(\{id, \dots\} + l + s + l', u)$. On peut donc appliquer l'hypothèse de récurrence aux expressions u avec les environnements de liaison $\{id, \dots\} + l$ et l', et il vient : $\bigwedge \texttt{correct}_{\texttt{B}}(\{id, \dots\} + l + l', u)$ et donc $\texttt{correct}_{\texttt{B}}(l + l', \{id = u, \dots\})$.

4. $e \equiv u \# v$
   De $\texttt{correct}_{\texttt{B}}(l+s+l', u \# v)$ on déduit $\texttt{correct}_{\texttt{B}}(l+s+l', u)$ et de $\texttt{C}(l+s+l', u)$ on tire $\texttt{C}(l+s+l', u)$. On peut donc appliquer l'hypothèse de récurrence et il vient $\texttt{correct}_{\texttt{B}}(l+l', u)$. Il en va de même pour v et donc on a : $\texttt{correct}_{\texttt{B}}(l + l', u \# v)$.

5. $e \equiv u \,\texttt{.}\, v$
   De $\texttt{correct}_{\texttt{B}}(l + s + l', u \,\texttt{.}\, v)$ on déduit $\texttt{correct}_{\texttt{B}}(l + s + l', u)$ et $\texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l + s + l', v)$. Et de $\texttt{C}(l + s + l', u \,\texttt{.}\, v)$ on déduit $\texttt{C}(l + s + l', u)$ et $\texttt{C}(\texttt{Dom}(u) + l + s + l', v)$. On peut donc appliquer l'hypothèse de récurrence à u avec les environnement de liaison l et l' et à v avec les environnements de liaison $\texttt{Dom}(u) + l$ et l'. Il vient $\texttt{correct}_{\texttt{B}}(l + l', u)$ et $\texttt{correct}_{\texttt{B}}(\texttt{Dom}(u) + l + l', v)$ d'où le résultat $\texttt{correct}_{\texttt{B}}(l + l', u \,\texttt{.}\, v)$.

□

LEMME X.22 $\forall e \in \Sigma, \forall u, \dots \in \Sigma, \forall l \in \mathcal{E}_{\mathrm{L}}, \quad \texttt{correct}(l, \{id = u, \dots\} \,\texttt{.}\, e) \wedge \texttt{C}(\{id, \dots\} + l, e) \Rightarrow \texttt{correct}(l, e)$

Preuve. En appliquant les corollaires des lemmes (X.20) et (X.21). □

X.6.4.3 Le théorème de correction de la réduction

Nous avons enfin les outils nécessaires à la preuve du théorème de correction de la réduction. Pour montrer :

THÉORÈME X.2 **(L'évaluation préserve la correction)**

$$\forall e \in \Sigma_{\mathcal{C}}, \quad e \twoheadrightarrow e' \Rightarrow e' \in \Sigma_{\mathcal{C}}$$

nous montrons :

THÉORÈME X.3 **(La réduction préserve la correction)**

$$\forall e \in \Sigma, \forall \rho \in \mathcal{E}_{\mathrm{E}}, \quad \texttt{correct}_{\texttt{E}}(\rho) \wedge \texttt{correct}(\rho, e) \wedge \rho \vdash e \rightarrow e' \Rightarrow \texttt{correct}(\rho, e')$$

Preuve.
La formulation du théorème avec $\rho = \epsilon$ permet d'établir que $\forall e \in \Sigma_{\mathcal{C}}, e \rightarrow e' \Rightarrow e' \in \Sigma_{\mathcal{C}}$. En itérant ce résultat, on a le théorème (X.2).

La démonstration du théorème se fait par induction sur la structure d'une preuve de réduction en inspectant chaque règle de réduction. On suppose vraie les prémisses de l'implication et on montre la conclusion.

1. Règle Ref^n
   On a $e \equiv e' \equiv id^n$ et donc $\texttt{correct}_{\texttt{B}}(\rho, e') = \texttt{correct}_{\texttt{B}}(\rho, e)$ qui est vrai par hypothèse.

2. Règle Ref_C
   Par hypothèse on a $\texttt{correct}_{\texttt{E}}(\rho)$ et on en déduit $\texttt{correct}_{\texttt{B}}(\rho, v)$ et donc, par le lemme (X.12), on a aussi $\texttt{correct}_{\texttt{B}}(\rho, v')$.

3. Règle $Ref_{\neg C}$
   On a $e \equiv e' \equiv id_n$ et donc $\texttt{correct}_{\texttt{B}}(\rho, e') = \texttt{correct}_{\texttt{B}}(\rho, e)$ qui est vrai par hypothèse.

4. Règle $\{\dots\}$
   Des hypothèses on déduit avec le lemme (X.16) que $\texttt{correct}_{\texttt{E}}(\texttt{Nenv}(\rho, \{id = u, \dots\}))$. On en déduit aussi que $\texttt{correct}_{\texttt{B}}(\{id, \dots\} + \texttt{Dom}_{\texttt{E}}(\rho), u)$ et en appliquant le lemme (X.6), que $\texttt{correct}_{\texttt{B}}(\texttt{Nenv}(\rho, e), u)$.

On a donc les hypothèses nécessaires pour appliquer l'hypothèse de récurrence aux u avec l'environnement d'évaluation $\mathtt{Nenv}(\rho, e)$ et déduire que l'on a $\mathtt{correct_B}(\mathtt{Nenv}(\rho, e), u')$ pour toute partie droite u tel que $\mathtt{Nenv}(\rho, e) \vdash u \rightarrow u'$. Donc on a bien $\bigwedge \mathtt{correct_B}(\mathtt{Nenv}(\rho, e), u)$, la conjonction portant sur toutes les parties droites de e, c'est-à-dire qu'on a $\mathtt{correct_B}(\rho, e')$.

5. Règle $\#_1$
   On a $\mathtt{correct_B}(\rho, \{id = u, \dots\})$ et $\mathtt{correct_B}(\rho, \{id' = u', \dots\})$ à partir de l'hypothèse $\mathtt{correct_B}(\rho, e)$. On en déduit $\mathtt{correct_B}(\rho, \{id = u, \dots, id' = u', \dots\})$ à l'aide du lemme (X.19). On applique alors le lemme (X.9) pour conclure qu'on a bien $\mathtt{correct}(\rho, \{id = u, \dots, id' = u', \dots\})$.

6. Règle $\#_2$
   On suppose que $\rho \vdash u \rightarrow u'$. De $\mathtt{correct_B}(\rho, u \# v)$ on tire $\mathtt{correct_B}(\rho, u)$. On peut donc appliquer l'hypothèse de récurrence et on déduit $\mathtt{correct}(\rho, u')$. Si u ne se réduit pas, on a par hypothèse $\mathtt{correct}(\rho, u')$. Il en va de même pour v. On a donc $\mathtt{correct}(\rho, u') \wedge \mathtt{correct}(\rho, v')$ c'est-à-dire $\mathtt{correct}(\rho, u \# v)$.

7. Règle $\centerdot_{S_1}$
   Il suffit d'appliquer le lemme (X.22) pour obtenir $\mathtt{correct}(\rho, v)$.

8. Règle $\centerdot_{S_2}$
   De l'hypothèse $\mathtt{correct}(\rho, e)$ on tire $\mathtt{correct}(\rho, u)$ et donc, si $\rho \vdash u \longrightarrow u'$, on peut appliquer l'hypothèse de récurrence pour déduire $\mathtt{correct}(\rho, u')$. Sinon, $\rho \vdash u \longrightarrow_0 u'$ avec $u' \equiv u$ et on a aussi $\mathtt{correct}(\rho, u')$.

   Par le lemme (X.16), avec l'hypothèse $\mathtt{correct_E}(\rho)$, on déduit qu'on a aussi $\mathtt{correct_E}(\mathtt{Nenv}(\rho, u'))$. De l'hypothèse on tire aussi $\mathtt{correct}(\mathtt{Nenv}(\rho, u), v)$ et on en déduit $\mathtt{correct}(\mathtt{Nenv}(\rho, u'), v)$ car $\mathtt{Dom_E}(\mathtt{Nenv}(\rho, u')) = \mathtt{Dom}(u') + \mathtt{Dom_E}(\mathtt{Lift_E}(\rho)) = \mathtt{Dom}(u) + \mathtt{Dom_E}(\mathtt{Lift_E}(\rho)) = \mathtt{Dom_E}(\mathtt{Nenv}(\rho, u))$ (on a $\mathtt{Dom}(u) = \mathtt{Dom}(u')$ car $u \in \textsc{Sys}$ donc $u' \in \textsc{Sys}$).

   On a donc $\mathtt{correct_E}(\mathtt{Nenv}(\rho, u'))$ et $\mathtt{correct}(\mathtt{Nenv}(\rho, u'), v)$ ce qui permet d'appliquer l'hypothèse de récurrence et il vient $\mathtt{correct}(\mathtt{Nenv}(\rho, u'), v')$.

   En combinant les deux résultats, on a $\mathtt{correct}(\rho, u') \wedge \mathtt{correct}(\mathtt{Nenv}(\rho, u'), v')$ c'est-à-dire $\mathtt{correct}(\rho, u' \centerdot v')$.

9. Règle $\centerdot_{\neg S_1}$
   De l'hypothèse $\mathtt{correct}(\rho, e)$ on tire $\mathtt{correct}(\rho, u)$ et donc on peut appliquer l'hypothèse de récurrence pour déduire $\mathtt{correct}(\rho, u')$.

   De l'hypothèse on tire aussi $\mathtt{correct}(\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho), v)$, donc $\mathtt{correct_B}(\star + \mathtt{Dom_E}(\rho), v)$ puisque $u \notin \textsc{Sys}$. On peut donc appliquer le corollaire (X.17) pour en inférer $\mathtt{correct_B}(\mathtt{Dom}(u') + \mathtt{Dom_E}(\rho), v)$. Il suffit alors d'appliquer le lemme (X.12) pour déduire $\mathtt{correct}(\mathtt{Dom}(u') + \mathtt{Dom_E}(\rho), v')$.

   Par conséquent on a $\mathtt{correct}(\rho, u') \wedge \mathtt{correct}(\mathtt{Dom}(u') + \mathtt{Dom_E}(\rho), v')$ c'est-à-dire $\mathtt{correct}(\rho, u' \centerdot v')$.

10. Règle $\centerdot_{\neg S_2}$
    Pour ce qui concerne u et u', le raisonnement est le même que dans le cas $\centerdot_{S_2}$ et on aboutit à $\mathtt{correct}(\rho, u')$.

    Puisqu'on a $\mathtt{correct}(\rho, u')$, en appliquant le lemme (X.16) on a $\mathtt{correct_E}(\mathtt{Nenv}(\rho, u'))$. De plus, $u, u' \notin \textsc{Sys}$ et donc $\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho) = \mathtt{Dom}(u') + \mathtt{Dom_E}(\rho) = \mathtt{Dom_E}(\mathtt{Nenv}(\rho, u'))$. Donc, à partir de $\mathtt{correct}(\mathtt{Dom}(u) + \mathtt{Dom_E}(\rho), v)$ déduit de l'hypothèse, on a $\mathtt{correct}(\mathtt{Nenv}(\rho, u'), v)$. Si $\mathtt{Nenv}(\rho, u') \vdash v \longrightarrow v'$, on peut alors appliquer l'hypothèse de récurrence à v et de déduire $\mathtt{correct}(\mathtt{Nenv}(\rho, u'), v')$. Si $\mathtt{Nenv}(\rho, u') \vdash v \longrightarrow_0 v'$, alors $v' \equiv v$ et on a encore $\mathtt{correct}(\mathtt{Nenv}(\rho, u'), v')$.

    Par suite, on a $\mathtt{correct}(\rho, u') \wedge \mathtt{correct}(\mathtt{Nenv}(\rho, u'), v')$ ce qui permet de conclure $\mathtt{correct}(\rho, u' \centerdot v')$ et d'achever la preuve.

□

Chapitre XI

Éléments d'implémentation et exemples de calculs sur les amalgames

XI.1 Éléments d'implémentation en Mathematica

Un premier évaluateur des amalgames a été développé de manière autonome en Mathematica. Il est fondé sur la sémantique opérationnelle que nous avons décrite dans le précédent chapitre. L'intégration du calcul des amalgames en 8½ est en cours d'implémentation et sera décrite dans la dernière partie de ce document.

Nous donnons ici quelques éléments de l'implémentation des règles de réduction des amalgames en Mathematica. On trouvera dans l'annexe B l'ensemble des règles Mathematica permettant une évaluation des exemples que nous donnons dans ce chapitre, excepté pour l'exemple de la structuration objet d'un programme 8½ qui fait intervenir l'aspect stream de 8½ (cf. section XI.3.4, page 169).

XI.1.1 Principe de l'implémentation

Les règles de la sémantique opérationnelle correspondent à la spécification d'un évaluateur. L'évaluation d'un terme des amalgames se fait par réductions successives, jusqu'à ce qu'une forme normale soit atteinte, quand elle existe.

Mathematica [Wol88] est un environnement de calcul *symbolique*, c'est-à-dire que c'est un système de réécriture qui manipule des arbres syntaxiques. Les règles de réécriture spécifiées en section X.3.2 page 137, doivent être traduites en règles Mathematica. À chaque règle de réécriture de la sémantique va correspondre une règle de réécriture Mathematica qui consiste à parcourir l'arbre syntaxique d'une expression des amalgames, et à effectuer un traitement en fonction du type du nœud de l'arbre traité. Nous avons montré que le système de réécriture était déterministe, et en particulier qu'à chaque expression s'appliquait au plus une règle. On peut donc associer à chaque règle de la sémantique une règle Mathematica unique. Les prémisses d'une règle de la sémantique correspondent à des calculs auxiliaires.

Avant de pouvoir effectuer un traitement sur les nœuds de l'arbre, il est nécessaire de représenter, en Mathematica, les deux objets principaux de la sémantique :

- une expression des amalgames,

- l'environnement d'évaluation ρ.

Nous ne détaillerons pas ici la définition de l'environnement de liaison. Le lecteur intéressé par sa représentation se reportera à l'annexe B où figure l'ensemble du code Mathematica d'évaluation des amalgames.

Élément de Σ	Traduction en Mathematica
id^n	`Up[n, id]`
id_n	`Scope[n, id]`
id	`Up[0, id]`
$\{\ldots\}$	`SYS[ ... ]`
id = e	`LET[id, e]`
$u \# v$	`Plus[u, v]` ou `u+v`
$u \,.\, v$	`Dot[u, v]` ou `u.v`

TAB. 1 –: *La traduction d'expressions d'amalgames en Mathematica.*

XI.1.2 Représentation d'une expression d'amalgames

L'objet de base du langage Mathematica est l'expression : c'est un objet de la forme `T[ ... ]` où `...` représente une ou plusieurs sous-expressions. On appelle `T` la *tête* de l'expression. En utilisant la représentation classique d'une expression sous la forme d'un arbre syntaxique, avec les opérateurs en position nodale, on représente en Mathematica une expression d'amalgame en définissant une tête différente pour chaque opérateur différent des amalgames. La table 1 décrit la traduction d'expressions d'amalgames en Mathematica.

Il apparaît que la traduction d'expressions des amalgames est immédiate. Par exemple, l'expression :

$$\{a = 1, b = \{c = a_1\}, d = u^1 \,.\, v, e = f \# g\} \tag{1}$$

est traduite en :

```
SYS[LET[a, 1], LET[b, SYS[LET[c, Scope[1, a]]]], LET[d, Dot[Up[1, u], Up[0, v]]],
    LET[e, Plus[Up[0, f], Up[0, g]]]]
```

Pour des raisons de simplicité, il a été défini en Mathematica un *lecteur* qui effectue la traduction d'une expression entrée sous la forme `Sys[`id `=` e`]` où e est une expression de $\Sigma_{\mathcal{I}}$ (seuls les opérateurs `+` et `.` sont utilisés en lieu et place de $\#$ et $\,.\,$). Par exemple, la forme externe de l'équation (1) est :

```
Sys[a=1, b=Sys[c=Scope[1, a]], d=Up[1, u].v, e=f+g]
```

Le lecteur effectue de plus une traduction des expression de $\Sigma_{\mathcal{I}}$ en expressions de $\Sigma_{\mathcal{C}}$ (cf. section X.2.2, page 133).

XI.1.3 Représentation d'un environnement d'évaluation ρ

Nous avons défini un environnement d'évaluation, ρ élément de $\mathcal{E}_{\mathrm{E}}$, comme une liste de fonctions qui associaient à un identificateur l'expression qui lui était associée. Le n^{e} élément de ρ correspond à l'ensemble des définitions qui apparaissent au n^{e} scope. Par exemple, dans l'expression suivante :

$$\{a = 1, b = 2, c = \{d = 1, e = 2\}\}$$

au niveau du système c, l'environnement d'évaluation ρ est égal à :

$$\rho \quad = \quad < [d \mapsto 1, e \mapsto 2], [a \mapsto 1, b \mapsto 2, c \mapsto \{d = 1, e = 2\}] >$$

Nous utilisons le fait que Mathematica est un langage permettant la manipulation de règles de réécriture pour représenter ρ. Les règles de réécriture sont représentées sous la forme de listes (les listes sont représentées en Mathematica entre accolades, chaque élément étant séparé par une virgule) de couples `l->r` où `l` représente le membre gauche de la règle et `r` le membre droit de la règle [vL90]. Par conséquent, l'environnement de l'exemple ci-dessus est représenté sous la forme d'une liste de listes de règles de réécriture :

```
{{d -> 1, e -> 2}, {a -> 1, b -> 2, c -> SYS[LET[d, 1], LET[e, 2]]}}
```

Cette représentation permet, lors de la substitution d'une référence liée par sa définition, d'utiliser la capacité de Mathematica à réécrire une expression par l'utilisation de l'opérateur `ReplaceAll`.

XI.1.4 Traduction d'une règle de la sémantique opérationnelle en Mathematica

Les règles de réécriture de la sémantique opérationnelle sont définies par inférence sur la structure des termes de Σ. Certaines règles sont gardées par une condition. La traduction de ces règles en Mathematica est immédiate. Certaines règles ne peuvent s'appliquer que si leurs prémisses ont pu être déduites. Pour cela, on a besoin de savoir si $\rho \vdash e \rightarrow_n e'$ s'applique avec $n = 0$ ou $n = 1$. On utilise le théorème reliant C() et la notion de forme normale (cf. section X.4.2, page 138) pour remplacer cette prémisse par un calcul du prédicat C().

On définit l'évaluateur `Eval[env_][e_]` où `env_` est un environnement d'évaluation et où `e_` peut s'instancier avec n'importe quelle expression qui correspond à la spécification de $env \vdash e \rightarrow$ `Eval[env][e]`. Par exemple, la règle de substitution d'une référence liée (cf. section 3, page 139) se traduit en Mathematica :

```
Eval[env_List][Scope[n_, id_]] :=  With[{e = id /. env[[n+1]]},
                                        If[RC[EnvToScope[env],e][Bind[EnvToScope[env]][e], Scope[n, id]]]]
```

Cette règle de réécriture nécessite deux arguments : un environnement (de type liste) et un argument de type `Scope` où les deux arguments de `Scope` sont nommés `n` et `id`[1]. Une variable locale, `e` est définie. Elle est égale à la valeur de `id` dans le `n`^{e} scope définit dans `env`[2]. Si la substitution de `e` dans `env` est C(), alors la valeur de l'évaluation est la liaison de `e` dans l'environnement calculé à partir de `env`, sinon l'expression `Scope[n_, id_]` est inchangée.

Il apparaît que le traitement de l'expression est identique à la spécification de la règle de la sémantique opérationnelle pour ce qui concerne le traitement de la réduction d'un élément de SCOPE.

XI.2 Exemple d'expressions d'amalgame purs

XI.2.1 Exemple du calcul de fonctions booléennes

Le calcul d'une fonction booléenne peut être effectué en utilisant uniquement les opérateurs de conjonction et de négation. Il est aussi possible de définir ces opérateurs à partir de leur tables de vérité [Aum77]. Nous donnons ici un exemple de traduction des opérations logiques `not` et `and` en amalgames.

XI.2.1.1 La fonction booléenne not

L'expression suivante[3] définit une fonction booléenne `not` où `valeur` est une donnée du premier système défini, la négation de `valeur` apparaissant comme la valeur de la seconde expression :

$$\begin{aligned} NOT \quad \equiv \quad \{ & valeur = vrai^0, \\ & premier = \{vrai = f^0, faux = v^0\} \;\mathbf{.}\; valeur_1, \\ & second = \{f = faux^0, v = vrai^0\} \;\mathbf{.}\; premier_1\} \end{aligned}$$

Le calcul de la négation de `valeur` (qui appartient à l'ensemble {`vrai`, `faux`}) s'effectue *séquentiellement* en deux temps :

1. Deux équations sont définies dans le premier système. Ces équations définissent deux definiens f et v accessibles respectivement par leur definiendum $vrai$ et $faux$. Si `valeur` est égal à $vrai$, alors c'est la référence f qui est sélectionnée, v sinon.

2. Une fois que le premier système s'est évalué en v ou f en fonction de `valeur`, la sélection de *premier* dans le second système permet de convertir la référence en un élément de l'ensemble des booléens {`vrai`, `faux`}.

1. Cette construction est similaire à la fonction ML : `let f = function x -> x+1` où le paramètre de la fonction `f` est identifié à la variable `x`.

2. La notation « `/.` » est synonyme de `ReplaceAll` qui applique une liste de règles de réécriture à un argument et a pour valeur le membre droit de la règle qui peut s'appliquer sur cet argument (s'il existe) ; l'expression l`[[`n`]]` a pour résultat le n^{e} élément de la liste l.

3. Qui s'inspire très librement de [Dam94c, pp 66].

L'algorithme utilise une conversion de `valeur` en un terme de $\{v, f\}$ en associant à une valeur vraie un terme qui représente la valeur fausse, et inversement, puis une conversion en un terme correct des booléens. Nous donnons les réductions de l'expression où *valeur* est instanciée avec *vrai* :

$$\begin{array}{lll}
NOT & \rightarrow & \{\mathcal{V}, premier = \{vrai = f^0, faux = v^0\} \centerdot vrai_0, second = \{f = faux^0, v = vrai^0\} \centerdot premier_1\} \\
& \rightarrow & \{\mathcal{V}, premier = \{vrai = f^0, faux = v^0\} \centerdot f^0, second = \{f = faux^0, v = vrai^0\} \centerdot premier_1\} \\
& \rightarrow & \{\mathcal{V}, premier = f^0, second = \{f = faux^0, v = vrai^0\} \centerdot premier_1\} \\
& \rightarrow & \{\mathcal{V}, premier = f^0, second = \{f = faux^0, v = vrai^0\} \centerdot f_0\} \\
& \rightarrow & \{\mathcal{V}, premier = f^0, second = \{f = faux^0, v = vrai^0\} \centerdot faux^0\} \\
& \rightarrow & \{\mathcal{V}, premier = f^0, second = faux^0\}
\end{array}$$

où $\mathcal{V}$ représente l'expression $valeur = vrai^0$.

XI.2.1.2 La fonction booléenne `and`

En utilisant un schéma de traitement similaire à celui adopté pour l'exemple précédent, nous montrons comment utiliser les amalgames pour représenter le calcul de la fonction logique dyadique `and`. L'expression suivante :

$$\begin{array}{lll}
AND & \equiv & \{r = \{ \quad \{v1 = faux, \\
& & \qquad\qquad v2 = vrai, \\
& & \qquad\qquad prem = \{faux = f, vrai = autre\} \centerdot v1, \\
& & \qquad\qquad et = \{f = faux, autre = v2\} \centerdot prem\}\}\}
\end{array}$$

définit les definiendum `v1` et `v2` qui sont les paramètres de l'expression. L'expression utilise le codage de la fonction `and` pour effectuer l'opération entre les deux valeurs : si `v1` est faux alors le résultat (donné par *et*) est égal a f qui est ensuite converti en la valeur booléenne $faux$, sinon c'est la valeur de $v1$ (à travers le definiens *autre*) qui est sélectionné. Nous donnons les réductions pour des valeurs de $(v1, v2)$ égales à $(faux, vrai)$:

$$\begin{array}{lll}
AND & \rightarrow & \{r = \{\mathcal{V}, prem = \{faux = f^0, vrai = autre^0\} \centerdot faux_0, et = \{f = faux^0, autre = vrai^0\} \centerdot prem_1\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = \{faux = f^0, vrai = autre^0\} \centerdot f^0, et = \{f = faux^0, autre = vrai^0\} \centerdot prem_1\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = f^0, et = \{f = faux^0, autre = vrai^0\} \centerdot prem_1\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = f^0, et = \{f = faux^0, autre = vrai^0\} \centerdot f_0\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = f^0, et = \{f = faux^0, autre = vrai^0\} \centerdot faux^0\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = f^0, et = faux^0\}\}
\end{array}$$

où $\mathcal{V}$ représente les définitions $v1 = faux, v2 = vrai$. Pour les valeurs $(vrai, vrai)$, nous avons :

$$\begin{array}{lll}
AND & \rightarrow & \{r = \{\mathcal{V}, prem = \{faux = f^0, vrai = autre^0\} \centerdot vrai_0, et = \{f = faux^0, autre = vrai^0\} \centerdot prem_1\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = \{faux = f^0, vrai = autre^0\} \centerdot autre^0, et = \{f = faux^0, autre = vrai^0\} \centerdot prem_1\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = autre^0, et = \{f = faux^0, autre = vrai^0\} \centerdot prem_1\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = autre^0, et = \{f = faux^0, autre = vrai^0\} \centerdot autre_0\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = autre^0, et = \{f = faux^0, autre = vrai^0\} \centerdot vrai^0\}\} \\
& \rightarrow & \{r = \{\mathcal{V}, prem = autre^0, et = vrai^0\}\}
\end{array}$$

où $\mathcal{V}$ représente les définitions $v1 = vrai, v2 = vrai$.

XI.2.2 Codage de l'arithmétique en amalgames

Nous allons illustrer le fait qu'on peut traduire les fonctions arithmétiques récursives en une expression des amalgames. Cela montre la puissance d'expression formelle du calcul sur les amalgames. Cependant, nous ne démontrerons pas ce résultat de manière rigoureuse.

Nous nous restreindrons à la classe des *fonctions totales*. En effet, le schéma de traduction que nous proposons n'assure pas la non-terminaison du calcul de l'amalgame associé à l'application d'une fonction sur des arguments n'appartenant pas au domaine de définition. Par contre, rien ne permet d'affirmer qu'il n'est pas possible de spécifier une autre notion de définissabilité par amalgame qui permettrait de faire coïncider fonctions « qui bouclent » et expressions n'ayant pas de forme normale.

Nous commençons par rappeler la définition des fonctions arithmétiques récursives[1], puis nous définirons une notion de *définissabilité par les amalgames* et nous esquisserons le codage des fonctions arithmétiques par les amalgames. Le codage proposé a été implémenté sous Mathematica par une fonction qui traduit une fonction arithmétique récursive en un terme de $\Sigma_{\mathcal{I}}$, lui-même traduisible en un terme de $\Sigma_{\mathcal{C}}$. L'exemple de l'addition est détaillé. Nous détaillons dans l'annexe C le source Mathematica du traducteur de l'arithmétique dans les amalgames.

XI.2.2.1 Rappel : les fonctions arithmétiques récursives

DÉFINITION XI.1 Une *fonction numérique* est une fonction (totale) $\varphi : \mathbb{N}^p \mapsto \mathbb{N}$.

Les fonctions numériques *de base* U_i^p, S, Z sont définies par :

$$U_i^p(n_0, \ldots, n_p) = n_i, \qquad 0 \leq i \leq p$$
$$S(n) = n + 1$$
$$Z() = 0$$

Notations : on abrège $n_1, \ldots, n_p$ par $\vec{n}$. Avec cette notation, $U_i^p(n_0, \ldots, n_p)$ devient $U_i^p(\vec{n})$.

DÉFINITION XI.2 Soit $\mathcal{A}$ un ensemble de fonctions numériques.

$\mathcal{A}$ est *clos par composition* si pour tout φ défini par

$$\varphi(\vec{n}) = \phi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$$

avec $\phi, \psi_i \in \mathcal{A}$, alors on a aussi $\varphi \in \mathcal{A}$.

$\mathcal{A}$ est *clos par récursion primitive* si pour tout φ défini par

$$\varphi(0, \vec{n}) = \phi(\vec{n})$$
$$\varphi(x + 1, \vec{n}) = \psi(x, \varphi(x, \psi_1(\vec{n}), \ldots, \psi_p(\vec{n})))$$

avec $\phi, \psi, \psi_i \in \mathcal{A}$, alors on a aussi $\varphi \in \mathcal{A}$.

$\mathcal{A}$ est *clos par minimisation* si pour tout φ défini par

$$\varphi(\vec{n}) = \text{Min}\{x : \phi(x, \vec{n}) = 0\}$$

avec $\phi \in \mathcal{A}$ et tel que $\forall \vec{n}, \exists m, \phi(m, \vec{n}) = 0$, alors on a aussi $\varphi \in \mathcal{A}$.

DÉFINITION XI.3 La classe $\mathcal{R}$ des *fonctions arithmétiques récursives* est le plus petit ensemble contenant les fonctions numériques de base. Il est clos par composition, récursion primitive et minimisation.

XI.2.2.2 Définissabilité par amalgames

DÉFINITION XI.4 **(Représentation des entiers dans Σ)** Pour chaque $n \in \mathbb{N}$ un terme $\ulcorner n \urcorner \in \Sigma$ est défini de la manière suivante :

$$\ulcorner 0 \urcorner \equiv \{b = \mathit{vrai}^0\}$$
$$\ulcorner n + 1 \urcorner \equiv \{p = \ulcorner n \urcorner, b = \mathit{faux}^0\}$$

DÉFINITION XI.5 **(Définissabilité par amalgames)** Soit φ une fonction numérique d'arité p. On dit que φ est *définissable par amalgame*, ou encore a-définissable, si il existe un système s tel que :

$$\forall \vec{n}, \quad s[\ulcorner n_1 \urcorner, \ldots, \ulcorner n_p \urcorner] \equiv \{a1 = \ulcorner n_1 \urcorner, \ldots, ap = \ulcorner n_p \urcorner\} \centerdot s \twoheadrightarrow \{\mathit{valeur} = \ulcorner \varphi(\vec{n}) \urcorner, \ldots\}$$

Les ... dans le résultat indiquent que s doit être un système qui doit spécifier au moins une définition pour *valeur* et qui peut contenir d'autres définitions si cela est nécessaire. Les noms $a1$, $a2$... sont, par convention les noms donnés aux arguments de φ et qui ne sont pas définis par ailleurs. Le terme s peut dépendre de ces noms.

1. La définition que nous donnons diffère dans les détails de la définition usuelle mais elle est équivalente. Nous adoptons cette définition car elle est plus pratique à traduire.

XI.2.2.3 Le codage de l'arithmétique

Les fonctions arithmétiques de base sont définissables par amalgame : il suffit de prendre les termes

$$\begin{array}{lcl} \boldsymbol{U_i^p} & \equiv & \{\mathit{valeur} = a\mathrm{i}\} \\ \boldsymbol{S} & \equiv & \{\mathit{valeur} = \{p = a1, b = \mathit{faux}^0\}\} \\ \boldsymbol{Z} & \equiv & \{\mathit{valeur} = \ulcorner 0 \urcorner\} \end{array}$$

La fonction numérique P, telle que $P(n+1) = n$ est définissable par amalgame, on prend :

$$\boldsymbol{P} \equiv \{\mathit{valeur} = a1 \,.\, p\}$$

Les fonctions définissables par amalgames sont closes par composition. En effet, soient $\phi, \psi_1, \dots \psi_p$ a-définies par $G, H1, \dots, H\mathrm{m}$, alors

$$\varphi(\vec{n}) = \phi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n}))$$

est définie par :

$$\begin{array}{lrcl} \{ & \mathit{valeur} & = & \mathit{args} \,.\, G, \\ & \mathit{args} & = & \{a1 = H1 \,.\, \mathit{valeur}, \dots, a\mathrm{m} = H\mathrm{m} \,.\, \mathit{valeur}\} \quad \} \end{array}$$

Il est maintenant nécessaire de définir une forme conditionnelle : $\lambda x, y, z.$(si $x = 0$ alors y sinon z). Cela se fait grâce au champ b de la représentation d'un nombre qui nous permet d'extraire une référence libre servant d'aiguillage. Mais attention : il ne faut pas que cette référence libre soit capturée et devienne liée. Par convention, le terme $\mathrm{ifZ}_{x,y,z}$ représente

$$\begin{array}{lrcl} \{ & \mathit{codeCond} & = & x \,.\, b \\ & \mathit{codeVrai} & = & y, \\ & \mathit{codeFaux} & = & z, \\ & \mathit{valIf} & = & \mathit{aiguillage}^0 \,.\, \mathit{codeCond}_1 \quad \} \end{array}$$

sachant que *aiguillage* sera défini plus tard comme :

$$\text{Branchement} \equiv \{\mathit{vrai} = \mathit{stop}^0 \,.\, \mathit{codeVrai}_2, \mathit{faux} = \mathit{stop}^0 \,.\, \mathit{codeFaux}_2\}$$

une fois qu'on a extrait *valIf* de ifZ. La référence libre *stop* a pour but d'empêcher la liaison des références libres vrai^0et faux^0qui apparaissent dans *codeVrai* et *codeFaux*. On s'en débarrasse en fournissant une définition inoffensive à *stop*, par exemple $\mathit{stop} = \{\}$.

La méthode utilisée pour définir un schéma d'appel récursif consiste à créer une expression correspondant au terme traduisant la fonction mais ne contenant aucune référence liée et à fournir la liaison de ces références au moment de l'appel de la fonction.

Les fonctions définissables par amalgames sont closes par récursion primitive. Nous allons détailler le schéma pour une fonction φ à deux variables, en utilisant la conditionnelle et le schéma récursif : soit la fonction φ définie par $\varphi(0, y) = \phi(y)$ et $\varphi(x+1, y) = \psi(x, \varphi(x, \psi_1(y)))$ avec ϕ a-défini par F, ψ a-défini par G et ψ_1 a-défini par H. Alors φ est a-défini par (x et y sont les noms utilisés pour les arguments de φ) :

$$\begin{array}{lcl} \{\mathit{valeur} & = & \mathit{value} \,.\, \mathit{finalval} \\ \quad \mathit{value} & = & \{ \quad \mathit{finalval} = \{\mathit{stop} = \{\}\} \,.\, (\mathit{recval} \,.\, \mathit{valIf}), \\ & & \quad \mathit{fct} = \mathrm{ifZ}_{x^0,\ G[y^0],\ F[x^0, \mathit{parameter}\,.\,(\mathit{call}\,.\,\mathit{valIf})]} \\ & & \quad \mathit{recval} = \{\mathit{call} = \{X = x^1 \,.\, p, Y = H\}, \mathit{aiguillage} = \text{Branchement}\} \,. \\ & & \qquad\qquad (\{x = \ulcorner n_1 \urcorner, y = \ulcorner n_2 \urcorner, \mathit{parameter} = \{x = X, y = Y\}\} \,.\, \mathit{fct})\}\} \end{array}$$

De la façon dont nous avons présenté les choses, il doit être évident au lecteur qu'il nous importe peu de décrémenter x ou de l'incrémenter dans le schéma de la récursion primitive : cela donne lieu naturellement à l'implémentation de la minimisation.

XI.2.2.4 Exemples de calcul d'expressions arithmétiques

À cause de l'imbrication des définitions, il est nécessaire de générer un nouveau nom pour les arguments lors de chaque application de fonction, ou bien de gérer les accès par des exposants. Ceci explique les numéros arbitraires derrière les noms de variables. Voici, à titre d'exemple, la traduction de $U_1^2(S(U_1^2), S(U_2^2))(1,0)$:

$$\begin{array}{rcl} \{valeur & = & \{val21 = arg51, arg50 = \{p = arg49, b = faux^0\}, arg51 = \{p = arg48, b = faux^0\}\} \,.\, val21, \\ arg48 & = & \{p = \{b = vrai^0\}, b = faux^0\}, \\ arg49 & = & \{b = vrai^0\}\} \end{array}$$

terme dont le résultat de l'évaluation est :

$$\begin{array}{rcl} \{valeur & = & \{p = \{p = \{b = vrai^0\}, b = faux^0\}, b = faux^0\}, \\ arg48 & = & \{p = \{b = vrai^0\}, b = faux^0\}, \\ arg49 & = & \{b = vrai^0\}\} \end{array}$$

comme on s'y attendait.

La définition de l'addition prend la forme suivante :

$$\begin{array}{l} add(0,n) = n \\ add(n+1,m) = add(n, S(m)) \end{array}$$

d'où les fonctions $\phi = U_1^1, \psi = U_2^2, \psi_1 = S$. La traduction de

$$add(2,1)$$

donne le terme (les variables Tmp$x$ sont des méta-variables utilisées pour afficher le terme par morceaux, ce ne sont pas des identificateurs de ID) :

$$\begin{array}{rcl} add(2,1) \equiv \{valeur & = & \mathrm{Tmp1} \\ arg82 & = & \{p = \{p = \{b = vrai^0\}, b = faux^0\}, b = faux^0\}, \\ arg83 & = & \{p = \{b = vrai^0\}, b = faux^0\}\} \end{array}$$

$$\begin{array}{rcl} \mathrm{Tmp1} \equiv \{recval & = & \mathrm{Tmp2}, \\ finalval & = & \{stop6 = \{\}\} \,.\, recval_1, \\ fct6 & = & \{valif = aiguillage^0 \,.\, codeCond_1, \\ & & \quad codeCond = x6^0 \,.\, b^0, \\ & & \quad codeVrai = \{val36 = arg86_0, arg86 = y6^0\} \,.\, val36_0, \\ & & \quad codeFaux = \{val37 = arg88_0, arg87 = x6^0, arg88 = call^0 \,.\, retvalue^0\} \,.\, val37_0 \\ & & \quad \}\} \,.\, finalval_0 \end{array}$$

$$\begin{array}{rcl} \mathrm{Tmp2} \equiv \{call & = & \{X6 = x6^1 \,.\, p^0, Y6 = Tmp3, retvalue = param^0 \,.\, fct6^0 \,.\, valif^0\}, \\ aiguillage & = & \{vrai = stop6^0 \,.\, codevrai^1, faux = stop6^0 \,.\, codefaux^1\}\} \\ & & \qquad .\{x6 = arg82_3, y6 = arg83_3, param = \{x6 = X6^0, y6 = Y6^0\}\} \,.\, fct6_2 \,.\, valif^0 \end{array}$$

$$\begin{array}{rcl} \mathrm{Tmp3} \equiv \{val34 & = & \{comp6 = arg85_1, val35 = \{p = comp6_1, b = faux^0\}\} \,.\, val35_0, \\ arg84 & = & x6^1, \\ arg85 & = & y6^1\} \,.\, val34_0 \end{array}$$

qui s'évalue[1] en le terme attendu :

$$\begin{array}{rcl} \{valeur & = & \{p = \{p = \{p = \{b = vrai^0\}, b = faux^0\}, b = faux^0\}, b = faux^0\}, \\ a82 & = & \{p = \{p = \{b = vrai^0\}, b = faux^0\}, b = faux^0\}, \\ a83 & = & \{p = \{b = vrai^0\}, b = faux^0\}\} \end{array}$$

1. L'évaluation nécessite 37 pas d'évaluation et 450 secondes environ sur une HP 9000/770 avec Mathematica (2.2). Évidement, ce n'est pas très efficace, mais le codage de l'arithmétique dans les amalgames purs est surtout un exercice formel.

XI.3 Extension du calcul sur les amalgames aux types de bases

Les règles que nous avons données pour la sémantique opérationnelle des amalgames définissaient des règles de réduction pour les amalgames purs c'est-à-dire uniquement l'ensemble définissant le fonctionnement des références libres, liées, l'opérateur de concaténation, de sélection et le système. Pour rendre cet évaluateur utilisable, il est nécessaire d'y ajouter les constantes et les opérations arithmétiques de base ainsi que la conditionnelle. Nous n'étendrons pas d'avantage la sémantique ici, car c'est l'objet de la dernière partie de ce document que d'étudier l'intégration des amalgames et des GBF dans le langage 81/2.

XI.3.1 Les règles de réduction

Nous étendons la grammaire des expressions pour prendre en compte de nouvelles expressions. Le langage $\Sigma_{\mathcal{I}}$ devient le langage $\Sigma_{\mathcal{I}+}$ égal à : $\Sigma_{\mathcal{I}}$ augmenté de la conditionnelle $\texttt{IfThenElse}(e_{if}, e_{then}, e_{else})$, des opérations binaires $+, -, *, /$ et des entiers $\mathbb{N}$. Les sous-langages Σ et $\Sigma_{\mathcal{C}}$ s'étendent naturellement aux nouvelles constructions ainsi que les fonctions précédemment définies.

XI.3.1.1 Réduction d'une opération binaire

$$binop_{Cte} \quad : \quad \frac{}{\rho \vdash c\, binop\, c' \longrightarrow c''} \quad \begin{cases} (c \in \mathbb{N}) \wedge (c' \in \mathbb{N}) \\ c'' = c\, binop\, c' \end{cases} \tag{2}$$

$$binop \quad : \quad \frac{\rho \vdash u \longrightarrow_n u' \quad \rho \vdash v \longrightarrow_m v'}{\rho \vdash u\, binop\, v \longrightarrow u'\, binop\, v'} \quad 1 \leq n+m \leq 2 \tag{3}$$

RÈG. 11: *Réduction d'une opération binaire entre deux constantes.*

La règle (2) définit la valeur d'une opération binaire entre deux constantes. La règle (3) spécifie la valeur d'une opération binaire entre deux *expressions*. Ces deux expressions doivent être réduites pour pouvoir réellement effectuer l'opération. Nous utilisons ici le même mécanisme de macro-expansion que nous avons décrit pour les précédentes règles de la sémantique. En effet, la règle sur les opérateurs binaires doit s'appliquer si au moins une des deux expressions u ou v ne sont pas des constantes. Plutôt que de spécifier un ordre de réduction (de la gauche vers la droite par exemple), nous spécifions uniquement que cette règle peut s'appliquer si une des réductions en prémisse peut s'exécuter.

XI.3.1.2 Réduction de la conditionnelle

$$\mathrm{Cond}_d \quad : \quad \frac{\rho \vdash e \longrightarrow e'}{\rho \vdash \texttt{IfThenElse}(e,u,v) \longrightarrow \texttt{IfThenElse}(e',u,v)} \quad e' \notin \textsc{Bool} \tag{4}$$

$$\mathrm{Cond}_t \quad : \quad \frac{\rho \vdash e \longrightarrow \texttt{true}}{\rho \vdash \texttt{IfThenElse}(e,u,v) \longrightarrow u} \quad e \in \textsc{Bool} \tag{5}$$

$$\mathrm{Cond}_f \quad : \quad \frac{\rho \vdash e \longrightarrow \texttt{false}}{\rho \vdash \texttt{IfThenElse}(e,u,v) \longrightarrow v} \quad e \in \textsc{Bool} \tag{6}$$

RÈG. 12: *Réduction de la conditionnelle.*

Le traitement de la conditionnelle par les règles (4), (5) et (6) est classique : la condition est évaluée jusqu'à ce que la condition soit un élément de BOOL. Tant que la condition est une expression, il n'est pas souhaitable d'évaluer les branches vraies ou fausses : seule une des deux branches doit être effectivement évaluée, celle qui est sélectionnée par la condition. Une fois la condition devenue un élément de BOOL, alors la valeur de la conditionnelle est la valeur de la branche correspondante.

Les opérations que nous avons définies précédemment s'étendent naturellement au traitement de la conditionnelle. On considère la conditionnelle comme un opérateur ternaire classique. On définit les extensions sur $\Sigma_{\mathcal{I}+}$ des fonctions définies sur les termes de Σ :

$$\begin{array}{lcl}
\mathtt{correct_B}(l, \mathtt{IfThenElse}(u,v,w)) & = & \mathtt{correct_B}(l,u) \wedge \mathtt{correct_B}(l,v) \wedge \mathtt{correct_B}(l,w) \\
\mathtt{correct_F}(l, \mathtt{IfThenElse}(u,v,w)) & = & \mathtt{correct_F}(l,u) \wedge \mathtt{correct_F}(l,v) \wedge \mathtt{correct_F}(l,w) \\
\mathtt{bind}(l, \mathtt{IfThenElse}(u,v,w)) & \equiv & \mathtt{IfThenElse}(\mathtt{bind}(l,u), \mathtt{bind}(l,v), \mathtt{bind}(l,w)) \\
\mathtt{IdToRef}(\mathtt{IfThenElse}(u,v,w)) & \equiv & \mathtt{IfThenElse}(\mathtt{IdToRef}(u), \mathtt{IdToRef}(v), \mathtt{IdToRef}(w)) \\
\mathtt{Lift}(n, \mathtt{IfThenElse}(u,v,w)) & \equiv & \mathtt{IfThenElse}(\mathtt{Lift}(n,u), \mathtt{Lift}(n,v), \mathtt{Lift}(n,w)) \\
\mathtt{C}(l, \mathtt{IfThenElse}(u,v,w)) & = & \mathtt{C}(l,u) \wedge \mathtt{C}(l,v) \wedge \mathtt{C}(l,w)
\end{array}$$

XI.3.2 Calcul d'une fonction récursive

Le calcul des fonctions récursives primitives dans le formalisme des amalgames utilise la conditionnelle que nous avons introduit au début de cette section. Un mécanisme de génération récursive d'arbre avec une décoration de l'arbre lors de sa construction permet la définition de fonctions récursives. Une expression qui est « gelée » est reproduite avec un compteur qui est décrémenté à chaque réplication. Quand la valeur du compteur est nulle, alors on fournit une définition qui « dégèle » l'expression et permet la réduction effective de l'arbre généré. Le schéma général de récursion est explicité par l'expression suivante :

$$\begin{array}{l}
\{parameter = \{x = x^1 - 1\}, \\
\ fct = \mathrm{If}(x^0 \leq 0, \mathrm{C}, F(x^0, arg^0 \centerdot fct^0)), \\
\ value = \{arg = parameter_1\} \centerdot (\{x = 4\} \centerdot fct_2)\}
\end{array}$$

Un germe (*parameter*) définit une constante (x) qui est égale à sa valeur dans le scope englobant, moins un. Le definiendum fct définit une expression qui est égale à la constante C si la constante x du scope courant est égale à zéro, sinon qui est égale à $arg^0 \centerdot fct^0$. Enfin, nous avons un système *value* qui définit une évaluation de fct dans un environnement où arg est lié à *parameter* et qui fournit une valeur à x. Ces définitions correspondent à une équation récursive dans la mesure où dans fct, arg^0 est une référence libre qui peut se lier à l'expression $arg = parameter_1$ ainsi que x et fct. La génération de la structure récursive cesse après x « dépliages » (ici 4) du germe. Par exemple, l'évaluation de l'expression ci-dessus a pour résultat :

$$\begin{array}{l}
\{parameter = \{x = x^1 - 1\}, \\
\ fct = \mathrm{If}(x^0 \leq 0, \mathrm{C}, F(x^0, arg^0 \centerdot fct^0)), \\
\ value = F(4, F(3, F(2, F(1, \mathrm{C}))))\}
\end{array}$$

où $F(a,b)$ représente une fonction quelconque de paramètre a et b.

L'utilisation de ce schéma général de récursion permet le calcul de fonctions récursives classiques telles que factorielle, fibonacci... Nous donnons à titre d'exemple les expressions correspondant au calcul de la fonction factorielle :

$$\begin{array}{l}
\{parameter = \{x = x^1 + -1\}, \\
\ fct = \mathrm{If}(x^0 \leq 0, 1, (arg^0 \centerdot fct^0)\, x^0), \\
\ value = \{arg = parameter_1\} \centerdot \{x = V\} \centerdot fct_2\}
\end{array}$$

et fibonacci :

$$\begin{array}{l}
\{parameter = \{x = x^1 + -1\}, \\
\ fct = \mathrm{If}(x^0 \leq 1, 1, arg^0 \centerdot fct^0 + arg^0 \centerdot (arg^0 \centerdot fct^0)), \\
\ value = \{arg = parameter_1\} \centerdot \{x = V\} \centerdot fct_2\}
\end{array}$$

où V représente la donnée du calcul. La figure 1 représente le détail du calcul de la fonction factorielle où V à la valeur 6, et qui correspond donc au calcul de 6!, obtenu en 28 étapes de réduction. Les définitions de *parameter* et fct ne changeant pas, on les a par conséquent remplacées par $\mathcal{U}$ et $\mathcal{V}$ pour des raisons de concision.

$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . \mathrm{If}(x_0 \leq 0, 1, arg_1 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . arg_1 . fct^0\ x_0\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = x_1 + -1\} . fct_3\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . \mathrm{If}(x_0 \leq 0, 1, arg_2 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . arg_2 . fct^0\ x_0\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = x_1 + -1\} . fct_4\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . \mathrm{If}(x_0 \leq 0, 1, arg_3 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . arg_3 . fct^0\ x_0\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = x_1 + -1\} . fct_5\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . \mathrm{If}(x_0 \leq 0, 1, arg_4 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . arg_4 . fct^0\ x_0\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = x_1 + -1\} . fct_6\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . \mathrm{If}(x_0 \leq 0, 1, arg_5 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . arg_5 . fct^0\ x_0\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = x_1 + -1\} . fct_7\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = 1\} . \mathrm{If}(x_0 \leq 0, 1, arg_6 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = 1\} . arg_6 . fct^0\ x_0\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = 1\} . \{x = x_1 + -1\} . fct_8\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = 1\} . \{x = 0\} . \mathrm{If}(x_0 \leq 0, 1, arg_7 . fct^0\ x_0)\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = 1\} . \{x = 0\} . 1\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\,\{x = 1\} . 1\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 3\,\{x = 2\} . 2\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 4\,\{x = 3\} . 6\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 5\,\{x = 4\} . 24\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 6\,\{x = 5\} . 120\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . \{x = 6\} . 720\}$$
$$\{\mathcal{U}, \mathcal{V}, value = \{arg = \{x = x^1 + -1\}\} . 720\}$$
$$\{\mathcal{U}, \mathcal{V}, value = 720\}$$

FIG. 1 –: *Le calcul complet de* 6! *en 28 étapes.* $\mathcal{U}$ *et* $\mathcal{V}$ *représentent respectivement les définitions de parameter et de* fct *qui sont constantes.*

XI.3.3 Exemple de simulation créé par programme : croissance cellulaire d'une racine

L'exemple de cette section n'a pas été choisi pour sa pertinence biologique, mais pour montrer sur un exemple simple la construction calculée d'un programme. C'est un exemple de système dont l'espace des états doit être calculé conjointement à l'évolution du système.

L'exemple correspond à la croissance et au fonctionnement d'une racine. On suppose qu'une racine est un ensemble de cellules organisées hiérarchiquement en arbre (au sens informatique du terme). L'activité d'une cellule c est modélisée par une variable dont la valeur dépend des cellules filles.

Il y a trois type de cellules : les cellules *terminales* qui sont « feuilles de la racine », les cellules de *branchement* qui ont deux cellules filles, et les cellules normales qui ont une seule cellule fille. Seules les cellules terminales sont capables de croissance : à chaque pas de temps, une cellule terminale se transforme en une cellule de branchement ou en une cellule normale.

La simulation consiste à construire à chaque pas de temps l'arbre (informatique) qui représente la racine, et à calculer simultanément la valeur qui caractérise chaque cellule.

Le programme correspondant est écrit entièrement dans le langage des amalgames, sans faire appel à la notion de stream [1]. On émule donc l'évolution de la racine dans le temps en la construisant comme on construit l'arbre des appels de *factorielle* :

$$
\begin{array}{lcl}
\{ & & \\
\quad \mathit{Cell} & = & \{sval = \mathit{Cval1},\, hval = \mathit{Cval3}\}, \\
\quad \mathit{BranchCell} & = & \{sval = \mathit{Cval2},\, hval = \mathit{Cval4}\}, \\
\quad \mathit{EndCell} & = & \{sval = 1,\, hval = \mathit{Cval3}\}, \\
& & \\
\quad \mathit{param} & = & \{x = x^1 - 1\}, \\
\quad \mathit{Fct} & = & \texttt{if}\ x \leq 0\ \texttt{then}\ ec
\end{array}
$$

1. Il est évident que dans le cadre d'une véritable simulation, on utiliserait l'aspect stream de 81/2, ce qui simplifierait la simulation (dans ce cas, il n'y aurait pas besoin de gérer la récursion). Cependant, nous n'abordons l'intégration des amalgames dans 81/2 que dans la partie suivante. Se contenter des amalgames pour coder y compris l'aspect temporel de la simulation, permet de présenter un exemple auto-consistant.

$$\texttt{else(if } Random() \texttt{ then } c \# \{next = arg \,.\, Fct\}$$
$$\texttt{else } cb \# \{nextLeft = arg \,.\, Fct,\ nextRight = arg \,.\, Fct\})$$

$$Thread \quad = \quad \{arg = param,\ ec = EndCell,\ c = Cell,\ cb = BranchCell\} \,.\, (\{x = 3\} \,.\, Fct)$$

$$\begin{aligned} RealThread \quad = \quad \{ & Cval1 = 1 + next \,.\, sval, \\ & Cval2 = 1 + nextLeft \,.\, sval + nextRight \,.\, sval, \\ & Cval3 = hval^1, \\ & Cval4 = hval^1/2, \\ & loop = val\} \,.\, (Thread \# \{hval = loop\}) \end{aligned}$$

}

La traduction en amalgame associe un système incomplet à chaque type de cellule. Ce système incomplet décrit l'activité de la cellule, mais les références vers les filles de la cellules (f, fl, fr) restent libres. On calcule l'arbre de la racine avec la variable *Thread*. Cet arbre étant obtenu, on fournit les définitions nécessaires pour instancier le calcul de la valeur de chaque cellule (à savoir, on définit $Cval1, Cval2, Cval3, Cval4$) : c'est l'équation associée à *RealThread*. x représente le nombre de pas d'évolution désiré. Voici un exemple de résultat (nous ne donnons que l'expression de *RealThread*) :

```
RealThread = {sval = 11
              hval = 11/2
              nextLeft = {sval = 7
                          hval = 11/4
                          nextLeft = {sval = 3
                                      hval = 11/8
                                      nextLeft = {sval = 1, hval = 11/8}
                                      nextRight = {sval = 1, hval = 11/8}
                                     }
                          nextRight = {sval = 3
                                       hval = 11/8
                                       nextLeft = {sval = 1, hval = 11/8}
                                       nextRight = {sval = 1, hval = 11/8}
                                      }
                         }
              nextRight = {sval = 3
                           hval = 11/2
                           next = {sval = 2
                                   hval = 11/2
                                   next = {sval = 1, hval = 11/2}
                                  }
                          }
              loop = 11
             }
```

Cet exemple montre comment on peut construire un espace arbitraire : l'arbre n'est pas un arbre binaire régulier, mais le hasard intervient dans sa construction. Quand la racine est construite, il faut « la faire fonctionner» ce qui correspond dans notre exemple à propager l'attribut *sval* des cellules terminales à la racine de l'arbre (par exemple, cet attribut peut représenter la valeur énergétique de la sève) et à faire redescendre l'attribut *hval*. Ce modèle, bien que primaire, permet par exemple de faire dépendre la croissance de la valeur des attributs (synthétisés ou hérités) et correspond donc à un mécaniste plausible de redistribution des ressources dans la plante [1].

XI.3.4 Structuration de code et programmation orientée objets

Les notions de système et de concaténation de systèmes permettent la manipulation d'environnements. La composition de systèmes permet de programmer des enregistrements extensibles. De plus, les valeurs symboliques et la capacité à fournir des définitions manquantes vont nous permettre de définir un comportement similaire à ceux que l'on trouve dans les langages orientés objets. Il est possible de définir et composer des fragments de programmes en exhibant un comportement de *classe* et d'*instanciation de classe* que l'on

1. C. Godin, communication personnelle, (CIRAD) janvier 1996.

trouve dans ces langages. Nous allons montrer à travers un exemple comment émuler un style programmation proche de ceux des langages à objets.

Un système représente les notions de *classe* et de *constructeur* qui sont utilisées pour créer une instance d'une classe. Les arguments nécessaires au constructeur sont les références libres du système. L'*instanciation* d'une classe correspond à la concaténation du système avec les arguments requis par le constructeur. L'ajout de définitions supplémentaires, par l'utilisation de la concaténation, correspond au mécanisme d'*héritage*.

Un système complet (sans références libres) correspond à un objet comme c'est le cas dans les langages à objets. Mais pour ce qui concerne 81/2, les valeurs de l'objet sont des tissus : il n'existe pas de notion de message ni de méthodes. Le modèle objet qui correspond le mieux à l'esprit de ce qui est faisable en 81/2 est le modèle « *embedding based* » où toutes les informations qui concernent l'objet se trouvent dans celui-ci. Il est évident que les mécanismes évolués de protection et d'encapsulation qui sont proposés par les langages à objets sont absents dans le modèle que nous proposons.

Pour illustrer ce style de programmation, on se propose de définir en suivant un style proche des langages orientés objets, une modélisation de la trajectoire d'une planète dans un mouvement circulaire uniforme autour d'une étoile elle-même en mouvement rectiligne uniforme. Pour ce faire, on définit une classe $Mobile$ d'objets animés d'un mouvement. La classe $Mobile$ est représentée par un système avec deux références libres :

- *initiale* qui représente la position initiale de l'objet,
- *dp* qui représente les déplacements élémentaires de l'objet.

À l'aide de ces références libres, qui sont des vecteurs de deux éléments correspondant aux axes Ox et Oy, le système $Mobile$ définit une position :

$$Mobile \quad = \quad \{position@0 = initiale; position = \$Mobile\ .\ position + dp\};$$

L'attribut *position* de $Mobile$ est un stream qui représente la trajectoire du mobile au cours du temps.

Une fois que $Mobile$ est défini, nous pouvons définir une nouvelle classe d'objets : les objets mobiles avec une vitesse uniforme. La classe $U\text{–}Trajectoire$ attend une position *initiale* (requise par la classe $Mobile$ dont il hérite) et un vecteur de vitesse pour pouvoir s'instancier :

$$U\text{–}Trajectoire \quad = \quad Mobile \# \{dp = vitesse\}$$

Le système $U\text{–}Trajectoire$ est un système avec toutes les définitions du système $Mobile$ car c'est un système étendu par les définitions de *dp* utilisé pour calculer les déplacements élémentaires avec une trajectoire uniforme (on suppose que *vitesse* est une constante et est égale à la différence de la trajectoire uniforme entre deux éléments consécutifs du stream). L'opération de concaténation rassemble ces deux systèmes et effectue les liaisons entre la référence libre *dp* qui apparaît dans $Mobile$ et la définition de *dp* dans le système anonyme complétant les définitions de $U\text{–}Trajectoire$.

On poursuit avec cet exemple en utilisant $Mobile$ pour représenter la trajectoire circulaire d'une planète en révolution autour d'une étoile (le soleil par exemple) qui se déplace suivant une trajectoire uniforme. La classe $C\text{–}Trajectoire$ attend un rayon, un centre pouvant être animé d'un mouvement quelconque, et une vitesse angulaire :

$$\begin{array}{lllll}
C\text{–}Trajectoire & = & Mobile \# \{ & initiale & = & \{centre\ .\ 0, angle + centre\ .\ 1\}, \\
 & & & dp & = & \{dx, dy\}, \\
 & & & ot & = & \$t, \\
 & & & t@0 & = & 0.0, \\
 & & & t & = & ot + vitesse_angulaire, \\
 & & & dx@0 & = & -angle * 2.0 * \cos(t) * \sin(t), \\
 & & & dx & = & -angle * 2.0 * \cos((t + ot)/2.0) * \sin((t - ot)/2.0) + \\
 & & & & & centre\ .\ 0 - \$centre\ .\ 0, \\
 & & & dy@0 & = & -angle * 2.0 * \sin(t) * \cos(t), \\
 & & & dy & = & -angle * 2.0 * \sin((t + ot)/2.0) * \cos((t - ot)/2.0) + \\
 & & & & & centre\ .\ 1 - \$center\ .\ 1\}
\end{array}$$

Il ne nous reste plus qu'à instancier les classes pour décrire le « système solaire » :

$$\begin{array}{lcl} Etoile & = & U\text{–}Trajectoire\,\#\,\{vitesse = \{1.0, 1.0\}, initiale = \{0.0, 0.0\}\}; \\ Planete & = & C\text{–}Trajectoire\,\#\,\{angle = 1.0, centre = Etoile\,.\,position\} \end{array}$$

La figure 2 illustre la trajectoire résultante de la planète, une trajectoire circulaire se déplaçant autour d'un centre, l'étoile, qui elle, est animée d'une translation uniforme.

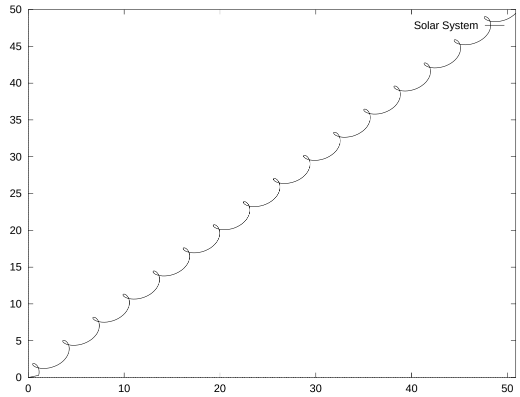


FIG. 2 –: *Simulation d'un mouvement circulaire uniforme autour d'un centre lui-même en mouvement rectiligne uniforme.*

Quatrième partie

$8_{1/2}{}_{\mathcal{D}}$: GBF et Amalgames dans $8_{1/2}$

Chapitre XII

Intégration des GBF et des amalgames dans $8_{1/2}$: l'interprète $8_{1/2}{}_{\mathcal{D}}$

Dans la deuxième partie de ce document, nous avons conçu et développé une nouvelle structure de données et des mécanismes déclaratifs de définition, pour la représentation des espaces homogènes : les *GBF*.

Nous avons défini et étudié dans la troisième partie de ce document une nouvelle structure de données et des mécanismes déclaratifs de définition, pour la représentation des espaces hétérogènes : les *amalgames*.

Dans cette quatrième partie, il est à présent temps d'étudier l'intégration de ces deux structures de données dans le cadre du langage $8_{1/2}$. Cette intégration est à l'origine d'une nouvelle sémantique formelle pour $8_{1/2}$, qui permet de manipuler les GBF et les amalgames comme de nouveaux types de collections. Cette nouvelle sémantique, appelée $\mathbf{8_{1/2}}{}_{\mathcal{D}}$, est spécifiée à l'aide de règles de sémantique opérationnelle et s'incarne dans un interprète en cours de développement, implémenté en CAML.

Structure de la partie. Cette quatrième partie est constituée de trois chapitres. Le premier chapitre expose les principes de l'intégration des GBF et des amalgames dans $8_{1/2}$. L'objectif est d'étendre la notion de tissu afin de disposer de streams de GBF et de streams d'amalgames. Cela n'est pas possible dans le cadre de l'ancienne sémantique des tissus à comportement statique, sémantique qui a été spécifiquement orienté pour permettre la compilation des programmes $8_{1/2}$ sur des architectures parallèles. Cela motive donc le développement d'une nouvelle sémantique qui permet d'interpréter les nouveaux programmes : cette nouvelle sémantique s'appelle $8_{1/2}{}_{\mathcal{D}}$.

Le second chapitre (le chapitre XIII) décrit la sémantique formelle du processus d'évaluation $8_{1/2}{}_{\mathcal{D}}$. Nous ne prouvons pas formellement que la nouvelle sémantique est une extension de la sémantique dénotationnelle proposée dans [Gia91a], mais nous montrons sur des exemples que $8_{1/2}{}_{\mathcal{D}}$ calcule bien la même horloge que la sémantique dénotationnelle du compilateur $8_{1/2}$.

Enfin, le troisième chapitre (le chapitre XIV) donne des exemples de programme $8_{1/2}{}_{\mathcal{D}}$ pris dans trois domaines d'application très différents : des structures de données dynamiques pour la combinatoire, des exemples de calcul symbolique et la modélisation de processus de croissance. Ces exemples complètent les exemples illustrant les amalgames et les GBF, et montrent l'intégration de ces deux nouveaux concepts dans $8_{1/2}{}_{\mathcal{D}}$. Tous les exemples du chapitre ont été évalués par l'interprète $8_{1/2}{}_{\mathcal{D}}$ dans sa version courante.

Structure du chapitre. Ce chapitre se décompose en trois sections. La première section introduit les principes d'intégration des GBF et des amalgames dans $8_{1/2}$. Le principe de base consiste à *découpler* le traitement des streams et des collections et d'introduire ces nouvelles structures de données comme des nouveaux types de collections.

La deuxième section traite de l'intégration des GBF en tant que nouveau type de collection. Nous n'avons pas eu le temps de traiter complètement l'intégration des GBF : nous nous sommes donc restreints à un sous-ensemble de ceux-ci, correspondant grossièrement à des tableaux dont la taille est dynamique.

La troisième section de ce chapitre traite de l'intégration des amalgames comme nouveau type de collection. Cette intégration nécessite un nouveau schéma d'évaluation qui correspond à un *interprète*.

XII.1 Principes de l'intégration des GBF et des amalgames dans 81/2

Un programme 81/2 est un système d'équations qui définissent des tissus. Un tissu est une structure de données qui combine les streams et les collections. Nous avons mentionné à la section II.4 qu'un tissu est une structure de données qui peut être vue soit comme un stream de collections, soit comme une collection de streams.

Afin d'intégrer les GBF et les amalgames dans 81/2, nous allons renoncer à ce point de vue et définir un *tissu comme un stream de collections*. Les collections permises peuvent être de deux types : GBF ou amalgame.

Le stream de GBF étend et généralise la notion de tissu homogène. Le stream d'amalgames étend et généralise la notion de système 81/2.

Nous verrons que l'introduction des GBF comme un type de collection requiert une gestion dynamique de la mémoire, puisque le support de la collection (les éléments qui ont une valeur définie) peut varier d'un tic à l'autre. En 81/2, le calcul du support de la collection correspond à l'inférence de la géométrie à la compilation. Dorénavant, il est nécessaire de calculer la géométrie d'un tissu à chaque tic.

Par ailleurs, l'introduction des amalgames comme un type de collection nécessite qu'un programme, correspondant à l'arbre syntaxique d'une expression, puisse être une valeur de premier ordre. Cette valeur est calculée comme le résultat d'une opération sur les amalgames, et une fois close, cette valeur peut s'évaluer comme une programme 81/2 classique.

La conséquence immédiate de l'intégration des amalgames dans un langage compilé est que le code généré par un compilateur doit inclure un *évaluateur*. Cela est rendu nécessaire par la phase dynamique d'évaluation qui intervient dans le traitement des amalgames. L'exécution du code aura un comportement *hybride* nécessitant des phases d'exécution de code compilé entrecoupées de phases d'interprétation de termes (non compilés). Cela ne constitue pas un problème car, en adoptant le point de vue des langages réflexifs [Smi84], on peut estimer que l'exécution d'un programme correspond, en réalité, à une évaluation dans une tour réflexive où chaque niveau évalue le niveau supérieur.

Nous sommes extrêmement confiants dans cette approche de compilation hybride [CP97, Ple96] où la séparation entre phase de compilation et phase d'exécution n'est plus aussi nette que dans les langages compilés classiques (Pascal, C). On trouvera un exemple d'approche similaire d'exécution hybride dans [BL84b, pp 290] où la séparation entre exécution et évaluation commence à être gommée.

Cependant, cette approche nécessite une refonte trop grande du schéma de compilation actuel. En effet, la sémantique dénotationnelle définie dans [Gia91a] restreint les programmes 81/2 traités à ceux qui exhibent un *comportement statique*.

Aussi, pour l'intégration des GBF et des amalgames dans 81/2, nous préférons développer une nouvelle sémantique, $81/2_{\mathcal{D}}$, qui correspond à une *approche totalement interprétée* du processus d'évaluation.

Avant de passer aux détails de l'intégration des GBF et des amalgames comme nouveaux types de collection, nous allons expliquer pourquoi le schéma statique de compilation correspondant à [Gia91a] est inadapté. Un programme a un comportement statique quand les paramètres de l'exécution sont connus dès la compilation. Par exemple, il est nécessaire, en 81/2 statique, de connaître, au moment de la compilation, l'espace mémoire qu'occupera une collection, cet espace mémoire étant fixe durant l'exécution. Autre exemple, l'ordonnancement des calculs d'un programme 81/2 doit être automatiquement inféré au moment de la compilation du programme. Cela implique le rejet de certains programmes ayant un cycle dans le graphe des dépendances *syntaxiques*, mais pas dans le graphe des dépendances *sémantiques*. Par exemple, le programme suivant :

$$
\begin{aligned}
b &= \ldots && (1)\\
c &= \ldots && (2)\\
x &= \mathtt{if}\, b\, \mathtt{then}\, y\, \mathtt{else}\, c && (3)\\
y &= \mathtt{if}\, \neg b\, \mathtt{then}\, x\, \mathtt{else}\, c && (4)
\end{aligned}
$$

définit quatre équations b, c, x et y exhibant une dépendance circulaire entre x et y, à travers les deux opérateurs `if`. Si on regarde plus précisément le graphe des dépendances des expressions (cf. figure 1) il apparaît qu'il est toujours possible de résoudre les dépendances entre les expressions x et y. En effet, la dépendance de x vers y apparaît à travers la branche vraie de la définition de x alors que la dépendance de y vers x apparaît à travers la branche vraie de la définition de y. Les gardes b de x et $\neg b$ de y assurent qu'il n'existe aucune dépendance entre x et y. Cette résolution a nécessité une connaissance de la sémantique de l'opérateur `if` alors qu'un ordonnancement statique nécessite une absence de dépendance entre *les deux branches* de la conditionnelle.

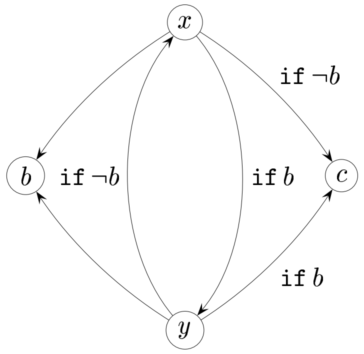


FIG. 1 –: *Graphe des dépendances des équations (1) à (4).*

Ce type de contrainte est nécessaire si on veut déterminer à la compilation la distribution des données [Mah96] et les schémas de communication qui en résultent [DG93, GDC94] sur une architecture parallèle à mémoire distribuée.

Cependant, ce type de contrainte n'est plus acceptable quand il s'agit de développer des structures de données de très haut-niveau : on est prêt à payer le prix d'une gestion dynamique des données quand elle seule peut répondre aux besoins cruciaux en expressivité.

XII.2 Intégration des GBF dans 81/2

Le temps nous a manqué pour traiter complètement l'intégration du calcul sur les GBF dans 81/2 ; c'est pourquoi nous nous sommes restreint à l'intégration d'une classe très particulière de GBF. Les restrictions sur les GBF utilisables sont les suivantes :

– le GBF doit être un GBF *abélien libre* ;

– le support du GBF (les éléments ayant une valeur différente de `nil`) doit être une région décrite par un élément de $[0 \ldots n_1] \times \cdots \times [0 \ldots n_d]$;

– les valeurs d'un GBF doivent être de type simple : entier, flottant... Ce ne peut être ni des imbrications de GBF, ni des amalgames.

Dans la suite, nous appellerons *tableau dynamique* ou plus simplement *tableau* cette restriction des GBF. Ce nom se justifie car à chaque tic de l'horloge, un tel GBF correspond à un tableau, mais ce tableau peut avoir une géométrie différente à chaque tic (rappelons que la géométrie d'un tableau est le nombre de dimensions et le nombre d'éléments dans chaque dimension du tableau).

Cette classe de GBF est suffisamment générale pour démontrer la faisabilité de l'intégration des GBF quelconques à valeur simple. En effet :

- Pour traiter les supports arbitraires bornés dans un GBF abélien libre, il suffit de gérer des unions de tableaux à chaque tic, plutôt qu'un tableau unique.

  L'extension est simple dans son principe, mais la gestion efficace de grandes unions de régions hypercubiques de $\mathbb{Z}^n$, nécessaire quand on veut manipuler des régions complexes, est délicate. Une étude détaillée est en cours [Sem93, Vit96].

- Le traitement des GBF abéliens non-libres se décompose par l'isomorphisme fondamental en un traitement sur un groupe abélien libre (et on est renvoyé au problème précédent) et aux modules de torsion. Il « suffit » donc d'étendre les calculs d'index nécessaires à la gestion des GBF abéliens libres par un calcul d'index *modulo* le rang du module de torsion.

  Les algorithmes en algèbres modulaires sont bien connus [NQ92, Coh93] et cette extension ne devrait donc pas poser de problème de principe.

- Si on peut intégrer les GBF abéliens comme un nouveau type de collection, alors l'intégration de familles de GBF quelconques dans $8_{1/2\mathcal{D}}$ ne pose pas de problèmes supplémentaires.

Ces restrictions sont motivées par le fait qu'il est ainsi possible, à peu de frais, de réutiliser la machine virtuelle existante, et que dans un premier temps, il n'est pas nécessaire d'augmenter beaucoup la syntaxe du langage.

XII.2.1 Nouveaux opérateurs pour la manipulation des tableaux dynamiques

Nous ne permettons pas toutes les expressions sur les GBF décrites dans la section V.5 pour deux raisons :

- Le résultat de certaines opérations sur des tableaux dynamiques n'est pas un tableau dynamique. C'est le cas par exemple pour l'union de GBF, ou la translation.
- Le temps nous a manqué pour implémenter l'ensemble des opérations définies.

Nous avons donc choisi de nous restreindre à un sous-ensemble significatif et facilement implémentable au-dessus de la machine virtuelle existante, ou ne nécessitant pas de grosses extensions de cette machine virtuelle. Nous allons présenter ces opérations.

XII.2.1.1 Opérateur $\#d>$: union et translation d'un GBF dynamique

L'opération de concaténation, « $\#_n$ », défini dans le langage $8_{1/2}$ (cf. section II.3.3.5, page 15) permet d'effectuer la concaténation de deux collections suivant la n^{e} dimension. Comme nous l'avons vu dans la seconde partie (cf. section V.5.2, page 58), l'opération de concaténation ainsi définie ne correspond plus à la concaténation des GBF. L'opération définie sur les GBF est de plus bas niveau.

Les programmes que l'on désire implémenter nécessitent cependant l'utilisation d'une concaténation, au sens des tableaux. On définit par conséquent l'opérateur $\#d>$ de $8_{1/2\mathcal{D}}$, correspondant à la concaténation de GBF. Cet opérateur effectue les deux opérations d'union et de translation de GBF, où d correspond à la dimension suivant laquelle la translation doit avoir lieu. Si le support de A est $[n_1, \ldots, n_d]$ alors :

$$A\#p> B \quad \equiv \quad A \# B.g_p^{n_p}$$

où g_i est le i^{e} générateur de la forme de A.

Pour des raisons de simplicité, dans le reste du document, on établit les correspondances suivantes :

$$\begin{array}{lcl} \#1> & \equiv & \# \\ \#2> & \equiv & \#\hat{} \end{array}$$

et on utilisera les seuls symboles « $\#$ » et « $\#\hat{}$ ».

XII.2.1.2 Un opérateur conditionnel sans α-extension

La définition de tableaux dynamiques nécessite de pouvoir comparer deux tableaux de taille différente. L'unique opérateur conditionnel existant en 81/2 est la conditionnelle qui correspond à la version data-parallèle de l'opérateur `if`. Une expression de la forme $s \equiv \texttt{if}\, c\, \texttt{then}\, t\, \texttt{else}\, f$ impose aux trois tableaux c, t et f d'avoir une géométrie et un type scalaire identique. Les éléments du tableau s correspondent aux éléments de t pour les éléments de c vrais et ceux de f pour les éléments de c faux.

Cet opérateur n'a plus de sens quand ses arguments sont des tableaux de tailles différentes. Nous proposons la définition d'un nouvel opérateur $81/2_{\mathcal{D}}$, l'opérateur `switch` qui permet la sélection du tableau t ou du tableau f en fonction de la condition *scalaire* c.

XII.2.1.3 Coupure et extension d'un tableau dynamique

L'opération de coupure-extension d'une collection suivant une géométrie particulière est essentiellement utilisée lors de la définition d'expressions impliquant un mécanisme de récursion spatiale (cf. section III.4, page 24). Cependant, cet opérateur implique un argument *constant*. La définition de tableaux dont les dimensions varient au cours du temps, nécessitent la définition d'un opérateur qui permette une expression dynamique comme argument de la coupure-extension.

Nous étendons la spécification de l'opérateur `e:[c]` de 81/2 (`c` étant une liste de constantes entières) en permettant l'utilisation d'un argument résultant d'un calcul et dynamique pour la spécification de la géométrie d'une collection. L'expression `e:f` permet de coercer la géométrie de `e` par le vecteur `f` qui spécifie le rang du résultat (i.e. le nombre d'éléments dans chaque dimension). Par exemple :

$$1 : \{1,2,3\} \Rightarrow \left\{ \left\{ \left\{ \begin{array}{c} 1 \\ 1 \\ 1 \end{array} \right\}, \left\{ \begin{array}{c} 1 \\ 1 \\ 1 \end{array} \right\} \right\} \right\}$$

(f est égal à $\{1,2,3\}$). Cet opérateur doit s'interpréter comme une version spécialisée de l'opérateur de restriction explicite introduit dans la section V.5 :

$$e : \{n_1, \ldots, n_d\} \quad \equiv \quad e\, \texttt{on}\, g_1^{i_1}.\ldots.g_d^{i_d}\, \texttt{where}\, 0 \leq i_1 < n_1, \ldots, 0 \leq i_d < n_d$$

Par ailleurs, nous permettons la définition d'un tableau par énumération de ses éléments (c'est une version bridée de la définition quantifiée de GBF) :

$$\{\ldots, e_i, \ldots\}$$

correspond à un tableau dont la valeur du i^{e} élément est e_i, etc.

XII.2.1.4 Rang d'un tableau dynamique

Nous avons présenté dans la section précédente l'opérateur de coercion d'une collection. Nous aimerions aussi être capable de déterminer la taille d'une collection afin de pouvoir effectuer des opérations sur celle-ci en fonction de sa taille (cf. section XIV.4.3, page 219). Nous définissons par conséquent l'opérateur $|e|$, dont la valeur est un vecteur d'entiers qui correspond au rang (cf. section II.3.2, page 12) de l'expression e.

XII.2.2 Réutilisation de la machine virtuelle de gestion des tableaux

Nous disposons déjà d'une machine virtuelle permettant de calculer sur des tableaux. En effet, le compilateur statique du langage est capable de générer un code cible `C` ou bien un code pour une machine virtuelle de manipulation de tableaux.

Cette machine virtuelle est de type SIMD : elle permet la manipulation de tableaux comme des touts à travers un ensemble restreint d'instructions élémentaires portant sur des tableaux. Les instructions se divisent en deux classes :

1. les instructions d'allocation de tableaux,

2. les instructions de calculs qui sont de type *code trois adresses*.

Cette machine est aisément portable sur une architecture matérielle vectorielle ou SIMD. Une version écrite en C, qui traite les éléments des tableaux séquentiellement, existe et est utilisable à partir un autre langage (comme par exemple le langage CAML), à travers une interface fonctionnelle. L'implémentation repose sur une représentation par une imbrication de vecteurs.

Bien que la détermination de la dimension des tableaux ait lieu au moment de la compilation, les instructions d'allocation de tableaux de la machine virtuelle peuvent être utilisées à tout moment : on peut donc faire de l'allocation dynamique. L'extension de cette machine virtuelle pour traiter quelques opérateurs supplémentaires nécessaires sur les tableaux dynamiques (comme la restriction) s'est révélée suffisament simple.

Par conséquent, nous réutilisons dans un premier temps cette machine virtuelle existante pour le développement d'un interprète (d'un sous-ensemble de) $8_{1/2\mathcal{D}}$, avant de concevoir et de développer une nouvelle machine virtuelle capable de traiter l'ensemble des GBF abéliens.

XII.3 Intégration des amalgames dans $8_{1/2}$

Pour intégrer les amalgames dans $8_{1/2}$, nous les considérons comme un nouveau type de collection : un amalgame est donc une valeur d'un type nouveau. Mais un amalgame est aussi un fragment de programme $8_{1/2}$. Il se pose donc un premier problème qui est celui de la notation des constantes de type amalgames. Nous verrons ensuite quelle horloge il faut associer à une constante de type amalgame.

XII.3.1 Dénotation d'un amalgame comme valeur immédiate

Un amalgame est à la fois une valeur et un fragment de programme $8_{1/2}$. Cela pose un problème : dans le texte d'un programme, comment distinguer la dénotation immédiate d'une constante de type amalgame et une expression à évaluer ?

Par exemple, 1234 est la dénotation immédiate d'un entier et $1234 + 5678$ est un programme qu'il faut calculer. La distinction entre ces deux types d'objet est simple car la syntaxe permet de les distinguer. Par contre, quand la constante qu'il faut dénoter peut s'interpréter aussi comme un programme à évaluer, il y a ambiguïté. Par exemple :

$$\{a = 1 + 2, b = x^0\}$$

représente t-il une constante du langage (de type amalgame) ou bien un programme dont l'évaluation associe 3 à a et un amalgame x^0 à b ?

Dans un langage comme Lisp[1] où les valeurs peuvent aussi être des programmes, ce problème est résolu en introduisant un mécanisme explicite de *quotation*. Par exemple :

(setq s ‘(cons ,(+ 1 2) l))

affecte à s une valeur de type *s-expression* qui représente un fragment de programme Lisp. L'opérateur de quotation « ‘ » indique que la liste qui suit ne fait pas partie du programme à évaluer mais doit être considérée comme une constante. La construction d'une constante représentant une s-expression fait souvent intervenir des valeurs calculées et donc on rajoute un mécanisme de « déquotation » : c'est le but de la virgule avant l'expression (+ 1 2). La s-expression construite correspond au programme Lisp qui concatène la valeur 3 au symbole l.

Contrairement à Lisp, nous ne voulons pas introduire de mécanisme de quotation explicite, tout au moins dans un premier temps. En effet, l'étude de l'évaluation des expressions qui impliquent des amalgames montre qu'on peut reposer sur des règles implicites simples pour décider de l'amalgame dénoté. Par exemple, l'expression :

$$1 + x^0$$

1. on a en tête ici un Lisp dynamique, comme par exemple Frantz Lisp.

dénote obligatoirement un amalgame. En effet, puisqu'on n'a pas de définition pour la référence x^0, on ne peut pas évaluer $1 + x^0$. Par suite, le programme :

$$\{a = 1 + x^0\}$$

est un système qui associe une constante de type amalgame à a, de la même façon exactement que :

$$\{a = 1\}$$

est un système qui associe une constante entière à a. Par contre, que faut-il penser de l'expression :

$$(1 + 2) + x^0$$

Il est clair que cette expression dénote une constante de type amalgame, car on ne peut pas l'évaluer comme une constante arithmétique, puisqu'on ignore la définition de x. Mais on pourrait évaluer $(1 + 2)$ avant de construire la constante immédiate. En explicitant les quotations à la Lisp, il faut décider si $(1+2)+x^0$ dénote $`((1 + 2) + x^0)$ ou bien $`(,(1 + 2) + x^0)$.

L'évaluation d'une expression mixte qui combine amalgame et tableau peut se faire en suivant deux stratégies :

- La réduction des expressions impliquant des tableaux a lieu *au plus tard* : on attend qu'une expression composée devienne une expression n'impliquant que des tableaux pour effectuer les opérations sur les tableaux. Dans ce cas, l'expression que nous venons de voir n'effectuera l'opération $(1 + 2)$ qu'une fois la référence libre x substituée par une expression de tableaux. Si ce n'est jamais le cas, alors cette opération n'aura jamais lieu. On parle de stratégie *tardive*.
- La réduction des expressions impliquant des tableaux a lieu *au plus tôt* : dès qu'une opération implique deux tableaux, alors la réduction de la sous-expression a lieu. Pour l'expression ci-dessus, l'amalgame dénoté est $3 + x^0$ Cette stratégie ne présuppose rien quant à la référence libre x. On qualifie cette stratégie de *hâtive*.

La stratégie hâtive est choisie afin d'effectuer *au plus tôt* les calculs, ce qui permet de *simplifier* les expressions le plus rapidement possible, et d'obtenir la valeur la plus *complète* pour une expression qui implique des références libres. Cette évaluation au plus tôt permet la définition de fragments de programmes *partiellement évalués*, alors la première stratégie *délègue* l'évaluation à l'exécution finale, celle qui fournit toutes les définitions aux références libres.

Remarquons que contrairement à Lisp et aux langages impératifs, une référence liée ne peut être *liée qu'à une seule définition*. Par suite, il importe peu d'évaluer hâtivement ou tardivement une expression : le calcul résultera toujours en la même valeur. La seule différence est le moment de l'évaluation. Par exemple, il importe peu que :

$$\{a = 1, b = a_0 + x^0\}$$

dénote :

$$\{a = 1, b = `(a_0 + x^0)\}$$

ou bien :

$$\{a = 1, b = 1 + `x^0\}$$

puisque la référence à a dans l'expression de définition de b est liée «pour toujours» à la même définition. Ceci justifie le choix d'une approche implicite de la dénotation d'un amalgame.

D'une façon similaire, on peut passer en revue le traitement de toutes les constructions du langage, y compris celles portant sur l'aspect stream d'un tissu. Par exemple, pour le traitement de l'opérateur d'échantillonnage :

$$1 \texttt{ when } b^0$$

correspond à ‘(1 `when` b^0). En effet, nous savons que le calcul de l'horloge d'une expression u `when` v nécessite le calcul de l'horloge de v. La référence b^0 étant libre, on ne peut pas calculer de valeur associée et donc 1 `when` b^0 doit être compris comme la notation d'un amalgame (c'est-à-dire une valeur d'un type spécial) et non comme un fragment de programme à évaluer. Par contre, l'expression

b^0 `when Clock`

est une expression évaluable et sa valeur est l'amalgame b^0.

Les règles permettant de désambigüer la dénotation immédiate d'un amalgame sont spécifiées par le prédicat `Code` () qui prend la valeur vrai si l'expression argument n'est pas un amalgame :

$$
\begin{array}{lcl}
\texttt{Code}(\{\ldots\}) & = & \texttt{C}(\{\ldots\}) \\
\texttt{Code}(e_1 \# e_2) & = & \texttt{Code}(e_1) \wedge \texttt{Code}(e_2) \\
\texttt{Code}(e_1 \texttt{ . } e_2) & = & \texttt{Code}(e_1) \wedge (e_2 \in \textsc{Id}) \\
\texttt{Code}(\$e) & = & \texttt{Code}(e) \\
\texttt{Code}(e_1 \texttt{ when } e_2) & = & \texttt{Code}(e_2) \\
\texttt{Code}(e_1 + e_2) & = & \texttt{Code}(e_1) \wedge \texttt{Code}(e_2) \\
 & \ldots &
\end{array}
$$

XII.3.2 Horloge d'une constante amalgame

De la même façon que la constante scalaire 1 dénote un stream d'entier, une constante de type collection dénote un stream de collection.

La question de la valeur de l'horloge d'une constante amalgame se pose donc. On peut décider d'attribuer à un amalgame une horloge identique à celle qui est définie pour les constantes arithmétiques : le domaine de 1 est défini pour tous les tics et seul le premier tic est un top.

Cette solution n'est cependant pas satisfaisante. En effet, dans ce cas une expression d'amalgame ne sera jamais réévaluée après le premier tic : une expression dépendant d'un amalgame ne peut alors pas progresser.

On choisit par conséquent d'attribuer aux amalgames la même horloge que la constante `Clock`, dont le domaine de définition est vrai pour tous les tics et dont tous les tics sont des tops.

Cette horloge permet d'écrire simplement des constantes de type amalgame, sans faire intervenir un mécanisme de quotation particulier, tout en s'assurant de la progression des streams impliquant des amalgames.

L'approche que nous avons choisi est implicite et simple. Il se peut qu'elle se révèle à l'usage trop grossière en empêchant de dénoter très précisément l'amalgame désiré et de contrôler finement les instants d'évaluation. Il faudra alors introduire un mécanisme de quotation afin de permettre l'expression explicite d'une constante amalgame et spécifier un calcul d'horloge sur les amalgames permettant de minimiser les évaluations à effectuer.

Chapitre XIII

Une sémantique pour $8_{1/2}{}_{\mathcal{D}}$

Dans ce chapitre nous définissons une sémantique formelle du processus d'évaluation $8_{1/2}{}_{\mathcal{D}}$. Nous ne prouvons pas formellement que la nouvelle sémantique est une extension de la sémantique dénotationnelle proposée dans [Gia91a], mais nous montrons sur des exemples caractéristiques que $8_{1/2}{}_{\mathcal{D}}$ calcule bien la même horloge que la sémantique dénotationnelle du compilateur $8_{1/2}$.

XIII.1 Syntaxe des expressions

Nous nous intéressons à un sous-ensemble de $8_{1/2}{}_{\mathcal{D}}$, le langage $\mathcal{L}$. La syntaxe abstraite de $\mathcal{L}$ est donnée par :

EXP	$::=$	c	les constantes, de type tableau ou amalgame
	$\mid$	id	les identificateurs,
	$\mid$	$id = e$	la définition de tissu,
	$\mid$	e . id	la sélection par un identificateur,
	$\mid$	e . n	la sélection par un entier,
	$\mid$	$e_1 \# e_2$	la concaténation de deux expressions (amalgame ou tableaux),
	$\mid$	$\{\dots, e, \dots\}$	la définition de tableaux,
	$\mid$	$\{\dots, id = e, \dots\}$	la définition de systèmes,
	$\mid$	e_1 *binop* e_2	les opérations binaires arithmétiques et logiques,
	$\mid$	*unop*(e)	les opérations unaires,
	$\mid$	`if` e_1 `then` e_2 `else` e_3	l'opération conditionnelle,
	$\mid$	`Clock` n	l'horloge,
	$\mid$	$\$e$	le délai,
	$\mid$	e_1 `fby` e_2	la transcription des quantifications,
	$\mid$	e_1 `when` e_2	l'échantillonnage,
	$\mid$	$e : [c]$	la coercion spatiale.

avec $\mathbb{N}$ le domaine des entiers ($n \in \mathbb{N}$), ID le domaine des identificateurs de tissus ($id \in$ ID), UNOP le domaine des opérateurs monadiques du langage ($unop \in$ UNOP) et BINOP le domaine des opérateurs dyadiques du langage ($binop \in$ BINOP).

Comme le décrit la syntaxe des expressions de $\mathcal{L}$, tous les opérateurs de $8_{1/2}$ définis dans [Gia91a] ne seront pas décrits dans ce chapitre. En effet, ceux-ci n'ont pas un traitement particulier justifiant leur description dans ce document. Par exemple, les opérateurs temporels `until`, `after`... peuvent être traduits grâce à l'utilisation de l'opérateur d'échantillonnage `when`. Nous n'expliciterons pas non plus le traitement des opérations géométriques de `scan` et de β-réduction dans la mesure où leur calcul est immédiat et n'est pas essentiel à la définition de la notion de réduction des expressions.

XIII.1.1 Traduction des quantifications temporelles

Nous avons vu dans le premier chapitre (cf. section II.2.3.3, page 11) que la spécification de la valeur d'un tissu à des instants distincts se faisait à l'aide d'équations quantifiées. Pour des raisons de simplification des expressions, on transforme une expression quantifiée en une expression utilisant l'opérateur `fby`, opérateur similaire à celui de même nom défini dans Lucid. Par exemple, l'expression :

$$\begin{array}{lcl} T@0 & = & 0; \\ T & = & \$T + 2; \end{array}$$

est transformée en son équivalent dans $\mathcal{L}$:

$$T \quad = \quad 0\,\texttt{fby}\,\$T + 2$$

Une expression sans quantification ne change pas ; la traduction d'équations qui ont plus d'un quantificateur, ou un quantificateur autre que @0, nécessite une opération de transformation plus complexe. Pour des raisons de simplicité, on restreint donc $\mathcal{L}$ aux seules expressions ayant *au plus* une quantification de la forme @0 en plus de la quantification universelle.

XIII.1.2 Traduction des quantifications spatiales

La définition des GBF permet une quantification suivant un coset (cf. section V.6.2, page 62). Pour des raisons de simplicité, on ne permet pas les quantifications spatiales en $81/2_{\mathcal{D}}$. Autrement dit, on ne traite pas les GBF récursifs.

XIII.2 Principes de la sémantique

La sémantique d'une expression est définie par trois grandeurs qui correspondent aux :

1. calcul de l'horloge d'un tissu (spécifié par deux relations de réduction, cf. section suivante),
2. calcul de la géométrie pour déterminer le type d'une expression,
3. calcul de la valeur d'une expression de $\mathcal{L}$.

Ces grandeurs sont calculées pour chaque tic t ($t \in \mathbb{N}$) de l'exécution du programme. Le calcul de l'horloge, la géométrie et la valeur d'une expression e dépendent du tic courant ainsi que des sous-expressions de e.

L'environnement de définition $\mathcal{E}_{\mathrm{E}}$ est une fonction partielle dont la signature est $\mathrm{I}\textsc{d} \to \mathcal{L}$. Notons $[id \mapsto e]$ la fonction qui associe l'expression e à l'identificateur id et qui est indéfinie pour tous les autres identificateurs. Si ρ est un environnement, alors $\rho' = \rho \uplus [id' \mapsto e']$ et $\rho'' = \rho \setminus id'$ sont les environnements tels que :

$$\rho'(id) = \begin{cases} e' & \text{si } id = id' \\ \rho(id) & \text{sinon} \end{cases}$$

et

$$\rho''(id) = \begin{cases} \rho(id) & \text{si } id \neq id' \\ \text{indéfini sinon} & \end{cases}$$

L'écriture $[id_1 \mapsto e_1, \ldots, id_n \mapsto e_n]$ est une abréviation de $[id_1 \mapsto e_1] \uplus \cdots \uplus [id_n \mapsto e_n]$. Les opérateurs « $\uplus$ » et « $\setminus$ » représentent respectivement l'augmentation et l'appauvrissement d'un environnement. La fonction `Dom()` appliquée à un environnement a pour valeur l'ensemble des identificateurs auxquels sont associés une définition. Par exemple :

$$\texttt{Dom}([id_1 \mapsto \ldots, id_2 \mapsto \ldots, id_2 \mapsto \ldots]) \quad = \quad \{id_1, id_2, id_3\}$$

XIII.2.1 Quatre relations de réduction pour calculer la valeur d'un tissu

Quatre relations de réduction sur une expression e à un instant t dans un environnement ρ sont définies pour calculer la valeur de e. Ces quatre relations de réduction permettent de calculer les informations nécessaires au calcul de la valeur, soit :

1. le calcul du dom d'une expression,
2. le calcul du top d'une expression,
3. le calcul de la géométrie d'une expression,
4. le calcul de la valeur d'une expression.

Les calcul du dom et du top d'une expression permettent le calcul de l'horloge d'une expression. Nous verrons dans la section suivante les raisons de la séparation de l'horloge d'une expression en deux calculs. Le calcul de la géométrie d'une expression permet de déterminer ses dimensions (si c'est un tableau). Enfin, on peut en calculer la valeur effective instantanée à partir du calcul de son horloge et de sa géométrie.

XIII.2.2 Séparation de l'horloge d'une expression en deux calculs

L'horloge d'un stream est un prédicat temporel qui assure que la valeur du stream doit être re-calculée à l'instant courant. Soit $hor(e)$ la fonction qui définit l'horloge du stream e. La fonction $hor(e)$ est une fonction qui à un tic t, a pour valeur un booléen. En première approximation, on peut dire que $hor(e)(t)$ est vraie si la valeur de e change à l'instant t, sinon elle est fausse.

Dans les définitions des horloges de stream que l'on trouve dans [Pla88, Jen95], la définition de l'horloge d'un stream est faiblement couplée avec la valeur du stream dans le sens suivant : un tic peut être dans l'horloge d'un stream alors que la valeur courante du stream est indéfinie. L'exemple le plus simple d'un tel stream est l'expression retardée $\$e$ qui partage la même horloge avec e mais dont la valeur est indéfinie au premier tic de $hor(e)$. Nous aimerions au contraire avoir la propriété suivante :

$$hor(e)(t) \quad \Rightarrow \quad Val(e)(t) \neq \texttt{nil} \tag{1}$$

($Val(e)(t)$ désigne la valeur associée à e à l'instant t) mais cette propriété ne peut pas être obtenue immédiatement. Regardons l'exemple suivant. Soit le stream défini par :

$$\begin{aligned} A@0 &= 0 && (2) \\ A &= (\$A+1)\,\texttt{when}\,\texttt{Clock} && (3) \end{aligned}$$

qui devrait correspondre à un compteur dont la valeur est incrémentée à chaque tic. Or ce n'est pas le cas si on suppose la propriété (1) vraie. Si on suppose cette propriété vraie, les équations (2) et (3) définissent alors un stream identique à la constante 0. En effet, l'équation (2) force le stream A à avoir pour valeur 0 au premier top de la constante 0 (c'est-à-dire au tic 0). La valeur de $\$A$ est indéfinie au tic 0. Si on assume la propriété (1) ou plutôt sa contraposée :

$$Val(e)(t) = \texttt{nil} \quad \Rightarrow \quad \neg hor(e)(t)$$

il est nécessaire que l'horloge de $\$A$ soit fausse au tic 0. Par conséquence nous avons $hor(\$A)(0) = \texttt{false}$. Par ailleurs, l'horloge de $M\,\texttt{when}\,\texttt{Clock}$ est vraie au tic t, à partir du moment où M possède une valeur définie ; si ce n'est pas le cas, on viole la propriété (1). Or M a une valeur définie s'il a déjà eu un top au tic t. Par conséquence, l'horloge de A, pour $t > 0$, est définie par :

$$\begin{aligned} hor(A)(t) &= OK(\$A+1, t) \qquad t > 0 \\ &= OK(\$A, t) \end{aligned}$$

où $OK(e,t)$ est un prédicat qui est vrai si e a déjà un top à un tic $\leq t$. Or l'horloge de A et de $\$A$ coïncident pour tous les tics ultérieurs au premier top. Nous avons donc aussi :

$$hor(\$A)(t) \quad = \quad OK(\$A, t) \qquad t > 0$$

Cette équation admet plusieurs solutions. Par exemple, $OK(\$A, t) =$ `false` pour tout t est une solution valide, de même que $OK(\$A, t) =$ `true` pour $t > 0$. Il est usuel, quand il est nécessaire de choisir une solution parmi un ensemble d'éléments ordonnés, de prendre la plus petite solution. La relation d'ordre sur les suites prise habituellement est la relation d'ordre *préfixe*. La solution «naturelle» ici correspond à :

$h_0 =$ `true`
$h_n =$ `false`

Le lecteur désirant trouver un développement plus formel à ce problème se référera à [Gia91a, DV96].

L'«effondrement» de l'horloge est dû à la confusion de deux prédicats : «avoir une valeur définie au tic t» et «changer (possiblement) de valeur au tic t». Pour éviter l'effondrement des horloges lors de la définition récursive de streams, nous séparons l'horloge d'un stream A en deux prédicats, *dom*() et *top*() ; nous donnons une interprétation intuitive de ces deux prédicats :

- le prédicat $dom(A)$, que l'on appelle par abus de langage le *domaine* de A, indique quand la valeur de A est accessible (i.e. différente de `nil`),
- le prédicat $top(A)$, que l'on appelle par abus de langage l'*horloge* de A, indique quand il est nécessaire d'effectuer des calculs pour maintenir les relations définies entre les expressions.

On trouvera dans [Gia91a, pp 76,77] une discussion plus détaillée sur cette problématique.

XIII.3 Les domaines sémantiques

XIII.3.1 La réduction $\xrightarrow{\mathcal{D}}$

Comme nous l'avons vu, le calcul de l'horloge d'une expression est effectué par le calcul de deux prédicats. Ces prédicats sont définis sous la forme de règles de réduction. La règle de réduction du domaine d'une expression à pour domaine $\mathbb{N} \times \mathcal{E}_{\mathrm{E}} \times \mathcal{L} \mapsto \textsc{Bool} \cup (\textsc{Id} \to \textsc{Bool})$ et se note $t, \rho \vdash e \xrightarrow{\mathcal{D}} dom$ pour exprimer que, au tic t, dans l'environnement ρ, le dom de e est égal à dom.

XIII.3.2 La réduction $\xrightarrow{\mathcal{T}}$

D'une façon similaire, le domaine de la règle de réduction définie pour calculer l'horloge d'une expression est $\mathbb{N} \times \mathcal{E}_{\mathrm{E}} \times \mathcal{L} \mapsto \textsc{Bool} \cup (\textsc{Id} \to \textsc{Bool})$ et se note $t, \rho \vdash e \xrightarrow{\mathcal{T}} top$ pour exprimer que, au tic t, dans l'environnement ρ, l'horloge de e est égale à top.

XIII.3.3 La réduction $\xrightarrow{\mathcal{G}}$

Le typage de la géométrie des tissus permet de vérifier des propriétés sur la géométrie des tissus, par exemple avoir pour valeur un système ou un tableau, et de calculer la dimension des collections. La géométrie d'un tissu dépend de la géométrie de ses sous-expressions. Nous associons aux expressions de $\mathcal{L}$ un type, qui peut être :

- *indéfini*, c'est le cas de la géométrie d'un tissu avant son premier top vrai. Nous faisons la différence entre un calcul qui ne termine pas et un calcul qui n'a pas de valeur définie, typiquement un tissu qui n'a jamais de top. Cette géométrie est notée `UndefGEO`.
- *un tableau* : dans ce cas on spécifie la géométrie du tissu par un type scalaire (`int`, `bool` ou `float`) qui correspond au type scalaire de ses éléments, ainsi que par le rang du tableau (représenté par une liste d'entiers).
- *un système* : la géométrie associée à un système est dans ce cas une fonction qui associe à chaque identificateur la géométrie de l'expression qu'il réfère.

– *un amalgame* : la géométrie d'un amalgame est égale à `AmalGeo`.

En conséquence, le domaine de la géométrie d'un tissu est :

$$\textsc{Geo} = \{\texttt{UndefGEO}, \texttt{AmalGeo}\} \cup \{\texttt{int}, \texttt{bool}, \dots\} \times \textit{int list} \cup (\textsc{Id} \to \textsc{Geo})$$

Le calcul de la géométrie d'une expression est effectué par la notion de réduction : $t, \rho \vdash e \xrightarrow{\mathcal{G}} g$ pour exprimer que e a pour géométrie g au tic t dans l'environnement ρ.

XIII.3.4 La réduction $\xrightarrow{\mathcal{V}}$

La valeur d'une expression dépend de la valeur de ses sous-expressions. La valeur d'une expression de $\mathcal{L}$ peut être :

– *indéfinie* : c'est le cas de l'expression $\{x = \$x\}$ qui n'a jamais de top. On note la valeur indéfinie `Undef`.

– *un tableau* : la valeur est dans ce cas une imbrication de vecteurs qui correspond à la valeur de ses éléments.

– *un système* : la valeur associée est une fonction qui, à chaque identificateur, associe la valeur de sa définition.

– *un amalgame* : la valeur associée est une expression de $\mathcal{L}$.

Les tableaux sont représentés par des imbrications de vecteurs homogènes. Si on note S le domaine des valeurs scalaires, alors le domaine des tableaux est : $Array(S)$. Le domaine des valeurs est :

$$\textsc{Val} = \{\texttt{Undef}\} \cup Array(S) \cup (\textsc{Id} \to \textsc{Val}) \cup \mathcal{L}$$

Nous ferons référence, dans la spécification de la sémantique, à une relation de réduction « $\xrightarrow{\mathcal{A}}$ ». Cette relation de réduction n'est pas spécifiée dans ce chapitre mais correspond à la relation que nous avons définie dans la troisième partie (cf. section X.3.2, page 136). Par conséquent, on identifie :

$$t, \rho \vdash e \xrightarrow{\mathcal{A}} e' \quad \equiv \quad \rho \vdash e \to e'$$

dans la mesure où l'aspect temporel des tissus (matérialisé par t, qui représente le tic courant) n'est pas important dans le cadre de la réduction des amalgames, la perte de l'information du tic courant n'est pas significatif. La structure de l'environnement ρ n'est pas exactement la même dans les deux relations de réduction, mais on peut aisément convertir l'environnement dans les deux domaines. En effet, afin de simplifier la présentation, on ne traite pas des opérateurs d'échappement id^n et on suppose qu'il n'y a pas de redéfinition d'un identificateur. Par ailleurs, on augmente le calcul sur les amalgames par une règle permettant de propager le calcul à travers des opérateurs étrangers (comme le délai, l'échantilloneur) :

$$\frac{\rho \vdash a \to a' \quad \rho \vdash b \to b' \quad \dots}{\rho \vdash \texttt{op}(a, b, \dots) \to \texttt{op}(a', b', \dots)}$$

XIII.4 Conventions

Dans ce qui suit, afin d'alléger les notations, nous adoptons les conventions suivantes :

– $t \in \mathbb{N}$ désigne le *tic* courant ;

– les variables *dom* et *top*, éléments de Bool, désignent le domaine et l'horloge d'un tissu ;

– les variables e, g, ts, v désignent respectivement un terme de $\mathcal{L}$, une géométrie, un type scalaire (élément de $\{\texttt{int}, \texttt{bool}, \dots\}$) et une valeur (élément de Val) ; *id* désigne un identificateur ;

- ρ désigne un environnement, i.e. une fonction de $\text{ID} \to \mathcal{L}$;
- `[|` $v_1; v_2, v_3$ `|]` est un élément de VAL, un vecteur de trois éléments ;
- *unop* et *binop* représentent respectivement des opérateurs arithmétiques unaires et binaires de $8_{1/2\mathcal{D}}$;
- $ite(cond, v_{true}, v_{false})$ est une fonction ayant pour résultat v_{true} si la condition *cond* est vraie, v_{false} si elle est fausse ; on note « == » le prédicat d'égalité entre deux entiers ; l'opérateur « % » représente le reste de la division entière.

XIII.5 Les règles de la sémantique opérationnelle de $8_{1/2\mathcal{D}}$

XIII.5.1 Le prédicat auxiliaire $OK(t, \rho, e)$

Nous définissons le prédicat $OK(t, \rho, e)$ qui est vrai, si, dans un environnement ρ à un tic t, l'expression e a déjà eu un tic qui était un top. La fonction $OK(t, \rho, e)$ est définie récursivement et par cas :

1. au tic 0, si le tic est un top alors le prédicat est vrai, et faux sinon,
2. si à un tic t ($t > 0$), le tic est un top, alors le prédicat est vrai ; si le tic n'est pas un top alors la valeur est égale à $OK(t-1, \rho, e)$, c'est-à-dire la valeur est vraie si e à eu un top au tic $t-1$.

$$OK_0 \quad : \qquad \frac{0, \rho \vdash e \xrightarrow{\mathcal{T}} top}{OK(0, \rho, e) = top}$$

$$OK_t \quad : \quad \frac{t, \rho \vdash e \xrightarrow{\mathcal{T}} \texttt{true}}{OK(t, \rho, e) = \texttt{true}} \quad t \neq 0 \qquad \frac{t, \rho \vdash e \xrightarrow{\mathcal{T}} \texttt{false} \quad OK(t-1, \rho, e) = p}{OK(t, \rho, e) = p} \quad t \neq 0$$

RÈG. 13: *Calcul de* $OK(t, \rho, e)$

XIII.5.2 Les règles par défaut

Lors de la définition de chaque règle, nous omettrons toujours les règles qui correspondent aux cas « par défaut ». Par exemple, pour ce qui concerne le calcul de la valeur (c'est aussi vrai pour le calcul de la géométrie), les règles de réduction ne doivent être appliquées que si le tissu doit être re-calculé, c'est-à-dire si le tic courant est un top pour le tissu. Cette condition est exprimée par la garde $t, \rho \vdash M \xrightarrow{\mathcal{T}} \texttt{true}$, qui apparaît dans la réduction de tous les termes de $\mathcal{L}$, où M est le terme qui est décrit dans la règle. Si le tic n'est pas un top, la valeur du tissu est celle du tissu au précédent top vrai, `Undef` si le tissu n'a jamais eu de top (nous utilisons à cet effet le prédicat $OK(t, \rho, e)$ que l'on vient de décrire).

Les règles par défaut ne s'appliquent pas pour le calcul du délai $\$e$. La traitement du délai est particulier comme nous le verrons (voir la règle (18)) . Par conséquent, pour les règles suivantes, le terme M représente une expression quelconque de $\mathcal{L}$ qui n'est pas un délai (expression différente de $\$e$).

La règle $\neg dom \Rightarrow \neg top$ indique qu'une expression ne peut pas avoir de top quand son domaine n'est pas vrai. La règle $Défaut_g$ indique que la géométrie d'une expression qui n'a pas de top à l'instant 0 est `UndefGEO`. Dans le cas contraire, si à un instant $t > 0$ l'horloge est fausse, alors la géométrie est égale à la géométrie de l'expression au précédent top. Il en est de même pour la valeur.

$$\neg dom \Rightarrow \neg top \quad : \quad \frac{t,\rho \vdash M \xrightarrow{\mathcal{D}} \texttt{false}}{t,\rho \vdash M \xrightarrow{\mathcal{T}} \texttt{false}}$$

$$D\acute{e}faut_g \quad : \quad \frac{0,\rho \vdash M \xrightarrow{\mathcal{T}} \texttt{false}}{0,\rho \vdash M \xrightarrow{\mathcal{G}} \texttt{UndefGEO}} \qquad \frac{t,\rho \vdash M \xrightarrow{\mathcal{T}} \texttt{false} \quad t-1,\rho \vdash M \xrightarrow{\mathcal{G}} g}{t,\rho \vdash M \xrightarrow{\mathcal{G}} g} \quad t \neq 0$$

$$D\acute{e}faut_v \quad : \quad \frac{0,\rho \vdash M \xrightarrow{\mathcal{T}} \texttt{false}}{0,\rho \vdash M \xrightarrow{\mathcal{V}} \texttt{Undef}} \qquad \frac{t,\rho \vdash M \xrightarrow{\mathcal{T}} \texttt{false} \quad t-1,\rho \vdash M \xrightarrow{\mathcal{V}} v}{t,\rho \vdash M \xrightarrow{\mathcal{V}} v} \quad t \neq 0$$

RÈG. 14: *Les règles par défaut*

XIII.5.3 Les constantes

XIII.5.3.1 Les constantes arithmétiques

$$Cte_d \quad : \quad \frac{}{t,\rho \vdash c \xrightarrow{\mathcal{D}} \texttt{true}} \quad c \in \{\texttt{true}, \texttt{false}, 1, 2, \dots\}$$

$$Cte_t \quad : \quad \frac{t,\rho \vdash c \xrightarrow{\mathcal{D}} \texttt{true}}{t,\rho \vdash c \xrightarrow{\mathcal{T}} (t == 0)} \quad c \in \{\texttt{true}, \texttt{false}, 1, 2, \dots\}$$

$$Cte_g \quad : \quad \frac{t,\rho \vdash c \xrightarrow{\mathcal{T}} \texttt{true}}{t,\rho \vdash \texttt{true} \xrightarrow{\mathcal{G}} \texttt{bool}\ [1]} \qquad \frac{t,\rho \vdash c \xrightarrow{\mathcal{T}} \texttt{true}}{t,\rho \vdash \texttt{false} \xrightarrow{\mathcal{G}} \texttt{bool}\ [1]} \qquad \frac{t,\rho \vdash c \xrightarrow{\mathcal{T}} \texttt{true}}{t,\rho \vdash c \xrightarrow{\mathcal{G}} \texttt{int}\ [1]}$$

$$Cte_v \quad : \quad \frac{t,\rho \vdash c \xrightarrow{\mathcal{T}} \texttt{true}}{t,\rho \vdash c \xrightarrow{\mathcal{V}} \texttt{[| c |]}} \quad c \in \{\texttt{true}, \texttt{false}, 1, 2, \dots\}$$

RÈG. 15: *Les constantes arithmétiques*

Dans $\mathcal{L}$, deux classes de constantes sont définies : les constantes booléennes, `true` et `false`, et les constantes numériques, entière et flottante.

Une constante possède un domaine qui est toujours vrai et possède un top uniquement au premier tic. La géométrie est de type scalaire entier pour les constantes numériques et de type booléen pour les constantes booléennes. La valeur d'une constante est le vecteur dont l'unique élément est égal à la constante.

REMARQUE Pour la spécification de la règle «Cte_v», on note `c` la valeur sémantique associée à la constante c.

XIII.5.3.2 Le tissu `Clock n`

Le tissu *Clock* n est un tissu de booléens toujours vrai. Son domaine est toujours vrai (sauf pour le cas $n = 0$ où le domaine n'est alors jamais défini) ; l'horloge est vraie tous les n tics ; sa géométrie est un vecteur de booléen de taille 1 et sa valeur est un booléen égal à `true`.

$$
\begin{array}{lcc}
Clock\ n_d & : & \dfrac{}{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{D}} (n \neq 0)} \\[2ex]
Clock\ n_t & : & \dfrac{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{D}} \mathtt{true}}{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{T}} ((t\%n) == 0)} \\[2ex]
Clock\ n_g & : & \dfrac{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{T}} \mathtt{true}}{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{G}} \mathtt{bool}\ [1]} \\[2ex]
Clock\ n_v & : & \dfrac{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{T}} \mathtt{true}}{t,\rho \vdash \mathtt{Clock}\,n \xrightarrow{\mathcal{V}} \mathtt{[|}\ \mathrm{true}\ \mathtt{|]}}
\end{array}
$$

RÈG. 16: *Le tissu* `Clock n`

XIII.5.4 Les identificateurs

$$
\begin{array}{lccc}
Id_d & : & \dfrac{id \notin \mathtt{Dom}(\rho)}{t,\rho \vdash id \xrightarrow{\mathcal{D}} \mathtt{false}} & \dfrac{t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{D}} d}{t,\rho \vdash id \xrightarrow{\mathcal{D}} d} \\[2ex]
Id_t & : & \dfrac{id \notin \mathtt{Dom}(\rho)}{t,\rho \vdash id \xrightarrow{\mathcal{T}} \mathtt{false}} & \dfrac{t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{D}} \mathtt{true} \quad t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{T}} top}{t,\rho \vdash id \xrightarrow{\mathcal{T}} top} \\[2ex]
Id_g & : & \dfrac{id \notin \mathtt{Dom}(\rho)}{t,\rho \vdash id \xrightarrow{\mathcal{G}} \mathtt{UndefGEO}} & \dfrac{t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{T}} \mathtt{true} \quad t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{G}} g}{t,\rho \vdash id \xrightarrow{\mathcal{G}} g} \\[2ex]
Id_v & : & \dfrac{id \notin \mathtt{Dom}(\rho)}{t,\rho \vdash id \xrightarrow{\mathcal{V}} \mathtt{Undef}} & \dfrac{t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{T}} \mathtt{true} \quad t,\rho \setminus id \vdash \rho(id) \xrightarrow{\mathcal{V}} v}{t,\rho \vdash id \xrightarrow{\mathcal{V}} v}
\end{array}
$$

RÈG. 17: *Les identificateurs*

Le calcul du domaine, de l'horloge, de la géométrie et de la valeur d'un identificateur correspond au calcul du domaine, de l'horloge, de la géométrie et de la valeur de l'expression associée, dans l'environnement qui ne contient plus la liaison associée à l'identificateur. En effet, il est nécessaire de retirer la liaison d'un identificateur pour gérer correctement les expressions récursives (cf. section XIII.6.3). La valeur d'un identificateur id dépend de l'appartenance de id à l'environnement ρ :

- si id appartient à ρ, alors sa valeur correspond à la valeur de l'expression associée ;
- si id n'appartient pas à ρ, alors l'horloge est fausse, la géométrie est égale à `UndefGEO` et la valeur est égale à `Undef`.

XIII.5.5 Les opérateurs temporels

XIII.5.5.1 L'opérateur de délai

Le délai est l'opérateur qui permet le *décalage* d'un stream dans le temps : il permet l'accès à la valeur d'un tissu au précédent top. Le domaine de $\$e$ est faux au premier tic et égal, à un tic t, au domaine de e au tic $t-1$. Nous avons dit précédemment que l'horloge d'un délai $\$e$ était la même que celle de l'expression e sauf pour le premier top. Ce n'est pas tout à fait exact.

$$Delay_d \;:\quad \frac{}{0, \rho \vdash \$e \xrightarrow{\mathcal{D}} \texttt{false}} \qquad \frac{t-1, \rho \vdash e \xrightarrow{\mathcal{D}} d}{t, \rho \vdash \$e \xrightarrow{\mathcal{D}} d}\; t \neq 0$$

$$Delay_t \;:\quad \frac{}{0, \rho \vdash \$e \xrightarrow{\mathcal{T}} \texttt{false}} \qquad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{D}} \texttt{true} \quad t, \rho \vdash e \xrightarrow{\mathcal{T}} top \quad OK(t-1, \rho, e)}{t, \rho \vdash \$e \xrightarrow{\mathcal{T}} top}\; t \neq 0$$

$$Delay_g t \;:\quad \frac{}{0, \rho \vdash \$e \xrightarrow{\mathcal{G}} \texttt{UndefGEO}} \qquad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{D}} \texttt{true} \quad OK(t, \rho, \$e) = \texttt{false} \quad t, \rho \vdash e \xrightarrow{\mathcal{G}} g}{t, \rho \vdash \$e \xrightarrow{\mathcal{G}} g}$$

$$Delay_g d \;:\quad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{T}} \texttt{true} \quad t-1, \rho \vdash e \xrightarrow{\mathcal{G}} g}{t, \rho \vdash \$e \xrightarrow{\mathcal{G}} g} \qquad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{T}} \texttt{false} \quad t-1, \rho \vdash \$e \xrightarrow{\mathcal{G}} g}{t, \rho \vdash \$e \xrightarrow{\mathcal{G}} g}$$

$$Delay_v t \;:\quad \frac{}{0, \rho \vdash \$e \xrightarrow{\mathcal{V}} \texttt{Undef}} \qquad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{D}} \texttt{true} \quad OK(t, \rho, \$e) = \texttt{false} \quad t, \rho \vdash e \xrightarrow{\mathcal{V}} v}{t, \rho \vdash \$e \xrightarrow{\mathcal{V}} v}$$

$$Delay_v d \;:\quad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{T}} \texttt{true} \quad t-1, \rho \vdash e \xrightarrow{\mathcal{V}} v}{t, \rho \vdash \$e \xrightarrow{\mathcal{V}} v} \qquad \frac{t, \rho \vdash \$e \xrightarrow{\mathcal{T}} \texttt{false} \quad t-1, \rho \vdash \$e \xrightarrow{\mathcal{V}} v}{t, \rho \vdash \$e \xrightarrow{\mathcal{V}} v}$$

RÈG. 18: *L'opérateur de délai* \$

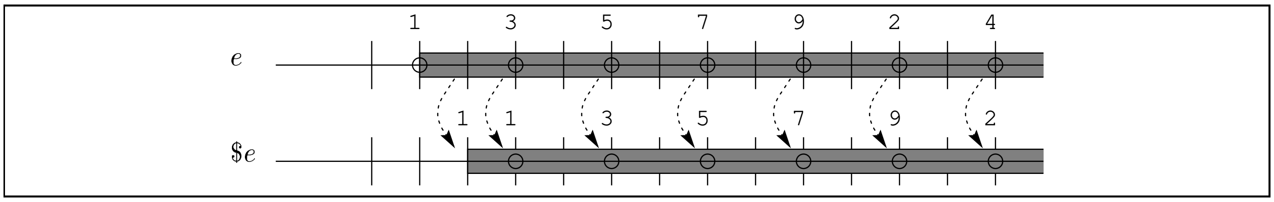


FIG. 1 –: *Calcul de la valeur de* $\$e$*. Les cercles représentent les tops de l'expression ; les barres verticales correspondent aux tics ; les parties grisées représentent le domaine de l'expression.*

Si le domaine du tissu e a pour valeur `true`, et que $\$e$ n'a pas encore eu de top, alors plutôt que de donner une géométrie et une valeur indéfinie à $\$e$, on préfère lui donner la géométrie et la valeur de e. La figure 1 détaille le calcul de l'horloge d'une expression impliquant un délai. En conséquence, les règles de calcul par défaut (règles (14)) ne s'appliquent pas pour le calcul des valeurs du délai.

XIII.5.5.2 L'opérateur `fby`

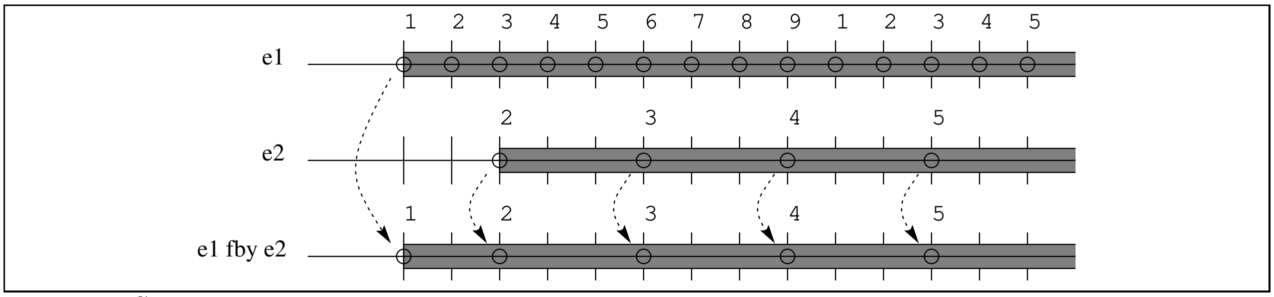


FIG. 2 –: *Calcul de l'horloge de* e_1 `fby` e_2*. L'expression* e_1 *possède un top tous les tic à partir du premier tic et l'expression* e_2 *possède un top tous les trois tics à partir du* 3[e] *tic. L'horloge de l'expression* `fby` *est celle de* e_1 *jusqu'au premier top de* e_1 *puis est égale à celle de* e_2*.*

Nous avons vu que l'opérateur `fby` était une construction qui permettait la traduction des équations quantifiées en une équation unique. Une expression $s \equiv e_1\, \texttt{fby}\, e_2$ se comporte comme e_1 jusqu'au premier top de e_1 inclus, puis se comporte ensuite comme e_2. Le domaine de s est celui de e_1. Le top de s est celui de e_1 jusqu'au premier top de e_1 puis est égal aux tops de e_2 ; il en est de même pour le calcul de la géométrie et de la valeur. La figure 2 est un exemple du calcul d'une expression $e_1\,\texttt{fby}\,e_2$ où e_1 et e_2 sont des tissus munis d'une horloge différente.

$$Fby_d \quad : \quad \frac{t,\rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1}{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{D}} d_1}$$

$$Fby_t \quad : \quad \frac{0,\rho \vdash e_1 \xrightarrow{\mathcal{T}} top_1}{0,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} top_1} \qquad \frac{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{D}} \texttt{true} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{T}} top_1 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{T}} top_2}{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} ite(OK(t-1,\rho,e_1), top_2, top_1)}$$

$$Fby_g 0 \quad : \quad \frac{0,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad 0,\rho \vdash e_1 \xrightarrow{\mathcal{G}} g_1}{0,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{G}} g_1}$$

$$Fby_g First \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad OK(t-1,\rho,e_1) = \texttt{false} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{G}} g_1}{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{G}} g_1}$$

$$Fby_g Next \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad OK(t-1,\rho,e_1) = \texttt{true} \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{G}} g_2}{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{G}} g_2}$$

$$Fby_v 0 \quad : \quad \frac{0,\rho, \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad 0,\rho \vdash e_1 \xrightarrow{\mathcal{V}} v_1}{0,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{V}} v_1}$$

$$Fby_v First \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad OK(t-1,\rho,e_1) = \texttt{false} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{V}} v_1}{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{V}} v_1}$$

$$Fby_v Next \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad OK(t-1,\rho,e_1) = \texttt{true} \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2}{t,\rho \vdash e_1\,\texttt{fby}\,e_2 \xrightarrow{\mathcal{V}} v_2}$$

RÈG. 19: *L'opérateur* `fby`

XIII.5.5.3 L'opérateur `when`

$$When_d 0 \quad : \quad \frac{0,\rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1 \quad 0,\rho \vdash e_2 \xrightarrow{\mathcal{D}} d_2 \quad 0,\rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2}{0,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{D}} d_1 \wedge d_2 \wedge v_2}$$

$$When_d t \quad : \quad \frac{t-1,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{D}} d \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{D}} d_2 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2}{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{D}} (d_1 \wedge d_2 \wedge v_2) \vee d} \quad t \neq 0$$

$$When_t \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{D}} \texttt{true} \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{T}} top_2 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2}{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{T}} (top_2 \wedge v_2)}$$

$$When_g \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{G}} g \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{G}} \texttt{bool}\ [1]}{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{G}} g}$$

$$When_v \quad : \quad \frac{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{V}} v}{t,\rho \vdash e_1\,\texttt{when}\,e_2 \xrightarrow{\mathcal{V}} v}$$

RÈG. 20: *L'opérateur* `when`

L'opérateur `when` est un opérateur d'échantillonnage, analogue temporel de l'opérateur `if ... then ... else`. Il permet la sélection des valeurs d'un tissu aux instants où un autre tissu de booléens a pour valeur `true`.

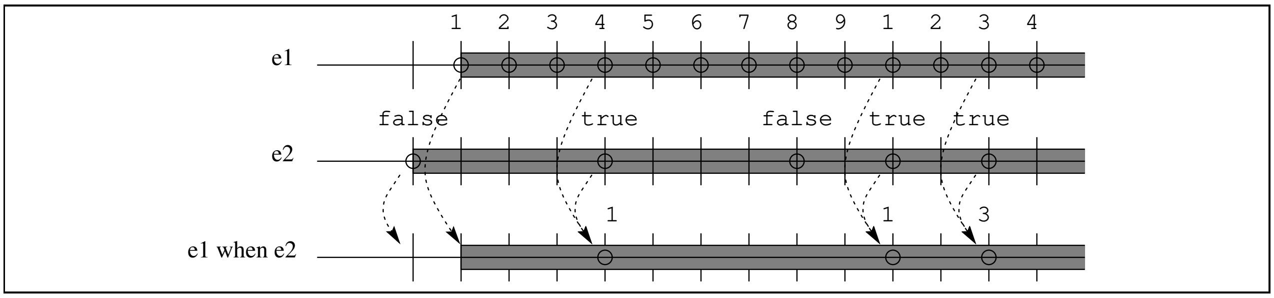


FIG. 3 –: *Calcul de la valeur de e_1 `when` e_2. Le domaine de l'expression est vrai à partir de l'instant où les domaines de e_1 et e_2 sont vrais et où e_2 a une valeur. L'expression possède un top pour chaque top de e_2 où e_2 est vrai, la valeur étant alors celle de e_1.*

Une fois que le domaine d'une expression $s \equiv e_1$ `when` e_2 est devenu vrai, il reste toujours vrai. Le domaine est vrai à partir du moment où le domaine des deux expression est vrai et où e_2 possède une valeur vraie. Une expression s possède un top aux instants où e_2 possède un top et prend la valeur vraie. La géométrie et la valeur d'une expression s sont celles de e_1.

XIII.5.6 Les deux formes de conditionnelle

XIII.5.6.1 L'opérateur `if` data-parallèle

$$Cond_d \;:\; \frac{t,\rho \vdash si \xrightarrow{\mathcal{D}} d_1 \quad t,\rho \vdash alors \xrightarrow{\mathcal{D}} d_2 \quad t,\rho \vdash sinon \xrightarrow{\mathcal{D}} d_3}{t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{D}} d_1 \wedge d_2 \wedge d_3}$$

$$Cond_t \;:\; \frac{t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{D}} \mathtt{true} \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{T}} top_1 \\ t,\rho \vdash alors \xrightarrow{\mathcal{T}} top_2 \\ t,\rho \vdash sinon \xrightarrow{\mathcal{T}} top_3 \end{array}}{t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{T}} \bigvee top_i} \quad i \in [1,3]$$

$$Cond_g \;:\; \frac{t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{T}} \mathtt{true} \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{G}} \mathtt{bool}\ l \\ t,\rho \vdash alors \xrightarrow{\mathcal{G}} \mathtt{ts}\ l \\ t,\rho \vdash sinon \xrightarrow{\mathcal{G}} \mathtt{ts}\ l \end{array}}{t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{G}} \mathtt{ts}\ l}$$

$$Cond \;:\; \frac{\begin{array}{l} t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{T}} \mathtt{true} \\ t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{G}} g \end{array} \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{V}} v_{si} \\ t,\rho \vdash alors \xrightarrow{\mathcal{V}} v_{alors} \\ t,\rho \vdash sinon \xrightarrow{\mathcal{V}} v_{sinon} \end{array}}{t,\rho \vdash \mathtt{if}\, si\, \mathtt{then}\, alors\, \mathtt{else}\, sinon \xrightarrow{\mathcal{V}} \mathtt{ite}_{g\times g\times g\to g}(v_{si}, v_{alors}, v_{sinon})}$$

RÈG. 21: *L'opérateur conditionnel `if` data-parallèle*

L'opérateur conditionnel `if` de 81/2 et 81/2D est un opérateur de sélection data-parallèle. Il permet la combinaison des éléments de deux collections de géométries et de types scalaires identiques suivant une troisième collection de type booléen qui joue le rôle de filtre : suivant les valeurs `true` ou `false` du tableau de booléens, les valeurs sont sélectionnées parmi le deuxième ou le troisième argument du `if`. Par exemple, l'expression :

$$\texttt{if}\,\{\texttt{true},\texttt{true},\texttt{false},\texttt{true}\}\,\texttt{then}\,\{1,2,20,4\}\,\texttt{else}\,\{10,11,3,13\} \Rightarrow \{1,2,3,4\}$$

est un exemple d'utilisation de l'opérateur `if`. Les valeurs sont sélectionnées point-à-point par les valeurs du premier argument de l'opérateur.

Les dom d'une expression $s \equiv \texttt{if}\, c\, \texttt{then}\, e_1\, \texttt{else}\, e_2$ sont égaux à la conjonction des dom de e_1 et de e_2 ; les tops de s sont égaux à la disjonction des tops de e_1 et e_2. Les arguments e_1 et e_2 doivent avoir une géométrie identique et la valeur de s est égale à la sélection point-à-point des éléments de e_1 et de e_2 suivant c.

XIII.5.6.2 L'opérateur conditionnel dynamique `switch`

$$Switch_d0 \quad : \quad \frac{OK(t,\rho,si) = \texttt{false}}{t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{D}} \texttt{false}}$$

$$Switch_d1 \quad : \quad \frac{0,\rho \vdash si \xrightarrow{\mathcal{T}} \texttt{true} \quad 0,\rho \vdash si \xrightarrow{\mathcal{V}} v \quad \begin{array}{l} 0,\rho \vdash alors \xrightarrow{\mathcal{D}} d_{alors} \\ 0,\rho \vdash sinon \xrightarrow{\mathcal{D}} d_{sinon} \end{array}}{0,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{D}} ite(v, d_{alors}, d_{sinon})}$$

$$Switch_d2 \quad : \quad \frac{t,\rho \vdash si \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash si \xrightarrow{\mathcal{V}} v \quad \begin{array}{l} t,\rho \vdash alors \xrightarrow{\mathcal{D}} d_{alors} \\ t,\rho \vdash sinon \xrightarrow{\mathcal{D}} d_{sinon} \\ t-1,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{D}} d \end{array}}{t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{D}} ite(v, d_{alors}, d_{sinon}) \vee d}$$

$$Switch_t \quad : \quad \frac{t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{D}} \texttt{true} \quad t,\rho \vdash si \xrightarrow{\mathcal{V}} v \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{T}} top \\ t,\rho \vdash alors \xrightarrow{\mathcal{T}} top_1 \\ t,\rho \vdash sinon \xrightarrow{\mathcal{T}} top_2 \end{array}}{t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{T}} ite(v, top_1, top_2) \vee (top \wedge OK(t,\rho, ite(v, alors, sinon)))}$$

$$Switch_g \quad : \quad \frac{t,\rho \vdash \texttt{if}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash si \xrightarrow{\mathcal{V}} b \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{G}} \texttt{bool}\ [1] \\ t,\rho \vdash alors \xrightarrow{\mathcal{G}} g_1 \\ t,\rho \vdash sinon \xrightarrow{\mathcal{G}} g_2 \end{array}}{t,\rho \vdash \texttt{if}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{G}} ite(b, g_1, g_2)}$$

$$Switch_v \quad : \quad \frac{\begin{array}{l} t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{T}} \texttt{true} \\ t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{G}} g \end{array} \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{V}} v_{si} \\ t,\rho \vdash alors \xrightarrow{\mathcal{V}} v_{alors} \\ t,\rho \vdash sinon \xrightarrow{\mathcal{V}} v_{sinon} \end{array} \quad \begin{array}{l} t,\rho \vdash si \xrightarrow{\mathcal{G}} g_{si} \\ t,\rho \vdash alors \xrightarrow{\mathcal{G}} g_{alors} \\ t,\rho \vdash sinon \xrightarrow{\mathcal{G}} g_{sinon} \end{array}}{t,\rho \vdash \texttt{switch}\, si\, \texttt{then}\, alors\, \texttt{else}\, sinon \xrightarrow{\mathcal{V}} \texttt{Switch}_{g_{si} \times g_{alors} \times g_{sinon} \to g}(v_{si}, v_{alors}, v_{sinon})}$$

RÈG. 22: *L'opérateur conditionnel dynamique* `switch`

Le comportement temporel de l'opérateur `switch` est différent de son homologue data-parallèle. Le domaine d'une expression $s \equiv \texttt{switch}\, c\, \texttt{then}\, e_1\, \texttt{else}\, e_2$ dépend du domaine de c et de la branche qui est sélectionnée par la valeur de c. Si c n'a jamais eu de top, où que la branche sélectionnée n'a pas son domaine vrai, alors le domaine de s est faux. Si c a déjà eu un top, alors le domaine de s est le domaine de l'argument sélectionné par la valeur de c (cf. figure 4). Une fois le domaine devenu vrai, il le reste. Une expression s possède un top à chaque fois que c ou que la branche sélectionnée possède un top. Le calcul de la géométrie de s ne contraint pas les trois opérandes à avoir les mêmes dimensions, la géométrie de s dépendant de la géométrie de la branche sélectionnée ; la valeur de s est égale à la valeur de la branche sélectionnée par c.

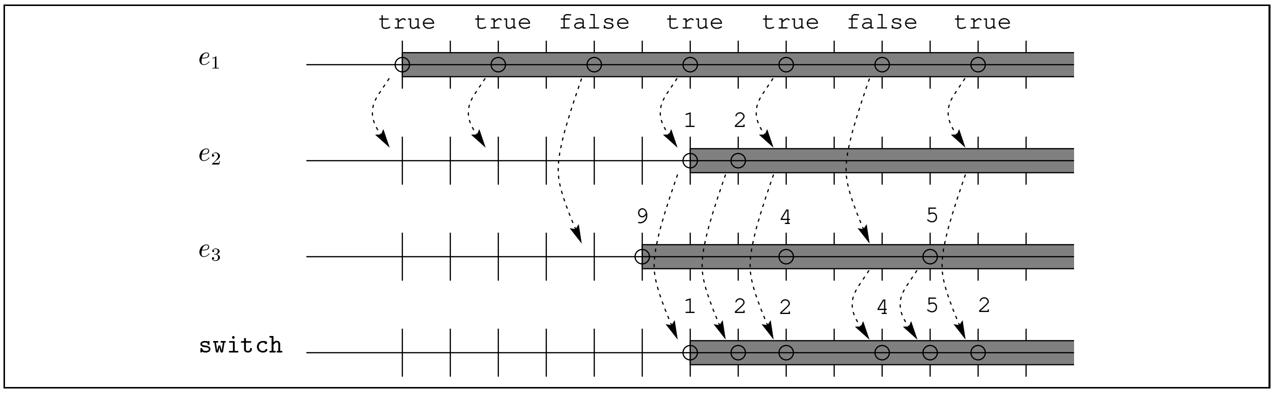


FIG. 4 –: *Calcul de la valeur de l'expression* `switch` e_1 `then` e_2 `else` e_3. *Le domaine de l'expression* `switch` *est vrai à partir du moment où les domaines de la condition* et *de la branche sélectionnée sont vrais. L'expression* `switch` *possède un top quand la condition ou la branche sélectionnée possèdent un top. La valeur de* `switch` *étant la valeur de la branche sélectionnée par la condition.*

XIII.5.7 Le calcul d'expressions

XIII.5.7.1 Les opérateurs unaires généraux

$$Unaire_d : \quad \frac{t, \rho \vdash e \xrightarrow{\mathcal{D}} d}{t, \rho \vdash unop\; e \xrightarrow{\mathcal{D}} d} \quad unop \in \{sin, cos, tan, e : [], \dots\}$$

$$Unaire_t : \quad \frac{t, \rho \vdash unop\; e \xrightarrow{\mathcal{D}} \mathtt{true} \quad t, \rho \vdash e \xrightarrow{\mathcal{T}} top}{t, \rho \vdash unop\; e \xrightarrow{\mathcal{T}} top} \quad unop \in \{sin, cos, tan, e : [], \dots\}$$

$$e : []_g : \quad \frac{t, \rho \vdash e : [\dots, n_i, \dots] \xrightarrow{\mathcal{T}} \mathtt{true} \quad t, \rho \vdash e \xrightarrow{\mathcal{G}} \mathtt{ts}\; l}{t, \rho \vdash e : [\dots, n_i, \dots] \xrightarrow{\mathcal{G}} \mathtt{ts}\; [\dots, n_i, \dots]}$$

$$Unaire_g : \quad \frac{t, \rho \vdash unop\; e \xrightarrow{\mathcal{T}} \mathtt{true} \quad t, \rho \vdash e \xrightarrow{\mathcal{G}} \mathtt{float}\; l}{t, \rho \vdash unop\; e \xrightarrow{\mathcal{G}} \mathtt{float}\; l} \quad unop \in \{sin, cos, tan, \dots\}$$

$$e : []_v : \quad \frac{t, \rho \vdash e : [\dots, n_i, \dots] \xrightarrow{\mathcal{T}} \mathtt{true} \quad t, \rho \vdash e \xrightarrow{\mathcal{V}} v \quad t, \rho \vdash e \xrightarrow{\mathcal{G}} g_e \quad t, \rho \vdash e : [\dots, n_i, \dots] \xrightarrow{\mathcal{G}} g}{t, \rho \vdash e : [\dots, n_i, \dots] \xrightarrow{\mathcal{V}} \mathtt{Shrink}_{g_e \to g}(e)}$$

$$Unaire_v : \quad \frac{t, \rho \vdash unop\; e \xrightarrow{\mathcal{T}} \mathtt{true} \quad t, \rho \vdash e \xrightarrow{\mathcal{V}} v \quad t, \rho \vdash e \xrightarrow{\mathcal{G}} g_e \quad t, \rho \vdash unop\; e \xrightarrow{\mathcal{G}} g}{t, \rho \vdash unop\; e \xrightarrow{\mathcal{V}} \mathtt{unop}_{g_e \to g}(v)} \quad unop \in \{sin, cos, tan, \dots\}$$

RÈG. 23: *Les opérateurs unaires*

L'horloge d'une opération unaire correspond à l'horloge de son argument. Les règles des opérateurs unaires comportent deux classes d'opérateurs.

Les opérateurs classiques de la logique et de l'arithmétique contraignent la géométrie de leur argument à être du type scalaire requis (booléen pour une opération logique et flottant pour une opération trigonométrique par exemple), la dimension étant quelconque, la valeur de l'expression unaire est naturellement une α-extension [1] de l'opérateur scalaire à l'argument.

L'opérateur de troncature-extension d'un tableau à une dimension quelconque spécifiée par une liste d'entiers étend une valeur de géométrie g à une valeur de géométrie g'. Le type scalaire ne change pas. De

1. L'α-extension dont il est question ici est bien évidemment l'extension point-à-point d'un opérateur scalaire à un tableau d'une dimension quelconque (cf. section II.3.3.1, page 14).

façon générale, une valeur est étendue par réplication du *motif* défini par la valeur initiale jusqu'à atteindre les bornes de la géométrie spécifiée en argument. Par exemple, l'extension du tableau $\{\{1,2\},\{3,4\}\}$ de géométrie $[2,2]$ à une géométrie $[3,3]$ est égale au tableau $\{\{1,2,1\},\{3,4,3\},\{1,2,1\}\}$.

XIII.5.7.2 Coupure et extension dynamique de la géométrie d'une collection

$$e_1 : e_{2\,d} \quad : \quad \frac{t,\rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{D}} d_2}{t,\rho \vdash e_1 : e_2 \xrightarrow{\mathcal{D}} d_1 \wedge d_2}$$

$$e_1 : e_{2\,t} \quad : \quad \frac{t,\rho \vdash e_1 : e_2 \xrightarrow{\mathcal{D}} \texttt{true} \quad t,\rho \vdash e \xrightarrow{\mathcal{T}} top_1 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{T}} top_2}{t,\rho \vdash e_1 : e_2 \xrightarrow{\mathcal{T}} (top_1 \vee top_2)}$$

$$e_1 : e_{2\,g} \quad : \quad \frac{t,\rho \vdash e : e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash e \xrightarrow{\mathcal{G}} \texttt{ts}\ l \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{G}} \texttt{int}\ [n] \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{V}} v}{t,\rho \vdash e_1 : e_2 \xrightarrow{\mathcal{G}} \texttt{ts}\ v}$$

$$e_1 : e_{2\,v} \quad : \quad \frac{t,\rho \vdash e_1 : e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash e \xrightarrow{\mathcal{V}} v \quad t,\rho \vdash e : f \xrightarrow{\mathcal{G}} g}{t,\rho \vdash e_1 : e_2 \xrightarrow{\mathcal{V}} \texttt{Shrink}_{g_e \to g}(v)}$$

RÈG. 24: *L'opérateur dynamique de coercion de la géométrie d'une collection*

La sémantique de la coercion spatiale dynamique est identique à la coercion statique (cf. section II.3.3.4, page 15) à la différence que, utilisant un tissu pour spécifier le second argument de l'opérateur, il est nécessaire de calculer celui-ci et de vérifier que c'est bien un vecteur, avant de pouvoir effectuer la coercion.

XIII.5.7.3 Rang d'une collection dynamique

$$|e|_d \quad : \quad \frac{t,\rho \vdash e \xrightarrow{\mathcal{D}} d}{t,\rho \vdash |e| \xrightarrow{\mathcal{D}} d}$$

$$|e|_t \quad : \quad \frac{t,\rho \vdash |e| \xrightarrow{\mathcal{D}} \texttt{true} \quad t,\rho \vdash e \xrightarrow{\mathcal{T}} top}{t,\rho \vdash |e| \xrightarrow{\mathcal{T}} top}$$

$$|e|_g \quad : \quad \frac{t,\rho \vdash |e| \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash e \xrightarrow{\mathcal{G}} \texttt{ts}\ l}{t,\rho \vdash |e| \xrightarrow{\mathcal{G}} \texttt{int [|}\ length(l)\ \texttt{|]}}$$

$$|e|_v \quad : \quad \frac{t,\rho \vdash |e| \xrightarrow{\mathcal{T}} \texttt{true} \quad t,\rho \vdash e \xrightarrow{\mathcal{G}} \texttt{ts}\ l}{t,\rho \vdash |e| \xrightarrow{\mathcal{V}} \texttt{[|}\ nth(l,1)\,;\ \ldots\,;\ nth(l,\texttt{lg}(l))\ \texttt{|]}}$$

RÈG. 25: *L'opérateur dynamique de calcul de la géométrie d'une collection*

La sémantique de l'opérateur $||$ est très proche de celle d'un opérateur unaire général. Le calcul de la dimension d'un tissu correspond au calcul de la géométrie du tissu, la valeur étant égale à un tableau ayant pour i^{e} valeur la i^{e} valeur de la géométrie du tissu.

XIII.5.7.4 Les opérateurs binaires

Le domaine des opérateurs binaires correspond à la conjonction des domaines des arguments et le top à la disjonction des tops des arguments. La géométrie et le type scalaire doivent être homogènes pour les

$$Binaire_d \quad : \quad \frac{t,\rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{D}} d_2}{t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{D}} d_1 \wedge d_2} \quad binop \in \{+,-,\wedge,\vee,\dots\}$$

$$Binaire_t \quad : \quad \frac{t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{D}} \mathtt{true} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{T}} top_1 \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{T}} top_2}{t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{T}} top_1 \vee top_2} \quad binop \in \{+,-,\wedge,\vee,\dots\}$$

$$Binaire_g \quad : \quad \frac{t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{T}} \mathtt{true} \quad t,\rho \vdash e_1 \xrightarrow{\mathcal{G}} \mathtt{ts}\ l \quad t,\rho \vdash e_2 \xrightarrow{\mathcal{G}} \mathtt{ts}\ l}{t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{G}} \mathtt{int}\ l} \quad binop \in \{+,-,\wedge,\vee,\dots\}$$

$$Binaire_v \quad : \quad \frac{\begin{array}{lll} t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{T}} \mathtt{true} & t,\rho \vdash e_1 \xrightarrow{\mathcal{V}} v_1 & t,\rho \vdash e_1 \xrightarrow{\mathcal{G}} g_1 \\ t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{G}} g & t,\rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2 & t,\rho \vdash e_2 \xrightarrow{\mathcal{G}} g_2 \end{array}}{t,\rho \vdash e_1\ binop\ e_2 \xrightarrow{\mathcal{V}} \mathtt{Bop}_{g_1 \times g_2 \to g}(v_1,v_2)} \quad binop \in \{+,-,\wedge,\vee,\dots\}$$

RÈG. 26: *Les opérateurs binaires*

deux arguments. De façon similaire aux opérateur unaires, l'opération binaire est naturellement α-étendue à la géométrie de ses arguments. La valeur du résultat correspond à l'application point-à-point de l'opérateur binaire à ses arguments.

XIII.5.8 Traitement des systèmes, des amalgames et des tableaux

La définition du domaine sémantique de la géométrie distingue les systèmes, les amalgames et les tableaux. Il est cependant possible, comme nous l'avons évoqué dans le chapitre précédent, de définir des expressions impliquant des systèmes, des amalgames et des tableaux. Les opérateurs définis particulièrement sur ces objets sont les opérateurs de définition, de concaténation et de sélection.

$$Geo_{\#} \quad : \quad \frac{t,\rho \vdash u \xrightarrow{\mathcal{G}} g_u \quad t,\rho \vdash v \xrightarrow{\mathcal{G}} g_v}{t,\rho \vdash u, v\ \# \xrightarrow{\mathcal{G}} Comb(\#, g_u, g_v)}$$

$$Geo_{\centerdot} \quad : \quad \frac{t,\rho \vdash u \xrightarrow{\mathcal{G}} g_u \quad t,\rho \vdash v \xrightarrow{\mathcal{G}} g_v}{t,\rho \vdash u, v\ \centerdot \xrightarrow{\mathcal{G}} Comb(\centerdot, g_u, g_v)}$$

RÈG. 27: *Calcul de la géométrie des opérations de concaténation et de sélection pour des opérandes système, amalgame, et tableau.*

Les règles (27) définissent le calcul du type de la géométrie d'une opération de concaténation et de sélection pour des opérandes de type système, tableau et amalgame. Nous avons de plus intégré la géométrie indéfinie. La valeur de la géométrie est spécifiée par la fonction $Comb(\#, g_u, g_v)$ dont les valeurs sont spécifiées dans la table 1.

Le détail de la *valeur* de la géométrie apparaît dans les sections suivantes qui décrivent la sémantique des systèmes et des tableaux ; de même, nous ne nous occupons pas ici de l'opération de définition, celle-ci étant spécifiée dans les sections suivantes.

Pour ce qui concerne le calcul de la valeur de ces expressions, les sections suivantes définissent la sémantique pour des opérations entre opérandes de type correct, c'est-à-dire qu'une opération de sélection par un identificateur n'a lieu que lorsque l'opérande gauche de la sélection est un système.

REMARQUE Pour le calcul de la valeur d'expression impliquant des objets de type géométrique différent, et pour lequel le type géométrique n'est pas erroné, il est nécessaire de coercer la valeur d'un des opérandes. Par exemple, lors d'une opération de concaténation d'un objet de type système et d'un objet de type amalgame, il

#	*Système*	*Tableau*	AmalGeo	UndefGEO
Système	*Système*	erreur !	AmalGeo	UndefGEO
Tableau	erreur !	*Tableau*	erreur !	UndefGEO
AmalGeo	AmalGeo	erreur !	AmalGeo	UndefGEO
UndefGEO	UndefGEO	UndefGEO	UndefGEO	UndefGEO

.	*Système*	*Tableau*	AmalGeo	UndefGEO
Système	AmalGeo	erreur !	AmalGeo	UndefGEO
Tableau	erreur !	*Tableau*	AmalGeo	UndefGEO
AmalGeo	AmalGeo	*Tableau*	AmalGeo	UndefGEO
UndefGEO	UndefGEO	UndefGEO	UndefGEO	UndefGEO

TAB. 1 –: *Type de la géométrie des opérations de concaténation et de sélection entre des systèmes, des amalgames et des tableaux. Tableau et Système correspondent à des calculs de géométrie à partir de la géométrie des arguments. Par exemple,* $[2,3]\,\#\,[4,3] = [6,3]$.

est nécessaire de coercer le système en amalgame afin de pouvoir effectuer le calcul entre les deux opérandes. La table 1 que nous venons de spécifier définit les coercions nécessaires, coercions que nous ne ferons pas apparaître dans les règles de la sémantique afin de ne pas les surcharger.

XIII.5.9 Traitement des systèmes

Comme nous venons de le voir, nous supposons qu'une expression est correctement typée lors de l'exécution d'une de ces trois opérations, c'est-à-dire qu'une opération définie sur des arguments de type géométrique système ne peut être effectuée que si ses arguments sont *effectivement* de type géométrique système.

XIII.5.9.1 La définition de systèmes

L'horloge, la géométrie et la valeur d'un système sont des fonctions qui associent aux identificateurs du système les horloges, les géométries et les valeurs de leurs définitions. Les règles de notre sémantique calculent ces valeurs et retournent la fonction qui associe à chaque identificateur la valeur de son definiens. Le calcul de ces valeurs se fait alors dans l'environnement augmenté des définitions apparaissant dans le système : par exemple, le système[1] $\{a = b_0, b = 1\}$ a besoin de connaître la définition de b pour résoudre le calcul de a.

$$\{sys\}_d \quad : \quad \frac{\rho' = \rho \uplus [\ldots, id_i \mapsto e_i, \ldots] \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{D}} d_i}{t, \rho \vdash \{\ldots, id_i = e_i, \ldots\} \xrightarrow{\mathcal{D}} [\ldots, id_i \mapsto d_i, \ldots]} \quad i \in [1,n]$$

$$\{sys\}_t \quad : \quad \frac{\rho' = \rho \uplus [\ldots, id_i \mapsto e_i, \ldots] \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{D}} \mathtt{true} \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{T}} top_i}{t, \rho \vdash \{\ldots, id_i = e_i, \ldots\} \xrightarrow{\mathcal{T}} [\ldots, id_i \mapsto top_i, \ldots]} \quad i \in [1,n]$$

$$\{sys\}_g \quad : \quad \frac{\rho' = \rho \uplus [\ldots, id_i \mapsto e_i, \ldots] \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{T}} \mathtt{true} \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{G}} g_i}{t, \rho \vdash \{\ldots, id_i = e_i, \ldots\} \xrightarrow{\mathcal{G}} [\ldots, id_i \mapsto g_i, \ldots]} \quad i \in [1,n]$$

$$\{sys\}_v \quad : \quad \frac{\rho' = \rho \uplus [\ldots, id_i \mapsto e_i, \ldots] \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{T}} \mathtt{true} \quad t, \rho' \vdash e_i \xrightarrow{\mathcal{V}} v_i}{t, \rho \vdash \{\ldots, id_i = e_i, \ldots\} \xrightarrow{\mathcal{V}} [\ldots, id_i \mapsto v_i, \ldots]} \quad i \in [1,n]$$

RÈG. 28: *L'opérateur de définition de systèmes*

1. Cette expression correspond bien à un système. En effet, elle ne comporte aucune référence libre. Toutes les références étant liées, il est uniquement nécessaire de propager les valeurs des définitions.

Remarque La sémantique d'un système, bien que différente de celle définie dans [Gia91a] assure qu'une expression de la forme $\{a = a\}$ ne boucle pas (cf. section XIII.6.1, preuve (1)).

XIII.5.9.2 La sélection par un identificateur

$$
\begin{array}{lll}
._d & : & \dfrac{t, \rho \vdash e_1 \xrightarrow{\mathcal{D}} d}{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{D}} d(id)} \\[2ex]
._t & : & \dfrac{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{D}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{T}} top}{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{T}} top(id)} \\[2ex]
._g & : & \dfrac{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{G}} g}{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{G}} g(id)} \\[2ex]
._v & : & \dfrac{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{V}} v}{t, \rho \vdash e_1 \;.\; id \xrightarrow{\mathcal{V}} v(id)}
\end{array}
$$

Règ. 29: *L'opérateur de sélection par un identificateur*

Nous supposons que l'expression de sélection par un identificateur est correctement typée, c'est-à-dire qu'une opération de sélection par un label n'apparaît qu'avec un système en partie gauche du point. Les dom et top sont donc des fonctions de Id $\rightarrow$ Bool, la géométrie une fonction de Id $\rightarrow$ Geo et la valeur une fonction de Id $\rightarrow$ Val. Les dom, top, géométrie et valeur correspondent à l'application de la fonction correspondante à la valeur du label dans l'environnement ρ.

XIII.5.9.3 La concaténation de systèmes

Nous supposons, lors de la concaténation de deux systèmes qu'il n'y a pas de collision de noms : l'intersection des identificateurs définis par chaque système est vide. Les opérandes d'une opération de concaténation étant supposées bien typées, les deux arguments de la concaténation sont bien des systèmes : il est donc valide de calculer le dom, le top la géométrie et la valeur de la concaténation en termes de composition des fonctions de ses arguments.

$$
\begin{array}{lll}
\#_d & : & \dfrac{t, \rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1 \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{D}} d_2}{t, \rho \vdash e_1 \# e_2 \xrightarrow{\mathcal{D}} d_1 \uplus d_2} \\[2ex]
\#_t & : & \dfrac{t, \rho \vdash e_1 \# e_2 \xrightarrow{\mathcal{D}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{T}} top_1 \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{T}} top_2}{t, \rho \vdash e_1 \# e_2 \xrightarrow{\mathcal{T}} top_1 \uplus top_2} \\[2ex]
\#_g & : & \dfrac{t, \rho \vdash e_1 \# e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{G}} g_1 \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{G}} g_2}{t, \rho \vdash e_1 \#_{e_2} \xrightarrow{\mathcal{G}} g_1 \uplus g_2} \\[2ex]
\#_v & : & \dfrac{t, \rho \vdash e_1 \# e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{V}} v_1 \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2}{t, \rho \vdash e_1 \# e_2 \xrightarrow{\mathcal{V}} v_1 \uplus v_2}
\end{array}
$$

Règ. 30: *L'opérateur de concaténation de systèmes*

XIII.5.10 Traitement des amalgames

$$\mathcal{L}_d : \frac{}{t, \rho \vdash c \xrightarrow{\mathcal{D}} \texttt{true}} \quad \texttt{Code}(c)$$

$$\mathcal{L}_t : \frac{}{t, \rho \vdash c \xrightarrow{\mathcal{T}} \texttt{true}} \quad \texttt{Code}(c)$$

$$\mathcal{L}_g : \frac{}{t, \rho \vdash c \xrightarrow{\mathcal{G}} \texttt{AmalGeo}} \quad \texttt{Code}(c)$$

$$\mathcal{L}_v : \frac{t, \rho \vdash c \xrightarrow{\mathcal{A}} v}{t, \rho \vdash c \xrightarrow{\mathcal{V}} v} \quad \texttt{Code}(c)$$

RÈG. 31: *Calcul d'une expression d'amalgame.*

Le calcul sur des expressions d'amalgame est entièrement réalisé par la sémantique que nous avons spécifiée dans le précédent chapitre. Par conséquent, il n'est pas nécessaire de reproduire ici les règles correspondant au calcul des opérations de concaténation et de sélection.

XIII.5.11 Le traitement des tableaux

XIII.5.11.1 La définition de tableaux

$$\{tab\}_d : \frac{t, \rho \vdash e_i \xrightarrow{\mathcal{D}} d_i}{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{D}} \bigwedge d_i} \quad i \in [1, n]$$

$$\{tab\}_t : \frac{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{D}} \texttt{true} \quad t, \rho \vdash e_i \xrightarrow{\mathcal{T}} top_i}{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{T}} \bigvee top_i} \quad i \in [1, n]$$

$$\{tab\}_g : \frac{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_i \xrightarrow{\mathcal{G}} \texttt{ts}\ l}{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{G}} \texttt{ts}\ n{::}\, l} \quad i \in [1, n]$$

$$\{tab\}_v : \frac{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_i \xrightarrow{\mathcal{V}} v_i}{t, \rho \vdash \{\ldots, e_i, \ldots\} \xrightarrow{\mathcal{V}} \texttt{[|}\ \ldots, v_i, \ldots\ \texttt{|]}} \quad i \in [1, n]$$

RÈG. 32: *L'opérateur de définition de tableaux*

Le calcul de l'horloge d'un tableau détermine la façon dont on pourra accéder à ses éléments. Plusieurs choix, pour la définition de l'horloge d'un tableau, sont possibles :

1. Tous les éléments du tableau ont la même horloge et l'horloge du tableau correspond à l'horloge de ses éléments. Tous les éléments ayant la même horloge, il n'est pas possible de définir des tableaux partiels ni des tableaux dont les élément changent à des rythmes différents.
2. L'horloge des éléments du tableau est quelconque. L'horloge du tableau est définie comme l'union des horloges de ses éléments mais on demande néanmoins à ce que tous les éléments aient leur domaine vrai. Dans ce cas, les tableaux dont les éléments évoluent à leur propre rythme sont permis alors que les tableaux partiels sont rejetés.
3. L'horloge des éléments du tableau est quelconque et l'horloge du tableau est l'union des horloges de ses éléments, sans restriction. Dans ce cas, les tableaux partiels sont admis, i.e. les tableaux dont le domaine de certains de leurs éléments est faux alors que le domaine d'autres éléments est vrai.

Nous choisissons pour le moment la seconde possibilité qui nous permet de définir des tableaux où chaque élément peut avoir une horloge différente mais où tous les éléments sont définis. Le domaine d'un tableau est donc la conjonction des domaines de ses éléments, le top étant la disjonction des tops des éléments du tableau. La géométrie des tableaux nécessite que les éléments du tableau aient tous la même géométrie (i.e. les tableaux sont homogènes). La valeur d'un tableau correspond à la valeur de l'agrégation de la valeur de ses éléments.

XIII.5.11.2 La sélection par un entier

$$\mathbf{.}_d \quad : \quad \frac{t, \rho \vdash e_1 \xrightarrow{\mathcal{D}} d}{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{D}} d}$$

$$\mathbf{.}_t \quad : \quad \frac{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{D}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{T}} top}{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{T}} top}$$

$$\mathbf{.}_g \quad : \quad \frac{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{G}} \texttt{ts}\; h{::}\,l \quad 0 \leq n < h}{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{G}} \texttt{ts}\; l}$$

$$\mathbf{.}_v \quad : \quad \frac{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{V}} v \quad \begin{array}{l} t, \rho \vdash e_1 \xrightarrow{\mathcal{G}} g_1 \\ t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{G}} g_2 \end{array}}{t, \rho \vdash e_1 \;\mathbf{.}\; n \xrightarrow{\mathcal{V}} \texttt{SelInt}_{g_1 \to g_2}(v, n)}$$

RÈG. 33: *L'opérateur de sélection par un entier*

L'horloge d'une sélection d'un tableau par un entier est l'horloge du tableau considéré. La géométrie de la sélection d'un tableau à n dimensions par un entier est un tableau à $n-1$ dimensions de même type scalaire que le tableau initial ; la valeur de la sélection d'un tableau est le n^{e} vecteur du tableau initial.

XIII.5.11.3 La concaténation de tableaux

$$\#_d \quad : \quad \frac{t, \rho \vdash e_1 \xrightarrow{\mathcal{D}} d_1 \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{D}} d_2}{t, \rho \vdash e_1 \,\#\, e_2 \xrightarrow{\mathcal{D}} d_1 \wedge d_2}$$

$$\#_t \quad : \quad \frac{t, \rho \vdash e_1 \,\#\, e_2 \xrightarrow{\mathcal{D}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{T}} top_1 \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{T}} top_2}{t, \rho \vdash e_1 \,\#\, e_2 \xrightarrow{\mathcal{T}} top_1 \vee top_2}$$

$$\#_g \quad : \quad \frac{t, \rho \vdash e_1 \,\#_n\, e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad t, \rho \vdash e_1 \xrightarrow{\mathcal{G}} \texttt{ts}\; l\,@\,(h_1{::}\,l') \quad t, \rho \vdash e_2 \xrightarrow{\mathcal{G}} \texttt{ts}\; l\,@\,(h_2{::}\,l') \quad length(l) == n}{t, \rho \vdash e_1 \,\#_n\, e_2 \xrightarrow{\mathcal{G}} \texttt{ts}\; l\,@\,(h_1{::}\,h_2){::}\,l'}$$

$$\#_v \quad : \quad \frac{t, \rho \vdash e_1 \,\#\, e_2 \xrightarrow{\mathcal{T}} \texttt{true} \quad \begin{array}{l} t, \rho \vdash e_1 \,\#_n\, e_2 \xrightarrow{\mathcal{G}} g \\ t, \rho \vdash e_1 \xrightarrow{\mathcal{G}} g_1 \\ t, \rho \vdash e_2 \xrightarrow{\mathcal{G}} g_2 \end{array} \quad \begin{array}{l} t, \rho \vdash e_1 \xrightarrow{\mathcal{V}} v_1 \\ t, \rho \vdash e_2 \xrightarrow{\mathcal{V}} v_2 \end{array}}{t, \rho \vdash e_1 \,\#_n\, e_2 \xrightarrow{\mathcal{V}} \texttt{\#}_{g_1 \times g_2 \to g}(v_1, v_2, n)}$$

RÈG. 34: *L'opérateur de concaténation de tableaux*

La concaténation de tableaux se comporte de la même façon que la définition de tableaux pour le calcul de l'horloge. Le dom est la conjonction des domaines de ses arguments et le top, la disjonction des tops. L'opération de concaténation permet la concaténation de tableaux suivant une dimension particulière : par exemple, dans le cas d'un tableau bi-dimensionnel, la concaténation suivant la première dimension est une concaténation « à droite », suivant la seconde dimension, c'est une concaténation « vers le bas » (cf. section II.3.3.5, page 15). Si on concatène un tableau à n dimensions suivant la dimension n', alors les géométries des $n - n' - 1$ premières et des $n - n'$ dernières dimensions doivent être identiques.

XIII.6 Preuves d'évaluation de programmes 8½$_{\mathcal{D}}$

Dans cette section nous allons dérouler « à la main » la sémantique formelle afin de calculer l'horloge de quelques expressions caractéristiques. Nous verrons ainsi que l'horloge calculée par 8½$_{\mathcal{D}}$ est la même que l'horloge calculée par la sémantique dénotationnelle initiale de 8½. Le but n'est pas de prouver que la nouvelle sémantique est une extension compatible de l'ancienne, ce n'est pas le but de cette thèse, mais de se convaincre que grosso modo, on calcule la même chose.

Les expressions caractéristique du calcul d'horloge 8½ sont les suivantes :

- $\{a = a\}$ définit (sans boucler) un stream dont le domaine n'est jamais vrai.
- $\$a + 1$ définit (sans boucler) un stream dont le domaine n'est jamais vrai.
- $\{a = 0\,\texttt{fby}(\$a + 1)\}$ définit un stream avec un seul top en $t = 0$.
- $\{a = 0\,\texttt{fby}(\$a + 1\,\texttt{when}\,\texttt{Clock})\}$ définit un compteur qui progresse au rythme de `Clock`.
- $sx = x\,\texttt{fby}(x + \$sx)$ possède la même horloge que celle de x.

Ces expressions sont caractéristiques du calcul d'horloge 8½ : par exemple, les expressions analogues en Lustre ou bien Signal ne présenteraient pas les mêmes horloges.

XIII.6.1 Calcul de la valeur d'une expression récursive simple

Nous montrons dans cette section que le calcul de la valeur d'une expression récursive simple termine et a pour résultat `Undef`.

Preuve 1 ***(Valeur de*** $\{a = a\}$ ***au tic*** t***)***
Le calcul de la valeur de $\{a = a\}$ termine. En effet, l'utilisation de la règle $\{sys\}_v$ pour le calcul de l'expression nécessite que le domaine et l'horloge des expressions prémisses soit vrai. Or le domaine de l'expression précédente est faux car :

$$\dfrac{\dfrac{\dfrac{\overline{id \notin \texttt{Dom}(\emptyset)}}{t, \emptyset \vdash \rho'(a) = a \xrightarrow{\mathcal{D}} \texttt{false}}\,Id_d \qquad t, \emptyset \vdash \rho'(a) = a \xrightarrow{\mathcal{T}} ?}{t, \rho' \vdash a \xrightarrow{\mathcal{T}} \texttt{false}}\,Id_t \qquad t, \rho' \vdash a \xrightarrow{\mathcal{V}} ?}{t, \emptyset \vdash \{a = a\} \xrightarrow{\mathcal{V}} ?}\,\{sys\}_v$$

où $\rho' = [a \mapsto a]$. Par conséquent, la règle par défaut $Défaut_v$ est utilisée de façon à calculer la valeur de l'expression :

$$\dfrac{t, \emptyset \vdash M \xrightarrow{\mathcal{T}} \texttt{false} \quad \begin{array}{c} \dfrac{\overline{0, \emptyset \vdash M \xrightarrow{\mathcal{T}} \texttt{false}}}{0, \emptyset \vdash M \xrightarrow{\mathcal{V}} \texttt{Undef}}\,Défaut_v \\ \vdots\ Défaut_v \\ t-1, \emptyset \vdash M \xrightarrow{\mathcal{V}} V_M = \texttt{Undef} \end{array}}{t, \emptyset \vdash M \xrightarrow{\mathcal{V}} V_M = \texttt{Undef}}\,Défaut_v$$

Le résultat obtenu est bien celui qui était attendu : `Undef`. □

XIII.6.2 Calcul de la valeur d'un délai

Nous allons montrer que le calcul de la valeur d'une expression simple impliquant un délai termine et a pour résultat `Undef`.

Preuve 2 ***(Calcul du domaine de*** $\$a + 1$ ***au tic*** t***)***
D'une façon similaire, le calcul d'une expression $\{a = \$a + 1\}$ ne boucle pas non plus. En effet, la règle $Delay_d$ assure que le domaine de $\$a + 1$ à un instant t quelconque est faux :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0,\rho' \vdash \$a \xrightarrow{\mathcal{D}} = \texttt{false}}}{\vdots\; Bop_d, \$_d, Cte_d}}{t-1,\emptyset \vdash \rho'(a) = \$a+1 \xrightarrow{\mathcal{D}} d_1 = \texttt{false}}}{t-1,\rho' \vdash a \xrightarrow{\mathcal{D}} d_1 = \texttt{false}}\,Id_d}{t,\rho' \vdash \$a \xrightarrow{\mathcal{D}} d_1 = \texttt{false}}\,\$_d \quad \dfrac{\dfrac{}{t,\rho' \vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}}{t,\rho' \vdash 1 \xrightarrow{\mathcal{D}} d2 = \texttt{true}}\,Cte_d}{t,\rho' \vdash \$a+1 \xrightarrow{\mathcal{D}} (d_1 = \texttt{false}) \wedge (d_2 = \texttt{true}) = \texttt{false}}\,Bop_d$$

où $\rho' = [a \to \$a + 1]$. Le dom et le top de l'élément d'un système doivent être vrais pour que la valeur de l'élément puisse être calculée. Ceux-ci étant faux, la règle sys_v ne peut donc pas s'appliquer :

$$\dfrac{\dfrac{t,\rho' \vdash \$a+1 \xrightarrow{\mathcal{D}} \texttt{false} \quad t,\rho' \vdash \$a \xrightarrow{\mathcal{T}} top_1 \quad t,\rho' \vdash 1 \xrightarrow{\mathcal{T}} top_2}{t,\rho' \vdash \$a+1 \xrightarrow{\mathcal{T}} top_1 \vee top_2}\,Bop_t \qquad t,\rho' \vdash \$a+1 \xrightarrow{\mathcal{V}} ?}{t,\emptyset \vdash \{a = \$a+1\} \xrightarrow{\mathcal{V}} ?}\,\{sys\}_v$$

La règle par défaut $Défaut_v$ est utilisée pour obtenir, comme valeur de la réduction de l'expression, `Undef` (cf. preuve (1)). □

XIII.6.3 Calcul de $\{a = 0\,\texttt{fby}(\$a + 1)\}$

XIII.6.3.1 Tic $t = 0$

Calculons la valeur de l'expression $A = \{a = 0\,\texttt{fby}(\$a + 1)\}$ au tic 0. Pour que cette expression ait une valeur au tic 0, il est nécessaire qu'elle ait un top vrai. Si c'est le cas, la valeur de A sera le résultat de l'application de la réduction : $0, \emptyset \vdash A \xrightarrow{\mathcal{V}} v$. Vérifions que A à bien un top en 0 :

Preuve 3 ***(Calcul du top de*** A ***au tic 0)***

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}}{0,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}\,Cte_d}{0,\rho' \vdash 0 \xrightarrow{\mathcal{T}} (0 == 0) \equiv \texttt{true}}\,Cte_t}{0,\rho' \vdash 0\,\texttt{fby}\,\$a+1 \xrightarrow{\mathcal{T}} \texttt{true}}\,Fby_t$$

où $\rho' = [a \mapsto 0\,\texttt{fby}(\$a + 1)]$. Nous pouvons maintenant calculer la valeur de A au top 0. □

Preuve 4 ***(Calcul de la valeur de*** $0\,\texttt{fby}\,\$a + 1$ ***au tic 0)***

$$\dfrac{\dfrac{\text{cf. la règle ci-dessus}}{\vdots\; Fby_t}\atop{0,\rho' \vdash 0\,\texttt{fby}\,\$a+1 \xrightarrow{\mathcal{T}} \texttt{true}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{0,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}}{0,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}\,Cte_d}{0,\rho' \vdash 0 \xrightarrow{\mathcal{T}} (0 == 0) \equiv \texttt{true}}\,Cte_t}{0,\rho' \vdash 0 \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}}\,Cte_v}{0,\rho' \vdash 0\,\texttt{fby}\,\$a+1 \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}}\,Fby_v$$

Par conséquent, au tic 0, la valeur de A est `[| 0 |]`. □

Preuve 5 ***(Calcul de la valeur de*** $\{a = A\}$***)***

$$\dfrac{\begin{array}{c}\text{cf. (3)}\\ \vdots\ Fby_t\\ 0,\rho' \vdash 0\,\texttt{fby}(\$a+1) \xrightarrow{\mathcal{T}} \texttt{true}\end{array} \qquad \begin{array}{c}\text{cf. (4)}\\ \vdots\ Fby_v\\ 0,\rho' \vdash 0\,\texttt{fby}(\$a+1) \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}\end{array}}{0,\emptyset \vdash A \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}}\ \{sys\}_v$$

□

XIII.6.3.2 Tic $t = 1$

Calculons maintenant la valeur de A au tic 1 afin de vérifier que le stream progresse. Nous procédons pour cela de manière analogue à ce qui a été réalisé précédemment : on calcule la valeur de l'horloge de A afin de s'assurer que celle-ci est égale à `true` :

Preuve 6 ***(Horloge de*** $0\,\texttt{fby}\,\$a+1$ ***au tic 1)***

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}}{1,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{1,\rho' \vdash 0\,\texttt{fby}\,\$a+1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Fby_d \qquad \begin{array}{c}\text{cf. remarque}\\ \text{ci-dessous}\\ \vdots\ Cte_t\\ 1,\rho' \vdash 0 \xrightarrow{\mathcal{T}} \texttt{true}\end{array} \qquad \begin{array}{c}\text{cf. (7)}\\ \vdots\ Binaire_t\\ 1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{T}} \texttt{false} \vee \texttt{false} \equiv \texttt{false}\end{array}}{1,\rho' \vdash 0\,\texttt{fby}\,\$a+1 \xrightarrow{\mathcal{T}} iteOK(t-1,\rho',0), t_2, t_1 = t_2}\ Fby_t$$

Il n'est pas nécessaire de calculer la valeur du top de la constante 0. En effet, $OK(0,\rho',0)$ ayant pour valeur `true`, ce sera le top de $\$a+1$ qui sera sélectionné. En effet :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}}{0,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{0,\rho' \vdash 0 \xrightarrow{\mathcal{T}} (0 == 0) \equiv \texttt{true}}\ Cte_t}{OK(0,\rho',0) = \texttt{true}}\ Ok_0$$

□

Preuve 7 ***(Calcul du top de*** $\$a+1$ ***au tic 1)***

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{cf. tic 0}\\ \vdots\ Id_d\\ 0,\rho' \vdash a \xrightarrow{\mathcal{D}} \texttt{true}\end{array}}{1,\rho' \vdash \$a \xrightarrow{\mathcal{D}} \texttt{true}}\ Delay_d}{1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Binaire_d \qquad \begin{array}{c}\text{cf. (8)}\\ \vdots\ Delay_t\\ 1,\rho' \vdash \$a \xrightarrow{\mathcal{T}} \texttt{false}\end{array} \qquad \dfrac{\dfrac{\dfrac{}{1,\rho' \vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}}{1,\rho' \vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{1,\rho' \vdash 1 \xrightarrow{\mathcal{T}} (1 == 0) \equiv \texttt{false}}\ Cte_t}{1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{T}} \texttt{false} \vee \texttt{false} \equiv \texttt{false}}\ Binaire_t$$

□

Preuve 8 ***(Calcul de l'horloge de*** $\$a$ ***au tic 1)***

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\text{cf. tic 0}\\ \vdots\ Fby_d,\ldots\\ 0,\emptyset \vdash \rho'(a) = 0\,\texttt{fby}(\$a+1) \xrightarrow{\mathcal{D}} \texttt{true}\end{array}}{0,\rho' \vdash a \xrightarrow{\mathcal{D}} \texttt{true}}\ Id_d}{1,\rho' \vdash \$a \xrightarrow{\mathcal{D}} \texttt{true}}\ Delay_d \qquad \dfrac{\begin{array}{c}\text{cf. tic 0}\\ \vdots\ Fby_d\\ 1,\emptyset \vdash \rho'(a) \xrightarrow{\mathcal{D}} \texttt{true}\end{array} \quad \begin{array}{c}\dfrac{a \notin \texttt{Dom}(\emptyset)}{1,\emptyset \vdash a \xrightarrow{\mathcal{T}} \texttt{false}}\ Id_t\\ \vdots\ Fby_t,\ldots\\ 1,\emptyset \vdash \rho'(a) \xrightarrow{\mathcal{T}} \texttt{false}\end{array}}{1,\rho' \vdash a \xrightarrow{\mathcal{T}} \texttt{false}}\ Id_t \qquad \dfrac{\begin{array}{c}\text{cf. tic 0}\\ \vdots\ Id_t\\ 0,\rho' \vdash a \xrightarrow{\mathcal{T}} \texttt{true}\end{array}}{OK(0,\rho,a) = \texttt{true}}\ OK_0}{1,\rho' \vdash \$a \xrightarrow{\mathcal{T}} \texttt{false}}\ Delay_t$$

□

XIII.6.3.3 Le système $\{a = 0\,\texttt{fby}(\$a + 1)\}$ définit un stream qui ne progresse pas

Par conséquent, l'horloge de A étant fausse au tic 1, le stream ne progresse pas. Il n'est pas nécessaire d'effectuer le calcul aux tics suivants car la preuve que nous venons de produire pour le tic 0 est utilisable pour tous les tics t tels que $t \geq 0$.

XIII.6.4 Calcul de $\{a = 0\,\texttt{fby}(\$a + 1\,\texttt{when}\,\texttt{Clock})\}$

Calculons la valeur de l'expression $A = \{a = 0\,\texttt{fby}(\$a + 1\,\texttt{when}\,\texttt{Clock})\}$ pour montrer que cette équation définit un stream qui progresse dans le temps (contrairement à l'expression précédente) et dont la valeur est incrémentée à chaque tic.

XIII.6.4.1 Tic $t = 0$

Au tic 0, l'équation A possède un top, car l'horloge de `fby` est identique à l'horloge que l'on vient de calculer pour l'expression précédente. De la même façon, sa valeur est égale à `[| 0 |]`.

XIII.6.4.2 Tic $t = 1$

Au tic 1, on calcule la valeur de A, sans détailler chaque étape. Nous séparons le calcul des preuves quand celles-ci apparaissent dans le calcul d'autre preuves ou quand il n'est pas possible de les définir en une seule fois (pour des raisons de place). La valeur au tic 1 de A est définie par la preuve (9).

Preuve 9 ***(Valeur de*** a ***au tic 1)***

$$
\cfrac{
\begin{array}{c}\text{cf. preuve (10)}\\ \vdots\ Fby_t\\ 1,\rho' \vdash A \xrightarrow{\mathcal{T}} \texttt{true}\end{array}
\quad
\begin{array}{c}\text{cf. preuve (17)}\\ \vdots\ Fby_v\\ 1,\rho' \vdash A \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}\end{array}
}{1,\emptyset \vdash \{a = 0\,\texttt{fby}(\$a+1\,\texttt{when}\,\texttt{Clock})\} \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}}\ \{sys\}_v
$$

où $\rho' = [a \mapsto 0\,\texttt{fby}(\$a + 1)\,\texttt{when}\,\texttt{Clock}]$. □

Preuve 10 ***(Horloge de*** A ***au tic 1)***

$$
\cfrac{
\cfrac{\cfrac{\cfrac{}{1,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{1,\rho' \vdash 0 \xrightarrow{\mathcal{D}} \texttt{true}}\ Fby_d}{1,\rho' \vdash A \xrightarrow{\mathcal{D}} \texttt{true}}
\quad
\begin{array}{c}\text{Cette branche n'est pas}\\ \text{sélectionnée}\\ \vdots\ Cte_t\\ 1,\rho' \vdash 0 \xrightarrow{\mathcal{T}} ?\end{array}
\quad
\begin{array}{c}\text{cf. preuve (11)}\\ \vdots\ When_t\\ 1,\rho' \vdash \$a + 1\,\texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{T}} \texttt{true}\end{array}
}{1,\rho' \vdash 0\,\texttt{fby}\,\$a + 1\,\texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{T}} ite(OK(0,\rho',0),\texttt{true},?) = \texttt{true}}\ Fby_t
$$

Il n'est pas nécessaire de calculer la valeur de l'horloge de la constante 0 au tic 1 (preuve de $1, \rho' \vdash 0 \xrightarrow{\mathcal{T}} ?$) car la branche sélectionnée par $OK(()0, \rho', 0)$ correspond à la preuve du calcul de l'horloge de $\$a + 1\,\texttt{when}\,\texttt{Clock}$ au tic 1. □

Preuve 11 ***(Horloge de*** $\$a + 1\,\texttt{when}\,\texttt{Clock}$ ***au tic 1)***

$$
\cfrac{
\begin{array}{c}\text{cf. (12)}\\ \vdots\ When_d\\ 0,\rho' \vdash \$a + 1\,\texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{D}} \texttt{false}\end{array}
\quad
\begin{array}{c}\text{cf. (13)}\\ \vdots\ Clock_t\\ 1,\rho' \vdash \texttt{Clock} \xrightarrow{\mathcal{T}} \texttt{true}\end{array}
\quad
\begin{array}{c}\text{cf. (14)}\\ \vdots\ Clock_v\\ 1,\rho' \vdash \texttt{Clock} \xrightarrow{\mathcal{V}} \texttt{true}\end{array}
}{1,\rho' \vdash \$a + 1\,\texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{T}} \texttt{true} \wedge \texttt{true} \equiv \texttt{true}}\ When_t
$$

□

Preuve 12 ***(Domaine de*** $\$a + 1\,\texttt{when}\,\texttt{Clock}$ ***au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (15)}\\ \vdots\ Clock_t\\ 0,\rho' \vdash \$a+1\ \texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{D}} \texttt{false}\end{array} \quad \begin{array}{c}\text{cf. (16)}\\ \vdots\ Delay_d\\ 1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{D}} \texttt{true}\end{array} \quad \begin{array}{c}\text{cf. (13)}\\ \vdots\ Clock_d\\ 1,\rho' \vdash \texttt{Clock} \xrightarrow{\mathcal{D}} \texttt{true}\end{array} \quad \begin{array}{c}\text{cf. (14)}\\ \vdots\ Clock_v\\ 1,\rho' \vdash \texttt{Clock} \xrightarrow{\mathcal{V}} \texttt{true}\end{array}}{0,\rho' \vdash \$a+1\ \texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{D}} \texttt{false}}\ When_d$$

□

Preuve 13 ***(Top de*** `Clock 1` ***au tic 1)***

$$\dfrac{\dfrac{}{1,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{D}} \texttt{true}(1 \neq 0) \equiv \texttt{true}}\ Clock_d}{\dfrac{1,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{D}} \texttt{true}}{1,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{T}} ((1\%1) == 0) \equiv \texttt{true}}\ Clock_t}$$

□

Preuve 14 ***(Valeur de*** `Clock 1` ***au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (13)}\\ \vdots\ Clock_t\\ 1,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{T}} \texttt{true}\end{array}}{1,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{V}} \texttt{[| true |]}}\ Clock_v$$

□

Preuve 15 ***(Domaine de*** $\$a+1$ `when Clock` ***au tic 1)***

$$\dfrac{\dfrac{\dfrac{}{0,\rho' \vdash \$a \xrightarrow{\mathcal{D}} \texttt{false}}\ Delay_d \quad \dfrac{}{0,\rho' \vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{0,\rho' \vdash \$a = 1 \xrightarrow{\mathcal{D}} (\texttt{false} \wedge \texttt{true}) \equiv \texttt{false}}\ Binaire_d \quad \begin{array}{c}\text{cf. (13)}\\ \vdots\ Clock_d\\ 0,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{D}} \texttt{true}\end{array} \quad \begin{array}{c}\text{cf. (14)}\\ \vdots\ Clock_v\\ 0,\rho' \vdash \texttt{Clock}\,1 \xrightarrow{\mathcal{V}} \texttt{true}\end{array}}{0,\rho' \vdash \$a+1\ \texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{D}} (\texttt{false} \wedge \texttt{true} \wedge \texttt{true}) \equiv \texttt{false}}\ When_d0$$

□

Preuve 16 ***(Domaine de*** $\$a+1$ ***au tic 1)***

$$\dfrac{\dfrac{\begin{array}{c}\text{cf. tic 0}\\ \vdots\ Id_d\\ 0,\rho' \vdash a \xrightarrow{\mathcal{D}} \texttt{true}\end{array}}{1,\rho' \vdash \$a \xrightarrow{\mathcal{D}} \texttt{true}}\ Delay_d \quad \dfrac{}{1,\rho' \vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{D}} (\texttt{true} \wedge \texttt{true}) \equiv \texttt{true}}\ Binaire_d$$

□

Preuve 17 ***(Valeur de*** A ***au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (10)}\\ \vdots\ Fby_t\\ 1,\rho' \vdash A \xrightarrow{\mathcal{T}} \texttt{true}\end{array} \quad \begin{array}{c}\text{cf. tic 0}\\ \vdots\ OK_0\\ OK(0,\rho',0) = \texttt{true}\end{array} \quad \dfrac{\begin{array}{c}\text{cf. (10)}\\ \vdots\ When_t\\ 1,\rho' \vdash \$a+1\ \texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{T}} \texttt{true}\end{array} \quad \begin{array}{c}\text{cf. (18)}\\ \vdots\ Binaire_v\\ 1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{V}} \texttt{+}(\texttt{[| 0 |]},\texttt{[| 1 |]}) \equiv \texttt{[| 1 |]}\end{array}}{1,\rho' \vdash \$a+1\ \texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}}\ When_v}{1,\rho' \vdash 0\ \texttt{fby}\ \$a+1\ \texttt{when}\,\texttt{Clock} \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}}\ Fby_vNext$$

□

Preuve 18 ***(Valeur de*** $\$a+1$ ***au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (19)}\\ \vdots\ Binaire_t\\ 1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{T}} \texttt{true}\end{array} \quad \begin{array}{c}\text{cf. (21)}\\ \vdots\ Delay_v\\ 1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}\end{array} \quad \begin{array}{c}\text{cf. (22)}\\ \vdots\ Defaut_v\\ 1,\rho' \vdash 1 \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}\end{array}}{1,\rho' \vdash \$a+1 \xrightarrow{\mathcal{V}} \texttt{+}(\texttt{[| 0 |]},\texttt{[| 1 |]}) \equiv \texttt{[| 1 |]}}\ Binaire_v$$

□

Preuve 19 ***(Horloge de** $\$a+1$ **au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (16)}\\ \vdots\ Binaire_d\\ 1,\rho'\vdash \$a+1 \xrightarrow{\mathcal{D}} \texttt{true}\end{array}\quad \begin{array}{c}\text{cf. (20)}\\ \vdots\ Delay_t\\ 1,\rho'\vdash \$a \xrightarrow{\mathcal{T}} \texttt{true}\end{array}\quad \dfrac{\dfrac{}{1,\rho'\vdash 1 \xrightarrow{\mathcal{T}}}\ Cte_d}{1,\rho'\vdash 1 \xrightarrow{\mathcal{T}} (1==0)\equiv \texttt{false}}\ Cte_t}{1,\rho'\vdash \$a+1 \xrightarrow{\mathcal{T}} (\texttt{true}\vee\texttt{false})\equiv\texttt{true}}\ Binaire_t$$

□

Preuve 20 ***(Horloge de** $\$a$ **au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (16)}\\ \vdots\ Delay_d\\ 1,\rho'\vdash \$a \xrightarrow{\mathcal{D}} \texttt{true}\end{array}\quad \begin{array}{c}\text{cf. tic 0}\\ \vdots\ OK_0\\ OK(0,\rho',a)=\texttt{true}\end{array}\quad \begin{array}{c}\text{cf. tic 0}\\ \vdots\ Id_t\\ 1,\rho'\vdash a \xrightarrow{\mathcal{T}} \texttt{true}\end{array}}{1,\rho'\vdash \$a \xrightarrow{\mathcal{T}} \texttt{true}}\ Delay_t$$

□

Preuve 21 ***(Valeur de** $\$a$ **au tic 1)***

$$\dfrac{\begin{array}{c}\text{cf. (19)}\\ \vdots\ Delay_t\\ 1,\rho'\vdash \$a \xrightarrow{\mathcal{T}} \texttt{true}\end{array}\quad \dfrac{\begin{array}{c}\text{cf. tic 0}\\ \vdots\ Fby_t\\ 0,\emptyset\vdash \rho'(a) \xrightarrow{\mathcal{T}} \texttt{true}\end{array}\quad \begin{array}{c}\text{cf. tic 0}\\ \vdots\ Fby_v\\ 0,\emptyset\vdash \rho'(a) \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}\end{array}}{0,\rho'\vdash a \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}}\ Id_v}{1,\rho'\vdash \$a \xrightarrow{\mathcal{V}} \texttt{[| 0 |]}}\ Delay_v$$

□

Preuve 22 ***(Valeur de la constante 1 au tic 1)***

$$\dfrac{\dfrac{\dfrac{}{1,\rho'\vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{1,\rho'\vdash 1 \xrightarrow{\mathcal{T}} (1\neq 0)\equiv\texttt{false}}\ Cte_t\quad \dfrac{\dfrac{\dfrac{}{0,\rho'\vdash 1 \xrightarrow{\mathcal{D}} \texttt{true}}\ Cte_d}{0,\rho'\vdash 1 \xrightarrow{\mathcal{T}} (0==0)\equiv\texttt{true}}\ Cte_t}{0,\rho'\vdash 1 \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}}\ Cte_v}{1,\rho'\vdash 1 \xrightarrow{\mathcal{V}} \texttt{[| 1 |]}}\ Defaut_v$$

□

XIII.6.4.3 Le système $\{a = 0\,\texttt{fby}(\$a+1)\,\texttt{when}\,\texttt{Clock}\}$ définit un stream qui progresse

Par conséquent, l'horloge de A étant vraie au tic 1, du fait de la quantification par `Clock`, le stream progresse au rythme de `Clock` et a pour valeur 1. De la même façon que pour l'exemple précédent, il n'est pas nécessaire d'effectuer le calcul aux tics suivants car la preuve que l'on vient de produire pour le tic 0 est utilisable pour tous les tics t tels que $t > 0$ où seule la valeur de $\$a$ change et est égale à la valeur du stream au tic $t-1$.

XIII.6.5 L'horloge de sx avec $sx = x\,\texttt{fby}(x+\$sx)$ est la même que celle de x

L'horloge de l'expression :

$$sx = x\,\texttt{fby}(x+\$sx)$$

est la même que celle de x :

1. pour les tics avant le premier top, et celui-ci inclus : nous avons vu que l'horloge de l'expression sx était l'horloge de l'argument gauche de `fby` (cf. la règle Fby_t et les deux exemples ci-dessus).

2. pour tous les autres tics : l'horloge de sx est la disjonction des horloges de x et $\$sx$. Elle inclue donc l'horloge de x et se réduit à celle-ci car l'horloge de sx est liée à `false` lors de son calcul dans l'environnement dû à la liaison $sx = \ldots$.

Chapitre XIV

Éléments d'implémentation et exemples

Dans ce chapitre nous esquissons des éléments de l'implémentation en CAML de la sémantique présentée au chapitre précédent. Nous donnons ensuite des exemples pris dans trois domaines d'application très différents :

- structure de données dynamiques pour la combinatoire,
- calcul symbolique,
- modélisation de processus de croissance.

Ces exemples complètent les exemples illustrant les amalgames et les GBF, et montrent l'intégration de ces deux nouveaux concepts dans $81/2_{\mathcal{D}}$. Tous les exemples de ce chapitre ont été évalués par la version présentée de l'interprète.

XIV.1 Éléments d'implémentation

Nous donnons quelques éléments de l'implémentation des règles d'évaluation de $81/2_{\mathcal{D}}$ en CAML. Cette implémentation est en cours. Tous les exemples de ce chapitre ont été évalués par le prototype dans son état actuel.

XIV.1.1 Principe de l'implémentation CAML

Une règle de réduction de la forme :

$$t, \rho \vdash e \xrightarrow{x} v$$

se traduit en une fonction récursive de la forme :

```
let rec fX(t, ρ, e) = v
```

La fonction $\mathtt{f_X}$ est définie par induction sur la structure de l'arbre syntaxique des expressions en CAML. L'évaluation de l'expression procède par décorations successives de l'arbre, ce qui nous permet de :

- supprimer ρ par une passe consécutive à la création de l'arbre qui associe à chaque occurrence d'un identificateur dans une expression sa définition. Cette phase de *liaison* intervient après la création de l'arbre et avant l'évaluation de l'expression.
- *mémoïser* [LT95] $\mathtt{f_X}$ en décorant l'arbre avec les valeurs de $\mathtt{f_X}$ pour le tic t courant et pour $t-1$ (qui apparaît parfois dans le calcul de v).
- calculer et de mémoïser certains autres prédicats (comme $OK()$, ...).

XIV.1.2 Le jeu d'instruction de la machine virtuelle

Les principales caractéristiques de la machine virtuelle sont :

- un jeu d'instructions extrêmement réduit (8 instructions),
- elle est *générique* : en effet, les fonctions sont paramétrées par la signature (la géométrie) de l'opération. L'opération notée $\mathtt{unop}_{g_e \to g}(v)$ correspond en fait à la fonction $\mathtt{unop}(op, g_e, g, v)$.
- elle est facilement SIMD-isable ou vectorisable

Nous donnons à titre indicatif l'ensemble des opérations de la machine virtuelle tableau avec leur équivalent :

e_1 **.** n	:	$\mathtt{SelInt}_{g_1 \to g_2}(v, n)$
$e_1 \#_n e_2$	:	$\mathtt{\#}_{g_1 \times g_2 \to g}(v_1, v_2, n)$
`if` *si* `then` *alors* `else` *sinon*	:	$\mathtt{ite}_{g \times g \times g \to g}(v_{si}, v_{alors}, v_{sinon})$
$e : [n]$	:	$\mathtt{Shrink}_{g_e \to g}(e)$
$unop(e)$	:	$\mathtt{unop}_{g_e \to g}(v)$
e_1 *binop* e_2	:	$\mathtt{Bop}_{g_1 \times g_2 \to g}(v_1, v_2)$
`switch` *si* `then` *alors* `else` *sinon*	:	$\mathtt{Switch}_{g_{si} \times g_{alors} \times g_{sinon} \to g}(v_{si}, v_{alors}, v_{sinon})$
$e : f$	:	$\mathtt{Shrink}_{g_e \to g}(v)$

L'opérateur de calcul de rang d'une collection n'apparaît pas car il correspond à un traitement en CAML de la géométrie d'une expression.

XIV.2 Structures de données dynamiques pour la combinatoire

XIV.2.1 Les « tableaux chinois »

Nous décrivons dans cette section une méthode originale utilisée pour calculer quelques suites classiques.

Le mathématicien chinois Li Shanlan a défini une méthode pour calculer des suites de nombres de façon « quasi automatique ». Sa méthode est similaire à la méthode utilisée pour le calcul du triangle de Pascal : une série de nombres est représentée sous une forme pyramidale où chaque élément est calculé par une fonction simple de ses voisins et prédécesseurs. Nous ne détaillerons pas ici la méthode utilisée pour définir ces structures, le lecteur intéressé se reportera à [Mar94, pp 96–106]. Nous donnons dans la figure 1 un exemple d'application de la méthode de Shanlan pour le calcul des nombres de Stirling et d'Euler. Nous décrivons dans les sections suivantes une implémentation directe en $8_{1/2}\mathcal{D}$ de ces calculs.

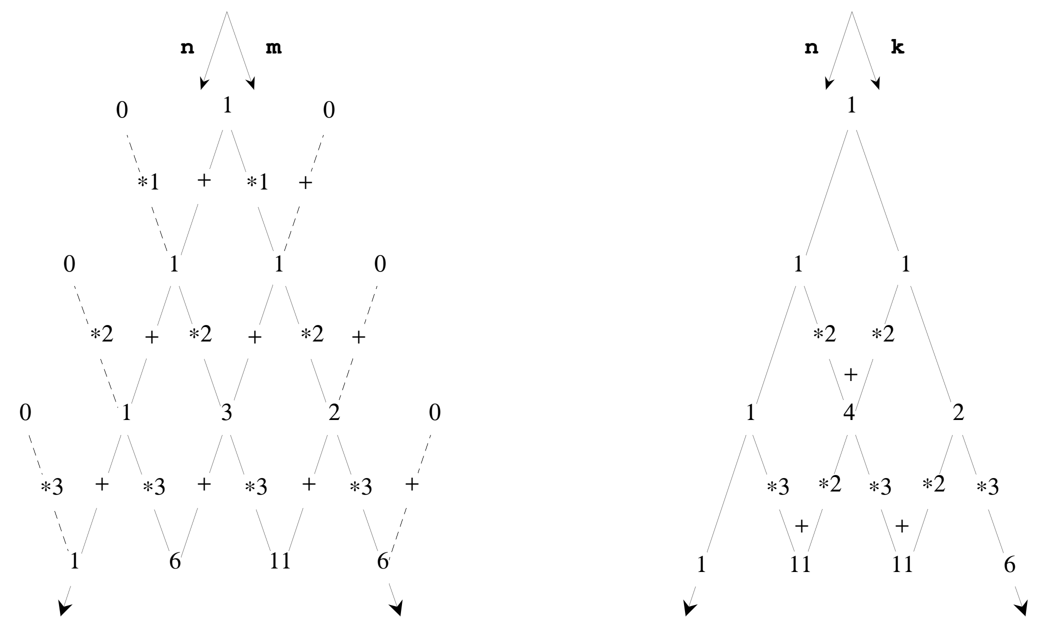
XIV.2.1.1 Calcul des coefficients binomiaux par la méthode du triangle de Pascal

Les coefficients binomiaux C_n^p peuvent être calculés de façon récursive : la valeur du point $(ligne, col)$ dans le triangle est la somme des points du triangle d'indice $(line-1, col)$ et $(line-1, col-1)$. On représente le triangle de Pascal en prenant pour les lignes l'aspect stream d'un tissu (les lignes du triangle de Pascal seront calculées dans le temps), les colonnes étant représentées par l'aspect collection du tissu. Le nombre de colonnes augmentant dans le temps, il est nécessaire d'avoir des tissus où la collection peut varier de taille à l'exécution. Chaque valeur du triangle de Pascal est définie par les relations de récurrences suivantes :

$$P(0,0) = 1 \quad (1)$$
$$P(i,i) = 1 \quad (2)$$
$$P(i,j) = P(i-1,j) + P(i-1,j-1) \quad (3)$$

Remarquons que la ligne l (telle que $l > 0$) est la somme de la ligne $(l-1)$ concaténée à droite avec 0 et de 0 concaténé à gauche avec la ligne $(l-1)$. Il nous est donc possible de définir une version du triangle de Pascal à l'aide des seuls opérateurs de concaténation et de délai. Le programme $8_{1/2}\mathcal{D}$ correspondant est :

```
t@0 = 1;
t   = ($t # 0) + (0 # $t) when Clock;
```



(a) Les 4 premières étapes du calcul des nombres de STIRLING.

(b) Les 4 premières étapes du calcul des nombres d'EULER.

FIG. 1 –: *Calcul des suites récurrentes par la méthode de Li* SHANLAN. *Exemple du calcul des nombres de* STIRLING *et d'*EULER.

L'initialisation de la collection à la valeur `1` à l'instant 0 provient de (1) et (2) ; l'équation universellement quantifiée exprime quant à elle la relation définie par (3). Les 5 premières valeurs du triangle de PASCAL sont :

```
Top : 0 : { 1 } : int[1]
Top : 1 : { 1, 1 } : int[2]
Top : 2 : { 1, 2, 1 } : int[3]
Top : 3 : { 1, 3, 3, 1 } : int[4]
Top : 4 : { 1, 4, 6, 4, 1 } : int[5]
```

XIV.2.1.2 Les nombres de Stirling

Les *nombres de* STIRLING de première espèce, $S(n,m)$, ont été définis par J. STIRLING et permettent, entre autre, le calcul les coefficients du développement du polynôme :

$$f(x) = x(x+1)\dots(x+n-1)$$

Les nombres de STIRLING ont de nombreuses applications en mathématiques ; ils servent par exemple dans certains algorithmes de multiplication en multiprécision, ils sont liés aux « nombres harmoniques » et ils ont des interprétations combinatoires (partitions d'ensembles). La suite des nombres de STIRLING est définie par les équations récurrentes suivantes :

$$S(0,0) = 1 \quad (4)$$
$$S(n,0) = 0 \qquad n \neq 0 \quad (5)$$
$$S(0,m) = 1 \qquad m \neq 0 \quad (6)$$
$$S(n,m) = (n-1)\,S(n-1,m) + S(n-1,m-1) \qquad n,m \neq 0 \quad (7)$$

Le calcul des nombres de STIRLING en $81/2_{\mathcal{D}}$ peut être réalisé en calculant n dans le temps et en associant le calcul de m à un vecteur. La figure 1 détaille la méthode de calcul des nombres de STIRLING. Le programme

$8_{1/2\mathcal{D}}$ correspondant est :

$$
\begin{array}{lcll}
s@0 & = & 0; & /*\text{ de (4) } */\\
s@1 & = & 1\,\texttt{when}\,Clock; & /*\text{ de (4) } */\\
s & = & (|\$f| * \$f) + (\$s \,\#\, 0); & /*\text{ de (7) } */\\
f@0 & = & 1; & /*\text{ de (6) } */\\
f & = & 1\#s; & /*\text{ de (6) } */
\end{array}
$$

L'opérateur $|x|$ calcule le rang du tissu x, c'est-à-dire un vecteur où l'élément i correspond au nombre d'éléments de la i^{e} dimension de x. Par conséquent, l'expression $|\$f|$ se comporte comme un compteur et correspond au coefficient $(n-1)$ utilisé pour multiplier chaque ligne précédente afin d'obtenir la ligne courante. Les 5 premières lignes du triangle de STIRLING sont :

```
Top: 0 : { 1 } : int[1]
Top: 1 : { 1, 1 } : int[2]
Top: 2 : { 1, 3, 2 } : int[3]
Top: 3 : { 1, 6, 11, 6 } : int[4]
Top: 4 : { 1, 10, 35, 50, 24 } : int[5]
```

XIV.2.1.3 Les nombres d'Euler

Les *nombres d'*EULER, $E(n, k)$, ont été définis par L. EULER. Ils permettent de calculer le nombre de permutations à n éléments se décomposant en p tranches décroissantes de longueur maximale. Par exemple, pour l'ensemble des six permutations de trois éléments $\{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$, il existe exactement quatre permutations qui sont formées de deux tranches décroissantes : $(1/3,2)$, $(4,1/3)$, $(2/3,1)$, $(3/2,1)$. Par conséquent, $E(3,2) = 4$. La suite des nombres d'EULER est définie par les équations récurrentes suivantes :

$$
\begin{array}{lll}
E(0,1) & = 1 & \\
E(0,k) & = 0 & n \neq 0\\
E(n,k) & = k\,E(n-1,k) + (n-k+1)\,E(n-1,k-1) & n,m \neq 0
\end{array}
\qquad
\begin{array}{r}(8)\\(9)\\(10)\end{array}
$$

L'implémentation du calcul des nombres d'EULER en $8_{1/2\mathcal{D}}$ suit le schéma que l'on a défini pour les nombres de STIRLING : on effectue le calcul de n dans le temps et on associe au calcul de k un vecteur. La figure 1 détaille la méthode de calcul des nombres d'EULER. Le programme $8_{1/2\mathcal{D}}$ correspondant est :

$$
\begin{array}{lcll}
iota & = & (\$iota \,\#\, (|\$iota| \,+\, 1))\,\texttt{when}\,Clock; & /*\text{ calcul de } k \text{ pour (10) } */\\
iota@0 & = & 1; & \\
riota & = & ((|\$riota| \,+\, 1) \,\#\, \$riota)\,\texttt{when}\,Clock; & /*\text{ calcul de } (n-k+1) \text{ pour (10) } */\\
riota@0 & = & 1; & \\
e & = & (0 \,\#\, (\$riota * \$e)) \,+\, ((\$iota * \$e) \,\#\, 0); & /*\text{ de (10) } */\\
e@0 & = & 1; & /*\text{ de (8) \& (9) } */
\end{array}
$$

La valeur de *iota* au tic n est le vecteur dont les éléments sont $1\dots(n-1)$. L'expression *riota* est la version « retournée » de *iota* (un vecteur de valeurs $(n-1)\dots 1$). Enfin, l'expression e représente le vecteur des nombres d'EULER. Les 5 premières valeurs du triangle d'Euler sont :

```
Top: 0 : { 1 } : int[1]
Top: 1 : { 1, 1 } : int[2]
Top: 2 : { 1, 4, 1 } : int[3]
Top: 3 : { 1, 11, 11, 1 } : int[4]
Top: 4 : { 1, 26, 66, 26, 1 } : int[5]
```

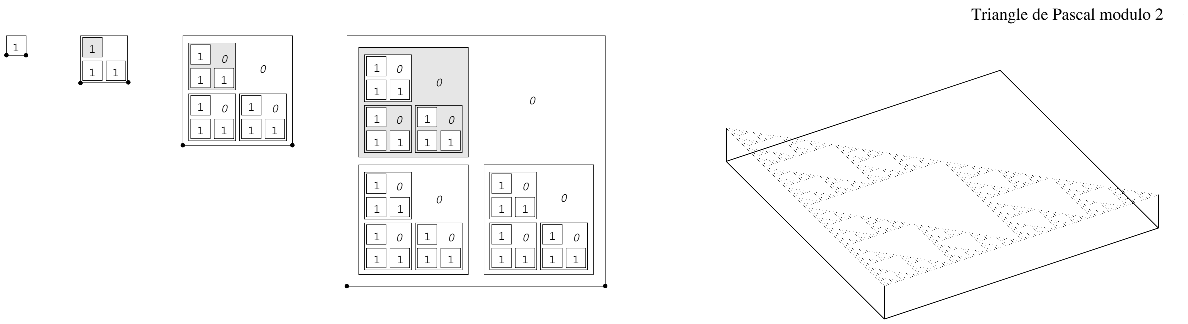
XIV.2.2 Le triangle de Pascal modulo 2

Nous revenons dans cette section sur le triangle de PASCAL. On peut s'intéresser à la parité des nombres qui figurent dans le triangle : on obtient ainsi un triangle de 0 et de 1, le triangle de PASCAL *modulo* 2. Ce triangle possède une structure récursive décrite par la figure 2.a. Si T est un triangle de Pascal modulo 2, alors le triangle obtenu en concaténant deux fois T à lui même de la manière décrite ci-après est encore un triangle de PASCAL modulo 2. La construction du nouveau triangle s'obtient en concaténant T une première fois en alignant le coin supérieur gauche de T sous le coin inférieur gauche, et en concaténant T une seconde fois au résultat en alignant le coin supérieur gauche de T sous le coin inférieur droit.

De manière plus générale, le triangle de PASCAL modulo n présente la même structure récursive si n est un nombre premier. Cette méthode de construction est intéressante car en n étapes on obtient un triangle de 2^n lignes.

Le programme $8\,1/2_{\mathcal{D}}$ correspondant manipule le carré qui englobe un triangle (en effet, on s'est restreint à ne manipuler que des régions rectangulaires). Le programme est très simple et traduit exactement le processus de construction :

```
p   = (($p # 0:|$p|) #^ ($p # $p)) when Clock;
p@0 = 1:[1, 1];
```



(a) Construction récursive du triangle de PASCAL modulo 2. Les points correspondent aux sommets où on réplique le triangle. Le triangle initial de la construction figure en grisé. On a figuré quatre étapes successives de construction.

(b) Résultat des 7 premières étapes du calcul du triangle de PASCAL modulo 2. L'affichage utilise l'interface avec Gnuplot que nous avons développé.

FIG. 2 –: *Le calcul du triangle de* PASCAL *modulo 2. Détail et résultat de l'exécution du programme* $8\,1/2_{\mathcal{D}}$.

L'évaluation de ce programme pour les 3 premiers tics a pour résultat :

```
Top: 0 : { { 1 } } : int[1, 1]

Top: 1 : { { 1, 0 }, { 1, 1 } } : int[2, 2]

Top: 2 :{ { 1, 0, 0, 0 },
          { 1, 1, 0, 0 },
          { 1, 0, 1, 0 },
          { 1, 1, 1, 1 } } : int[4, 4]

Top: 3 : { { 1, 0, 0, 0, 0, 0, 0, 0 },
           { 1, 1, 0, 0, 0, 0, 0, 0 },
           { 1, 0, 1, 0, 0, 0, 0, 0 },
           { 1, 1, 1, 1, 0, 0, 0, 0 },
           { 1, 0, 0, 0, 1, 0, 0, 0 },
           { 1, 1, 0, 0, 1, 1, 0, 0 },
           { 1, 0, 1, 0, 1, 0, 1, 0 },
           { 1, 1, 1, 1, 1, 1, 1, 1 } } : int[8, 8]
```

XIV.3 Calcul symbolique en $8_{1/2}{}_{\mathcal{D}}$

L'utilisation conjointe des amalgames et des streams $8_{1/2}{}_{\mathcal{D}}$ va permettre d'effectuer des opérations de calcul symbolique en $8_{1/2}$. En effet, une expression ouverte correspond à un morceau d'arbre syntaxique soit encore à une expression *symbolique*. Les trois sections suivantes illustrent l'utilisation d'expressions symboliques en $8_{1/2}{}_{\mathcal{D}}$.

XIV.3.1 Calcul par développement limité

Un grand nombre de suites mathématiques nécessitent le calcul symbolique d'une première suite de valeurs (par exemple, un développement limité suivant la méthode des développements en séries de TAYLOR) puis l'instanciation de cette suite avec des valeurs numériques fournissant des valeurs initiales, afin d'obtenir la suite désirée. Nous reproduisons cette approche ici à travers le calcul formel du développement limité de l'exponentielle :

$$\begin{aligned} e^x &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \dots \end{aligned}$$

Chaque terme de la somme correspond au terme précédent multiplié par x et divisé par un entier croissant. Nous utilisons ce schéma pour calculer la suite des sommes partielles : à un instant i on dispose d'une approximation correspondant à la somme des n premiers termes. A chaque itération, on ajoute un terme à la suite : le terme qui est ajouté est le dernier terme multiplié par x et divisé par la factorielle du numéro d'itération courant.

Le codage en $8_{1/2}{}_{\mathcal{D}}$ utilise la dimension stream du tissu pour représenter les génération successives des termes et utilise les amalgames pour représenter les termes. On définit une expression $fact$ qui calcule $n!$ où n est le numéro de la génération. L'expression $terme$ représente le terme en x et est égal, à chaque génération au terme de la génération précédente multiplié par x. Enfin, l'expression exp représente le calcul de l'exponentielle qui est égal au terme de la génération précédente plus le terme courant divisé par la factorielle courante. Le programme $8_{1/2}{}_{\mathcal{D}}$ est :

$$\begin{array}{lllll} n@0 & = & 0.0; & n & = \$n + 1.0 \texttt{ when Clock}; \\ fact@0 & = & 1.0; & fact & = n * \$fact \texttt{ when Clock}; \\ terme@0 & = & 1.0; & terme & = (\$terme * x) \texttt{ when Clock}; \\ exp@0 & = & 1.0; & exp & = (\$exp + terme/fact) \texttt{ when Clock}; \end{array}$$

La référence x reste symbolique car aucun definiendum de nom x n'existe. Les 4 premières valeurs de la suite sont :

```
{ 1 } : float[1]
{ 1 } + (({ 1 } * x) / { 1 })
({ 1 } + (({ 1 } * x) / { 1 })) + ((({ 1 } * x) * x) / { 2 })
(({ 1 } + (({ 1 } * x) / { 1 })) + ((({ 1 } * x) * x) / { 2 })) + (((({ 1 } * x) * x) * x) / { 6 })
```

En définissant une expression qui fournit une valeur à la référence libre x : $v = \{x = 1.0, r = exp\}$ on obtient la réduction de l'expression symbolique :

```
{ 1 } : float[1]
{ 1, 1 } : float[2]
{ 1, 2 } : float[2]
{ 1, 2.5 } : float[2]
{ 1, 2.66667 } : float[2]
{ 1, 2.70833 } : float[2]
{ 1, 2.71667 } : float[2]
{ 1, 2.71806 } : float[2]
{ 1, 2.71825 } : float[2]
{ 1, 2.71828 } : float[2]
```

On trouvera dans [GG91] un ensemble de calculs symboliques utilisant le formalisme des L systèmes et qui sont directement implémentables en $8_{1/2}{}_{\mathcal{D}}$ en utilisant le formalisme des amalgames.

XIV.3.2 Structures de données régulières et calcul de fonctions récursives primitives

Nous souhaitons calculer au temps t, les t premières valeurs d'une fonction récursive primitive. Rappelons qu'une fonction f est dite *récursive primitive* si la valeur de f pour l'entier n dépend de la valeur de f pour des entiers n' tels que $n' < n$.

L'exemple ci-après utilise l'opérateur de coercion, pour implémenter la sélection du n^{e} élément d'un vecteur. Quand la taille de la collection est supérieure à la valeur de la coercion, alors une troncature est effectuée. Si l'argument est négatif, cette troncature commence par la fin de la collection[1]. La fonction `sel` a pour valeur le 10^{e} élément (en partant de la fin) de la collection passée en argument. Le stream `fibo` est un stream, dont le nombre d'éléments augmente à chaque top, et dont la valeur concaténée est égale à la somme des dernières et avant dernières valeurs de la collection. Le stream `fibo` correspond à l'implémentation du calcul de la suite de Fibonacci. Le programme $8_{1/2\mathcal{D}}$ est :

$$functionsel(a, idx) = (a : \{-idx - 1\}).0;$$

$$fibo@0 = \{1, 1\}; fibo = \$fibo\,\#\,\{\,\mathtt{sel}(\$fibo, 0) + \mathtt{sel}(\$fibo, 1)\}\,\mathtt{when\,Clock};$$

Les 5 premières valeurs de la suite calculée par `fibo` sont :

```
Top: 0 : { 1, 1 } : int[2]
Top: 1 : { 1, 1, 2 } : int[3]
Top: 2 : { 1, 1, 2, 3 } : int[4]
Top: 3 : { 1, 1, 2, 3, 5 } : int[5]
Top: 4 : { 1, 1, 2, 3, 5, 8 } : int[6]
```

Remarque Une variante du programme précédant utilisant deux pas de récursion dans le temps pour accéder aux deux dernières valeurs du tissu *fibo*, est :

$$\begin{array}{lcl} fibo@0 & = & \{1\}; \\ fibo@1 & = & \$fibo\,\#\,\$fibo\,\mathtt{when\,Clock}; \\ fibo & = & \$fibo\,\#\,\{sel(\$fibo, 0) + sel(\$\$fibo, 0)\}\,\mathtt{when\,Clock}; \end{array}$$

XIV.3.3 Factorisation d'un calcul par une expression symbolique et instanciation multiple de cette expression

L'exemple de la section XIV.3.1 utilisait les amalgames pour permettre une définition symbolique d'une série mathématique puis l'instanciation de cette suite pour obtenir la *valeur* du calcul.

On peut aussi vouloir instancier plusieurs fois une suite formelle avec des valeurs initiales différentes pour montrer, par exemple, que la valeur de la suite n'est pas sensible aux conditions initiales. Prenons par exemple la suite de fibonnaci dont le rapport entre le $(n+1)^{e}$ et le n^{e} terme tend vers le nombre d'or $(1+\sqrt{5})/2$, soit approximativement 1.618. Nous pouvons vérifier cette propriété de la suite en définissant une expression symbolique représentant les $(n+1)^{e}$ et n^{e} termes de la suite et calculer leur rapport pour des valeurs initiales différentes. Pour ce faire, on défini le programme $8_{1/2\mathcal{D}}$ suivant :

$$\begin{array}{lcl} fib & = & \$fib + \$\$fib\,\mathtt{when\,Clock}; \\ fib@0 & = & f0; \\ fib@1 & = & f1\,\mathtt{when\,Clock}; \end{array}$$

où les références libres $f0$ et $f1$ vont permettre un calcul symbolique des termes de la suite. La valeur du n^{e} calcul de fib sera égal au n^{e} terme de la suite. Nous donnons les 7 premières valeurs de la suite :

```
s0
s1
s1 + s0
(s1 + s0) + s1
```

1. Ce fonctionnement de la troncature n'est pas pris en compte par la sémantique que nous avons proposé. Son intégration est cependant immédiate et ne présente aucune difficulté majeure.

```
((s1 + s0) + s1) + (s1 + s0)
(((s1 + s0) + s1) + (s1 + s0)) + ((s1 + s0) + s1)
((((s1 + s0) + s1) + (s1 + s0)) + ((s1 + s0) + s1)) + (((s1 + s0) + s1) + (s1 + s0))
```

On remarquera que les valeurs calculées pour chaque instant représentent l'arbre de calcul de la suite. Pour calculer la valeur du rapport $fib(n+1)/fib(n)$ pour $n = 9$, on définit l'expression suivante :

$$\begin{array}{lcl} fibo & = & \{f = fib/\$fib; \\ & & \quad v1 = \{s0 = 1.0; s1 = 1.0\} \;.\; f, \\ & & \quad v2 = \{s0 = 2.0; s1 = 3.0\} \;.\; f\} \;\texttt{when}\,\texttt{Clock}\,10; \end{array}$$

dont l'horloge `when Clock` 10 ne possède un top que tous les 10 tics. On instancie la suite fib avec deux valeurs différentes dans deux systèmes différents : la suite n'est calculée qu'une seule fois mais elle est réduite avec des valeurs initiales différentes. Le résultat au 2^{e} top (correspondant au 10^{e} tic) est :

```
{ ... , 1.61818, 1.61806 }
```

Cette méthode factorise les calcul d'un arbre de calculs fonctionnels et elle peut être utilisée dans un grand nombre de calculs du même type. Par exemple, pour le précédent calcul de fib, une fois que le calcul initial de l'« arbre des calculs » est effectué, il est possible d'obtenir de nombreux résultats par réductions successives, dans différents environnements, de l'expression symbolique. En utilisant la forme symbolique de la série qui calcule l'exponentielle, on peut calculer l'exponentielle d'arguments différents.

Remarquons qu'il est possible d'effectuer des *évaluations partielles* d'expressions en ne fournissant qu'un certain nombre de définitions pour les références libres. Suivant la stratégie d'évaluation des expressions composées que nous avons adoptée, la réduction des expressions impliquant des tableaux se fait *au plus tôt* (cf. section XII.3.1, page 181).

XIV.4 Exemples de modélisations et de simulations de structures croissantes

XIV.4.1 Le crible d'Eratosthenes

Le crible d'ERATOSTHENES est un exemple paradigmatique de programmation par l'utilisation de tâches dynamiques dans le cadre de programmation concurrente. Une tâche est associée à chaque nouveau nombre premier généré et est connectée aux tâches précédemment définies. Ces tâche jouent le rôle d'un filtre. À chaque génération d'un nombre, celui ci traverse le filtre ainsi créé et s'il arrive effectivement en queue de la file, c'est que c'est un nombre premier.

Nous décrivons une solution alternative, dans un style plus « data-parallèle », en utilisant des tableaux dynamiques. Le programme que nous définissons consiste en un générateur de nombres croissants et en un tableau de nombres premiers. À l'initialisation, le tableau des nombres premiers est initialisé avec la seule valeur 2. Lors de la génération d'un nombre, s'il n'est divisible par aucun des nombres premiers c'est qu'il est lui-même un nombre premier, et on le concatène dans ce cas au tableau des nombres premiers. Cette croissance du tableau résultat permet une évolution dynamique du tableau des nombres premiers, en fonction d'un calcul.

La table 1 détaille les 6 premiers instants de l'exécution du programme $8_{1/2\mathcal{D}}$ suivant :

$$\begin{array}{lcl} generateur@0 & = & 2; \qquad generateur = \$generateur + 1 \;\texttt{when}\,\texttt{Clock}; \\ extension & = & generateur : |\$crible|; \\ modulo & = & extension \;\%\; \$crible; \\ zero & = & (modulo == (0 : |modulo|)); \\ reduit & = & or\backslash zero; \\ crible@0 & = & generateur; \qquad crible = \$crible \,\#\, generateur \;\texttt{when}(\texttt{not}\, reduit); \end{array}$$

Le tissu entier croissant *generateur* est étendu à la même taille que le tableau *crible* contenant les nombres premiers déjà calculés. Le tissu *modulo* correspond au reste de la division du nombre généré par tous les nombres premiers. Les éléments du tissu booléen *zero* sont vrais quand le nombre généré est divisible par un des nombres premiers déjà connus. Enfin, le tissu *reduit* est égal à la β-réduction du tissu booléen : si *reduit* est faux, alors c'est que le nombre généré n'est divisible par aucun des nombres premiers connus. Dans ce cas, le nombre généré est concaténé au tableau des nombres premiers.

Top	0	1	2	3	4	5
generateur	{2}	{3}	{4}	{5}	{6}	{7}
extension		{3}	{4, 4}	{5, 5}	{6, 6, 6}	{7, 7, 7}
modulo		{1}	{0, 1}	{1, 2}	{0, 0, 1}	{1, 1, 2}
zero		{0}	{1, 0}	{0, 0}	{1, 1, 0}	{0, 0, 0}
reduit		{0}	{1}	{0}	{1}	{0}
crible	{2}	{2, 3}	{2, 3}	{2, 3, 5}	{2, 3, 5}	{2, 3, 5, 7}

TAB. 1 –: *Les 6 premières étapes du calcul des nombres premiers par la méthode du crible d'*ERATOSTHENES.

XIV.4.2 Un modèle de croissance, les D0L systèmes

XIV.4.2.1 Présentation des L systèmes

Les *L systèmes* sont un formalisme de réécriture *parallèle* développé par A. LINDENMAYER en 1968 [Lin68]. Ce système de réécriture est parallèle, car contrairement aux systèmes de réécriture classiques où un seul terme est réécrit lors d'une étape de réécriture, tous les termes pour lesquels il existe une règle de production sont réécrits. Depuis sa définition, ce formalisme a été l'objet d'une étude intensive et a été utilisé dans des domaines aussi variés que la description d'interactions cellulaires [Lin68] ou la spécification d'un modèle de calculs parallèles [PH92].

Un *L système* est un triplet $G = (\Sigma, h, \omega)$ où Σ est un alphabet, h une substitution finie de Σ (sur l'ensemble des sous-ensembles de Σ^*) et ω, appelé *l'axiome* du système, est un élément de Σ^+. Le processus de dérivation parallèle utilisé dans les L systèmes permet la description de phénomènes évoluant simultanément dans le temps et dans l'espace (processus de croissance, description et simulation du développement de plantes, création dynamique de graphes [Boe95], etc.). Les L systèmes, depuis leur définition pour la spécification de processus de croissance [Lin68], ont montré qu'ils étaient un paradigme efficace pour la description de nombreux systèmes dynamiques (par exemple, dans le domaine des images de synthèse [PLH+90], le calcul symbolique [GG91], la modélisation du développement cellulaire [dBFP92]...). La puissance du formalisme des L systèmes provient de sa généricité : les L systèmes sont un formalisme de réécriture de termes. L'interprétation des termes générés lors des dérivations de l'axiome dépend de la nature du processus que l'on désire modéliser. Chaque sémantique différente associée aux éléments de Σ permet de décrire l'évolution dans le temps (l'espace des dérivations de l'axiome) et dans l'espace (l'espace du terme dérivé) d'un système dynamique.

Nous allons nous restreindre à la forme la plus simple des L systèmes : les D0L systèmes. La lettre « D » signifie que le système de réécriture est déterministe, c'est-à-dire qu'il n'existe au plus qu'une règle de production pour chaque élément de Σ. Nous sommes ainsi assurés que la séquence de dérivation est unique, alors que dans les L systèmes non déterministes, où il est possible d'appliquer à chaque étape de dérivation plus d'une règle, il existe plus d'une séquence de dérivation. L'argument numérique du L système spécifie le nombre d'interactions dans le processus de réécriture ; ici, les 0L systèmes sont « context free ».

Nous allons montrer que le formalisme des D0L systèmes peut être facilement implémenté en $8_{1/2\mathcal{D}}$.

XIV.4.2.2 Une traduction des D0L systèmes en $8_{1/2\mathcal{D}}$

Nous sommes intéressés par la description du processus de dérivation en $8_{1/2\mathcal{D}}$ [Mic96b] des D0L systèmes. La dérivation $H(w)$ d'un mot $w \in \Sigma^*$ est définie de la façon suivante :

$$\begin{array}{lll} H(a) & = h(a) & \text{pour } a \in \Sigma \\ H(u \,.\, v) & = H(u) \,.\, H(v) & \text{pour } u, v \in \Sigma^+ \end{array}$$

où H est l'homomorphisme étendant h. Une traduction d'un D0L système en $8_{1/2\mathcal{D}}$ est basée sur le principe suivant :

- un mot est un *vecteur* d'éléments de Σ,
- un tissu W est associé à l'axiome ω et la valeur $W(t)$ de W au temps t est $H^t(\omega)$.

Pour implémenter ce mécanisme en $8_{1/2\mathcal{D}}$, chaque règle définissant h est traduite en deux équations $8_{1/2\mathcal{D}}$ suivant la règle de traduction suivante :

$$a \rightarrow a_1 \ldots a_n \text{ se traduit en } \begin{cases} A@0 = {}'a' \\ A = (\$A_1 \,\#\, \ldots \,\#\, \$A_n)\, \texttt{when}\, Clock \end{cases} \tag{11}$$

et un mot $w = b_1 \ldots b_n$ se traduit en $W = B_1 \,\#\, \ldots \,\#\, B_n$.

Nous montrons que pour chaque mot w, $W(t) = H^t(w)$. Montrons d'abord par récurrence sur t que $A(t) = H^t(a)$. Cette propriété est vraie pour $t = 0$. Supposons la propriété vraie pour $t' < t$. Alors,

$$\begin{array}{rcl} A(t) & = & (\$A_1)(t) \,\#\, \ldots \,\#\, (\$A_n)(t) \\ \text{(par la propriété de \$)} & = & A_1(t-1) \,\#\, \ldots \,\#\, A_n(t-1) \\ \text{(par hypothèse)} & = & H^{t-1}(a_1) \,\centerdot\, \ldots \,\centerdot\, H^{t-1}(a_n) \\ (H \text{ homomorphisme}) & = & H^{t-1}(a_1 \ldots a_n) \\ & = & H^{t-1}(H(a)) \\ & = & H^t(a) \end{array}$$

De plus,

$$\begin{array}{rcl} W(t) & = & B_1(t) \,\#\, \ldots \,\#\, B_n(t) \\ & = & H^t(b_1) \,\centerdot\, \ldots \,\centerdot\, H^t(b_n) \\ & = & H^t(b_1 \ldots b_n) \\ & = & H^t(w) \end{array}$$

Il est donc possible, par la seule utilisation de l'opérateur de concaténation et de délai, d'implémenter en $8_{1/2\mathcal{D}}$ toute la classe des D0L systèmes, en considérant chaque règle de transition comme étant une équation récursive $8_{1/2\mathcal{D}}$. Nous décrivons dans la section suivante, un exemple classique de L système, et sa transcription en $8_{1/2\mathcal{D}}$.

XIV.4.2.3 Un exemple de D0L système : le développement d'un organisme uni-dimensionnel

Nous donnons ici un exemple paradigmatique de formalisation d'un processus de croissance en L système. Nous décrivons les états du développement d'un organisme uni-dimensionnel : un organisme filamentaire, à travers la définition d'un D0L système. Chaque étape de dérivation du système de réécriture, représentera un état de développement de l'organisme. Les règles de production permettent aux cellules de l'organisme, de rester dans le même état, de changer d'état, de se diviser en plusieurs cellules ou bien de disparaître.

Pour chaque cellule, deux états, a et b sont possibles. L'état a correspond à une division cellulaire, alors que l'état b, correspond à un état d'attente, durant une étape de division. Les règles de production et les 5 premières dérivations sont :

$$\begin{array}{llcll} \boldsymbol{\omega} & : b_r & \quad & t_0 & : b_r \\ p_1 & : a_r \rightarrow a_l\, b_r & & t_1 & : a_r \\ p_2 & : a_l \rightarrow b_l\, a_r & & t_2 & : a_l\, b_r \\ p_3 & : b_r \rightarrow a_r & & t_3 & : b_l\, a_r\, a_r \\ p_4 & : b_l \rightarrow a_l & & t_4 & : a_l\, a_l\, b_r\, a_l\, b_r \end{array}$$

La polarité des cellules est donnée par les suffixes l et r. Un arbre de dérivation du processus de croissance est illustré par la figure 3 (emprunté partiellement à [LJ92]). Les règles de changement de polarité de cet exemple sont très proches de celles de la bactérie bleue-verte *Anabaena catenula* [MW72, KL87].

Une implémentation des règles de production en $8_{1/2\mathcal{D}}$ est immédiate. En suivant les règles données en (11), le présent système peut être traduit en $8_{1/2\mathcal{D}}$ par :

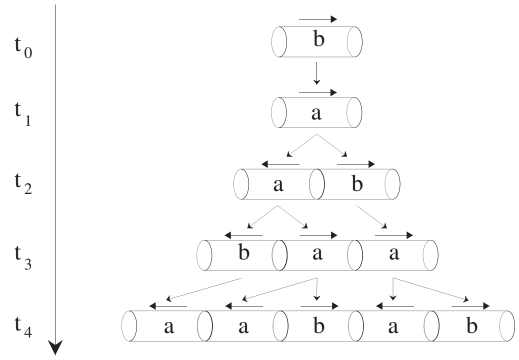


FIG. 3 –: *Les 5 premières dérivations du processus de croissance de la bactérie* Anaeba catenula *modélisé par un D0L système (la polarité des cellules est indiquée avec une flèche).*

```
w    = ar;
ar@0 = 'ar'; ar = $al # $br when Clock;
al@0 = 'al'; al = $al # $br when Clock;
br@0 = 'br'; br = $ar when Clock;
bl@0 = 'bl'; bl = $al when Clock;
```

Les 5 premiers pas de l'exécution du programme ci-dessus sont :

```
Top: 0 : { br } : char[1]
Top: 1 : { ar } : char[1]
Top: 2 : { al, br } : char[2]
Top: 3 : { bl, ar, ar } : char[3]
Top: 4 : { al, al, br, al, br } : char[5]
```

Remarquons que le nombre de cellules à la n^{e} génération est égal à la n^{e} valeur de la suite de FIBONACCI [FIB57].

XIV.4.3 Un autre modèle de croissance : la croissance de l'ammonite

Nous reprenons l'exemple de la croissance de l'ammonite que nous avons déjà abordé dans la première partie (cf. section III.3, page 22) mais en utilisant cette fois la capacité de $8_{1/2\mathcal{D}}$ à implémenter *dynamiquement* le processus.

Le processus de croissance est cette fois décrit par concaténation successive d'un tissu de la bonne dimension et au bon endroit. La modélisation du phénomène de croissance reste identique à celui que nous avons décrit dans la première partie, seule la programmation du modèle diffère.

Un tissu entier *cpt* est utilisé comme compteur, puis utilisé par le tissu entier *dir* pour indiquer suivant laquelle des 4 directions l'ammonite est actuellement en train de se développer. Tout le processus de croissance est modélisé par le tissu *gnomon*. Suivant la valeur de *dir* qui est un compteur modulo 4, le tissu *gnomon* concatène le tissu t à la bonne taille (par l'utilisation de l'opérateur `:| |`) et à la bonne position. La concaténation utilise les opérateurs $\#_0$ et $\#_1$ pour permettre, respectivement, une concaténation horizontale et verticale. Ces opérateurs sont notés # et #^ respectivement, dans le code $8_{1/2\mathcal{D}}$. Pour pouvoir interpréter graphiquement le résultat du calcul, nous utilisons le tissu u qui contient le numéro de génération. Le programme $8_{1/2\mathcal{D}}$ est :

$$
\begin{array}{lcl}
cpt@0 & = & 1; \quad cpt = \$cpt + 1\,\texttt{when}\,\texttt{Clock};\\
dir & = & (cpt\%4);\\
gnomon & = & cpt : |\$t|;\\
t@0 & = & \{\{1,1\}\};\\
t & = & \texttt{switch}(dir == 1)\,\texttt{then}(\$t)\,\#\hat{}\,gnomon\,\texttt{else}\,\texttt{switch}(dir == 2)\,\texttt{then}((\$t)\,\#\,gnomon)\\
 & & \texttt{else}\,\texttt{switch}(dir == 3)\,\texttt{then}\,gnomon\,\#\hat{}\,(\$t)\,\texttt{else}\,gnomon\,\#\,(\$t);
\end{array}
$$

On remarque que ce programme est particulièrement concis. Les 7 premières générations de l'ammonite sont illustrées par la figure 4.

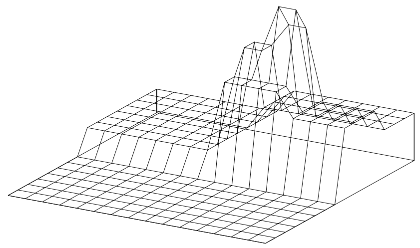


FIG. 4 –: *Les 7 premières étapes de la croissance de l'ammonite. Les cellules générées au même instant se trouvent à la même hauteur.*

Conclusion

Résumé des travaux

Nous avons défini dans ce travail deux nouvelles structures de données les *GBF* et les *amalgames*. Ces structures de données permettent la représentation dynamique de l'espace dans un langage déclaratif de simulation, le langage 81/2.

Plus précisément, nous avons tout d'abord défini la notion de *forme*. Une forme correspond à un type définissant un ensemble de déplacements possibles dans une structure de données. Cette forme définit la *topologie* des déplacements autorisés. Elle repose sur la présentation finie d'une structure de groupe et définit par conséquent une notion de voisinage *régulier*. Nous avons défini un ensemble d'opérations (produit libre, produit cartésien...) sur les formes qui permettent la construction de formes complexes, à partir de formes plus simples. La forme représente le type de base de la notion de GBF. Un GBF est une fonction partielle qui associe une *valeur* à un *élément* de la forme. Cette nouvelle structure de données permet la définition d'espaces plus complexes que les seules régions rectangulaires permises par les tableaux classiques. Nous avons proposé une implémentation des GBF abéliens. Enfin, nous avons illustré par des exemples classiques de simulation de systèmes dynamiques l'expressivité de la nouvelle structure de données.

Les GBF permettent de spécifier de façon extrêmement compacte une notion de voisinage régulier. Par contre, la représentation d'espaces irréguliers nécessite la définition de l'ensemble des points de l'espace ainsi que des chemins permettant de se déplacer d'un point à son voisin. Un tel graphe représente, par exemple, la structure des relations liant les variables d'un SD. Autrement dit, cet espace correspond au graphe des dépendances des équations qui modélisent le SD. Nous avons défini une nouvelle structure de données, les *amalgames*, permettant la construction dynamiques de DFG. Le formalisme des amalgames est composé de trois opérateurs, l'amalgamation « $\{\dots\}$ », la sélection « **.** » et la concaténation « # ». Les amalgames permettent la définition d'expressions *ouvertes*, et la définition d'un mécanisme de clôture des expressions par capture de nom rend possible la programmation *incrémentielle*. Nous avons défini une sémantique formelle des amalgames et nous avons décrit un évaluateur implémenté en Mathematica. Nous avons illustré la puissance d'expressivité du formalisme par des exemples. On remarquera en particulier un codage de l'arithmétique dans les amalgames.

Enfin, après avoir étudié ces structures de données pour elles mêmes, nous avons introduit les GBF et les amalgames dans le langage 81/2. La cohabitation des streams, des GBF et des amalgame n'est pas simple. Nous nous sommes volontairement restreints à un sous-cas des GBF, les GBF abéliens libres de support décrivant une région de $[0 \dots n_1] \times \cdots \times [0 \dots n_d]$. Les deux nouveaux types de données sont considérés comme des collections de types différents. Les tissus 81/2 deviennent des streams de GBF ou bien des streams d'amalgames. Nous avons défini une sémantique opérationnelle d'un évaluateur et une implémentation en CAML est en cours. Des exemples montrent que ces restrictions permettent tout de même la définition de programmes significatifs pour le domaine visé.

Tous les exemples spécifiés dans ce document ont été exécutés à l'aide des sémantiques définies et implémentées. Bien que les implémentations ne soient pas entièrement finalisées, elles sont suffisamment robustes pour permettre l'exécution des exemples décrits dans ce document. De façon similaire, les sorties graphiques de ce document ont toutes été réalisées par l'environnement que nous avons développé.

Poursuite des travaux

Les travaux que nous avons décrit dans ce document comportent de nombreuses lacunes. En effet, pour des raisons de temps, un certain nombre de problèmes n'ont pas trouvé de réponse entièrement satisfaisante.

Les GBF

Le travail effectué sur les GBF ne concerne qu'un sous-ensemble des GBF, les GBF abéliens. Nous avons donné dans la deuxième partie un schéma d'implémentation général pour les GBF. Nous avons en particulier détaillé cette implémentation pour les GBF abéliens (cf. chapitre VI, page 69). Mais nous n'avons pas développé l'implémentation d'une famille intéressante de GBF non-abélien, comme par exemple les groupes libres (qui permettent la représentation de structures d'arbres) et nous n'avons pas pu étudier l'intégration de ces deux types de formes.

Si l'on dispose des mécanismes nécessaires à l'implémentation des GBF (cf. section V.7, page 64) le problème d'intégration réside uniquement dans la détermination des mécanismes correspondant à une construction de forme. Par exemple, si l'on dispose d'un mécanisme pour tester l'appartenance à un coset dans une forme F et dans une forme G, dispose-t-on du mécanisme pour tester l'appartenance à un coset de $F \times G$?

Ce type de question est résolu dans le cas des formes abéliennes, car à chaque construction on sait associer une présentation du résultat, et tous les mécanismes nécessaires sont dérivables de la présentation. Il se peut que cela ne soit pas le cas pour une forme quelconque ; cela dépendra donc de propriétés supplémentaires spécifiques à la forme. Un travail est donc à faire pour chaque famille de formes non-abéliennes qu'on désire permettre dans le langage.

Par ailleurs, nous avons évoqué la stratégie d'évaluation data-driven. Cette stratégie pose de nombreux problèmes qu'il faut étudier plus attentivement. En particulier, il serait très intéressant de disposer d'une description du domaine de définition d'une définition récursive de GBF. Dans le cas général, même abélien, c'est sans doute hors de portée. Mais en dimension 2 ou 3 (groupe abélien à deux ou trois générateurs), les outils de la géométrie discrète sont peut-être utilisables pour résoudre ce problème. On sait par exemple décrire l'ensemble qui résulte de l'intersection de deux droites discrètes dans $\mathbb{Z}^2$.

Les amalgames

Il existe d'autres opérations intéressantes de combinaison des amalgames. Par exemple, une autre version de la sélection peut être envisagée, qui ne se réduit que si l'opérande droit ne comporte plus aucune référence libre. Cet opérateur correspond à une évaluation « avec mémoire des environnements rencontrés ». Un autre exemple est donné par une version de la concaténation permettant une liaison tardive des références. Par exemple, l'expression $\{a = 1, b = \{c = a^0\} \widetilde{\#} \{a = 9\}\}$ se réduirait en $\{a = 1, b = \{c = 9, a = 9\}\}$. Ce mécanisme peut correspondre à un mécanisme de délégation.

L'intégration des amalgames et des GBF dans le langage déclaratif 8½ pose de très nombreux problèmes qui n'ont pu qu'être mentionnés. Par exemple, il reste à définir un véritable calcul d'horloge sur les streams d'amalgames. Attribuer l'horloge de `Clock` à un amalgame n'est pas satisfaisant, car cela implique une évaluation pour chaque tic de l'amalgame.

Les mécanisme de complétion des expressions ouvertes des amalgames repose sur la notion de nom. La capture de nom permet comme nous l'avons dit une complétion dynamique des expressions. L'intégration des amalgames dans 8½ représente plus généralement l'introduction dans un langage déclaratif de la capture de nom. Toutes les implications de cette introduction n'ont pas été explorées. En particulier, il serait intéressant d'étudier les interactions avec la notion de *fonction*, absente de 8½.

Nous n'avons défini aucun système de type pour les amalgames. Si l'on voulait établir une comparaison, on pourrait dire que les amalgames, dans leur définition actuelle, correspondent au λ-calcul non typé. Il serait nécessaire de définir un système de type pour les amalgames afin de rejeter autant que possible, avant leur réduction, la classe des expressions qui divergent. Cependant, cette définition ne semble pas simple. La présence de références libres compliquant la tâche.

Les amalgames représentent des objets *en soi* : ce sont des calculs qui doivent se compléter. Il est donc décevant de ne disposer que d'une sémantique opérationnelle. Dans un premier temps, nous aimerions développer une sémantique dénotationnelle. Cependant, le domaine à associer à un système n'est pas immédiatement apparent (ce n'est en tout cas pas un environnement). Dans un deuxième temps, nous aimerions développer une sémantique équationnelle qui ferait des amalgames un objet «de plein droit».

Perspectives

Enfin, les travaux menés dans ce document ouvrent de très larges horizons quant aux évolutions des concepts que nous avons défini.

Vers une unification des amalgames et des GBF

On aimerait unifier les notions d'amalgame et de GBF afin de les englober dans une théorie commune qui soit plus générale et plus puissante que les deux formalismes qui existent actuellement de manière indépendante. En effet, un amalgame correspond à une collection d'éléments hétérogènes, dont les dépendances sont irrégulières et ad hoc, alors qu'un GBF est une collection d'éléments homogènes dont les dépendances sont régulières. Hormis cette distinction, les autres différences entre ces deux notions peuvent s'estomper, mais il faudrait :

- *Un traitement symétrique des amalgames et des GBF* : on peut définir en 8½ un amalgame dont les éléments sont des GBF. Mais on aimerait pouvoir aussi définir des GBF d'amalgames. Par exemple, une collection de wlumfs. Bien que cela soit théoriquement possible, 8½$_{\mathcal{D}}$ ne traite pas ce type de construction car les amalgames et les GBF ne sont pas encore gérés de façon symétrique.

- *Accès par un identificateur ou par un label* : en 8½ et en 8½$_{\mathcal{D}}$, chaque élément d'un amalgame est accessible à la fois par sa position et par son nom[1]. Cela permet de considérer des amalgames avec de multiples définitions du même identificateur. La position de la définition permet de lever l'ambiguïté de la référence.

  En fait, une fois ce schéma de nommage adopté, il est très tentant de l'étendre aux GBF en le généralisant comme [Gar94] : une référence serait alors constituée d'un label (le nom) et d'un index (la position). Un GBF correspondrait à des éléments indexés avec un label et une position vide. Un amalgame correspondrait à des éléments accédés par un nom *et* une position vide.

Ces éléments sont en faveur d'un rapprochement des notions de GBF et d'amalgame. Il existe un lien fort entre la notion d'amalgame, ou de champ de données, et de fonction. En effet, amalgames et champs sont des collections dont les éléments sont accessibles par un nom ou un index. On peut donc considérer ces structures de données comme des fonctions (des noms ou des index vers les valeurs). Traditionnellement, il existe deux point de vue pour concevoir les fonctions : le point de vue intensionnel et le point de vue extensionnel (cf. section IV.2.3, page 35). Selon ces distinctions, il est clair que les amalgames et les champs sont des structures de données extensionnelles et non pas des objets intensionnels. Néanmoins, il faut se rappeler que la principale caractéristique d'une collection est d'être manipulée comme un tout : les opérations permises sur les champs ne font pas référence aux constituants du champ et on construit des amalgames par concaténation d'amalgames. Autrement dit, ces structures de données sont *manipulées* intensionnellement même si elles sont *implémentées* extensionnellement. Les tableaux 1 et 2, dans l'annexe E, comparent les notions de fonction, d'enregistrement, de tableau, d'amalgame et de champ.

Tout cela montre qu'il existe une structure commune entre ces diverses notions qui reste à mettre en évidence et à étudier.

1. Nous n'avons pas développé ce point dans ce document, mais dans l'interprète 8½$_{\mathcal{D}}$, il est possible d'accéder aux éléments d'un amalgame par une position.

Amalgame, GBF et représentations topologiques

Les GBF sont utilisés pour décrire des espaces homogènes. Les amalgames permettent la description d'espaces ad hoc. Pour décrire un objet physique quelconque, un cône par exemple, on ne peut utiliser un GBF car le sommet du cône n'aura pas le même nombre de voisins qu'un autre point du cône. D'autre part, l'utilisation d'un amalgame nécessite d'expliciter tous les voisinages. Entre ces deux formalismes, il y a certainement la place pour une structure de données permettant des descriptions « raisonnables ». En topologie, la notion d'*atlas* permet de « raccommoder » des cartes, chacune décrivant un espace localement homogène différent. La notion actuelle d'amalgame de GBF est rudimentaire car on ne peut à travers un amalgame que combiner des GBF de même type. Notre espoir est donc de développer une notion plus fine d'amalgame de GBF qui jouerait ce rôle d'atlas.

Par ailleurs, un GBF correspond à un objet topologique bien connu : un groupe de déplacement. En revanche, à quel objet correspond un amalgame ? La figure 9, page 107 représente un amalgame sous la forme d'un graphe de graphes, ce qui va plus loin qu'un DFG qui n'est qu'un graphe à un seul niveau hiérarchique. Il est alors tentant de se représenter un amalgame comme un *complexe simpliciel.* Les complexes simpliciaux correspondent à une structure riche qu'on sait caractériser. Notre intuition est que cette caractérisation peut servir à typer les amalgames (déterminer si une expression possède une forme normale par exemple).

Amalgames, GBF et processus de morphogénèses

Amalgames et GBF peuvent être considérés comme des outils préliminaires mais indispensables pour la simulation de processus de morphogénèse. En effet, les enrichissements qu'ils apportent à la notion de collection, permettent trois choses :

– La notion d'amalgame permet de construire un programme 81/2 comme résultat d'un calcul. Avec ce mécanisme, on peut envisager des programmes dont la *structure* à l'instant t dépend fondamentalement de l'évaluation du programme à l'instant $t-1$.

– La notion de groupe permet de définir explicitement la notion de voisinage et de décrire des *topologies* plus riches que celles accessibles à travers la notion de tableau. Il devient donc possible de décrire ou de construire des formes plus complexes. Par ailleurs, la notion de voisinage devenant explicite, il devient possible de formaliser précisément ce qu'est un calcul *local.*

– La gestion paresseuse et la partialité des GBF permet la définition de *géométries* plus compliquées que celle des tableaux. En effet, un tableau correspond à une zone rectangulaire de $\mathbb{Z}^2$, parallélépipédique de $\mathbb{Z}^3$, etc. Quand on concatène des tableaux, le résultat est astreint à être un tableau. Avec des collections paresseuses, conçues comme des fonctions à support borné dans $\mathbb{Z}^n$, la concaténation n'est plus restreinte à s'appliquer à des tableaux conformes, et on peut manipuler des régions bornées quelconques.

En regard des processus réels de morphogénèse, et des mécanismes mis en jeu, les structures de données évoquées ci-dessus sont peut-être encore trop simples. L'étude d'applications permettra de préciser ce point. Par ailleurs, nous espérons que les concepts développés pour l'étude de la morphogénèse pourront être importés utilement en informatique et enrichir les concepts de programmation.

Annexe A

Code source du calcul de l'équation de diffusion–réaction

Ce source C est une ré-écriture du code de Greg TURK [Tur91] (greg.turk@cc.gatech.edu), écrit en 1991, suite à [BL74]. Il est donné à titre de comparaison avec le programme 81/2 de la première partie (cf. section III.2, page 20). Nous rappelons à l'issue du programme C le programme 81/2 équivalent.

A.1 Le programme C

```
extern double drand48();

#define SIZE 60            /* Nombre de cellules dans la ligne */

double x_init = 4.0;       /* Concentration initiale de "x" */
double y_init = 4.0;       /* Concentration initiale de "y" */

double x[SIZE];            /* Concentration chimique "x" */
double y[SIZE];            /* Concentration chimique "y" */

double dx[SIZE];           /* changements pour "x" */
double dy[SIZE];           /* changements pour "y" */

double diff1 = 0.25;       /* Taux de diffusion pour "x" */
double diff2 = 0.0625;     /* Taux de diffusion pour "y" */

long   right[SIZE];
long   left[SIZE];

double beta[SIZE];         /* Le substrat aléatoire       */
double beta_init = 12.0;   /* Valeur initiale du substrat */
double beta_rand = 0.05;   /* Distribution du substrat    */


double react_speed = 1.0   /* Vitesse de réaction (par rapport à la diffusion) */

main (int argc, char* argv[])
{
    unsigned long t, end_date;
    unsigned long i;
    double xdiff, ydiff;
    double rsp;

    end_date = (argc <= 1 ? 0 : atol(argv[1]));
    rsp = react_speed / 16.0;
    for (i = 0; i < SIZE; i++)
    {
        x[i]      = x_init;
```

```
        y[i]      = y_init;

        beta[i]   = beta_init + drand48() * 2.0 * beta_rand - beta_rand;

        right[i]  = (i + SIZE - 1) % SIZE;
        left[i]   = (i + 1) % SIZE;
    }

    for (t = 0; t <= end_date; t++)
    {
        if (t)
        {
            for (i = 0; i < SIZE; i++)
            {
                y[i] += dy[i];

                if (y[i] < 0.0)
                {
                   y[i] = 0.0;
                }

                x[i] += dx[i];
            }
        }

        for (i = 0; i < SIZE; i++)
        {
            ydiff = y[right[i]] + y[left[i]] - 2 * y[i];
            dy[i] = rsp * (x[i] * y[i] - y[i] - beta[i]) + ydiff * diff2;

            xdiff = x[right[i]] + x[left[i]] - 2 * x[i];
            dx[i] = rsp * (16.0 - x[i] * y[i]) + xdiff * diff1;
        }
    }
    return 0;
}
```

A.2 Le programme 8$_{1/2}$

$$
\begin{array}{lcl}
iota & = & '60; \\
droite & = & \texttt{if}(iota == 0)\,\texttt{then}\,59\,\texttt{else}(iota-1); \\
gauche & = & \texttt{if}(iota == 59)\,\texttt{then}\,0\,\texttt{else}(iota+1); \\
rsp & = & 1.0/16.0; \\
diff1 & = & 0.25; \\
diff2 & = & 0.0625; \\
x & = & \$x + \$dx\,\texttt{when}\,\texttt{Clock}; \quad x@0 = 4.0; \\
y & = & \max(0.0, \$y + \$dy)\,\texttt{when}\,\texttt{Clock}; \quad y@0 = 4.0; \\
beta & = & 12.0 + rand(0.05 * 2.0) - 0.05; \\
xdiff & = & x(droite) + x(gauche) - 2.0 * x; \\
ydiff & = & y(droite) + y(gauche) - 2.0 * y; \\
dx & = & rsp * (16.0 - x * y) + xdiff * diff1; \\
dy & = & rsp * (x * y - y - beta) + ydiff * diff2;
\end{array}
$$

Annexe B

Implémentation du calcul des amalgames en Mathematica

B.1 Les termes des amalgames, leur écrivain 8½ et LaTeX

On définit ici les têtes des termes représentant un amalgame en Mathematica. Ces termes ont deux formes externes correspondant à une sortie formatée à la 8½ et à la LaTeX.
Les expressions suivantes décrivent les têtes des termes :

```
Unprotect[Dot];
ClearAttributes[Dot, OneIdentity];
ClearAttributes[Dot, Flat];
Unprotect[Plus];
ClearAttributes[Plus, Orderless];

SetAttributes[Sys, HoldAll];
Sys[defs___] := ReleaseHold[SYS [Hold[defs] //. Set -> Let]]
Let[l_, r_]  := LET[l, r]
Up[x_Symbol] := Up[1, x]

Scope[x_Symbol]                        := Scope[0, x]
Scope[Scope[n_, x_Symbol]]             := Scope[n+1, x]
Scope/: Scope[x_, z_] < Scope[y_, z_] := x < y
```

Les expressions suivantes définissent les sorties formatées à la 8½ :

```
Format[SYS[]] := "{}"
Format[SYS[x__]] :=  If[{} === Cases[{x}, SYS[___], Infinity],
                        SequenceForm[{x}],
                        SequenceForm["{", ColumnForm[{x}], "}"]]

Format[LET[x_, y_]]       := HoldForm[x = y]
Format[Scope[n_, x_]]     := SequenceForm[x, Subscript[n]]
Format[Up[n_, x_]]        := SequenceForm[x, Superscript[n]]
Format[Dot[x_Dot, y_Dot]] := SequenceForm["(", x, ") . (", y, ")"]
Format[Dot[x_, y_Dot]]    := SequenceForm[x, ". (", y, ")"]
Format[Dot[x_Dot, y_]]    := SequenceForm["(", x, ") .", y]
```

Les expressions suivantes définissent les sorties formatées à la LaTeX:

```
Unprotect[Plus];
Format[Scope[n_, x_], TeXForm] := StringForm["‘{ ‘}{‘‘}", n, ToString[x]]
Format[Up[n_, x_], TeXForm]    := StringForm["‘{ ‘}{‘‘}", n, ToString[x]]
Format[LET[x_, y_], TeXForm]   := StringForm["{= ‘ ‘}{‘‘}", ToString[x], y]

Format[SYS0[], TeXForm]        := "systeme{}\\\\"
Format[SYS0[l___], TeXForm]    := SequenceForm["{{}", Format[TexList[l], TeXForm], "}\\\\"]
Format[SYS[], TeXForm]         := "{{}}"
Format[SYS[l__], TeXForm]      :=  SequenceForm["{{}", Format[TexList[l], TeXForm], "}"]
```

```
Format[TexList[l_], TeXForm]       := Format[l, TeXForm]
Format[TexList[l__, ld_], TeXForm] := SequenceForm[Format[TexList[l], TeXForm], ", ", Format[ld, TeXForm]]

Format[Dot[x_, y_], TeXForm]       := SequenceForm["{.", Format[x, TeXForm], "}{", Format[y, TeXForm], "}"]


Format[Plus[a_, b_], TeXForm] := If[MatchQ[a, _Integer] || MatchQ[b, _Integer],
                                    SequenceForm[Format[a, TeXForm], " + ", Format[b, TeXForm]],
                                    SequenceForm["{ #", Format[a, TeXForm], "}{", Format[b, TeXForm], "}"]]
```

B.2 La liaison des expressions

Préalablement à la définition de la liaison, nous devons définir les fonction Comb(), Dom(), index() et Lift() :

```
Comb[UnknownP, _]  := UnknownP
Comb[_ , UnknownP] := UnknownP
Comb[_SYS, _SYS]   := UnknownP
Comb[_, _]         := Unknown

Dom[s_SYS]   := Cases[s, LET[id_, _] -> id]
Dom[_Scope]  := UnknownP
Dom[_Up]     := Unknown
Dom[l_ + r_] := Comb[Dom[l], Dom[r]]
Dom[l_ . r_] := Dom[r]
Dom[_]       := UnknownP

Index[{}, id_, n_]               := Unknown
Index[{Unknown, l___}, id_, n_]  := Unknown
Index[{UnknownP, l___}, id_, n_] := Unknown
Index[{s_, l___}, id_, n_]       := If[MemberQ[s, id], n, Index[{l}, id, n+1]]

Lift[s_SYS]                     := Lift[0][s]

Lift[n_Integer][Scope[p_, id_]] := If[n <= p, Scope[p+1, id], Scope[p, id]]
Lift[n_Integer][x_Up]           := x
Lift[n_In][x_ ? AtomQ]          := x
Lift[n_Integer][s_SYS]          := Map[Lift[n+1], s]
Lift[n_Integer][LET[id_, e_]]   := LET[id, Lift[n][e]]
Lift[n_Integer][x_ . y_]        := Lift[n][x] . Lift[n+1][y]
Lift[n_Integer][x_ + y_]        := Lift[n][x] + Lift[n][y]
Lift[n_Integer][x_]             := x[[0]]@@Map[Lift[n], List@@x]
```

Puis nous pouvons définir la fonction de liaison des expressions :

```
Bind[s_SYS] := Bind[{}][s]

Bind[l_List][x:Up[n_, id_]] := If[n > Length[l], x,
                                 With[{index = Index[Drop[l, n], id, n]},
                                   If[index === Unknown, x, Scope[index, id]]]]
Bind[l_List][x_Scope]     := x
Bind[l_List][x_ ? AtomQ]  := x
Bind[l_List][s_SYS]       := Map[Bind[Prepend[l, Dom[s]]], s]
Bind[l_List][LET[x_, e_]] := LET[x, Bind[l][e]]
Bind[l_List][x_ . y_]     := Bind[l][x] . Bind[Prepend[l, Dom[x]]][y]
Bind[l_List][x_ + y_]     := Bind[l][x] + Bind[l][y]
Bind[l_List][x_]          := x[[0]]@@Map[Bind[l], List@@x]
```

B.3 La traduction de $\Sigma_{\mathcal{I}}$ en $\Sigma_{\mathcal{C}}$

Nous définissons les notions de corrections des référence liées, puis des références libres :

```
correctB[l_List][_Up]            := True
correctB[l_List][Scope[n_, id_]] := (n+1 <= Length[l]) && MemberQ[l[[n+1]], id]
correctB[l_List][_ ? AtomQ]      := True
correctB[l_List][LET[_, x_]]     := correctB[l][x]
correctB[l_List][s_SYS]          := LazyAnd[correctB[Prepend[l, Dom[s]]], s]
correctB[l_List][x_ . y_]        := correctB[l][x] && correctB[Prepend[l, Dom[x]]][y]
correctB[l_List][x_ + y_]        := correctB[l][x] && correctB[l][y]
correctB[l_List][x_]             := LazyAnd[correctB[l], x]

correctF[l_List][Up[n_, id_]] := (n >= Length[l]) || (Unknown === Index[Drop[l, n], id, 0])
correctF[l_List][_Scope]      := True
correctF[l_List][_ ? AtomQ]   := True
correctF[l_List][LET[_, x_]]  := correctF[l][x]
correctF[l_List][s_SYS]       := LazyAnd[correctF[Prepend[l, Dom[s]]], s]
correctF[l_List][x_ . y_]     := correctF[l][x] && correctF[Prepend[l, Dom[x]]][y]
correctF[l_List][x_ + y_]     := correctF[l][x] && correctF[l][y]
correctF[l_List][x_]          := LazyAnd[correctF[l], x]
```

et enfin la notion de correction des termes de $\Sigma_{\mathcal{C}}$:

```
Correct[l_List][x_] := correctB[l][x] && correctF[l][x]
Correct[x_]         := correctB[{}][x] && correctF[{}][x]
```

Nous pouvons alors définir la fonction de traduction d'un terme de $\Sigma_{\mathcal{I}}$ en un terme de $\Sigma_{\mathcal{C}}$ si la liaison est `correct()` :

```
IsVar[x_Symbol] := ! MemberQ[Attributes[x], (Locked | Protected)]
IsVar[_] := False

Id2Ref[s_SYS]       := Map[Id2Ref, s]
Id2Ref[LET[l_, r_]] := LET[l, Id2Ref[r]]
Id2Ref[x_Up]        := x
Id2Ref[x_Scope]     := x
Id2Ref[x_ ? IsVar]  := Up[0, x]
Id2Ref[x_ ? AtomQ]  := x
Id2Ref[x_]          := x[[0]]@@Map[Id2Ref, List@@x]

Trad[exp_] := With[{cexp = Bind[Id2Ref[exp]]},
                 If[Correct[cexp], cexp,
                    If[correctB[{}][cexp],
                          "Erreur (???) : terme de ΣI non traduisible (pb. avc les Refs LIBRES)",
                          "Erreur: terme de ΣI non traduisible (pb. avc les Refs LIEES)"]]]
```

B.4 Le prédicats C() et les fonctions annexes

Nous définissons les prédicats C() et les fonctions de manipulation d'environnements.

```
RC[l_List][_Scope]      := False
RC[l_List][Up[n_, _]]   := (n >= Length[l]) || !MemberQ[Drop[l, n], UnknownP]
RC[l_List][e_ ? AtomQ]  := True
RC[l_List][LET[_, r_]]  := RC[l][r]
RC[l_List][e1_ + e2_]   := !(MatchQ[e1, _SYS] && MatchQ[e2, _SYS]) && RC[l][e1] && RC[l][e2]
RC[l_List][e1_ . e2_]   := !MatchQ[e1, _SYS] && RC[l][e1] && RC[Prepend[l, Dom[e1]]][e2]
RC[l_List][e_SYS]       := LazyAnd[RC[Prepend[l, Dom[e]]], e]
RC[l_List][e_]          := LazyAnd[RC[l], e]

NewEnv[env_List, s_SYS] := Prepend[LiftEnv[env], Cases[s, LET[id_, e_] :> Rule[id, e]]]
NewEnv[env_List, x_]    := Prepend[LiftEnv[env], Dom[x]]

LiftEnv[env_List] := Map[LiEn, env]
LiEn[Unknown]     := Unknown
LiEn[UnknownP]    := UnknownP
LiEn[l_List]      := Cases[l, Rule[id_, e_] :> Rule[id, Lift[0][e]]]

EnvToScope[l_List] := Map[EToS, l]
EToS[Unknown]      := Unknown
EToS[UnknownP]     := UnknownP
```

```
EToS[l_List]       := Cases[l, Rule[id_, e_] -> id]

SetAttributes[LazyAnd, HoldRest]
LazyAnd[f_, l_] := ReleaseHold[And@@Map[f, Hold@@l]]
```

B.5 L'interprète proprement dit

Nous disposons maintenant de toutes les fonctions nécessaires à la définition de l'interprète des amalgames. Celui-ci consiste en une fonction d'évaluation qui effectue une opération de réduction sur une expression. Nous utilisons l'opérateur de point-fixe de Mathematica, gardé par le prédicat C() pour définir la relation de réduction « $\twoheadrightarrow$ » (cf. section X.3.2, page 136).

```
EvalT[e_]   := FixedPoint[Eval[{}],e, RC]
Eval[s_SYS] := (saveCurrentExp = s; Eval[{}][s])

Eval[env_List][LET[l_, r_]]    := LET[l, Eval[env][r]]
Eval[env_List][s_SYS]          := Map[Eval[NewEnv[env, s]], s]
Eval[env_List][e_Up]           := e

Eval[env_List][Scope[n_, id_]] :=  With[{e = id /. env[[n+1]]},
                                        If[RC[EnvToScope[env],e][Bind[EnvToScope[env]][e], Scope[n, id]]]]
Eval[env_List][e_ ? AtomQ]     := e

Eval[env_List][SYS[s1___] + SYS[s2___]] := Bind[EnvToScope[env]][SYS[s1, s2]]
Eval[env_List][l_ + r_]                 := Eval[env][l] + Eval[env][r]

Eval[env_List][Dot[s_SYS, r_]] := If[RC[EnvToScope[env]][r], r,
                                     With[{ns = Eval[env][s]}, Dot[ns, Eval[NewEnv[env, ns]][r]]]]

Eval[env_List][l_ . r_] := With[{ll = Eval[env][l]},
                                If[MatchQ[ll, _SYS],
                                   With[{nr = Bind[Prepend[EnvToScope[env], Dom[ll]]][r]},
                                                                               ll.nr],
                                   ll . Eval[NewEnv[env, ll]][r]]]

Eval[env_List][If[c_, v_, f_]] := With[{cond = Eval[env][c]}, If[cond, v, f]]
Eval[env_List][e_]             := e[[0]]@@Map[Eval[env], List@@e]
```

B.6 Exemple de session Mathematica

Nous reproduisons dans la figure 1 page suivante un exemple de session Mathematica où nous évaluons une expression des amalgames. L'expression correspond à un calcul de négation logique de l'argument spécifié par `val`.

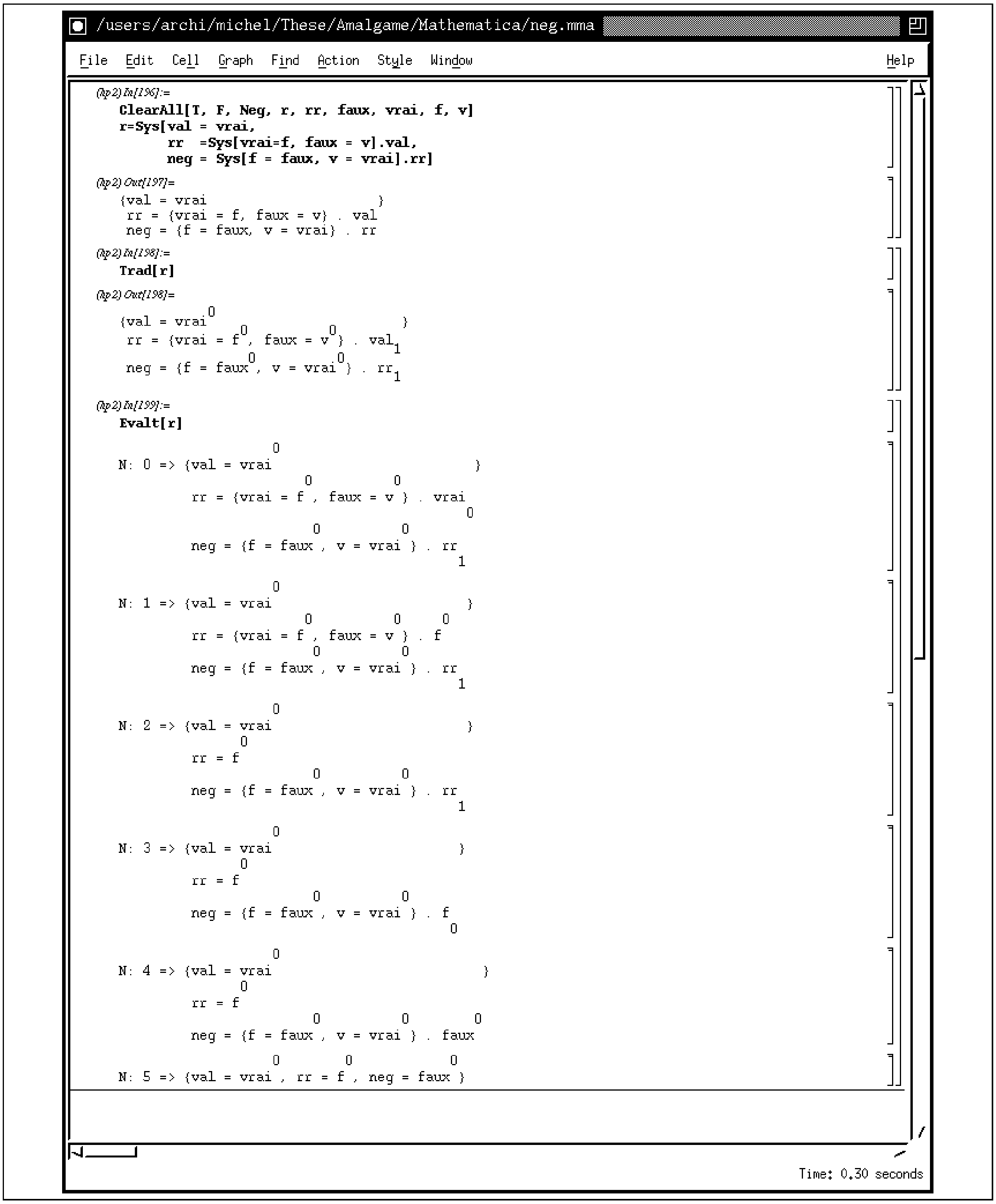


FIG. 1 –: *Exemple de session sous Mathematica. L'expression r correspondant à un système de trois élément est défini. Une fois l'expression de $\Sigma_{\mathcal{I}}$ entrée sous Mathematica, elle s'affiche sous la forme 8.1/2. Nous donnons le résultat de la fonction de traduction de l'expression de $\Sigma_{\mathcal{I}}$ en $\Sigma_{\mathcal{C}}$. Puis nous évaluons l'expression qui se réduit en 5 étape en le résultat escompté.*

Annexe C

Le codage de l'arithmétique en amalgames

Nous détaillons ici le source Mathematica du traducteur de l'arithmétique dans les amalgames :

```
Amar[x_] := Amar[0, {}, x]

Amar[i_Integer, l_List, NoAmar[x_]] := x

Amar[i_Integer, l_List, Zero] := Sys[b = vrai]
Amar[i_Integer, l_List, n_Integer] := Int[n]

Amar[i_Integer, l_List, Suc[x_]] := Sys[p = Amar[i+1, l, x], b = faux]
Amar[i_Integer, l_List, Pre[x_]] := Amar[i, l, x].p
Amar[i_Integer, l_List, ZeroQ[x_]] := Amar[i, l, x].b

Amar[i_Integer, l_List, Rond[f_, g_]] := With[{comp = Unique["comp"], val = Unique["val"]},
                                          With[{tmp = Amar[i+1, l, g], suite = Amar[i+1, {comp}, f]},
                                            Sys[comp = tmp, val = suite] . val]]

Amar[i_Integer, l_List, Rond[f__, g_, h_]] := With[{comp = Unique["comp"], val = Unique["val"]},
                                               With[{tmp = Amar[i+1, l, h],
                                                 suite = Amar[i+1, {comp}, Rond[f, g]]},
                                                 Sys[comp = tmp, val = suite] . val]]

Amar[i_Integer, l_List, RecPrim[c_, g_, h_]] :=
 With[{x = Unique["x"], y = Unique["y"], X = Unique["X"], Y = Unique["Y"],
       fct = Unique["fct"], start=Unique["start"], stop=Unique["stop"]},
 With[{ny = Amar[i, {Up[x], Up[y]}, h[U1, U2]],
           body = Amar[i, {Up[0, x], Up[0, y], call.retvalue},
                          IfZeroSymbo[U1, c[U2], g[U1, U3]]]},
                Sys[recval = Sys[call = Sys[X=Up[x].p, Y = ny, retvalue=param.(fct.valif)],
                              aiguillage = Sys[vrai=stop.Up[codeVrai], faux=stop.Up[codeFaux]]].
                                     (Sys[x=Up[2, l[[1]]], y=Up[2, l[[2]]], param=Sys[x = X, y = Y]].
                                             (Up[2,fct].valif)),
                        finalval = Sys[stop=Sys[]].recval,
                        fct = body
                   ].finalval]]

Amar[i_Integer, l_List, IfZeroSymbo[argc_, argv_, argf_]] :=
        With[{targc = Amar[i+1, l, argc], targv = Amar[i+1, l, argv], targf = Amar[i+1, l, argf]},
                Sys[valif = aiguillage.Up[codeCond],
                        codeCond = targc.b,
                        codeVrai = targv,
                        codeFaux = targf] ]

Amar[i_Integer, l_List, f_[param___Integer]] :=
        With[{args = Table[Unique["arg"], {Length[{param}]}]},
          SYS@@Prepend[MapThread[LET[#1, Amar[i+1, l, #2]]&, {args, {param}}],
                             LET[Reponse, Amar[i+1, args, f]]]]
```

```
Amar[i_Integer, l_List, f_[param___]] :=
               With[{val = Unique["val"], args = Table[Unique["arg"], {Length[{param}]}]},
                 SYS@@Prepend[MapThread[LET[#1, Amar[i+1, l, #2]]&, {args, {param}}],
                                         LET[val, Amar[i+1, args, f]]].val]
```

Annexe D

8,5$_{\mathcal{D}}$: un environnement pour 8$_{1/2}$$_{\mathcal{D}}$

Nous présentons brièvement dans cette annexe l'environnement de développement `8,5`$_{\mathcal{D}}$ pour le langage 8$_{1/2}$$_{\mathcal{D}}$. Nous présentons uniquement la liaison avec le logiciel de tracé graphique Gnuplot [WKC+90]. Pour une description complète de 8$_{1/2}$, le lecteur se reportera à [Gia91a] et les extensions spécifiques à 8$_{1/2}$$_{\mathcal{D}}$ sont décrites dans la quatrième partie de ce document.

D.1 L'environnement `8,5`$_{\mathcal{D}}$

Une commande, dans l'environnement `8,5`$_{\mathcal{D}}$ commence toujours par le point d'exclamation « `!` ». Une commande peut apparaître après le signe d'invite[1] (le symbole «`=>`» de début de ligne, qui invite l'utilisateur à entrer une expression) ou bien après le délimiteur de fin d'expression « `;` ». La table 1 récapitule les mots-clés de `8,5`$_{\mathcal{D}}$ qui se rapportent plus particulièrement aux opérations graphiques.

`!gc`	`!kill`	`!killf`	`!kills`	`!laser`	`!list`	`!load`	`!output`
`!plot`	`!dplot`	`!print`	`!replot`	`!reset`	`!rot`	`!send`	`!set`
`!show`	`!spin`						

TAB. 1 –: *Les commandes de l'environnement 8,5$_{\mathcal{D}}$.*

D.2 Principe de l'intégration

L'environnement `8,5`$_{\mathcal{D}}$ permet de visualiser le résultat d'un programme sous forme graphique, ce au cours du temps (on parle dans ce cas de visualisation «temps réel» d'un programme) ou bien à la fin de l'évaluation du programme (on parle alors de visualisation «post-mortem» d'un programme). Pour permettre cela, `8,5`$_{\mathcal{D}}$ fournit à l'utilisateur des contextes graphiques (*gc*) qui définissent l'état de sortie graphique d'une expression 8$_{1/2}$$_{\mathcal{D}}$. Ces contextes graphiques s'inspirent de la philosophie de `X11` : on attribue un gc à une expression 8$_{1/2}$$_{\mathcal{D}}$. Ce sont les attributs du gc attaché à l'équation qui spécifient le mode de tracé de l'expression. Ainsi, il n'existe qu'une unique fonction de tracé d'une expression, seules les valeurs des gc changent.

L'interaction avec `8,5`$_{\mathcal{D}}$ est réalisé à travers le découpage de l'opération de tracé en trois phases :

1. Une phase initiale intervenant avant l'évaluation d'un tissu. Cette phase, le *prélude*, correspond à la vérification de la cohérence des requêtes de tracé associé à un tissu et l'initialisation de Gnuplot.

2. Une phase d'*interlude* qui est exécutée pour chaque tic de l'évaluation d'un tissu et qui correspond au traitement de la valeur du tissu en fonction du tracé qui doit être effectué. En particulier, si on désire

1. Ou *prompt* en anglais

visualiser pour chaque tic la valeur d'un tissu, il est nécessaire de rafraîchir le contenu de la fenêtre de visualisation.

3. Une phase terminale, qui intervient à l'issue de l'évaluation d'un tissu. Cette phase de *postlude* correspond à la terminaison du processus de tracé. Si le tissu n'est visualisé qu'a l'issue de l'exécution, il est alors nécessaire d'effectuer le tracé.

La définition des caractéristiques du tracé se fait par uniquement par l'intermédiaire de *contextes graphiques* qui sont « attachés » aux tissu que l'on désire visualiser. Par conséquent, le tracé des tissus est totalement transparent pour le programmeur, qui décide à l'issue de la programmation les tissus qu'il désire visualiser.

D.3 Les contextes graphiques $8{,}5_{\mathcal{D}}$

Les contextes graphiques sont des objets de l'environnement 8,5 qui permettent de définir les caractéristiques du tracé de tissus $8_{1/2}$ lors de l'exécution d'un programme 8,5. Il est nécessaire de définir un gc pour spécifier les attributs du tracé. Un gc peut être partagé par plusieurs tissus : dans ce cas, tous les tissus partageant un même gc seront tracés sur la même fenêtre graphique, avec les même attributs. Un gc est un système $8_{1/2}$, de type `GC`. Un gc existe par défaut :

```
GC default = {color = 1; linestyle = lines; title = "Web output"; refresh; tick; memory;
              norealtime; noparametric; window = 0; geometry = none}
```

Des fonctions de manipulation de gc ont été définies et permettent la création, la duplication, la mise à jour et l'attachement et le détachement d'un gc à un tissu. La commande `!gc` *nom* `=` *defs* définit un gc *nom* munis des attributs *defs*. Il n'est pas nécessaire de définir l'ensemble des attributs de *nom*. Si on n'en spécifie qu'un certain nombre, les attributs non spécifiés auront une valeur par défaut : la valeur correspondante du gc `default`. La copie d'un gc se fait grâce à la fonction `!gc` nom_1 `=` nom_2 où nom_1 et nom_2 représentent tout deux des noms de gc.

Une fois un gc défini, il est nécessaire de l'attacher à un tissu. Cette opération se fait par l'intermédiaire de la commande `!set gc` nom_{tissu} `=` nom_{gc} où nom_{tissu} est le nom d'un tissu défini dans l'environnement $8{,}5_{\mathcal{D}}$ et nom_{gc} est un nom de gc précédemment défini. L'attachement d'un gc à un tissu suffit pour effectuer la sortie graphique des valeurs du tissu. Dans ce cas, l'exécution du tissu doit se faire par l'intermédiaire des instructions `!plot` (pour un schéma d'évaluation *statique*) et `!dplot` (pour un schéma d'évaluation dynamique).

REMARQUE Si on attache un gc à un tissu qui n'est pas évalué directement mais qui se trouve dans le graphe des dépendances de l'expression évaluée, alors le tracé du tissu est automatiquement effectué. Par conséquent, si l'on désire tracer un ensemble de tissus, il suffit de créer une dépendance entre les divers tissus, par l'intermédiaire d'un système par exemple.

REMARQUE La dernière définition des gc ne permet que l'attachement d'un gc à une équation quantifiée universellement. En effet, dans un schéma d'évaluation dynamique, le traitement des équations multiplement quantifiées ne permet pas une gestion simple des gc. Il est cependant toujours possible de se ramener a ce cas par la définition de nouvelles équations.

D.4 Sémantique des attributs d'un contexte graphique

Nous allons détailler la sémantique des attributs d'un gc. Nous pouvons répartir les attributs d'un contexte graphique en dix catégories :

- `tick/notick` : opération de sélection de tic/top,
- `memory/nomemory` : opération de prise en compte du temps lors du tracé,
- `realtime/norealtime` : opération de sélection de tracé en temps réel,
- `parametric/noparametric/order` : opérations de sélection du type de tracé,

- `linestyle` : opération de sélection de style de ligne du tracé,
- `window` : opération de sélection de fenêtre résultat,
- `color` : opération de sélection de la couleur du tracé,
- `title` : opération de sélection du nom de la fenêtre,

Ces attributs se décomposent en deux classes principales :

1. les attributs qui permettent la définition d'un *mode* particulier de tracé,
2. les attributs qui permettent de donner une *information* sur la façon dont doit être effectué la sortie.

Les attributs de la première catégorie sont organisés par paires : `memory` et `nomemory`, `realtime` et `norealtime`... Les attributs de la seconde catégorie nécessitent un argument : `linestyle` nécessite en plus de donner une information d'attribut sur le format du point utilisé pour le tracé...

Nous allons décrire chacune des catégories, en donnant à chaque fois un exemple de l'utilisation. Pour détailler les commandes ; nous donnerons à chaque fois que ce sera nécessaire des exemples mettant en évidence les caractéristiques des attributs des gc. Nous commençons par la description des attributs de la première catégorie puis détaillerons les attributs de la seconde catégorie.

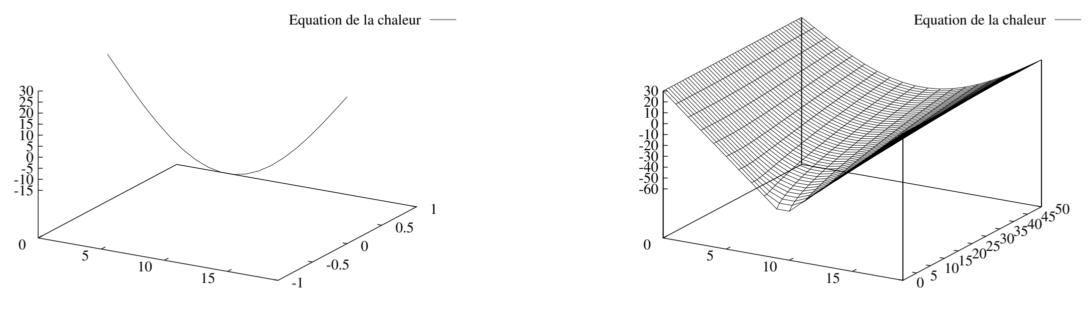
D.4.1 Attribut `tick`, `notick`

Pour chaque tic n de la simulation, un tissu t a une valeur. Cependant, si ce tic n'est pas un top, alors la valeur du tissu t au tic n est identique à la valeur de ce même tissu au tic $n-1$.

L'utilisation de l'attribut `tick` et `notick` permet de choisir s'il est nécessaire d'effectuer une sortie graphique pour tous les tics d'un tissu, ou uniquement pour les tics qui correspondent à des tops.

D.4.2 Attribut `memory`, `nomemory`

Lors du tracé d'un tissu 81/2, il est possible de considérer le tissu à une date n ou bien la succession des valeurs du tissu jusqu'à n. C'est dans ce cas, l'*histoire*, ou la *trajectoire* des éléments du tissu qui nous intéresse. Ces deux modes de tracé sont possibles par l'utilisation des attributs `nomemory` et `memory` respectivement.



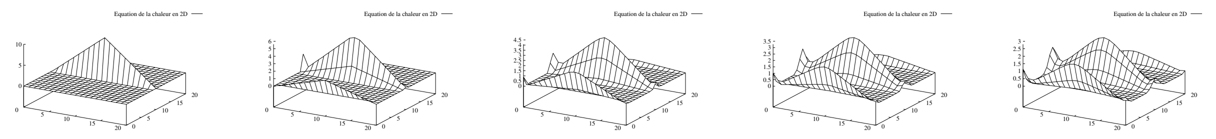
(a) Attribut `nomemory`. (b) Attribut `memory`.

FIG. 1 –: *Tracé d'équations avec ou sans mémoire.*

Par exemple, la figure 1 définit un programme résolvant le problème de la diffusion de la chaleur dans une barre de métal (cf. section VII.2, page 74). Les figure 1.b et 1.c illustrent le tracé sans ou avec l'attribut `memory`.

D.4.3 Attribut `realtime` = n, `norealtime`

L'attribut `realtime` = n permet de choisir entre un tracé en temps réel, c'est-à-dire un tracé suivant lequel le graphique est rafraîchi tous les n tics. La spécification de l'attribut `norealtime` permet la seule mise à jour du graphique à l'issue de l'exécution du programme. On parle de tracé en *temps réel* ou *post-mortem*.

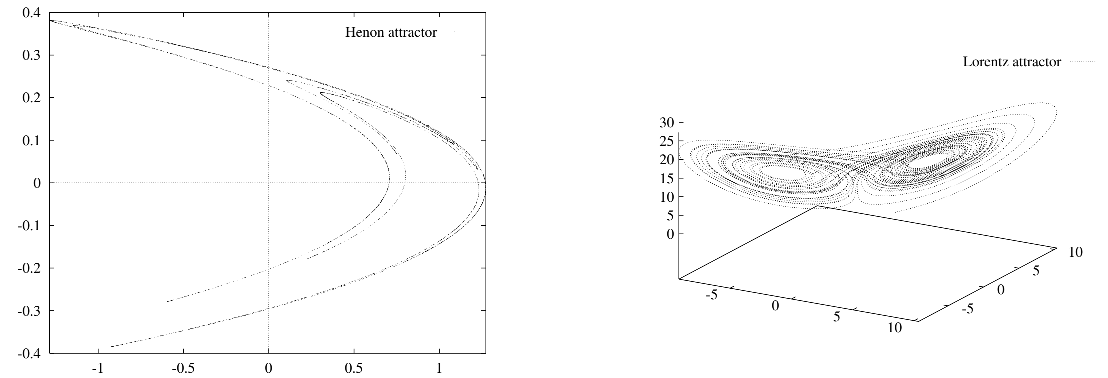


(a) Tic 0. (b) Tic 1. (c) Tic 2. (d) Tic 3. (e) Tic 4.

FIG. 2 –: *Tracé d'équations en temps réel.*

Nous reprenons l'exemple du calcul de la diffusion de la chaleur dans une barre de métal de la section précédente, mais calculons cette fois la diffusion dans un tore. Les figure 2.a à 2.e définissent l'évolution de la température dans la surface, au cours des 5 premiers tops ; cette exécution correspond à l'utilisation de l'attribut `realtime = 1`.

D.4.4 Attribut `parametric`, `noparametric`



(a) Attribut `parametric` en 2 dimensions : attracteur de HÉNON.

(b) Attribut `parametric` en 3 dimensions : attracteur de LORENTZ.

FIG. 3 –: *Tracé d'équations avec spécification explicite des coordonnées x, y et z.*

Jusqu'à présent, nous n'avons rien dit sur la façon dont étaient tracé les tissu 8$_{1/2}$. En effet, les exemples que nous avons vu représentaient graphiquement les tissus vectoriels et de dimensions 2 en utilisant le temps comme dimension supplémentaire. Ces représentation graphiques sont *implicites* : la première dimension collection des tissus est représentée suivant l'axe des X, la seconde dimension suivant l'axe des Y.

Il est cependant parfois nécessaire d'utiliser deux ou trois tissus scalaires pour spécifier la coordonnée d'un point. Cela est possible grâce à l'utilisation de l'attribut `parametric` avec une spécification d'un axe X,

Y ou Z par l'attribut `order = ` o où o a pour valeur `x`, `y` ou `z` . Par exemple, si l'on désire tracer la courbe correspondant aux attracteurs de HÉNON, dont le programme 81/2 est :

$$x@0 = 0; \quad x = 1 + \$y - (a * \$x * \$x)\ \texttt{when}\ \texttt{Clock};$$
$$y@0 = -0.2; \quad y = b * \$x\ \texttt{when}\ \texttt{Clock};$$

ou de LORENTZ :

$$x@0 = 2; \quad x = \$x + ((-1 * a) * \$x * dt) + (a * \$y * dt)\ \texttt{when}\ \texttt{Clock};$$
$$y@0 = 3; \quad y = \$y + (b * \$x * dt) - (\$y * dt) - (\$z * \$x * dt)\ \texttt{when}\ \texttt{Clock};$$
$$z@0 = 1; \quad z = \$z + ((-1 * c) * \$z * dt) + (\$x * \$y * dt)\ \texttt{when}\ \texttt{Clock};$$

il est nécessaire de définir les gc suivant :

```
!gc x={parametric; order=x; title="Henon attractor"};
!gc y={parametric; order=y; title="Henon attractor"};
```

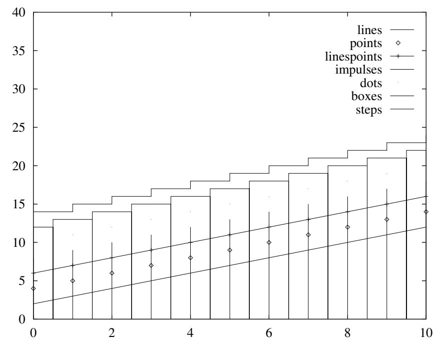
ou :

```
!gc x={parametric; order=x; title="Lorentz attractor"};
!gc y={parametric; order=y; title="Lorentz attractor"};
!gc z={parametric; order=z; title="Lorentz attractor"};
```

La figure 3 illustre l' utilisation des attributs `parametric` pour tracer une courbe paramétrique en 2 et 3 dimensions.

D.4.5 Attribut `linestyle`

Attribut	Représentation graphique
`lines`	les points sont liés par des segments
`points`	les points sont affichés par des croix
`linespoint`	les points sont liés par des points plus gros
`impulses`	les points liés par des pointillés
`dots`	les points sont affichés par des points fins
`boxes`	les points sont représentés par des rectangles
`steps`	les points sont représentés par des marches



TAB. 2 –: *Les attributs de liaison de points Gnuplot accessibles par un gc.*

L'attribut `linestyle` permet de changer le mode de liaison de deux points successifs. Les différents modes offerts sont ceux de Gnuplot. La table 2 décrit et illustre par un exemple les différents mode offerts.

D.4.6 Les attributs `color`, `window` et `title`

L'attribut `color` permet de définir la couleur du tracé. L'attribut `window` définit sur quelle fenêtre `X11` sera effectué le tracé. Il est possible de tracer des tissus sur un maximum de 10 fenêtres différentes. L'utilisation des gc permet de tracer un même tissu sur plusieurs fenêtres ; plusieurs tissus peuvent être tracés sur une même fenêtre, si leurs dimensions sont identiques (il faut que les tissus soient tous deux des scalaires ou des vecteurs). Cette capacité à tracer plusieurs tissus sur une même fenêtre permet la comparaison immédiate des éléments des tissus. Enfin, l'attribut `title` permet de donner un nom à une fenêtre de traçage.

Toutes les sorties graphiques incluses dans ce document ont été effectuées en utilisant le système que l'on vient de définir.

Annexe E

Comparaison des notions de fonctions, enregistrements, tableaux, amalgames et champs

ENREGISTREMENT	AMALGAME	FONCTION	CHAMP	TABLEAU
domaine				
label $\in$ Identificateurs	*terme* $\in$ Langage	élément quelconque	*point* $\in$ Groupe	*index* $\in [0..n_1] \times \cdots \times [0..n_d]$
codomaine				
valeur quelconque	valeur quelconque	valeur quelconque	valeur quelconque	valeur quelconque
définition partielle/totale				
fct partielle (un enregistrement ne définit de valeur que pour ses slots)	fct totale (un amalgame est un environnement qui permet d'évaluer n'importe quelle expression)	fct partielle ou totale	fct partielle (un champ n'a de valeur que pour certain de ses points)	fct totale (la valeur d'un tableau est définie pour chacun de ses index)
stricticité des constructeurs				
strict (sauf en Haskell)	non (si on considère une expression ouverte comme une valeur indéfinie)	non (les fonctions d'ordre supérieure ne sont pas nécessairement strictes)	non (la composition de champs n'est pas nécessairement stricte)	strict (sauf en Haskell)
évaluation paresseuse ou gloutonne				
gloutonne (mais évaluation paresseuse des constructeurs d'un produit en Haskell)	paresseuse (dans certain cas, on n'évalue pas A lors de l'évaluation de $A \centerdot B$)	les deux existent (appel par nom ou par valeur)	paresseuse	gloutonne (mais évaluation paresseuse en langageHaskell)
définition récursive				
non	oui	oui	oui	non (sauf pour les tableaux systoliques, en Crystal ou Alpha par exemple)

TAB. 1 –: *Comparaison des notions de fonctions, enregistrements, tableaux, amalgames et champs, 1re partie.*

ENREGISTREMENT	AMALGAME	FONCTION	CHAMP	TABLEAU
opération extensionelle (sélection d'un éléments)				
valeur d'un slot	évaluation d'un terme dans l'environnement fourni par l'amalgame	application	valeur d'un point	valeur associée à un index
opération extensionelle (définition d'un élément)				
définition d'un enregistrement en extension	définition par $\{\ldots\}$	définition en extension (par exemple les `map` en `SETL`)	définition quantifiée (mais le cas où une équation correspond à la spécification de la valeur d'un seul point est un cas dégénéré)	définition en extension (à la `C`), ou par une boucle d'itération `for`
opération intensionelle (composition)				
non applicable (un label ne peut être une valeur)	imbrication des **.**	composition $f \circ g$	non applicable (la valeur en un point n'est pas un élément d'un groupe)	gather
opération intensionelle (application d'une fonction aux éléments)				
non applicable (type des éléments non-homogène)	non applicable (type des éléments non-homogène)	α-extension : (ce sont les fonctions d'ordre supérieur)	oui	oui
opération intensionelle (β-réduction de la structure)				
non applicable (type des éléments non-homogène)	non applicable (type des éléments non-homogène)	β-réduction : généralement non (le domaine d'une fonction est usuellement infini), oui en `SETL` (pour les `map` qui ont un domaine fini)	β-réduction suivant un coset arbitraire	β-réduction suivant un axe d'un tableau (en `APL`)

TAB. 2 –: *Comparaison des notions de fonctions, enregistrements, tableaux, amalgames et champs, 2^e partie.*

Bibliographie en relation avec ce travail

[GMS95] Jean-Louis Giavitto, Olivier Michel, and Jean-Paul Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Sciences*, pages 209–215, Beaune (France), 2–4October 1995. Springer-Verlag.

[GSM92] Jean-Louis Giavitto, Jean-Paul Sansonnet, and Olivier Michel. Inférer rapidement la géometrie des collections. In *Workshop on Static Analysis, Bordeaux*, 1992.

[MDVS96] Olivier Michel, Dominique De Vito, and Jean-Paul Sansonnet. $8_{1/2}$: data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II: Proc. of the 9th Int. Symp. on Lucid and Intensional Programming*. World Scientific, May 1996.

[MG94] Olivier Michel and Jean-Louis Giavitto. Design and implementation of a declarative data-parallel language. In *post-ICLP'94 workshop W6 on Parallel and Data Parallel Execution of Logic Programs*, S. Margherita Liguria, Italy, 17June 1994. Uppsala University, Computing Science Department.

[MG95] Olivier Michel and Jean-Louis Giavitto. Typer une collection par la présentation d'un groupe. In *Journées du GDR Programmation*, Grenoble, 23–24November 1995. GDR Programmation du CNRS.

[MGS94] Olivier Michel, Jean-Louis Giavitto, and Jean-Paul Sansonnet. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In Institute for System Programming of the Russian Ac. of Sci., editor, *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21–23September 1994. Office of Naval Research USA & Russian Basic Research Foundation.

[Mic95] Olivier Michel. Design and implementation of $8_{1/2}$, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, December 1995.

[Mic96a] Olivier Michel. Design and implementation of $8_{1/2}$, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.

[Mic96b] Olivier Michel. Introducing dynamicity in the data-parallel language $8_{1/2}$. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 678–686. Springer-Verlag, August 1996.

[Mic96c] Olivier Michel. Les amalgames : un mécanisme pour la structuration et la construction incrémentielle de programmes déclaratifs. In *Journées du GDR Programmation*, Orléans, 20–22September 1996. GDR Programmation du CNRS.

[Mic96d] Olivier Michel. A straightforward translation of D0L Systems in the declarative data-parallel language $8_{1/2}$. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 714–718. Springer-Verlag, August 1996.

[MV94] Olivier Michel and Dominique De Vito. 8,5 un environnement de développement pour le langage $8_{1/2}$. In *Journées du GDR Programmation*, Lille, 22–23September 1994. GDR Programmation du CNRS.

[VM96] Dominique De Vito and Olivier Michel. Effective SIMD code generation for the high-level declarative data-parallel language $8_{1/2}$. In *Euro Micro '96*, pages 114–119. IEEE Computer Society, 2–5September 1996.

Bibliographie du document

[AFJW95] Edward A. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.

[Ama96] Robert M. Amadio. Some notes on the π-calculus. draft, unpublished, February 1996. CNRS, Sophia-Antipolis.

[Apo94] Maria-Virginia Aponte. La langage de modules de Standard ML. École de jeunes chercheurs en programmation, March 1994.

[ASS95] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Parallel and Distributed Systems*, 6(7), July 1995.

[Ast91] Edigio Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, New York, N.Y., 1991.

[Aum77] M. Aumiaux. *Logique binaire et ordinateurs*. Masson, 2 edition, 1977.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[AW93] Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming and Computer Architecture*, pages 31–41. ACM Press, June 1993.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? A functional style and its algebra of programs. *Com. ACM*, 21:613–641, August 1978.

[Bai90] James Bailey. Implementing scientific algorithms on the Connection Machine supercomputer. Technical Report 90-1, Thinking Machines, 1990.

[Bar90] Henk Barendregt. Functional programming and lambda calculus. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 7, pages 321–363. The MIT Press, New York, N.Y., 1990.

[Bar84] Henk Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1981 (1st ed) revised 84.

[BB91] A. Benveniste and G. Berry. Special section: Another look at real-time programming. *Proc. of the IEEE*, 79(9):1268–1336, September 1991.

[BCM88] J.-P. Bânatre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.

[BF82] Klaus J. Berkling and Elfriede Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55(1–3):89–101, October/November/December 1982.

[Bir87] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design, NATO ASI Series, vol. F36*, pages 217–245. Springer-Verlag, 1987.

[BKF94] S. B. Baden, S. R. Kohn, and S. F Fink. Programming with LPARX. Technical Report CSE/CS-94-377, U. of California at San-Diego, July 1994.

[BL74] J. Bard and I. Lauder. How well does turing's theory of morphogenesis work? *Journal of Theoretical Biology*, 45:501–531, 1974.

[BL84a] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Proceedings, International Symposium on the Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 1984.

[BL84b] Rod Burstall and Butler Lampson. A kernel language for modules and abstract data types. Technical Report 1, DEC SRC, September 1984.

[Ble89] Guy Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[Ble93] Guy Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[BLQ92] D. Balsara, M. Lemke, and D. Quinlan. *Adaptative, Multilevel and hierachical Computational strategies*, chapter AMR++, a C++ object-oriented class library for parallel adaptative mesh refinment in fluid dynamics application, pages 413–433. Amer. Soc. of Mech. Eng., November 1992.

[Boe95] E. J. W. Boers. Using L systems as graph grammars: G2L systems. Technical Report TR 95-30, Leiden University, 1995.

[Bou89] Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 1*, pages 149–161. LNCS 351. Springer, March 1989.

[Can91] D. C. Cann. Retire FORTRAN? A debate rekindled. In Anne Copeland MacCallum, editor, *Proceedings of the 4th Annual Conference on Supercomputing*, pages 264–272, Alburquerque, NM, USA, November 1991. IEEE Computer Society Press.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Car85] Luca Cardelli. Amber. In G. Cousineau, P-L. Curien, and B. Robinet, editors, *Combinators and Functional Languages.*, volume 242 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 1985.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL '96: The* 24^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 15–17 January 1997. ACM.

[Cas92] Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[CC82] Tsu-Wu J. Chou and George E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM Journal on Computing*, 11(4):687–708, November 1982.

[CFVN61] J. G. Charney, R. Fjörtoft, and J. Von Neumann. Numerical integration of the barotropic vorticity equation. In *Œuvres complètes de J.* Von Neumann, volume 6, pages 413–430. Pergamon Press, 1961.

[CGG+91] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Library Reference Manual*. Springer-Verlag, Berlin, 1991.

[Che86] M. C. Chen. A parallel language and its compilation to multiprocessor machines or VLSI. In *Principles of Programming Languages*, pages 131–139, Florida, 1986.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[CM89] K. Chandy and J. Misra. *Parallel Program Design - a Foundation*. Addison Wesley, 1989.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994) ; available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[Coh93] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics*. Springer-Verlag, 1993.

[col95] collectif. *Encyclopædia Universalis*, volume 10, chapter groupes (Mathématiques), page 987. 1995.

[CP97] Ron K. Cytron and Michael Plezbert. Does just-in-time equal better-late-than-never? In *Conference Record of POPL '96: The* 24^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, 15–17 January 1997. ACM.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, West Germany, January 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[cpl95] Working paper for draft proposed international standard for information systems programming language C++. Report, ANSI X3J16 Comittee, WG21, 1995.

[Dam94a] Laurent Dami. Functional programing with dynamic binding. Technical report, CUI Genève, 1994.

[Dam94b] Laurent Dami. Functions and names, without name capture. Technical report, CUI Genève, 1994.

[Dam94c] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. Ph.D. thesis, University of Geneva, 1994.

[Dam95] Laurent Dami. Type inference for λN, with application to record operations. Second Draft, April 1995.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[dBFP92] M. J. M. de Boer, F. D. Fracchia, and P. Prusinkiewicz. A model for cellular development in morphogenetic fields. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 351–370. Springer Verlag, February 1992.

[Del94] Franck Delaplace. *Compilation des communications dans un langage data-parallèle pour les architectures à réseau de communications compilées*. PhD thesis, Université de Paris-Sud, centre d'Orsay, LRI, France, February 1994.

[DG93] F. Delaplace and C. Germain. Test d'identification des communications statiques. In *5ièmes rencontres sur le parallélisme, Brest*. Univ. Brest & CNRS, 1993.

[DM88] Olivier Danvy and K. Malmkjær. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT*, pages 327–341, New York, NY, 1988. ACM.

[DMN68] Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard. SIMULA 67. common base language. Technical Report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968. Revised Edition: Publication No. S-22.

[dRS84] J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, TX*, pages 331–347, New York, NY, 1984. ACM.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, 24–26 May 1996.

[DV96] Dominique De Vito. Semantics and compilation of sequential streams into a static SIMD code for the declarative data-parallel language $8_{1/2}$. Technical Report 1044, Laboratoire de Recherche en Informatique, May 1996. 34 pages.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[FHP86] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. In Stephen Wolfram, editor, *Theory and applications of cellular automata*, number 1 in Advanced serues on complex systems, pages 358–351. World Scientific, May 1986. Appeared in the Physical Review Letters.

[FI73] A. D. Falkoff and K. E. Iverson. The design of APL. *IBM Journal of Research and Development*, pages 324–333, July 1973.

[Fib57] Leonardo Pisano Fibonacci. *Liber abbaci. (Il liber abbaci di Leonardo Pisano)*. Boncompagni, B., Rome, 1857.

[FKB96] S. J. Fink, S. R. Kohn, and S. B. Baden. Flexible communication mechanisms for dynamic structured applications. In A. Ferreira, J Rolim, Y. Saard, and T. Yang, editors, *Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, volume 1117 of *Lecture Notes in Computer Sciences*. Springer-Verlag, August 1996.

[FMSD95] J. T. Feo, P. J. Miller, S. K. Skedzielewski, and S. M. Denton. SISAL 90 users's guide. Technical report, Lawrence Livermore National Laboratory, December 1995. Draft 0.96.

[Fre95] Daniel Fredholm. Intensional aspects of function definitions. *Theoretical Computer Science*, 152(1):1–66, 11 December 1995. Fundamental Study.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 January 1996.

[FW84] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, TX*, pages 348–355, New York, NY, 1984. ACM.

[GAK94] Jacques Garrigue and H. Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *Principles of Programming Languages*, Portland, 1994.

[Gar94] Jacques Garrigue. *Label-selective lambda-calculi and transformation calculi.* PhD thesis, The university of tokyo, doctoral school of science, department of information science, 1994.

[Gar95] Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming.* World Scientific, 1995.

[GBBG86] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal, a dataflow oriented language for signal processing. *IEEE-ASSSP*, 34(2):362–374, 1986.

[GDC94] C. Germain, F. Delaplace, and R. Carlier. A static execution model for data-parallelism. *Parallel Processing Letter*, 4(4):367–378, 1994.

[GG91] N. S. Goel and M. D. Goodwin. Symbolic computation using L systems. *Applied mathematics and computation*, 42:223–253, 1991.

[GG94] Peter Grogono and Mark Gargul. A graph model for object oriented programming. *ACM SIGPLAN Notices*, 29(7):21–28, July 1994.

[GG96] W. Gellerich and M.M Gutzmann. Massively parallel programming languages – a classification of design approaches. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems 1996*, volume I, pages 110–118. ISCA, 1996. (`http://www.informatik.uni-stuttgart.de/ifi/ps/Gellerich/parlang-s.ps`).

[Gia91a] Jean-Louis Giavitto. *Otto e Mezzo : un modèle MSIMD pour la simulation massivement parallèle.* PhD thesis, Université de Paris-Sud, centre d'Orsay, December 1991.

[Gia91b] Jean-Louis Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.

[Gib94] Jeremy Gibbons. An introduction to the bird-meertens formalism. In *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, 1994.

[GJL87a] David Gelernter, Suresch Jagannathan, and Thomas London. Environments as first-class objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 98–110. ACM, ACM, January 1987.

[GJL87b] David Gelernter, Suresh Jagannathan, and Thomas London. Parallelism, persistence and meta-cleanliness in the symmetric lisp interpreter. In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 274–282. ACM, ACM, June 1987. Also available as SIGPLAN Notices 22(7) July 1987.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, pages 674–681. Addison-Wesley, 1983.

[GS93] Jean-Louis Giavitto and Jean-Paul Sansonnet. $8_{1/2}$: data-parallélisme et data-flow. *Techniques et Sciences de l'Ingénieur*, 12 - Numéro 5, 1993. Numéro spécial Langages à Parallélisme de Données.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic publishers, 1993.

[Har93] Robert Harper. Introduction to Standard ML. Technical report, School of Computer Science, Carnegie Mellon University, September 1993.

[HC89] P. M. Hilfinger and P. Colella. *Frontiers in Applied Mathematics*, chapter FIDIL: A Language for Scientific Programming, pages 97–138. SIAM, 1989.

[HC93] P. M. Hilfinger and P. Colella. FIDIL reference manual. Technical Report UCB/CSD 93-759, Computer Science Division, University of California, Berkeley, California 94720, May 1993.

[Hey96] Marie-Claude Heydemann. Support du cours réseaux d'interconnections. DEA Architectures Parallèles, 1996.

[HF92] Paul Hudak and J. H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5), May 1992.

[HHR93] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra Appl.*, 192:137–163, 1993.

[HM91] G. Hains and L. M. R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréeal, 1991.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 341–354. ACM, January 1990. An expanded version is found in CMU-CS-89-197, October 1989.

[HMS+94] Y.-S. Hwang, B. Moon, S. Sharma, R. Das, and J. Saltz. Runtime support to parallelize adaptive irregular programs. In *Proceeding of the Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.

[HO91] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software-Practice and Experience*, 21(4):375–??, April 1991.

[HO96] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, August 1996.

[Ili89] C. S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the hermite and smith normal forms of an integer matrix. *SIAM Journal on Computing*, 18(4):658–669, August 1989.

[Ive87] K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), September 1987.

[JA92] Suresh Jagannathan and Gul Agha. A reflective model of inheritance. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 350–371, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Jag89] Suresh Jagannathan. A programming language supporting first-class parallel environments. Technical Report 434, MIT LCS, 1989.

[Jag94a] Suresh Jagannathan. Dynamic modules in higher-order languages. In *International Conference on Computer Languages*. IEEE, 1994.

[Jag94b] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

[Jen95] T. P. Jensen. Clock analysis of synchronous dataflow programs. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Evaluation*, San Diego CA, June 1995.

[Joh83] S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* ACM Distinguished Dissertations. ACM Press, 1983.

[Kah92] Stefan Kahrs. Context rewriting. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *Conditional Term Rewriting Systems, Third International Workshop*, LNCS 656, pages 21–35, Pont-à-Mousson, France, July 8–10, 1992. Springer-Verlag.

[KB79] Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, November 1979.

[KB94] S. R. Kohn and S. B. Baden. A robust parallel programming model for dynamic non-uniform scientific computation. Technical Report TR-CS-94-354, U. of California at San-Diego, March 1994.

[KL87] C. G. Koster and A. Lindenmayer. Discrete and continuous models for heterocyst diffrentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica*, 36:249–273, 1987.

[Klo87] Carlos Delgado Kloos. *Semantics of digital circuits*, volume 285 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1987.

[KR93] David Keppel and Stephen Russell. Faster dynamic linking for SPARC V8 and system V.4. Technical Report 93-12-08, University of Washington, Department of Computer Science, December 1993.

[Lam88] John Lamping. A unified system of parameterization for programming languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 316–326. ACM, ACM, July 1988.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, March 1966. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

[LB78] S. Mac Lane and G. Birkhoff. *algèbre : les grands théorèmes.* Gautier–Villars, 1978.

[LB88] Butler Lampson and Rod Burstall. Pebble, A kernel language for modules and abstract data types. *Information and Computation*, 76(2/3):278–346, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 1–50.

[LC94] B. Lisper and J.-F. Collard. Extent analysis of data fields. Technical Report TRITA-IT R 94:03, Royal Institute of Technology, Sweden, January 1994.

[Leg91] Fabrice Legrand. Implémentation d'un langage data-flow synchrone pour la simulation des systèmes dynamiques discrets, Septembre 91. Rapport de stage du DEA AMIN de l'Université de Paris-Sud, Orsay.

[LF93] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, Charleston, South Carolina, January 1993.

[LF96] Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: toward a theory of incremental program construction. In *International Conference on Functional Programming*. ACM, May 1996.

[Lin68] A. Lindenmayer. Mathematical models for cellular interactions in development, parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[Lis93] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*. ACM, ACM Press, June 1993.

[Lis96] B. Lisper. Data parallelism and functional programming. In *Proc. ParaDigme Spring School on Data Parallelism*. Springer-Verlag, March 1996. Les Ménuires, France.

[LJ92] A. Lindenmayer and H. Jürgensen. Grammars of development: discrete-state models for growth, differentiation, and gene expression in modular organisms. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 3–21. Springer Verlag, February 1992.

[LQ92] M. Lemke and D. Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Parallel Processing : CONPAR 92 - VAPP V*, volume 634 of *Lecture Notes in Computer Sciences*. Springer-Verlag, September 1992.

[LS89] G. Lapaime and Patrick Sallé. Plasma-II: an actor approach to concurrent programming. *ACM SIGPLAN Notices*, 24(4):81–83, April 1989.

[LT95] Yanhong A. Liu and Tim Teitelbaum. Caching intermediate results for program improvement. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–201, La Jolla, CA, June 1995. ACM Press.

[Mae91] Patti Maes. A bottom-up mechanism for behavior selection in an artificial creature. In Bradford Book, editor, *proceedings of the first international conference on simulation of adaptative behavior*. MIT Press, 1991.

[Mah96] A. Mahiout. *Placement et ordonnancement de programmes « dataflow » à parallélisme de données sur les architecture parallèles*. PhD thesis, Université de Paris-Sud, centre d'Orsay, July 1996.

[Mar94] Jean-Claude Martzloff. Li shanlan. *Pour la science*, pages 96–106, January 1994.

[Mau89] C. Mauras. Definition of Alpha: a language for systolic programmation. Technical Report 482, INRIA, June 1989.

[MDVS96] Olivier Michel, Dominique De Vito, and Jean-Paul Sansonnet. $8_{1/2}$: data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II: Proc. of the 9th Int. Symp. on Lucid and Intensional Programming*. World Scientific, May 1996.

[Mee89] Lambert Meertens. Constructing a calculus of programs. In J. L. A. van de Snepscheut, editor, *Proceedings of the International Conference on Mathematics of Program Construction*, volume 375 of *LNCS*, pages 66–90, Berlin, June 1989. Springer.

[Mee96] Philip David Meese. The one hour Java applet — everything you need to know to build your first Java applet. using this tutorial, at the end of one or two hours you will have a working Java bar-chart applet and HTML page. *Datamation*, 42(5):51–, 1996.

[MGS94] Olivier Michel, Jean-Louis Giavitto, and Jean-Paul Sansonnet. A data-parallel declarative language for the simulation of large dynamical systems and its compilation. In Institute for System Programming of the Russian Ac. of Sci., editor, *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, 21–23September 1994. Office of Naval Research USA & Russian Basic Research Foundation.

[Mic95] Olivier Michel. Design and implementation of $8_{1/2}$, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, December 1995.

[Mic96a] Olivier Michel. Design and implementation of $8_{1/2}$, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.

[Mic96b] Olivier Michel. A straightforward translation of D0L Systems in the declarative data-parallel language $8_{1/2}$. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 714–718. Springer-Verlag, August 1996.

[Mil80] Robin Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, October 1991.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil93] Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, January 1993.

[Mis94] J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.

[MPW92] Robin Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, pages 1–40 & 41–77, September 1992.

[MW72] G. J. Mitchinson and M. Wilcox. Rule governing cell division in anaeba. *Nature*, 239:110–11, 1972.

[Nie91] Oscar Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Proceedings of Object-Based Concurrent Computing (ECOOP '91)*, pages 1–20. LNCS 612. Springer, July 1991.

[Nie93] Oscar Nierstrasz. Composing active objects. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[NO94] Susumu Nishimura and Atsushi Ohori. A calculus for exploiting data parallelism on recursively defined data (Preliminary Report). In *International Workshop TPPP '94 Proceedings (LNCS 907)*, pages 413–432. Springer-Verlag, November 94.

[NQ92] P. Naudin and C. Quitté. *Algorithmique Algébrique*. Masson, 1992.

[NSF91] Grand challenges: High performance computing and communications. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, DC 20550, 1991.

[Ode93] Martin Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Department of Computer Science, Yale University, May 1993.

[Ode94] Martin Odersky. A functional theory of local names. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 48–59, Portland, Oregon, January 17–21, 1994. ACM Press.

[Pau92] Larry Paulson. *ML for the Working Programmer*. Cambridge University Press, second, paperback edition, 1992.

[PH92] P. Prusinkiewicz and J. Hanan. L systems: from formalism to programming languages. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 193–211. Springer Verlag, February 1992.

[PKL93] J. A. Plaice, R. Khédri, and R. Lalement. From abstract time to real time. In *ISLIP'93: Proc. of the 6th Int. Symp. on Lucid and Intensional programming*, 1993.

[Pla88] J. A. Plaice. *Sémantique et compilation de LUSTRE un langage déclaratif synchrone*. PhD thesis, Institut national polytechnique de Grenoble, 1988.

[Ple96] Michael Plezbert. Continuous compilation. Master's thesis, Washington University, Sever Institute of Technology, Dep. of C.S., 1996.

[PLH$^+$90] P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, et al. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report 19, Aarhus University, 1981.

[PQ93] R. Parsons and D. Quinlan. Run-time recognition of task parallelism within the p++ parallel array class library. In *Proceedings of the Workshop of Scalable Parallel Libraries*, 1993.

[PQ94] R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Conference on Object-Oriented Numerics*, 1994.

[PRT93] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993. Obtained by anon ftp from ftp.dcs.ed.ac.uk pub/bcp/pilay.ps.

[QD96] Christian Queinnec and David DeRoure. Sharing code through first-class environments. *ACM SIGPLAN Notices*, 31(6):251–261, June 1996.

[Que96] Christian Queinnec. DMEROON, overview of a distributed class-based causally-coherent data model. *Lecture Notes in Computer Science*, 1068:297–309, 1996.

[RCe+92] Jonathan Rees, William Clinger, editors, H. Abelson, N. I. Adams IV, D. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. J. Sussman, , and M. Wand. **revised**^{4} report on the algorithmic language scheme. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), Cambridge, Massachusetts, November 1992.

[Ric93] Rice University, Houston, Texas. *High Performance Fortran Language Specification*, 1.1 edition, May 93.

[Rók93] Zsuzsanna Róka. One-way cellular automata on CAYLEY graphs. *Lecture Notes in Computer Science*, 710:406–??, 1993.

[Rók94a] Zsuzsanna Róka. *Automates cellulaires sur graphe de* CAYLEY. PhD thesis, École normale supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, 1994.

[Rók94b] Zsuzsanna Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 September 1994.

[Rók95a] Zsuzsanna Róka. The firing squad synchronization problem on CAYLEY graphs. *Lecture Notes in Computer Science*, 969:402–??, 1995.

[Rók95b] Zsuzsanna Róka. Simulations between cellular automata on CAYLEY graphs. *Lecture Notes in Computer Science*, 911:483–??, 1995.

[Rém92] Didier Rémy. Typing record concatenation for free. Research Report 1739, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[Rém93] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[Rém94] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Sciences*, pages 321–346. Springer-Verlag, April 1994.

[Rou96] François Rouaix. *Caml Applets User Guide*. INRIA Rocquencourt, April 1996.

[Roz87] Marie-Noëlle Rozin. Etude pratique des ressources de la programmation plane, July 1987. Mémoire de maitrise sous la direction de P. Greussay.

[Roz90] Marie-Noëlle Rozin. Opérateurs planaires appliqués à la construction de configurations combinatoires sur processeur simd. *Techniques et Sciences de l'Ingénieur*, 9(5), May 1990.

[San93] Davide Sangiorgi. From π-calculus to higher-order π-calculus — and back. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 151–166, Orsay, France, April 13–17, 1993. Springer-Verlag.

[Saw92] M Sawyer. A summary of fortran 90. Technical Report TR-92-04, Edinburgh, Parallel Comp Centre, 92.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: and introduction to SETL*. Springer-Verlag, 1986.

[SE94] Bjärne Strøustrup and Margaret A. Ellis. *The annotated C++ reference manual*. Addison-Wesley, 1994. 3rd edition.

[Sem93] Luigi Semenzato. The INFIDEL virtual machine. Technical Report UCB/CSD 93-761, Computer Science Division, University of California, Berkeley, 25 July 1993.

[Sem94] L. Semenzato. *An abstract machine for partial differential equations*. PhD thesis, U. of California at Berkeley, 1994.

[SG84] N. Schmitz and J. Greiner. Software aids in PAL circuit design, simulation and verification. *Electronic Design*, 32(11), May 1984.

[Sij89] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.

[Ski93] D. B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[SM91] Hans-Paul Schwefel and Reinhard Manner, editors. *Parallel problem solving from nature: 1st workshop, PPSN I, Dortmund, FRG, October 1–3, 1990: proceedings*, volume 496 of *Lecture Notes in Computer Science*, New York, NY, USA, 1991. Springer-Verlag Inc.

[Smi66] D. Smith. A basis algorithm for finitely generated abelian groups. *Math. Algorithms*, 1(1):13–26, January 1966.

[Smi82] Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, Laboratory of Computer Science, Massachusetts Institute of Technology and Cambridge, MA, 1982.

[Smi84] B. C. Smith. Reflection and semantics in lisp. In *Conference record of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, Salt Lake City, UT USA, 1984.

[Sou96] Julien Soula. Champs de données basés sur les groupes en $8_{1/2}$, Juin 1996. Rapport de stage du DEA Architectures Parallèles de l'Université de Paris-Sud.

[Ste82] Guy L. Steele, Jr. An overview of common LISP. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 98–107. ACM, ACM, August 1982.

[Ste84] G. L. Steele. *Common Lisp*. Digital Press, Burlington, 1984.

[Str94] Bjärne Strøustrup. *The design and evolution of C++*. Addison-Wesley, 1994.

[Sun95a] Sun Microsystems. *The Java^{TM} Language Specification*, 1.0 beta edition, October 1995.

[Sun95b] Sun Microsystems. *The Java^{TM} Virtual Machine Specification*, 1.0 beta edition, August 1995.

[Tal91] Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. Academic Press, 1991.

[Tal93] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112(1):99–143, 26 April 1993.

[TE68] G. L. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS Conference Proceedings*, volume 32, pages 403–408, 1968.

[Tha94] Satish R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124(1):127–148, 14 February 1994.

[Thi86] Thinking Machines Corporation, Cambridge M. *The Essential *Lisp Manual*, 1986.

[Tho94] D'Arcy W. Thompson. *Forme et croissance.* Seuil, 1994. préface de S. J. Gould, avant-propos de A. Prochiantz.

[Tor93] T. Torgersen. Parallel scheduling of recursively defined arrays: Revisited. *Journal of Symbolic Computation*, 16:189–226, 1993.

[Tur52] A. M. Turing. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc. of London*, Series B: Biological Sciences(237):37–72, 1952.

[Tur91] Greg Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 289–298, July 1991.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, December 1987.

[Vic88] Steven J. Vickers. *Topology Via Logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[Vit96] Dominique De Vito. Représentation de régions de $\mathbb{Z}^n$ par des unions de régions hypercubiques en `C++`. (personnal memo), December 1996.

[vL90] J. van Leeuwen, editor. *Handbook in theoretical computer science*. Elsevier Science Publishers, 1990.

[VP92] Eric Violard and Guy-Rene Perrin. PEI: a language and its refinement calculus for parallel programming. *Parallel Computing*, 18(10):1167–1184, October 1992.

[VP93] Eric Violard and Guy-Rene Perrin. PEI: a single unifying model to design parallel programs. In *PARLE'93*. LNCS 694, Springler Verlag, 1993.

[WA76] W. W. Wadge and E. A. Ashcroft. Lucid - A formal system for writing and proving programs. *SIAM Journal on Computing*, 3:336–354, September 1976.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language*. Academic Press U. K., 1985.

[Wad81] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[Wan93] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, pages 1–15, 93.

[Wat91] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. on Prog. Languages and Systems*, 13(1):52–98, January 1991.

[Weg87] Peter Wegner. Dimensions of object-based language design. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 168–182, Orlando, FL USA, [12] 1987. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 22, number 12.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988. Also to appear in Lisp and Symbolic Computation.

[Whi73] A. White. *Graphs, groups and surfaces*. Mathematics Studies. North-Holland, 1973.

[WKC+90] Thomas Williams, Colin Kelley, John Campbell, David Kotz, and Russell Lang. *GNUPLOT An Interactive Plotting Program*, 31 August 1990. Available in several Internet archives, including the Free Software Foundation collection on `prep.ai.mit.edu`. GNUPLOT can produce output for many different devices, including LaTeX picture mode, Postscript, and the X Window System.

[Wol88] Stephen Wolfram. *Mathematica*. Addison-Wesley, Redwood City, CA, 1988.

[Zim92] E. V. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *International Symposium, DISO'92, Bath, U.K.*, number 721 in Lecture Notes in Computer Sciences. Springer Verlag, April 1992.

Table des matières du document

Résumé

Les travaux de recherche effectués dans cette thèse s'inscrivent dans le cadre du projet **81/2** qui développe des structures de données et de contrôle *expressives* et efficaces pour la simulation de systèmes dynamiques. L'objectif de ce travail est de concevoir, étudier et développer des représentations dynamiques de l'espace dans un cadre déclaratif.

Nos travaux ont consisté à introduire dans 81/2 deux nouvelles structures de données, les *GBF* et les *amalgames*, en proposant une formalisation et en étudiant leur implémentation. Les GBF permettent de représenter des espaces réguliers et homogènes, tandis que les amalgames permettent de construire, par calcul, des espaces hétérogènes et *ad-hoc*. Ces deux nouvelles notions trouvent directement leur application dans le domaine de la simulation des systèmes hautement dynamiques (comme par exemple les processus de croissance en biologie). Elles trouvent aussi une application directe en informatique classique, en fournissant un nouveau cadre théorique pour 1) la spécification, l'analyse et l'implémentation de données récursives (les GBF permettent en particulier de considérer les arbres et les tableaux dans le même cadre théorique) ; 2) la conception et la formalisation des nouveaux mécanismes de programmation incrémentielle qui commencent à apparaître dans des langages tels que Java (les amalgames permettent en particulier de conjuguer à la fois un mécanisme d'instanciation par capture implicite et d'extension des programmes). GBF et amalgames sont d'abord étudiés pour eux-mêmes puis sont introduits et intégrés au langage déclaratif 81/2 pour donner lieu à la définition du langage $81/2_{\mathcal{D}}$.

Nous avons montré, par de nombreux exemples significatifs, la pertinence des choix effectués. Ceux-ci mettent en évidence le gain en expressivité apporté par l'enrichissement de la notion d'espace, et des primitives permettant la définition d'objets sur ces espaces. Les notions de GBF et d'amalgame permettent la définition, de façon extrêmement concise, de structures de données régulières et irrégulière, dans un cadre déclaratif, et ouvrent de nouvelles voies pour la paramétrisation et la construction incrémentielle de programmes.

Abstract

The work presented in this thesis is part of the **81/2** project that develops *expressive* and efficient data and control constructs for the simulation of dynamical systems. The aim of this work is to define, study and develop dynamical representations of space within a declarative framework.

We have introduced two new data structures in 81/2, the *GBF* and the *amalgams*, by proposing a formalization and studying their implementation. GBF allow the definition of regular and homogeneous spaces, while the amalgams allow the construction, through some computations, of heterogeneous and *ad-hoc* spaces. These two new notions have direct applications in the domain of simulation of highly dynamical systems (as for example growing processes in biology). They also find a direct application in computer science by defining a new framework for 1) the definition, analysis and implementation of recursive data (the GBF define for example a unified framework for the notion of array and tree); 2) the definition and formalization of new incremental programming mechanisms that are arising in recent languages like Java (amalgams allow the formalization of an instantiation mechanism by implicit name capture, and program extension). GBF and amalgams are first studied on their own and are afterwards introduced and integrated into the declarative framework of the language 81/2 to define the language $81/2_{\mathcal{D}}$.

We have shown, through numerous and significant examples, the pertinence of our choices. They put into evidence the gain of expressivity brought by the improvement of the notion of space, and by the primitives allowing the definition of objects onto those spaces. The notions of GBF and Amalgam allow the definition, in a very concise manner, of regular and irregular data structures, within a declarative framework, and open some new perspectives for the parameterization and the incremental construction of programs.