N° d'ordre : 2567

THÈSE

présentée à

l'Université Louis Pasteur de Strasbourg

par

STÉPHANE GENAUD

pour obtenir le grade de

Docteur de l'Université Louis Pasteur
Mention SCIENCES
Spécialité INFORMATIQUE

Transformations de programmes PEI : application au parallélisme de données

soutenue le 9 Janvier 1997 devant la commission d'examen composée de :

M. Luc Bougé, Rapporteur,
Professeur à l'Ecole Normale Supérieure de Lyon

M. Patrice Quinton, Rapporteur,
Professeur à l'Université de Rennes I

M. Christian Lengauer, Examinateur,
Professeur à l'Université de Passau

Mme Catherine Mongenet, Rapporteur interne,
Maître de Conférences à l'Université Louis Pasteur de Strasbourg

M. Guy-René Perrin, Directeur de Thèse,
Professeur à l'Université Louis Pasteur de Strasbourg

Remerciements

Je tiens à remercier MM. Luc Bougé et Patrice Quinton d'avoir accepté d'être les rapporteurs de cette thèse, et d'avoir porté autant d'intérêt à ces travaux. Je remercie également M. Christian Lengauer, qui a su, malgré l'obstacle linguistique, examiner cette thèse avec la plus grande clairvoyance. Je remercie également Mme Catherine Mongenet d'avoir présidé ce jury.

Je ne remercierai jamais assez les initiateurs de ce travail, M. Eric Violard et M. Guy-René Perrin. M. Violard, avec qui j'ai partagé ces trois années de recherche, a su me faire profiter de sa rigueur scientifique remarquable, et de son enthousiasme pour ce projet. Je remercie de tout coeur M. Guy-René Perrin, qui a été un directeur de thèse merveilleux. Il a su user, de ses grandes qualités, tant scientifiques qu'humaines, pour m'encourager lorsque mon scepticisme prenait le dessus. Je le remercie de m'avoir fait découvrir la recherche, et de m'avoir démontré que c'était passionnant.

Je suis reconnaissant envers tous ceux, qui, de près ou de loin, m'ont soutenu durant cette période. Je remercie particulièrement les membres de l'ICPS, pour leur grande disponiblité, et leur sympathie. Enfin, je remercie de tout coeur mes amis proches et ma famille, pour leur soutien constant et leur inébranlable foi en mon travail.

Table des matières

Chapitre 1

Introduction

Durant ces dix dernières années, l'idée d'utiliser des machines parallèles pour effectuer les traitements informatiques les plus complexes s'est progressivement imposée et, aujourd'hui, les grands constructeurs d'ordinateurs proposent tous des machines parallèles pour le calcul intensif. Ces machines, pouvant inclure des centaines de processeurs et dont la puissance théorique atteindra bientôt le Téraflops, semblent bien être la solution aux limites technologiques de l'intégration des processeurs entrevues il y a quelques années.

Cependant, si l'idée de faire coopérer plusieurs processeurs sur la résolution d'un même problème semble naturelle, l'écriture des applications pour ces machines est une tâche difficile. En plus de la coordination des instructions dans le temps, comme en programmation séquentielle, le concepteur de l'algorithme parallèle doit organiser l'exécution des instructions dans l'espace, c'est-à-dire répartir ces instructions, et éventuellement les données qu'elles traitent, sur les différents processeurs. On imagine donc sans peine la difficulté de concevoir de telles applications. Prendre en compte le critère de la performance de leur exécution complique encore cette tâche. En effet, les processeurs communiquant entre eux par un mécanisme lent par rapport aux accès dans leurs mémoires privées, on constate des dégradations très importantes des performances dès que l'application engendre de nombreuses communications inter-processeurs. L'objectif des recherches actuelles est donc, d'une part d'inventer un moyen d'exploiter efficacement ces machines, c'est-à-dire de se rapprocher, pour chaque application, de la puissance théorique maximale offerte par la machine, et d'autre part de pouvoir concevoir, vérifier et maintenir les programmes le plus facilement possible. Ces problèmes ont été étudiés dans le cadre séquentiel et, il est nécessaire d'y apporter également une réponse pour la programmation parallèle.

Les voies explorées pour l'exploitation de ces machines sont de deux natures : la parallélisation automatique de progammes séquentiels et l'utilisation de modèles de programmation adaptés aux machines parallèles. La première voie est économiquement très prometteuse, étant donné l'immense quantité d'applications déjà écrites dans les modèles de programmation séquentiels. Malheureusement, si on sait aujourd'hui extraire automatiquement du parallélisme des nids de boucles, les structures itératives dynamiques (comme la structure `while`[24]) posent toujours de sérieux problèmes. Les limitations actuelles de ces compilateurs paralléliseurs font qu'ils ne peuvent prétendre pour l'instant, être une solution unique satisfaisante pour les utilisateurs. Concernant les modèles de programmation, de nombreuses propositions ont été faites, souvent inspirées de l'architecture sous-jacente de la machine. Respectant la classification de Flynn[19], les langages les plus répandus se scindent en deux catégories : les langages à *parallélisme de contrôle*, calquant l'architecture MIMD-DM, et les langages à *parallélisme de données* utilisés initialement sur des machines SIMD.

Les programmes à parallélisme de contrôle sont constitués de plusieurs processus séquentiels s'exécutant librement sur les différents processeurs, les synchronisations nécessaires entre processus se faisant par échanges explicites de message. C'est aujourd'hui le modèle le plus employé, notamment à travers les bibliothèques PVM[20] et MPI[42]. La raison de ce succès tient certainement à leur disponibilité sur tous les types de machines, y compris sur les réseaux de stations de travail, ce qui a fait d'elles un standard *de facto*.

Les programmes à parallélisme de données, ou data-parallèles, extraient le parallélisme de la structure des données, qui sont mémorisées dans des *variables parallèles*. L'application d'une instruction portant sur une variable parallèle, a pour effet d'exécuter en parallèle l'instruction sur chaque élément de la variable. Le programme apparaît comme un seul et même code séquentiel agissant sur des variables d'un type nouveau. Ce modèle de programmation, bien que très adapté aux structures de données régulières, couvre de larges domaines d'applications [26]. Les communications inter-processeurs requises sont cachées à l'utilisateur, et la tâche de répartir les éléments des variables parallèles ainsi que la génération des communications nécessaires incombe au compilateur. Il faut ici citer le High Performance Fortran (HPF) [27] qui a pour objectif de s'imposer comme le standard en matière de langage data-parallèle, car défini lors d'un forum rassemblant utilisateurs, chercheurs et constructeurs. Cependant, la norme définie n'a pas permis de mettre au point à ce jour, des compilateurs dont les performances convaincraient les utilisateurs. Le fait que les communications soient masquées au programmeur par exemple, rend difficile la génération d'un code efficace en terme de communications. Cette lacune est aussi un argument qui fait pencher le choix des utilisateurs vers PVM.

Pourtant, ce modèle possède à nos yeux des qualités intéressantes par rapport au modèle à parallélisme de contrôle, notamment en terme de clarté du code, de portabilité, et d'abstraction par rapport au modèle d'exécution. Le fait que ce modèle soit synchrone et déterministe facilite aussi beaucoup la compréhension du comportement des programmes écrits. Il subsiste toutefois des questions délicates concernant la maîtrise du modèle de programmation, la transformation et la preuve des programmes. Le formalisme PEI proposé par [56] tente d'apporter des solutions à ces problèmes. Nous montrerons qu'il possède l'essentiel des caractéristiques du modèle data-parallèle, et peut être utilisé pour modéliser les programmes écrits dans des langages data-parallèles. A côté d'autres travaux sur la sémantique et la preuve des programmes data-parallèles (*cf.* [5] par exemple), PEI fournit un cadre formel pour l'expression de ces programmes, dans la lignée des langages d'équations récurrentes comme ALPHA[39] ou CRYSTAL[12]. Grâce à ce support formel, définissant clairement la sémantique des opérations, il devient possible de raisonner sur les programmes. Une application intéressante est par exemple un contrôleur de type vérifiant que les opérations sont appliquées sur des domaines cohérents. Mais l'apport le plus significatif de PEI est certainement la possiblité de transformer les programmes, sur la base d'un calcul de raffinement défini à partir de la sémantique des opérations du langage. Ce procédé de raffinement peut servir entre autres, à montrer l'équivalence de différents programmes, ou dériver des solutions différentes à partir d'un programme initial. Ces aspects nous semblent être des caractéristiques intéressantes, voire indispensables pour le génie logiciel dans le domaine de l'algorithmique parallèle. Quant au placement des données, il est explicite en PEI et fait partie intégrante du programme. Les transformations que l'on peut effectuer sur un programme modifient par conséquent ce placement des données ; cette modification du placement constitue d'ailleurs souvent une motivation à part entière pour tenter de trouver un programme plus efficace après transformation.

Les possibilités qu'apporte PEI concernent donc pleinement le problème de l'exploitation efficace des machines parallèles évoqué précédemment. L'objectif est de faire profiter l'utilisateur de ce formalisme, et de l'assister à travers des outils, qui à terme doivent automatiser le plus de choses possibles. Dans cette thèse, nous essaierons de montrer ce que pourrait être un environnement logiciel de conception de programmes parallèles reposant sur le formalisme PEI. Cet environnment doit inclure les étapes de conception des programmes, de transformation, et enfin celle de la génération de code data-parallèle.

Le premier chapitre expose les principes fondamentaux du formalisme, ainsi que les aspects syntaxiques à travers quelques exemples simples. Suit dans un second chapitre, une définition des règles et du calcul de raffinement tels qu'ils sont définis en PEI. Nous montrerons aussi dans ce chapitre les prototypes d'outils qui peuvent être associés à la construction des programmes et à leur raffinement.

Le troisième chapitre illustre l'utilisation du calcul de raffinement pour effectuer des transformations plus globales sur un programme. Il s'agit en fait de l'application de plusieurs règles de raffinement selon un schéma se présentant fréquemment. Nous qualifions ces transformations de *stratégies* et nous les employons sur un exemple.

Le quatrième chapitre sera consacré à l'outil de traduction de PEI vers HPF. Nous exposons ici les mécanismes permettant de passer d'un énoncé équationnel Pei à un programme impératif data-parallèle, et les restrictions imposées sur ces énoncés pour pouvoir les traduire. Nous montrons ensuite ce qui a été implanté dans l'outil actuel. Enfin, nous proposons quelques pistes pour améliorer ce traducteur dans le but de profiter de la sémantique particulière d'HPF. Le langage HPF possède la particularité par rapport aux autres langages data-parallèles, d'être insensible, du point de vue de la correction du programme, au placement des données. Ceci permet d'écrire des algorithmes dont le texte est proche du programme séquentiel. Cette caractéristique paraît intéressante pour la vérification intuitive de la correction du programme, par simple comparaison avec les algorithmes séquentiels existants.

En conclusion, nous replaçons Pei dans le contexte des travaux comparables, et nous exposons les particularités de ce langage. Nous soulignons l'intérêt de ce type de formalisme, en rappelant les techniques de transformations développées dans le troisième chapitre. Nous insistons sur la nécessité de disposer d'outils pour manipuler un tel formalisme, et nous avançons des idées de fonctionnalités supplémentaires que nous pourrions intégrer à ces outils. Nous parlons également des objectifs visés avec la traduction vers le langage HPF, et nous concluons sur les améliorations possibles de cette traduction.

Chapitre 2

Définition du formalisme Pei

Depuis quelques années, un certain nombre de travaux sur l'analyse statique des programmes ont abouti à des techniques de parallélisation automatique : recherche de boucles parallèles, distribution des calculs et des données, *etc.* Les techniques de réécriture de nids de boucles, en vue de leur parallélisation, reposent sur des manipulations de nature géométrique du domaine d'itération ou des domaines d'indices des tableaux, sous des hypothèses fortes, comme la linéarité des fonctions de dépendance et des bornes d'itération, par exemple. Des considérations du même type se retrouvent en programmation data-parallèle, dans les études sur l'alignement et la distribution des données, ou des communications induites. Ces remarques montrent l'importance d'une certaine forme de géométrie dans la sémantique des programmes parallèles, et l'intérêt de pouvoir manipuler les indices des tableaux de manière abstraite, au même titre que les définitions des calculs, en exprimant le placement des valeurs dans un *référentiel* qui permette de distinguer chacune d'elles par des coordonnées discrètes. La transformation du placement des valeurs peut alors permettre de définir des calculs efficaces sur une architecture parallèle donnée.

C'est ainsi que l'activité de programmation peut fondamentalement reposer sur un très petit nombre de primitives nécessaires à l'écriture des programmes. Nous en mettons trois en évidence :

- l'application de fonctions sur des valeurs, pour définir les calculs,
- le déplacement de valeurs dans le référentiel, pour exprimer les dépendances, et par conséquent les éventuelles communications,
- le changement de référentiel, pour transformer la structure des données et des calculs.

Ces idées très simples sont le fondement du formalisme Pei que nous allons maintenant rappeler. La définition originale de Pei se trouve dans la thèse d'Eric Violard [56] et les articles [58],[59] affinent les idées présentées dans sa thèse.

2.1 Introduction à la programmation Pei

2.1.1 Les champs de données

De manière générale, considérons qu'un problème est spécifié par une relation entre des *multi-ensembles* de valeurs, appelés ses *entrées* et ses *sorties*. En Pei, les valeurs d'un multi-ensemble sont placées dans un référentiel pour former un *champ de données*[1].

Evidemment, pour un multi-ensemble donné, il y a une infinité de référentiels envisageables, et pour un référentiel choisi, plusieurs placements possibles. Un placement possible pour le multi-ensemble de valeurs $V = \{1, 2, -3, 1\}$, est celui qui consiste à associer des indices de $\mathbb{Z}$, $\{0, 1, 2, 3\}$ par exemple, à chaque élément de V, comme le montre la figure 2.1(a). Définissons ce placement comme formant le champ de données `V`. Ce multi-ensemble peut également être placé dans un domaine d'indices (i, j) de $\mathbb{Z}^2$, tels que $(0 \leq i, j \leq 1)$ par exemple (figure 2.1(b)). On forme ainsi un autre champ de données `M`. Ces deux champs expriment différemment le même multi-ensemble de valeurs, et nous disons qu'ils sont *équivalents* (pour plus de précisions sur cette notion d'équivalence, on se reportera à la section 2.3.4). Formellement, il existe une bijection permettant de passer du premier pla-

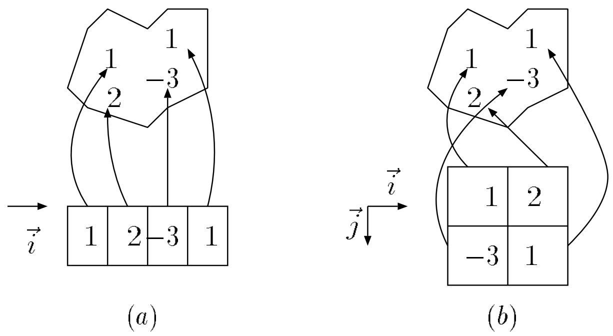

Fig. 2.1 - *Deux placements différents d'un multi-ensemble de valeurs*

cement au second. En l'occurrence, cette bijection est $\sigma(i) = (i\ mod\ 2, i\ div\ 2)$. En Pei nous exprimons cette relation entre les deux champs de données par l'équation suivante :

$$\begin{array}{ll} & \texttt{M} = \texttt{align}::\texttt{V} \\ \text{avec} & \texttt{align = }\lambda\texttt{(i)}|\texttt{(0}\leq\texttt{i}\leq\texttt{3).(i mod 2,i/2)} \end{array}$$

1. A la façon des *tableaux* dans les langages impératifs

L'application d'une fonction comme `align` sur un champ de données est appelée une *opération*, et rend comme résultat un autre champ de données. L'équation précédente exprime l'*égalité* entre les champs de part et d'autre de l'équation, c'est-à-dire `M` et `(align :: V)`. Ceci signifie qu'ils ont les mêmes valeurs, et que ces valeurs sont placées de la même manière sur le même référentiel.

2.1.2 Les opérations et l'opérateur de superposition

Les opérations

Nous distinguons trois sortes d'opérations en PEI.

- Celle que nous venons de voir modifie le référentiel dans lequel sont représentées les valeurs et nous l'appelons *changement de base*. L'application d'un changement de base est dénotée par le symbole : :

- Une autre opération modifie le placement des valeurs dans ce référentiel. Elle permet, par exemple, d'exprimer les communications dans une machine parallèle. Nous l'appelons opération de *routage*, et son application est dénotée par le symbole $\lhd$ [2]. La figure 2.2 montre le placement des valeurs du champ de données `W` défini par `W` $=$ `V` $\lhd$ `shift`, avec `shift=`λ`(i).((i-1) mod 4)`.

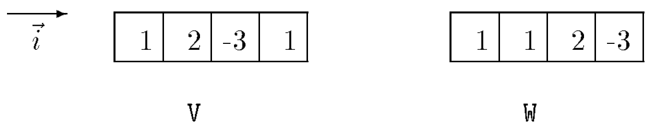

FIG. 2.2 - *Le champ* `W` *défini par un routage des valeurs de* `V`

  Noter que la fonction intervenant dans une opération de routage n'est pas nécessairement injective. Ceci permet d'exprimer des opérations de type diffusion : par exemple, l'équation `W` $=$ `V` $\lhd$ `spread`, avec `spread=`λ`(i)|(0<=i<4).(0)` diffuse la valeur placée au point d'indice 0 de `V` pour former le champ `W`. Ceci est illustré par la figure 2.3.

- La troisième opération est celle qui change les valeurs, appelée opération *fonctionnelle*. Elle permet d'appliquer un même calcul aux valeurs d'un champ de don-

2. L'opération est suffixée car la composition s'applique à droite (*cf.* section 2.3.1)

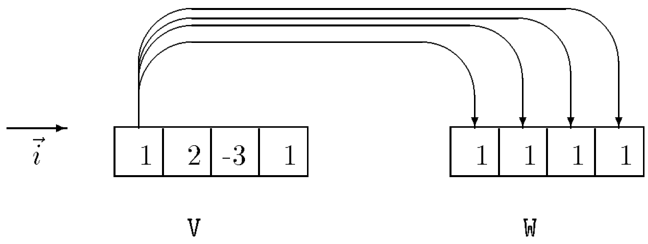

FIG. 2.3 - *Le champ* W *défini par un routage (diffusion) des valeurs de* V

nées. L'application de l'opération fonctionnelle est dénotée par le symbole $\triangleright$ [3]. Par exemple, W = inc $\triangleright$ V, avec `inc=`λ`(a).(a+3)`, est un champ dont chaque valeur est calculée à partir de la valeur de V placée au même point (figure 2.4).

FIG. 2.4 - *Le champ* W *défini par une opération fonctionnelle sur* V

On définit également l'inverse de l'opération de routage et de l'opération fonctionnelle, quelle que soit la fonction, qui sont respectivement la *réduction géométrique* et *la réduction fonctionnelle* [59]. Nous n'utiliserons que la réduction géométrique dans cette présentation. La réduction géométrique d'un champ de données X par une fonction g (dénotée par g ;$\triangleright$ X), place en tout point de coordonnées z, une séquence des valeurs de X situées en tous les points y, $y \in dom($g$)$, tels que g$(y) = z$. Par exemple, le champ Y défini par Y = merge ;$\triangleright$ X avec `merge=`λ`(i)|(0<=i<3).(0)` a toutes ses valeurs placées au point d'indice 0, dans une séquence d'ordre quelconque. La figure 2.5 montre une définition possible du champ Y.

Nous disons que cette opération, est l'inverse de l'opération géométrique car quand la fonction est bijective, la réduction est égale à l'opération géométrique définie par la fonction inverse.

3. L'opération est préfixée car la composition s'applique à gauche (*cf.* section 2.3.1)

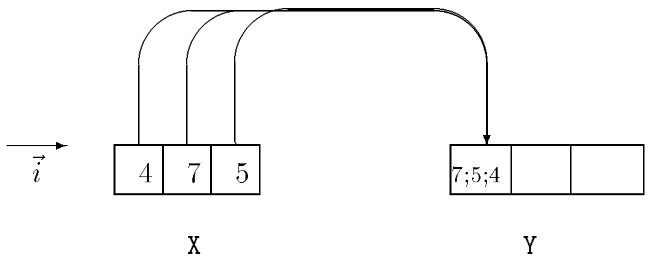

FIG. 2.5 - *Le champ* `Y` *défini par une réduction géométrique des valeurs de* `X`

Opérateur de superposition

En plus de ces trois opérations et de leurs inverses, on définit en PEI, un opérateur interne de superposition, noté `/&/`. La superposition de plusieurs champs de données permet de définir un autre champ de données, dont les valeurs sont des séquences. Chaque séquence est la concaténation des valeurs des champs qui sont placées aux mêmes indices dans le référentiel. Ainsi, la figure 2.6 montre le champ résultant de la superposition de `X` et `W`.

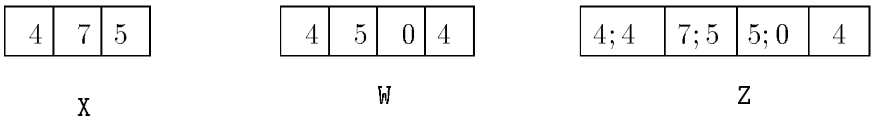

FIG. 2.6 - *Le champ* `Z` *défini par* `Z` = `X/&/W`

2.1.3 Les programmes PEI

Un *énoncé* PEI est un ensemble d'équations permettant de définir des champs de données en sortie en fonction des champs de données en entrée. Lorsqu'ils satisfont aux équations, l'ensemble de ces champs de données, et les champs intermédiaires pouvant apparaître dans les équations, constituent une *solution* du système d'équations. Les champs de données en entrée et en sortie sont précisés dans l'entête de l'énoncé. Les fonctions apparaissant dans les opérations, définissent le contexte de l'énoncé, sous la forme de λ-expressions. Dans la théorie PEI, certains énoncés seulement sont des programmes.

Définition 1 (programme PEI)
Un énoncé PEI est un *programme* si, pour tout jeu de champs de données en entrée, il existe au plus un ensemble de champs de données en sortie définissant une solution du système d'équations.

Cette définition a pour conséquence de ne pas considérer comme des programmes, les énoncés ayant des résultats non-déterministes.

Nous pouvons d'ores et déjà illustrer à l'aide de quelques exemples simples, les primitives du langage :

Exemple 1 *Addition d'une matrice et de sa transposée*

$\mathtt{MatSum : A} \mapsto \mathtt{C}$

$$\begin{cases} \mathtt{A} = \mathtt{matrix} :: \mathtt{A} \\ \mathtt{T} = \mathtt{A} \lhd \mathtt{transp} \\ \mathtt{C} = \mathtt{add} \rhd \mathtt{(A\ /\&/\ T)} \end{cases}$$

$$\begin{array}{lcl} \mathtt{matrix} & = & \lambda\mathtt{(i,j)}|\mathtt{(0}\leq\mathtt{i}<\mathtt{n}\,\&\,\mathtt{0}\leq\mathtt{j}<\mathtt{n)}.\mathtt{(i,j)} \\ \mathtt{transp} & = & \lambda\mathtt{(i,j)}|\mathtt{(0}\leq\mathtt{i}<\mathtt{n}\,\&\,\mathtt{0}\leq\mathtt{j}<\mathtt{n)}.\mathtt{(j,i)} \\ \mathtt{add} & = & \lambda\mathtt{(a;b)}.\mathtt{(a+b)} \end{array}$$

- la première équation est une précondition sur le champ de données en entrée `A`. Elle exprime que `A` n'est pas modifié par l'application du changement de base `matrix`. Ce changement de base est l'identité sur le domaine carré $(0 \leq i, j < n)$: comme nous le verrons par la suite, ceci signifie que le domaine discret sur lequel on place les valeurs de `A` est inclus dans ce carré.

- la deuxième équation définit le champ `T` par une opération de routage appliquée sur `A`. De manière générale, un routage exprime des dépendances sur le référentiel : le champ $\mathtt{X} \lhd \mathtt{g}$ est tel que ses valeurs de coordonnées $z \in dom(\mathtt{g})$ sont issues des points de coordonnées $\mathtt{g}(z)$ de `X`. En l'occurrence sur l'exemple, les valeurs placées aux points de coordonnées (i, j) dans `T` sont celles placées en (j, i) dans le champ `A`.

- la troisième équation définit le champ de données en sortie `C`. La partie droite définit la superposition de `A` et `T`, *i.e.* l'ensemble des couples `(a;b)`, avec $\mathtt{a} \in \mathtt{A}$ et $\mathtt{b} \in \mathtt{T}$. L'opération fonctionnelle définie ici consiste en l'application de la fonction `add` à tous les éléments de ce champ.

$\Diamond$

Exemple 2 *Somme de n nombres*

Pour calculer la somme de n nombres placés initialement sur le champ `A`, nous pouvons amener l'ensemble des valeurs en un point unique et y former la séquence de ces valeurs. On applique ensuite une opération fonctionnelle pour sommer cette séquence. La manière la plus générale et la plus simple d'exprimer ce déplacement est d'utiliser l'opération de réduction géométrique : on précise seulement la destination des valeurs, sans dire comment elles sont acheminées vers cette destination. Cette expression traduit en fait la définition globale du scalaire $t = \sum_{i=0}^{n-1} a_i$. Le programme s'écrit simplement[4] :

```
RedSum : A ↦ T
{ A = vector :: A
{ T = sum ▷ (red ;▷ A)
 vector  =  λ(i)|(0<=i<n).(i)
 red     =  λ(i)|(0<=i<n).(n-1)
 sum     =  id # λ(a;b).(a+sum(b))
◊
```

On peut également expliciter le chemin suivi par les valeurs pour atteindre la destination. Dans ce but, nous utilisons une définition récursive pour définir le champ `R` :

```
PrefixSum : A ↦ R
{ A = vector :: A
{ R = add ▷ (A /&/ (R ◁ prev))
 vector  =  λ(i)|(0<=i<n).(i)
 prev    =  λ(i)|(0<i<n).(i-1)
 add     =  id # λ(a;b).(a+b)
```

Nous calculons dans ce programme la somme de n entiers mais également les sommes partielles, ou sommes préfixes, c'est-à-dire les valeurs $r_i = \sum_{j=0}^{i} a_j$ pour $i \in [0 \ldots n-1]$. On vérifie aisément que le champ `R` contenant les sommes partielles des n nombres placés sur `A` est solution de l'équation `R = add ▷ (A /&/ (R ◁ prev))`. Pour s'en convaincre, la figure 2.7 montre que le champ `R` contenant les sommes partielles des entiers consécutifs de 1 à 6 est solution de cette équation pour le champ `A` donné sur l'exemple.

Notons qu'on peut obtenir le même résultat que celui calculé par le programme `RedSum`, en définissant `T= R ◁ last`, avec `last = λ(i)|(i=n-1).(i)`, sélectionnant ainsi uniquement la valeur placée en $i = n - 1$.

---

4. La fonction `id` et le symbole # sont définis en section 2.2

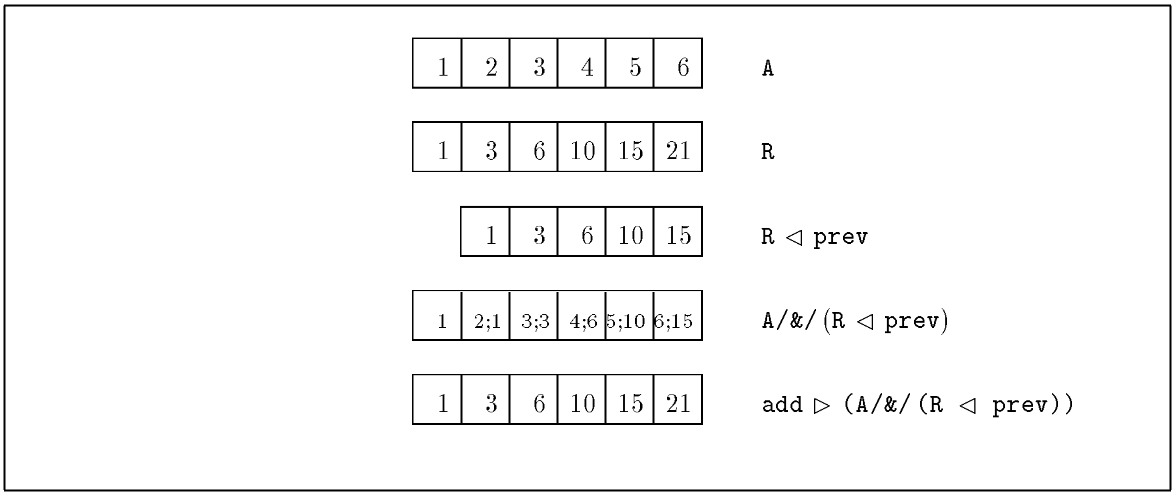

FIG. 2.7 - *Le champ* `R` *contient les sommes partielles du champ* `A`

Vérifier qu'un champ appartient à la solution du système d'équations est la façon la plus abstraite de considérer un programme PEI. On peut évidemment souhaiter pouvoir calculer une telle solution, c'est-à-dire associer une sémantique *opérationnelle* à ces équations. Nous y reviendrons en détail dans le dernier chapitre. Sur cet exemple simple on pourrait naturellement déterminer le champ solution en calculant pour chaque étape de la récurrence, la somme de la séquence de valeurs obtenue à cette étape, ceci par superposition de `A` avec ce même champ dont les valeurs ont été décalées sur la droite d'une position. Le champ solution est finalement obtenu lorsque le décalage vers la droite n'est plus possible, les valeurs n'étant plus dans le domaine de la fonction `prev`.

En somme, le programme `PrefixSum` est une version plus précise, concernant les communications induites, que le programme `RedSum`. Comme nous le verrons dans le chapitre 3, nous dirons que ce programme *raffine* `RedSum`.

Exemple 3 *Détection de palindromes*

Un vecteur V est un palindrome si $V_i = V_{n-1-i}, \forall i \in [0 \ldots n-1]$. Pour écrire le programme PEI, qui détecte si un vecteur est un palindrome, nous n'avons pas besoin de plus de primitives que pour le programme `RedSum`. Dans le programme suivant, la valeur booléenne du champ `R` placée en $i = 0$ indique si le placement des valeurs du champ de données `A` forme un palindrome.

$$\texttt{Palind : A} \mapsto \texttt{R}$$
$$\begin{cases} \texttt{T} = \texttt{equ} \rhd \texttt{(A /\&/ (A} \lhd \texttt{opp))} \\ \texttt{R} = \texttt{and} \rhd \texttt{(red ;} \rhd \texttt{T)} \end{cases}$$

$$\begin{array}{lcl} \texttt{equ} & = & \lambda\texttt{(a;b).(a = b)} \\ \texttt{opp} & = & \lambda\texttt{(i)|(0}\leq\texttt{i}\leq\texttt{n/2).(n-1-i)} \\ \texttt{red} & = & \lambda\texttt{(i)|(0}\leq\texttt{i}\leq\texttt{n/2).(0)} \\ \texttt{and} & = & \texttt{id} \;\#\; \lambda\texttt{(a;b).(a\& (and b))} \end{array}$$

$\Diamond$

2.2 Objets du langage

Nous venons de voir intuitivement quelques programmes Pei. Nous détaillons maintenant les notations utilisées, et les objets du langage Pei.

- Les programmes : la structure d'un programme Pei est la suivante. Un programme P possède en entrée un n-uplet $(\texttt{I}_1, \ldots, \texttt{I}_n)$ de champs de données, et un m-uplet $(\texttt{O}_1, \ldots, \texttt{O}_m)$ de champs de données en sortie.

  Les champs en entrée et les champs en sortie sont liés par des équations appartenant au système d'équations S, éventuellement à travers des équations portant sur des champs de données intermédiaires.

  $$P \;:\; (\texttt{I}_1, \ldots, \texttt{I}_n) \mapsto (\texttt{O}_1, \ldots, \texttt{O}_m)$$
  $$\{\ \mathrm{S}$$
  [définitions de fonctions]

- Les équations entre champs de données : chaque équation (non-orientée) du système S exprime l'égalité des expressions de champs de données de part et d'autre du symbole égale. Une expression de champ de données est soit celle d'un champ simple, soit celle d'une superposition de champs de données, soit celle d'une suite d'opérations. Par exemple, `plus` $\rhd$ `(A /&/ B)` ou encore `(carre` $\rhd$ `A)` $\lhd$ **`shift`** sont des expressions de champ de données.

- Les opérations : nous avons vu que les opérations consistaient en l'application d'une fonction sur un champ de données. La notation des fonctions en Pei est dérivée du lambda-calcul. Une fonction `f` de domaine dom(`f`) = {`x`| `P(x)`} est notée par l'expression λ`(x)|P(x).f(x)`. Par commodité, nous pouvons écrire λ`(x).f(x)` au lieu de λ`(x)|true.f(x)`, et λ`(x)|P(x)` au lieu de λ`(x)|P(x).(x)`.

Une fonction `f` définie sur des sous-domaines disjoints est notée `f1` # `f2`, et le domaine d'une fonction composée `f` $\circ$ `g` est {`x` $\in$ dom(`g`) | `g(x)` $\in$ dom(`f`)}.

Nous utilisons également la fonction identifiée par le mot-clé `id`, qui filtre les séquences à un élément. Cette fonction appliquée à un élément rend cet élément. Elle est fréquemment utilisée pour définir des fonctions récursives dans les opérations fonctionnelles. Ainsi, la somme récursive d'une séquence de valeurs s'écrit : `sum` = `id` # λ`(a;b).(a+sum(b))`. Dans cette définition, la fonction `sum`, sur le domaine des séquences à un élément, rend cet élément. Sur le domaine des séquences à plus d'un élément, elle rend la somme du premier élément plus la somme du reste de la séquence.

Une fonction peut apparaître directement dans l'expression de champ de données, comme dans l'expression `A`=`B` $\lhd$ λ`(i).(i+1)`, ou bien être référencée par un nom de fonction, la définition de la fonction étant reportée en dehors du système d'équations. C'est la notation que nous utilisons le plus fréquemment pour des raisons de lisibilité.

2.3 Sémantique

Nous allons maintenant exposer la sémantique des définitions PEI, toutes basées sur celle du champ de données. Rappelons qu'un champ de données est un placement des valeurs d'un multi-ensemble de valeurs V dans un référentiel, tel que l'on puisse distinguer chacune des valeurs par des coordonnées entières. Ce placement est une application $v : \mathbb{Z}^n \mapsto V$. Nous avons vu en introduction que les champs de données `V`, dont les valeurs $\{1, 2, -3, 1\}$ sont placées dans cet ordre sur l'intervalle $[0 \ldots 3]$ de $\mathbb{Z}$, et `M`, qui place ces mêmes valeurs sur les points de coordonnées $0 \leq i, j < 2$ de $\mathbb{Z}^2$, ne diffèrent que d'une bijection $\sigma(i) = (i \ mod \ 2, i \ div \ 2)$.

Plus généralement, pour un même multi-ensemble de valeurs donné, si $v : \mathbb{Z}^n \mapsto V$ est le placement définissant un champ de données `V` et $v' : \mathbb{Z}^p \mapsto V$ celui définissant un champ de données `V'`, on a $v = v' \circ \sigma$, où σ est une bijection de $dom(v)$ sur $dom(v')$. La figure 2.8 montre cette relation sur l'exemple de la section 2.1.1

Ceci veut dire que tout choix de placement définissant un champ de données est arbitraire et diffère d'une bijection par rapport à ce qu'on pourrait considérer comme un *pla-*

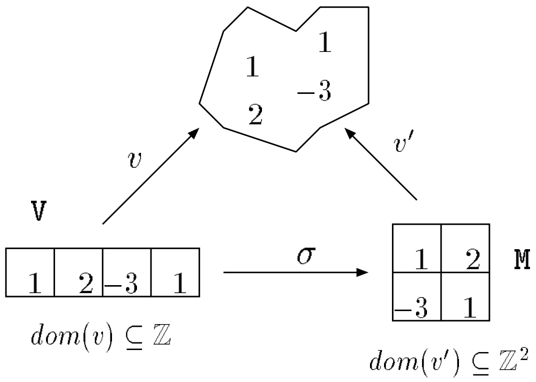

FIG. 2.8 - *Relation entre deux placements équivalents*

cement de référence. En intégrant cette bijection à la sémantique des champs de données, on peut ainsi facilement montrer la relation entre deux champs de données représentant le même multi-ensemble : ceci deviendra explicite dans l'interprétation fonctionnelle définie en section 2.3.5. Nous caractérisons donc un champ de données `X` par une paire de fonctions $(v : \sigma)$. De plus, nous imposons la contrainte suivante : $dom(v)$, que nous appelons le *domaine des valeurs* doit être inclus dans le domaine de la bijection σ, que nous appelons *domaine de référence* des valeurs, afin qu'un changement éventuel de placement concerne toutes les valeurs du champ de données.

Définition 2 (champ de données)
Soit V un multi-ensemble de valeurs. Soit v une fonction de $\mathbb{Z}^n \mapsto V$ et σ une bijection $\mathbb{Z}^n \mapsto \mathbb{Z}^p$, alors la paire $(v : \sigma)$ définit un champ de données si $dom(v) \subseteq dom(\sigma)$.

2.3.1 Sémantique des opérations

Ceci nous permet de définir la sémantique de l'application de chacune des trois opérations sur un champ `X`$= (v : \sigma)$:

Opération fonctionnelle

Soit `f` une fonction de V dans un ensemble de valeurs W, telle que $img(v) \subseteq dom(\mathtt{f})$, l'opération fonctionnelle $\mathtt{Y} = \mathtt{f} \rhd \mathtt{X}$ définit `Y` comme $(\mathtt{f} \circ v : \sigma)$.

Opération de routage (ou opération géométrique)

Soit g une fonction de $\mathbb{Z}^n \mapsto dom(v)$, telle que $dom(\mathtt{g}) \subseteq dom(\sigma)$, l'opération de routage `Y = X` $\lhd$ `g` définit Y comme étant $(v \circ \mathtt{g} : \sigma)$.

La condition sur les domaines de g et σ assure que le résultat Y est bien un champ de données, c'est-à-dire que sa paire $(v_Y : \sigma_Y)$ respecte $dom(v_Y) \subseteq dom(\sigma_Y)$.

Réciproquement, soit g une fonction partielle $dom(v) \mapsto dom(\sigma)$, la *réduction géométrique* définit g ;$\rhd$ X comme étant un champ de données $(w : \sigma)$ tel que $dom(w) = \mathtt{g}(dom(v))$ et $w(z)$ est une séquence des valeurs $v(y)$, telle que $\mathtt{g}(y) = z$. Par conséquent la propriété suivante lie la réduction géométrique et l'opération géométrique.

Proposition 1 Le champ de données g ;$\rhd$ X est égal à X $\lhd$ $\mathtt{g}^{-1}$ si et seulement si g est bijective.

Opération de changement de base

Soit h une bijection d'une partie de $\mathbb{Z}^n$ dans $\mathbb{Z}^p$, le changement de base `Y = h::X` définit Y comme étant $(v \circ \mathtt{h}^{-1} : \sigma \circ \mathtt{h}^{-1})$, si $dom(v) \subseteq dom(\mathtt{h})$.

2.3.2 Sémantique de la superposition

Soient deux fonctions f et g, nous notons $f|_{dom(g)}$ la partie de f définie sur dom(g). Ainsi $f|_{dom(g)} = g$ signifie que $dom(\mathtt{g}) \subseteq dom(\mathtt{f})$ et que f et g sont égales sur $dom(g)$. Deux champs de données X1$= (v_1 : \sigma_1)$ et X2$= (v_2 : \sigma_2)$ peuvent être superposés si $\sigma_1|_{dom(\sigma_2)} = \sigma_2$ ou $\sigma_2|_{dom(\sigma_1)} = \sigma_1$. La sémantique de l'opérateur de superposition est alors définie, pour `(X1/&/X2)`, par :

$$\begin{cases} (v_1 : \sigma) & \text{sur} \quad dom(v_1) \backslash dom(v_2) \\ (w : \sigma) & \text{sur} \quad dom(v_1) \cap dom(v_2) \\ (v_2 : \sigma) & \text{sur} \quad dom(v_2) \backslash dom(v_1) \end{cases}$$

où $w(z) = v_1(z);v_2(z)$ et $\sigma = \begin{cases} \sigma_1 \text{ si } dom(\sigma_2) \subseteq dom(\sigma_1) \\ \sigma_2 \text{ si } dom(\sigma_1) \subseteq dom(\sigma_2). \end{cases}$

2.3.3 Sémantique des équations

Nous avons vu qu'une équation définit l'égalité des expressions de champs de données de part et d'autre du symbole égale. Nous précisons maintenant la sémantique associée à une équation, qui s'exprime elle aussi à l'aide des fonctions de placement des valeurs et de la bijection σ. Par la suite nous notons $e(\texttt{C})$ une expression de champs de données fonction du champ C.

Soient les champs $\texttt{X}_1, \ldots, \texttt{X}_n$ et $\texttt{Y}_1, \ldots, \texttt{Y}_m$, et les deux expressions de champs de données $e(\texttt{X}_1, \ldots, \texttt{X}_n)$ et $e'(\texttt{Y}_1, \ldots, \texttt{Y}_m)$, respectivement égales aux paires de fonctions $(v_\texttt{X}{:}\sigma_\texttt{X})$ et $(v_\texttt{Y}{:}\sigma_\texttt{Y})$. Alors la sémantique de l'équation E :

$$e(\texttt{X}_1, \ldots, \texttt{X}_n) = e'(\texttt{Y}_1, \ldots, \texttt{Y}_m)$$

est $(v_\texttt{X} = v_\texttt{Y}) \wedge (\sigma_\texttt{X} = \sigma_\texttt{Y}) \wedge C$, où C est la conjonction de toutes les conditions imposées par les opérations apparaissant dans E. Si ce prédicat est vrai, nous dirons que les champs $\texttt{X}_1, \ldots, \texttt{X}_n, \texttt{Y}_1, \ldots, \texttt{Y}_m$ *vérifient* l'équation.

Par exemple, étant donné les champs $\texttt{A}{=}(v_\texttt{A}{:}\sigma_\texttt{A})$ et $\texttt{B}{=}(v_\texttt{B}{:}\sigma_\texttt{B})$, la sémantique de l'équation $\texttt{A} = \texttt{add} \rhd \texttt{(size :: B)}$ est $(v_\texttt{A} = \texttt{add} \circ v_\texttt{B} \circ \texttt{size}^{-1} \wedge \sigma_\texttt{A} = \sigma_\texttt{B} \circ \texttt{size}^{-1} \wedge dom(v_\texttt{B}) \subseteq dom(\texttt{size}) \wedge img(v_\texttt{B}) \subseteq dom(\texttt{add}))$.

Nous pouvons étendre la définition précédente à un système d'équations, car les équations du système sont reliées entre elles par un opérateur de conjonction. La sémantique d'un système d'équations est naturellement la conjonction des sémantiques de chaque équation. De manière analogue, nous dirons que les champ de données $\texttt{X}_1, \ldots, \texttt{X}_n$ *vérifient* un système d'équations si et seulement si $(v_{\texttt{X1}}{:}\sigma_{\texttt{X1}})$,..., $(v_{\texttt{Xn}}{:}\sigma_{\texttt{Xn}})$ sont solution du système.

2.3.4 Equivalence des champs de données

Etant donné deux champs de données, nous considérons qu'ils peuvent être équivalents à deux niveaux. Au premier niveau, appelé *équivalence faible*, ils représentent un même multi-ensemble de valeurs. Au second niveau, appelé *équivalence forte*, ils représentent le même multi-ensemble de valeurs et le même placement de référence.

Définition 3 (équivalence forte)
Soient deux champs de données X et Y. Nous disons que X et Y sont *fortement équivalents* si et seulement si il existe une bijection h telle que $\texttt{Y} = \texttt{h :: X}$ et $dom(\sigma_\texttt{X}) \subseteq dom(\texttt{h})$.

La définition de l'opération de changement de base permet de donner une idée intuitive de l'équivalence forte : elle modifie le placement du même multi-ensemble de valeurs mais garde le même placement de référence.

Par exemple, considérons la permutation h définie par $\lambda\texttt{(i)|(0<=i<4).((i-1) mod 4))}$ et le champ de données $\texttt{W} = \texttt{h :: V}$, V étant le champ de données présenté sur la figure 2.8, dont on suppose que M est le placement de référence. Nous observons sur la figure 2.9 que dans M, la valeur 2 de W est placée au même point que la valeur 2 de V par l'application de $\sigma_{\mathtt{W}}$: $v_{\mathtt{W}}(0) = v_{\mathtt{V}} \circ \mathtt{h}^{-1}(0) = v_{\mathtt{V}}(1) = 2$ and $\sigma_{\mathtt{W}}(0) = \sigma_{\mathtt{V}} \circ \mathtt{h}^{-1}(0) = \sigma_{\mathtt{V}}(1) = (1{,}0)$. Ceci est vrai pour toutes les valeurs, donc W et V sont fortement équivalents.

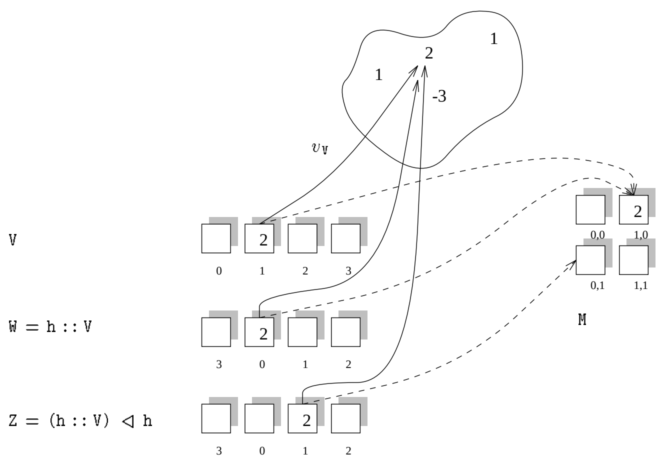

FIG. 2.9 - *Equivalences forte et faible entre champs de données*

Considérons à présent la définition $\mathtt{Z} = (\mathtt{h} :: \mathtt{V}) \lhd \mathtt{h}$. Z et V ne sont pas fortement équivalents car $\sigma_{\mathtt{Z}} \neq \sigma_{\mathtt{V}}$ ($\sigma_{\mathtt{Z}} = \sigma_{\mathtt{V}} \circ \mathtt{h}^{-1}$) alors que $v_{\mathtt{Z}} = v_{\mathtt{V}}$: nous dirons qu'ils sont faiblement équivalents. Contrairement au cas précédent, la valeur 2 de Z est placée au point $\sigma_{\mathtt{Z}}(1) = \sigma_{\mathtt{V}} \circ \mathtt{h}^{-1}(1) = \sigma_{\mathtt{V}}(2) = (0{,}1)$ de M.

Définition 4 (équivalence faible)
Soit X et Y deux champs de données. Nous disons que X et Y sont *faiblement équivalents* si et seulement si ils représentent un même multi-ensemble de valeurs.

Nous utiliserons l'équivalence faible des champs de données dans le raffinement de programmes sous la forme spécifique suivante, qui utilise des opérations PEI :

Caractérisation *Soient* X *un champ de données et* Y *une expression de champ de données fonction de* X*.* X *et* Y *sont* faiblement équivalents *si et seulement si il existe trois bijections* h*,* h′ *et* g*, telles que :*

– *ou bien* $(\mathtt{h} :: \mathtt{X}) \lhd \mathtt{g} = \mathtt{h}' :: \mathtt{Y}$ *et* $\mathtt{h}(dom(v_{\mathtt{X}})) = img(\mathtt{g})$

– *ou bien* $\mathtt{h} :: (\mathtt{X} \lhd \mathtt{g}) = \mathtt{h}' :: \mathtt{Y}$ *et* $dom(v_{\mathtt{X}}) = img(\mathtt{g})$.

Pour démontrer cette caractérisation nous montrons d'abord la propriété suivante, qui lie un champ de données X et un champ Y défini par une expression fonction de X.

Propriété 1 Soient X un champ de données et Y une expression de champ de données fonction de X. Il existe alors une paire de bijections (h,h′) telles que

$$\begin{cases} \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1} = \sigma_{\mathtt{Y}} \circ \mathtt{h'}^{-1} & (a) \\ dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h}) & (b) \\ dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h}') & (c) \end{cases}$$

Preuve. cf. Annexe A

Preuve. (caractérisation)

($\Rightarrow$) Supposons X et Y faiblement équivalents. Il existe alors une bijection α telle que $v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \alpha$, $img(\alpha) = dom(v_{\mathtt{X}})$ et $dom(\alpha) = dom(v_{\mathtt{Y}})$.

D'après la propriété 1, il existe donc deux bijections h et h′ telles que $\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1} = \sigma_{\mathtt{Y}} \circ \mathtt{h'}^{-1}$ avec $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})$ et $dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h}')$.

Soit g la bijection $\mathtt{h} \circ \alpha \circ \mathtt{h'}^{-1}$, et considérons le champ $(\mathtt{h} :: \mathtt{X}) \lhd \mathtt{g}$. Montrons que ce champ est correctement défini :

- $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})$.
- $dom(\mathtt{g}) \subseteq dom(\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1})$. En effet, par définition $dom(\mathtt{g}) = \mathtt{h}'(dom(\mathtt{h} \circ \alpha))$; or $dom(\mathtt{h} \circ \alpha) \subseteq dom(\alpha) = dom(v_{\mathtt{Y}}) \subseteq dom(\sigma_{\mathtt{Y}})$. On a donc $dom(\mathtt{g}) \subseteq \mathtt{h}'(dom(\sigma_{\mathtt{Y}})) = dom(\sigma_{\mathtt{Y}} \circ \mathtt{h'}^{-1}) = dom(\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1})$.
- $img(\mathtt{g}) \subseteq dom(v_{\mathtt{X}} \circ \mathtt{h}^{-1})$. On a même l'égalité : en effet, par définition $img(\mathtt{g}) = \mathtt{h}(img(\alpha \circ \mathtt{h'}^{-1}))$. On a donc $dom(\alpha) = dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h}')$, donc $img(\alpha \circ \mathtt{h'}^{-1}) = img(\alpha)$, c'est-à-dire $img(\mathtt{g}) = \mathtt{h}(img(\alpha)) = \mathtt{h}(dom(v_{\mathtt{X}})) = dom(v_{\mathtt{X}} \circ \mathtt{h}^{-1})$.

Le champ est égal à :

$$\begin{array}{lll} & (\mathtt{h} :: \mathtt{X}) \lhd \mathtt{g} & \\ = & (v_{\mathtt{X}} \circ \mathtt{h}^{-1} \circ \mathtt{g} : \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) & \\ = & (v_{\mathtt{X}} \circ \mathtt{h}^{-1} \circ \mathtt{h} \circ \alpha \circ \mathtt{h'}^{-1} : \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) & \text{(définition de g)} \\ = & (v_{\mathtt{X}} \circ \alpha \circ \mathtt{h'}^{-1} : \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) & \text{(car } dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})\text{)} \\ = & (v_{\mathtt{Y}} \circ \mathtt{h'}^{-1} : \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) & \text{(définition d' } \alpha\text{)} \\ = & (v_{\mathtt{Y}} \circ \mathtt{h'}^{-1} : \sigma_{\mathtt{Y}} \circ \mathtt{h'}^{-1}) & \text{(de la définition de h et h}'\text{)} \\ = & \mathtt{h}' :: \mathtt{Y} & \text{(car } dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h}')\text{)} \end{array}$$

De manière analogue, on peut définir $\mathtt{g}$ comme $\alpha \circ \mathtt{h'}^{-1} \circ \mathtt{h}$ et en déduire :
$\mathtt{h :: (X \lhd g) = h' :: Y}$

($\Leftarrow$) Supposons que $\mathtt{(h :: X) \lhd g = h' :: Y}$ avec $\mathtt{h}(dom(v_{\mathtt{X}})) = img(\mathtt{g})$. Alors, nous avons $v_{\mathtt{X}} \circ \mathtt{h}^{-1} \circ \mathtt{g} = v_{\mathtt{Y}} \circ \mathtt{h'}^{-1}$ c'est-à-dire $v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{h}^{-1} \circ \mathtt{g} \circ \mathtt{h'}$ car $dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h'})$. Soit α la bijection $\mathtt{h}^{-1} \circ \mathtt{g} \circ \mathtt{h'}$. Il vient $v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \alpha$.

Par définition de l'opération géométrique, $dom(\mathtt{g}) \subseteq dom(\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) = dom(\sigma_{\mathtt{Y}} \circ \mathtt{h'}^{-1}) \subseteq dom(\mathtt{h'}^{-1}) = img(\mathtt{h'})$. On peut donc conclure que $img(\alpha) = \mathtt{h}^{-1}(img(\mathtt{g} \circ \mathtt{h'}))$ est donc égale à $\mathtt{h}^{-1}(img(\mathtt{g})) = \mathtt{h}^{-1} \circ \mathtt{h}(dom(v_{\mathtt{X}})) = dom(v_{\mathtt{X}})$. X et Y sont donc faiblement équivalents.

□

Exemple 4 *Illustration de la caractéristation de l'équivalence faible*

La figure 2.10 montre deux champs faiblement équivalents X et Y. Les domaines de valeurs de ces deux champs sont supposés égaux (représentés par la partie grisée) mais leurs domaines de référence diffèrent, celui de X étant le vecteur, et celui de Y étant la première face du cube. Pour montrer que ces champs sont faiblement équivalents plusieurs possiblités s'offrent pour le choix des fonctions h, h′ et g pour établir l'égalité $\mathtt{(h :: X) \lhd g = h' :: Y}$ de la caractérisation.

- on peut choisir $\mathtt{h = g = \lambda(i).(i)}$ et $\mathtt{h' = \lambda(i,j,k)|(i{=}j \,\&\, k{=}0).(i)}$, pour réduire le domaine de référence de l'expression à droite de l'égalité à celui du vecteur X dans $\mathbb{Z}$.
- la deuxième possibilité est de choisir $\mathtt{h = \lambda(i).(i,i)}$, $\mathtt{g = \lambda(i,j)|(i{=}j).(i,j)}$, et $\mathtt{h' = \lambda(i,j,k)|(i{=}j \,\&\, k{=}0).(i,i)}$, pour que les domaines de référence des deux expressions soient représentés dans $\mathbb{Z}^2$.
- enfin, on peut choisir $\mathtt{h = \lambda(i).(i,i,0)}$ et $\mathtt{g = h' = \lambda(i,j,k)|(i{=}j \,\&\, k{=}0).(i,j,k)}$ pour que le domaine de référence de l'expression de gauche soit égal à celui de l'expression de droite dans $\mathbb{Z}^3$.

◊

2.3.5 Interprétation fonctionnelle

A partir des définitions sémantiques des opérations on peut préciser une *interprétation fonctionnelle* $[\![\mathtt{X}]\!]$ d'un champ de données $\mathtt{X}{=}(v : \sigma)$, c'est-à-dire donner une fonction qui

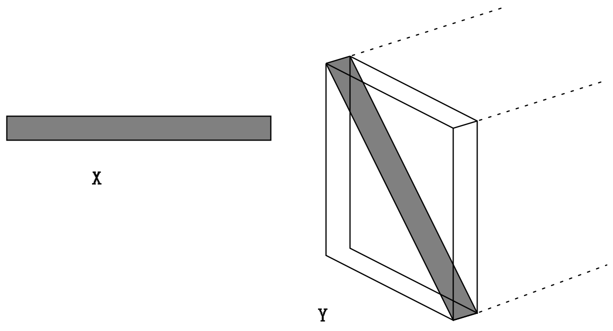

FIG. 2.10 - *Deux champs de données faiblement équivalents*

définit les valeurs d'un champ de données, quel que soit le placement de référence. Sa définition fait donc abstraction des différences éventuelles de placement d'un multi-ensemble de valeurs.

Définition 5 (interprétation fonctionnelle)
$[\![\mathtt{X}]\!] = v \circ \sigma^{-1}$

Proposition 2 $[\![\mathtt{h} :: \mathtt{X}]\!] = [\![\mathtt{X}]\!] = v \circ \sigma^{-1}$.

Preuve. De la sémantique de l'opération de changement de base et de la définition précédente, nous avons en effet :
$$\begin{aligned} [\![\mathtt{h} :: \mathtt{X}]\!] &= (v \circ \mathtt{h}^{-1}) \circ (\sigma \circ \mathtt{h}^{-1})^{-1} \\ &= (v \circ \mathtt{h}^{-1}) \circ (\mathtt{h} \circ \sigma^{-1}) \\ &= v \circ \sigma^{-1} \\ &= [\![\mathtt{X}]\!] \end{aligned}$$
□

Ainsi, le changement de base appliqué à un champ de données ne modifie pas son interprétation fonctionnelle. En conséquence, deux champs fortement équivalents, qui peuvent avoir des placements différents, ont la même interprétation fonctionnelle. De même, pour les champs de données obtenus par une opération fonctionnelle ou une opération de routage, nous avons la propriété suivante :

Proposition 3
$[\![\mathtt{f} \rhd \mathtt{X}]\!] = \mathtt{f} \circ [\![\mathtt{X}]\!]$
et
$[\![\mathtt{X} \lhd \mathtt{g}]\!] = [\![\mathtt{X}]\!] \circ \sigma \circ \mathtt{g} \circ \sigma^{-1}$

Exemple 5 *Interprétation fonctionnelle d'un champ résultat*

Considérons le programme suivant :

$$\mathtt{P} \;:\; \mathtt{V} \mapsto \mathtt{B}$$
$$\begin{cases} \mathtt{V} = \mathtt{align} :: \mathtt{M} \\ \mathtt{A} = \mathtt{M} \lhd \mathtt{spread} \\ \mathtt{B} = \mathtt{inc} \rhd \mathtt{A} \end{cases}$$

$$\begin{array}{lcl} \mathtt{align} & = & \lambda\mathtt{(i,j)|(i=0).(j)} \\ \mathtt{spread} & = & \lambda\mathtt{(i,j).(0,j)} \\ \mathtt{inc} & = & \lambda\mathtt{(a).(a+1)} \end{array}$$

Nous cherchons l'interprétation fonctionnelle du champ résultat B en fonction de l'entrée V. De l'équation $\mathtt{V} = \mathtt{align} :: \mathtt{M}$ et de la sémantique du changement de base, nous tirons les égalités : $v_V = v_M \circ \mathtt{align}^{-1}$ et $\sigma_V = \sigma_M \circ \mathtt{align}^{-1}$. Comme il n'y a pas de changement de base entre les champs A, B et M, nous avons aussi $\sigma_A = \sigma_B = \sigma_M$. De là,

$$\sigma_B \;=\; \sigma_V \circ \mathtt{align}$$

Nous avons d'autre part :

$$\begin{array}{rcl} [\![\mathtt{B}]\!] & = & [\![\mathtt{inc} \rhd \mathtt{A}]\!] = \mathtt{inc} \circ [\![\mathtt{A}]\!] \\ & = & \mathtt{inc} \circ [\![\mathtt{M} \lhd \mathtt{spread}]\!] = \mathtt{inc} \circ [\![\mathtt{M}]\!] \circ \sigma_M \circ \mathtt{spread} \circ {\sigma_M}^{-1} \\ & = & \mathtt{inc} \circ v_M \circ {\sigma_M}^{-1} \circ \sigma_M \circ \mathtt{spread} \circ {\sigma_M}^{-1} \\ & = & \mathtt{inc} \circ v_M \circ \mathtt{spread} \circ {\sigma_M}^{-1} \\ & = & \mathtt{inc} \circ v_V \circ \mathtt{align} \circ \mathtt{spread} \circ {\sigma_M}^{-1} \end{array}$$

c'est-à-dire :

$$v_B \circ {\sigma_B}^{-1} = \mathtt{inc} \circ v_V \circ \mathtt{align} \circ \mathtt{spread} \circ {\sigma_M}^{-1}$$

d'où :

$$v_B = \mathtt{inc} \circ v_V \circ \mathtt{align} \circ \mathtt{spread}$$

Nous pouvons ainsi évaluer la valeur associée à tout point $v_B(i,j)$ en calculant $(\mathtt{inc} \circ v_V \circ \lambda\mathtt{(i,j).(j)})\ (i,j)$. L'interprétation fonctionnelle est la base du traducteur PEI vers CAML[60] inclus dans l'environnement PEI, dont nous parlerons en détail dans la section 3.5. $\Diamond$

Chapitre 3

Raffinement et abstraction

3.1 Définition générale du raffinement

Le raffinement est un procédé classique de construction ou de transformation de programmes [2, 44], qui consiste à obtenir un programme satisfaisant toute spécification que satisfaisait un programme initial. La définition du raffinement est communément établie de la manière suivante, à partir d'une propriété de satisfaction choisie, notée **sat** :

Définition 6 Soient les programmes P et P', on dit que P est raffiné par P', et nous notons $P \sqsubseteq P'$, si et seulement si, pour toute spécification S,

$$P \ \mathsf{sat} \ S \Rightarrow P' \ \mathsf{sat} \ S$$

Si la propriété choisie est la correction totale par exemple, on utilisera les *weakest preconditions* de Dijkstra, notées $wp(P, post)$, où P est un énoncé et $post$ un prédicat : $wp(P, post)$ est le plus faible prédicat pour lequel l'exécution de P termine dans un état satisfaisant $post$. La définition générale du raffinement devient alors :

$$P \sqsubseteq P' \text{ ssi } \forall r \ (wp(P, r) \Rightarrow wp(P', r)).$$

La notion de raffinement est fréquemment utilisée dans un procédé continu de construction d'énoncés successifs, depuis une spécification jusqu'à un programme exécutable, prenant en compte tel ou tel modèle d'exécution. Elle définit une relation d'ordre partiel entre les énoncés, dite *relation de raffinement*. Le raffinement est généralement étendu aux spécifications elles-mêmes, avec, pour des spécifications de type [*pre*,*post*], les deux cas particuliers :

$[p, q] \sqsubseteq [p', q]$ si $p \Rightarrow p'$ (affaiblissement des préconditions) et

$[p, q] \sqsubseteq [p, q']$ si $q' \Rightarrow q$ (renforcement des postconditions).

Knapp [31] présente la définition du raffinement de spécifications sous une autre forme, pour l'appliquer aux programmes UNITY[10]. Bien qu'équivalente, la formulation en est différente du fait de la forme des spécifications utilisées (en logique modale) : la sémantique d'UNITY est définie sous la forme d'un système de transitions d'états et la logique utilisée permet d'exprimer des propriétés sur ces états. Soient s et s' deux états, on note $s(p)$ et $s'(q)$ le fait que les états satisfont les propriétés p et q respectivement. Intuitivement, dans ce type d'approche, une spécification S a la forme suivante :

$$S : \forall s(p)\ \exists s'(q).(s \rightsquigarrow s') \qquad (1)$$

où la notation $s \rightsquigarrow s'$ signifie que s' est un état "atteignable" à partir de l'état s. Le raffinement est alors défini de la manière suivante :

Définition 7 Soient deux spécifications S et S' de la forme (1), $S \sqsubseteq S'$ ssi $S' \Rightarrow S$

3.2 Raffinement des énoncés PEI

3.2.1 Définitions

La définition du raffinement des énoncés PEI s'inspire de la définition de Knapp, mais en restant dans la logique du premier ordre. Considérons un énoncé PEI, dont les entrées et les sorties sont `D` et `R` respectivement :

$$\mathtt{P} : \mathtt{D} \mapsto \mathtt{R}$$
$$\begin{cases} S_1 \\ S_2 \end{cases}$$

Nous pouvons considérer, parmi les équations du système, deux types d'équations : celles qui portent uniquement sur des entrées, et celles qui comportent des champs de données intermédiaires ou des sorties. Nous appelons les premières *pré-équations*, les secondes *post-équations*. Dans la structure schématique de l'énoncé `P` ci-dessus, les pré-équations appartiennent à l'ensemble d'équations S_1, les post-équations à l'ensemble d'équations S_2.

Définition 8 (pré-équation et post-équation)
Une *pré-équation* d'un énoncé `P` est une équation liant uniquement des champs de données en entrée de `P`. Une équation d'un énoncé `P` est une *post-équation* si elle n'est pas une pré-équation.

Nous pouvons maintenant préciser ce que spécifie un énoncé PEI. Nous associons à l'énoncé `P`, une propriété $sat(\mathtt{P})$, dont nous dirons qu'elle est satisfaite par `P`. Cette propriété précise que pour tout champ de données `D` vérifiant S_1, la solution est formée d'un champ de données `R` vérifiant S_2. Ce que nous noterons par :

$$sat(\mathtt{P}) : S_1 \Rightarrow S_2$$

Par exemple, à l'énoncé suivant :

$$\mathtt{P1} : \mathtt{A} \mapsto \mathtt{B}$$
$$\begin{cases} \mathtt{A} = \mathtt{matrix} :: \mathtt{A} \\ \mathtt{B} = \mathtt{A} \lhd \mathtt{transp} \end{cases}$$
$$\begin{array}{lcl} \mathtt{matrix} & = & \lambda\mathtt{(i,j)}|\mathtt{(0}\leq\mathtt{i,j<n).(i,j)} \\ \mathtt{transp} & = & \lambda\mathtt{(i,j)}|\mathtt{(0}\leq\mathtt{i,j<n).(j,i)} \end{array}$$

nous associons la propriété :

$$\begin{array}{ll} & (v_{\mathtt{A}} = v_{\mathtt{A}} \circ \mathtt{matrix}^{\mathtt{-1}} \wedge \sigma_{\mathtt{A}} = \sigma_{\mathtt{A}} \circ \mathtt{matrix}^{\mathtt{-1}} \wedge dom(v_{\mathtt{A}}) \subseteq dom(\mathtt{matrix})) \\ \Rightarrow & (v_{\mathtt{B}} = v_{\mathtt{A}} \circ \mathtt{transp} \wedge \sigma_{\mathtt{B}}{=}\sigma_{\mathtt{A}}) \wedge \\ & dom(\mathtt{transp}) \subseteq dom(\sigma_{\mathtt{A}}) \wedge img(\mathtt{transp}) \subseteq dom(v_{\mathtt{A}})) \end{array}$$

ce que nous noterons sous la forme :

$$(\mathtt{A} = \mathtt{matrix} :: \mathtt{A}) \Rightarrow (\mathtt{B}{=}\mathtt{A} \lhd \mathtt{transp})$$

Définition 9 (raffinement d'énoncés)
Nous disons que `P` est *raffiné* par $\mathtt{P}'$, ou que `P` *abstrait* $\mathtt{P}'$, et nous notons $\mathtt{P} \sqsubseteq \mathtt{P}'$, si et seulement si $sat(\mathtt{P}') \overset{\equiv}{\Rightarrow} sat(\mathtt{P})$.

La notation $\overset{\equiv}{\Rightarrow}$ est une relation d'implication permettant de prendre en compte l'équivalence des champs de données, dont la définition est la suivante. Dans cette définition, nous notons $S[\mathtt{X}\backslash\mathtt{X}']$ le système d'équations dont une occurrence de `X` a été substituée par $\mathtt{X}'$, et $S[\mathtt{X}\backslash\mathtt{X}']^\star$ quand toutes les occurrences de `X` ont été remplacées par $\mathtt{X}'$ dans S.

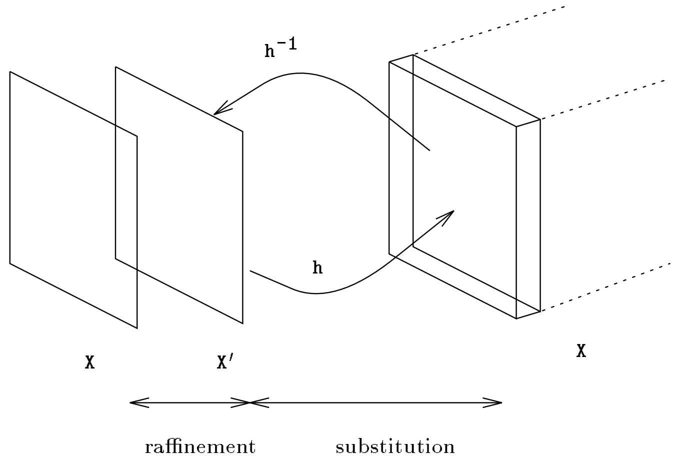

FIG. 3.1 - *Raffinement par substitution de champs équivalents*

Définition 10 (implication modulo l'équivalence)
Soit `P1` et `P2` deux énoncés et S le système d'équations de `P1`. Nous disons que sat(`P2`) *implique* sat(`P1`) *modulo l'équivalence* (forte ou faible) de champs de données, et nous notons $sat(\mathtt{P2}) \overset{\equiv}{\Rightarrow} sat(\mathtt{P1})$ si et seulement si $sat(\mathtt{P2}) \Rightarrow sat(\mathtt{P1'})$ est vérifiée, avec :

1. ou bien $\mathtt{P1'} = \mathtt{P1}\{S[\mathtt{X}\backslash e(\mathtt{X})]^\star\}$ où $e(\mathtt{X})$ est une expression dépendant de `X`, satisfaisant la propriété suivante : pour tout champ de données dénoté par cet `X`, il existe un champ de données, disons $\mathtt{X}'$, qui lui est fortement équivalent, tel que l'expression $e(\mathtt{X}')$ dénote le même champ de données que `X`,

2. ou bien $\mathtt{P1'} = \mathtt{P1}\{S[\mathtt{E}\backslash e(\mathtt{E})]\}$ où $e(\mathtt{E})$ est une expression dépendant de `E`, satisfaisant la propriété suivante : pour tout champ de donnée dénoté par cette expression `E`, il existe un champ de données, disons $\mathtt{E}'$, qui lui est faiblement équivalent, et tel que l'expression $e(\mathtt{E}')$ dénote le même champ de données que `E`.

D'après la définition 10.1, on peut évidemment remplacer les occurrences de `X` par l'expression $e(\mathtt{X}')$ qui lui est égale. Intuitivement, le raffinement consite alors à substituer $e(\mathtt{X})$ à $e(\mathtt{X}')$, sachant que `X'` est fortement équivalent à `X`. L'explication intuitive est similaire pour le cas 10.2 de l'équivalence faible.
La figure 3.1 illustre l'explication précédente : le champ `X` exprimé dans $\mathbb{Z}^3$ est égal à $\mathtt{h} :: \mathtt{X}'$, où $\mathtt{X}'$ est exprimé dans $\mathbb{Z}^2$. Le raffinement consiste à substituer $\mathtt{h} :: \mathtt{X}$ à $\mathtt{h} :: \mathtt{X}'$.

L'équivalence de deux énoncés est définie à partir du raffinement :

Définition 11 (équivalence d'énoncés)
Nous dirons que deux énoncés `P` et `P'` sont *équivalents*, et on note $\mathtt{P} \equiv \mathtt{P'}$ si et seulement si ils se raffinent mutuellement.

Exemple 6 *Raffinement par substitution d'un champ fortement équivalent*

Considérons les deux programmes suivants, `P2` et `P3`. Nous cherchons à montrer que $\mathtt{P2} \sqsubseteq \mathtt{P3}$ en utilisant l'implication modulo l'équivalence de champs de données.

$$
\begin{array}{l}
\mathtt{P2\ :\ X} \mapsto \mathtt{Y} \\
\left\{\begin{array}{l} \mathtt{X} = \mathtt{carre} :: \mathtt{X} \\ \mathtt{Y} = \mathtt{X} \end{array}\right. \\
\quad \mathtt{carre} = \lambda\mathtt{(i,j)|(0}\leq\mathtt{i,j<n).(i,j)}
\end{array}
\qquad
\begin{array}{l}
\mathtt{P3\ :\ X} \mapsto \mathtt{Y} \\
\left\{\begin{array}{l} \mathtt{X} = \mathtt{carre} :: \mathtt{X} \\ \mathtt{transp} :: \mathtt{Y} = \mathtt{X} \end{array}\right. \\
\quad \mathtt{carre} = \lambda\mathtt{(i,j)|(0}\leq\mathtt{i,j<n).(i,j)} \\
\quad \mathtt{transp} = \lambda\mathtt{(i,j)|(0}\leq\mathtt{i,j<n).(j,i)}
\end{array}
$$

Considérons `P2'` obtenu à partir de `P2` en remplaçant `Y` par $(\mathtt{transp} :: \mathtt{Y})$, conformément à la partie 1 de la définition 10. Nous vérifions immédiatement $sat(\mathtt{P3}) \Rightarrow sat(\mathtt{P2'})$ car `P3` et `P2'` sont les mêmes énoncés. D'autre part, soit $\mathtt{Y}'$ le champ $(\mathtt{transp}^{\mathtt{-1}} :: \mathtt{Y})$: sachant de l'énoncé `P2` que $dom(\sigma_{\mathtt{Y}}) \subseteq dom(\mathtt{transp}^{\mathtt{-1}})$, $\mathtt{Y}'$ est alors fortement équivalent à `Y` et $\mathtt{transp} :: \mathtt{Y}' = \mathtt{transp} :: (\mathtt{transp}^{\mathtt{-1}} :: \mathtt{Y}) = \mathtt{Y}$. Nous avons par conséquent $sat(\mathtt{P3}) \overset{\equiv}{\Rightarrow} sat(\mathtt{P2})$ et par conséquent $\mathtt{P2} \sqsubseteq \mathtt{P3}$.
$\Diamond$

Cet exemple illustre les propriétés suivantes, directement issues des définitions du raffinement (9 et 10) en utilisant l'implication modulo l'équivalence. Elles permettent en effet d'obtenir des programmes raffinés en substituant un champ par un champ fortement ou faiblement équivalent.

Propriété 2 Soit `P` un énoncé et `X` un identificateur de champ. Si $dom(\sigma_{\mathtt{X}}) \subseteq img(\mathtt{h})$ alors $\mathtt{P} \sqsubseteq \mathtt{P}\{S[\mathtt{X}\backslash\mathtt{h} :: \mathtt{X}]^{\star}\}$

Preuve. cf. Annexe A.

Remarque. La preuve met en évidence un champ de données $\mathtt{X}' = \mathtt{h}^{\mathtt{-1}} :: \mathtt{X}$ qui est tel que $\mathtt{X} = \mathtt{h} :: \mathtt{X}'$. Le raffinement a donc consisté à substituer $\mathtt{h} :: \mathtt{X}$ à $\mathtt{h} :: \mathtt{X}'$.

Propriété 3 Soit `P` un énoncé où `X` est un identificateur de champ n'apparaissant que sous la forme `h::X`. Si $dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h}) \wedge img(\mathtt{h}) \subseteq dom(\mathtt{h})$ nous avons alors : $\mathtt{P} \sqsubseteq \mathtt{P}\{S[\mathtt{h::X} \backslash \mathtt{X}]^{\star}\}$

Preuve. cf. Annexe A.

Propriété 4 Soit `P` un énoncé. Nous avons :

(1) si `h` est une inverse gauche de `g`,
$dom(v_{\mathtt{E}}) = dom(\mathtt{g})$ et $\mathtt{h}(dom(\mathtt{g})) \subseteq dom(\sigma_{\mathtt{E}}) \subseteq img(\mathtt{h})$.
$\mathtt{P} \sqsubseteq \mathtt{P}\{S[\mathtt{E} \backslash \mathtt{h::(E} \lhd \mathtt{g}^{-1})]\}$

(2) si `h` est une inverse droite de `g`,
$dom(v_{\mathtt{E}}) = img(\mathtt{g})$ et $dom(\mathtt{g}) \subseteq dom(\sigma_{\mathtt{E}}) \subseteq img(\mathtt{h})$.
$\mathtt{P} \sqsubseteq \mathtt{P}\{S[\mathtt{E} \backslash (\mathtt{h::E}) \lhd \mathtt{g}^{-1}]\}$

Preuve. cf. Annexe A.

Notons également un cas particulier de cette dernière proposition, pour lequel $\mathtt{h} = \mathtt{g}^{-1}$. Dans ce cas la propriété 4 devient :

Propriété 5 Soit `P` un énoncé, `E` une expression de champ de données, et `h` une bijection. Si $dom(v_{\mathtt{E}}) = dom(\sigma_{\mathtt{E}}) = dom(\mathtt{h}) = img(\mathtt{h})$ nous avons :

$$
\begin{array}{llll}
(1) & \mathtt{P} & \sqsubseteq & \mathtt{P}\{S[\mathtt{E} \backslash \mathtt{h::(E} \lhd \mathtt{h})]\} \\
(2) & \mathtt{P} & \sqsubseteq & \mathtt{P}\{S[\mathtt{E} \backslash (\mathtt{h::E}) \lhd \mathtt{h}]\}
\end{array}
$$

Cette dernière propriété est celle que nous avions utilisée dans le chapitre 2 pour introduire l'équivalence faible et est illustrée par la figure 2.9.

3.2.2 Cas particuliers

Nous donnons ici un ensemble d'exemples classiques qui n'utilisent que l'implication simple.

Exemple 7 *Raffinement par introduction d'un champ intermédiaire*

Considérons les deux programmes suivants, `P2` et `P4`.

$$\begin{array}{l} \texttt{P2 : X} \mapsto \texttt{Y} \\ \left\{ \begin{array}{l} \texttt{X} = \texttt{carre::X} \\ \texttt{Y} = \texttt{X} \end{array} \right. \\ \texttt{carre} = \lambda\texttt{(i,j)|(0}\leq\texttt{i,j<n).(i,j)} \end{array} \qquad \begin{array}{l} \texttt{P4 : X} \mapsto \texttt{Y} \\ \left\{ \begin{array}{l} \texttt{X} = \texttt{carre::X} \\ \texttt{Z} = \texttt{X} \\ \texttt{Y} = \texttt{Z} \end{array} \right. \\ \texttt{carre} = \lambda\texttt{(i,j)|(0}\leq\texttt{i,j<n).(i,j)} \end{array}$$

Nous sommes ici dans le cas particulier où P2' est le même énoncé que P2. P2 ne diffère de P4 que par la présence du champ de données intermédiaire Z, et leurs propriétés respectives sont :

$$\begin{array}{l} sat(\texttt{P2}) : (\texttt{X} = \texttt{carre::X}) \Rightarrow (\texttt{Y} = \texttt{X}) \\ sat(\texttt{P4}) : (\texttt{X} = \texttt{carre::X}) \Rightarrow (\texttt{Z} = \texttt{X} \wedge \texttt{Y} = \texttt{Z}) \end{array}$$

Nous vérifions immédiatement $sat(\texttt{P4}) \Rightarrow sat(\texttt{P2})$ car la relation d'égalité sur les champs de données est naturellement transitive, et $\texttt{P2} \sqsubseteq \texttt{P4}$.

De plus, de la définition 9, puisqu'il existe un champ de données Z tel que $(\texttt{Y} = \texttt{X}) \Rightarrow (\texttt{Z} = \texttt{X} \wedge \texttt{Y} = \texttt{Z})$ est vrai, nous avons aussi la relation $\texttt{P4} \sqsubseteq \texttt{P2}$, ce qui signifie que ces deux énoncés sont équivalents.

Par conséquent, l'ajout ou la suppression d'un champ intermédiare donne toujours un nouvel énoncé équivalent.
$\Diamond$

Des procédés classiques de raffinement consistent à «affaiblir» une pré-équation, ou «renforcer» une post-équation. En PEI, ces procédés sont également applicables, à travers la propriété suivante :

Propriété 6 Soient $\texttt{P}\{S_1, S_2\}$ et $\texttt{P'}\{S'_1, S'_2\}$ deux énoncés, où S_1 et S'_1 sont des ensembles de pré-equations, et S_2, S'_2 des ensembles de post-equations. Nous avons :

$$\begin{array}{llll} \texttt{P} \sqsubseteq \texttt{P'} & \text{si} & S_2 = S'_2 & \text{et } (S_1 \Rightarrow S'_1) \text{ est vérifiée} \\ & \text{ou si} & S_1 = S'_1 & \text{et } (S'_2 \Rightarrow S_2) \text{ est vérifiée} \end{array}$$

Preuve. Supposons $S_1 \Rightarrow S'_1$, alors $(S'_1 \Rightarrow S_2) \Rightarrow (S_1 \Rightarrow S_2)$. De même, supposons $S'_2 \Rightarrow S_2$, alors $(S_1 \Rightarrow S'_2) \Rightarrow (S_1 \Rightarrow S_2)$.
$\Box$

Exemple 8 *Affaiblissement des pré-équations*

$\texttt{P4} : \texttt{X} \mapsto \texttt{Y}$
$$\left\{ \begin{array}{l} \texttt{X} = \texttt{vecteur}::\texttt{X} \\ \texttt{Y} = \texttt{X} \end{array} \right.$$
spécifié par $sat(\texttt{P4})$:
$(\texttt{X} = \texttt{vecteur}::\texttt{X}) \Rightarrow (\texttt{Y}{=}\texttt{X})$

$\texttt{P5} : \texttt{X} \mapsto \texttt{Y}$
$$\left\{ \texttt{Y} = \texttt{X} \right.$$
spécifié par $sat(\texttt{P5})$:
vrai $\Rightarrow$ $(\texttt{Y}{=}\texttt{X})$

La propriété précédente affirme que $\texttt{P4} \sqsubseteq \texttt{P5}$, ce qui peut également être établi par $sat(\texttt{P5}) \Rightarrow sat(\texttt{P4})$: en effet, $sat(\texttt{P5})$ se simplifie en $(\texttt{Y} = \texttt{X})$ et la propriété $(\texttt{Y} = \texttt{X}) \Rightarrow ((\texttt{X} = \texttt{vecteur}::\texttt{X}) \Rightarrow (\texttt{Y} = \texttt{X}))$ est vérifiée.
$\Diamond$

Exemple 9 *Renforcement des post-équations*

$\texttt{P6} : \texttt{X} \mapsto \texttt{Y}$
$$\left\{ \begin{array}{l} \texttt{vecteur}::\texttt{Z} = \texttt{X} \\ \texttt{Y} = \texttt{vecteur}::\texttt{Z} \end{array} \right.$$
spécifié par $sat(\texttt{P6})$: vrai $\Rightarrow$ $(\texttt{vecteur}::\texttt{Z} = \texttt{X} \wedge \texttt{Y} = \texttt{vecteur}::\texttt{Z})$

Nous avons $\texttt{P5} \sqsubseteq \texttt{P6}$ car $sat(\texttt{P6}) \Rightarrow sat(\texttt{P5})$ s'écrit $(\texttt{vecteur}::\texttt{Z} = \texttt{X} \wedge \texttt{Y} = \texttt{vecteur}::\texttt{Z}) \Rightarrow (\texttt{Y} = \texttt{X})$, qui est vérifiée.
$\Diamond$

3.3 Calcul de raffinement

De manière plus opérationnelle, nous définissons maintenant un *calcul de raffinement* qui permet de construire un programme raffinant un programme donné, sans utiliser l'implication modulo l'équivalence. Ce calcul stipule que le raffinement d'un énoncé peut se faire par la substitution d'une équation de cet énoncé par une autre. Nous notons par la suite $\texttt{P}\{S[E\backslash E']\}$ l'énoncé $\texttt{P}$ dans lequel l'équation E a été substituée par l'équations E'. Nous précisons maintenant les moyens d'obtenir, à partir d'une équation E, un équation E' telle que $\texttt{P}\{S[E]\} \sqsubseteq \texttt{P}\{S[E\backslash E']\}$. Nous introduisons la notation $E \longrightarrow E'$, qui signifie que l'on substitue E par E' en utilisant un des deux procédés suivant :

1. L'application d'un changement de base aux deux membres d'une équation, conduit aux deux règles de substitution suivantes :

(I)	$\mathtt{Y} = \mathtt{X}$	$\longrightarrow$	$\mathtt{h}::\mathtt{Y} = \mathtt{h}::\mathtt{X}$ et $dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h}) \wedge dom(\sigma_{\mathtt{Y}}) \subseteq dom(\mathtt{h})$
et réciproquement,			
(II)	$\mathtt{h}::\mathtt{Y} = \mathtt{h}::\mathtt{X}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{X}$ et $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})$

2. L'application d'une des règles élémentaires dont nous donnons la liste dans les trois tables qui suivent. Quand des conditions apparaissent dans la partie droite de la règle, on doit s'assurer, pour pouvoir appliquer la règle, que le programme obtenu après substitution, requiert également ces conditions (elles doivent donc pouvoir se déduire des autres équations du programme obtenu). Notons que si on peut appliquer la règle, le programme obtenu est équivalent puisque la règle réciproque ne suppose aucune condition.

A titre d'exemple, examinons les règles (5a) et (5b). Dans la règle (5a) la partie droite de la règle n'est une expression correcte que si $(\mathtt{X} \lhd \mathtt{f1})$ est un champ de données, ce qui n'est pas requis par la partie gauche de la règle. Considérons par exemple les fonctions $\mathtt{f1}{=}\lambda\mathtt{(i)|(2}{\leq}\mathtt{i}{\leq}\mathtt{n).(i-1)}$ et $\mathtt{f2}{=}\lambda\mathtt{(i)|(1}{\leq}\mathtt{i}{\leq}\mathtt{n-1).(i+1)}$. De la définition de l'opération géométrique, la partie gauche de la règle est correcte si $dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\sigma)$. Dans la partie droite, l'expression doit vérifier à la fois $dom(\mathtt{f1}) \subseteq dom(\sigma)$ et $dom(\mathtt{f2}) \subseteq dom(\sigma)$, qui se simplife en $dom(\mathtt{f1}) \cup dom(\mathtt{f2}) \subseteq dom(\sigma)$. En l'occurrence, ceci signifie que le champ de la partie droite doit vérifier $[1 \ldots n] \subseteq dom(\sigma)$ alors que la partie gauche ne nécessite que $[1 \ldots n{-}1] \subseteq dom(\sigma)$. Le fait que le champ soit correctement défini par l'expression en partie droite, implique qu'il est bien défini par l'expression en partie gauche.
Si en plus, on ajoute une contrainte imposant $dom(\sigma) = [1 \ldots n]$, par exemple en ajoutant l'équation $\mathtt{X} = \lambda\mathtt{(i)|(1}{\leq}\mathtt{i}{\leq}\mathtt{n)}::\mathtt{X}$, alors on déduit de cette information contextuelle qu'on peut également appliquer la règle dans le sens opposé, comme l'indique la règle (5b).

Le lien entre la réécriture d'une équation d'un énoncé et le raffinement est établi à l'aide de la propriété suivante :

Propriété 7 Soient deux équations E et E' et un énoncé $\mathtt{P}\{S\}$:

Si E est une pré-équation et $E \longrightarrow E'$ alors $\mathtt{P}\{S[E \backslash E']\} \sqsubseteq \mathtt{P}$

Si E est une post-équation et $E \longrightarrow E'$ alors $\mathtt{P} \sqsubseteq \mathtt{P}\{S[E \backslash E']\}$

Preuve. Examinons les deux procédés permettant d'effectuer une substitution d'équation :

1. **Règle (I)** Il faut d'abord établir la preuve pour le premier cas de substitution. Soient p les conditions induites par la partie gauche de la règle (I), et p' les

	(fonctionnelle-base)		
(1a)	$\mathtt{Y} = \mathtt{f1} \rhd (\mathtt{f2}::\mathtt{X})$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f2}::(\mathtt{f1} \rhd \mathtt{X})$
(1b)	$\mathtt{Y} = \mathtt{f2}::(\mathtt{f1} \rhd \mathtt{X})$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f1} \rhd (\mathtt{f2}::\mathtt{X})$
	(géométrique-fonctionnelle)		
(2a)	$\mathtt{Y} = (\mathtt{f1} \rhd \mathtt{X}) \lhd \mathtt{f2}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f1} \rhd (\mathtt{X} \lhd \mathtt{f2})$
(2b)	$\mathtt{Y} = \mathtt{f1} \rhd (\mathtt{X} \lhd \mathtt{f2})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{f1} \rhd \mathtt{X}) \lhd \mathtt{f2}$
	(base-géométrique)		
(3a)	$\mathtt{Y} = (\mathtt{f2}::\mathtt{X}) \lhd \mathtt{f1}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f2}::(\mathtt{X} \lhd \mathtt{f2}^{-1} \circ \mathtt{f1} \circ \mathtt{f2})$ et $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq img(\mathtt{f2}) \wedge dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f2})$
(3b)	$\mathtt{Y} = \mathtt{f2}::(\mathtt{X} \lhd \mathtt{f1})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{f2}::\mathtt{X}) \lhd \mathtt{f2} \circ \mathtt{f1} \circ \mathtt{f2}^{-1}$ et $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq dom(\mathtt{f2})$
	(réduction-base)		
(3c)	$\mathtt{Y} = \mathtt{f1} \,;\!\rhd (\mathtt{f2}::\mathtt{X})$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f2}::(\mathtt{f2}^{-1} \circ \mathtt{f1} \circ \mathtt{f2} \,;\!\rhd \mathtt{X})$ et $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq img(\mathtt{f2}) \wedge dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f2})$
(3d)	$\mathtt{Y} = \mathtt{f2}::(\mathtt{f1} \,;\!\rhd \mathtt{X})$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f2} \circ \mathtt{f1} \circ \mathtt{f2}^{-1} \,;\!\rhd (\mathtt{f2}::\mathtt{X})$ et $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq dom(\mathtt{f2})$

TAB. 3.1 - *Règles impliquant des opérations de type différent*

conditions induites par la partie droite de cette même règle :

$$
\begin{array}{lcl}
p & = & (v_{\mathtt{Y}} = v_{\mathtt{X}} \wedge \sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}}) \\
p' & = & (v_{\mathtt{Y}} \circ \mathtt{h}^{-1} = v_{\mathtt{X}} \circ \mathtt{h}^{-1} \wedge \sigma_{\mathtt{Y}} \circ \mathtt{h}^{-1} = \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1} \\
 & & \wedge\, dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h}) \wedge dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h}) \\
 & & \wedge\, dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h}) \wedge dom(\sigma_{\mathtt{Y}}) \subseteq dom(\mathtt{h}))
\end{array}
$$

Considérons d'abord que $\mathtt{X} = \mathtt{Y}$ est une pré-équation dans un énoncé `P`. Alors, la propriété de `P`, et celle de l'énoncé `P'` obtenue par application de la règle (I) ont la forme suivante :

$$
\begin{array}{lcl}
sat(\mathtt{P}) & : & (p_1 \wedge p) \Rightarrow p_2 \\
sat(\mathtt{P}') & : & (p_1 \wedge p') \Rightarrow p_2
\end{array}
$$

	(composition fonctionnelle)		
(4a)	$\mathtt{Y} = (\mathtt{f1} \circ \mathtt{f2}) \rhd \mathtt{X}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f1} \rhd (\mathtt{f2} \rhd \mathtt{X})$
(4b)	$\mathtt{Y} = \mathtt{f1} \rhd (\mathtt{f2} \rhd \mathtt{X})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{f1} \circ \mathtt{f2}) \rhd \mathtt{X}$
	(composition géométrique)		
(5a)	$\mathtt{Y} = \mathtt{X} \lhd (\mathtt{f1} \circ \mathtt{f2})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{X} \lhd \mathtt{f1}) \lhd \mathtt{f2}$
(5b)	$\mathtt{Y} = (\mathtt{X} \lhd \mathtt{f1}) \lhd \mathtt{f2}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{X} \lhd (\mathtt{f1} \circ \mathtt{f2})$
			et $img(\mathtt{f2}) \subseteq dom(\mathtt{f1}) \subseteq dom(\sigma_{\mathtt{X}})$
	(composition réduction)		
(6)	$\mathtt{Y} = (\mathtt{f1} \circ \mathtt{f2}) ;\rhd \mathtt{X}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f1} ;\rhd (\mathtt{f2} ;\rhd \mathtt{X})$
	(composition base)		
(7a)	$\mathtt{Y} = (\mathtt{f1} \circ \mathtt{f2}) :: \mathtt{X}$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f1} :: (\mathtt{f2} :: \mathtt{X})$
(7b)	$\mathtt{Y} = \mathtt{f1} :: (\mathtt{f2} :: \mathtt{X})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{f1} \circ \mathtt{f2}) :: \mathtt{X}$
			et $\mathtt{f1}$ et $\mathtt{f2}$ bijectives

TAB. 3.2 - *Règles concernant la composition des fonctions*

où p_1 et p_2 traduisent respectivement les pré- et post-équations de P. D'après la propriété 6, nous devons montrer que $(p_1 \wedge p') \Rightarrow (p_1 \wedge p)$ est toujours vraie pour conclure que P' $\sqsubseteq$ P. Il suffit donc de montrer que $p' \Rightarrow p$.
En effet, de $v_{\mathtt{Y}} \circ \mathtt{h}^{-1} = v_{\mathtt{X}} \circ \mathtt{h}^{-1}$ et $\sigma_{\mathtt{Y}} \circ \mathtt{h}^{-1} = \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}$, nous déduisons respectivement $v_{\mathtt{X}}|_{dom(h)} = v_{\mathtt{Y}}|_{dom(h)}$ et $\sigma_{\mathtt{X}}|_{dom(h)} = \sigma_{\mathtt{Y}}|_{dom(h)}$, et comme $dom(v_{\mathtt{X}})$, $dom(v_{\mathtt{Y}})$, $dom(\sigma_{\mathtt{X}})$ et $dom(\sigma_{\mathtt{Y}})$ sont inclus dans $dom(\mathtt{h})$, il vient $v_{\mathtt{X}} = v_{\mathtt{Y}}$ et $\sigma_{\mathtt{X}} = \sigma_{\mathtt{Y}}$. Donc $sat(\mathtt{P}') \Rightarrow sat(\mathtt{P})$ et P' $\sqsubseteq$ P.

Dans le cas où les équations de la règle (I) sont des post-équations, alors les propriétés de P et P' ont la forme suivante :

$$
\begin{array}{lcl}
sat(\mathtt{P}) & : & p_1 \Rightarrow (p_2 \wedge p) \\
sat(\mathtt{P}') & : & p_1 \Rightarrow (p_2 \wedge p')
\end{array}
$$

Nous avons montré que $p' \Rightarrow p$ et donc ici, $sat(\mathtt{P}') \Rightarrow sat(\mathtt{P})$ d'après la propriété 6. Par conséquent, P $\sqsubseteq$ P'.

Règle (II) Dans ce qui suit, nous ne n'établissons la preuve que pour le cas, où les équations sont des post-équations. La démonstration pour une pré-équation est la même que précédemment.

	(distributivité géométrique)		
(8a)	$\mathtt{Y} = (\mathtt{X1}\ \mathtt{/\&/}\ \mathtt{X2}) \lhd \mathtt{f}$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{X1} \lhd \mathtt{f})\ \mathtt{/\&/}\ (\mathtt{X2} \lhd \mathtt{f})$
(8b)	$\mathtt{Y} = (\mathtt{X1} \lhd \mathtt{f})\ \mathtt{/\&/}\ (\mathtt{X2} \lhd \mathtt{f})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{X1}\ \mathtt{/\&/}\ \mathtt{X2}) \lhd \mathtt{f}$ et $dom(\mathtt{f}) \subseteq dom(\sigma_{\mathtt{X1}}) \cap dom(\sigma_{\mathtt{X2}}) \wedge img(\mathtt{f}) \subseteq dom(v_{\mathtt{X1}}) \cap dom(v_{\mathtt{X2}})$
	(partition géométrique)		
(9a)	$\mathtt{Y} = \mathtt{X} \lhd (\mathtt{f1} \# \mathtt{f2})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{X} \lhd \mathtt{f1})\ \mathtt{/\&/}\ (\mathtt{X} \lhd \mathtt{f2})$
(9b)	$\mathtt{Y} = (\mathtt{X} \lhd \mathtt{f1})\ \mathtt{/\&/}\ (\mathtt{X} \lhd \mathtt{f2})$	$\longrightarrow$	$\mathtt{Y} = \mathtt{X} \lhd (\mathtt{f1} \# \mathtt{f2})$ et $dom(\mathtt{f1}) \cap dom(\mathtt{f2}) = \{\}$
	(partition réduction)		
(10)	$\mathtt{Y} = (\mathtt{f1} \# \mathtt{f2})\ ;\!\rhd \mathtt{X}$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{f1}\ ;\!\rhd \mathtt{X})\ \mathtt{/\&/}\ (\mathtt{f2}\ ;\!\rhd \mathtt{X})$
	(distributivité base)		
(11a)	$\mathtt{Y} = (\mathtt{f} :: \mathtt{X1})\ \mathtt{/\&/}\ (\mathtt{f} :: \mathtt{X2})$	$\longrightarrow$	$\mathtt{Y} = \mathtt{f} :: (\mathtt{X1}\ \mathtt{/\&/}\ \mathtt{X2})$
(11b)	$\mathtt{Y} = \mathtt{f} :: (\mathtt{X1}\ \mathtt{/\&/}\ \mathtt{X2})$	$\longrightarrow$	$\mathtt{Y} = (\mathtt{f} :: \mathtt{X1})\ \mathtt{/\&/}\ (\mathtt{f} :: \mathtt{X2})$

TAB. 3.3 - *Règles concernant l'opérateur* `/&/`

Soient les énoncés, `P` ayant $\mathtt{h} :: \mathtt{X} = \mathtt{h} :: \mathtt{Y}$ parmi ses post-équations, et `P'` obtenu par la règle (II). Leurs propriétés sont :

$$
\begin{aligned}
sat(\mathtt{P}) \quad &: \quad p_1 \Rightarrow (p_2 \wedge v_{\mathtt{Y}} \circ \mathtt{h}^{-1} = v_{\mathtt{X}} \circ \mathtt{h}^{-1} \wedge \sigma_{\mathtt{Y}} \circ \mathtt{h}^{-1} = \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1} \\
&\qquad \wedge dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h}) \wedge dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})) \\
sat(\mathtt{P}') \quad &: \quad p_1 \Rightarrow (p_2 \wedge v_{\mathtt{Y}} = v_{\mathtt{X}} \wedge \sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}} \\
&\qquad \wedge dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h}))
\end{aligned}
$$

Nous avons ici $sat(\mathtt{P}') \Rightarrow sat(\mathtt{P})$ car $(v_{\mathtt{Y}} = v_{\mathtt{X}} \wedge \sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}}) \Rightarrow (v_{\mathtt{Y}} \circ \mathtt{h}^{-1} = v_{\mathtt{X}} \circ \mathtt{h}^{-1} \wedge \sigma_{\mathtt{Y}} \circ \mathtt{h}^{-1} = \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1})$.

2. **Règles élémentaires** : Il faut ensuite établir la preuve pour chacune des règles listées dans les tables. Nous n'en montrons ici que quelques-unes.

   – Par exemple, pour la règle (5a) nous devons montrer que l'énoncé `P` ayant parmi ses post-équations $\mathtt{Y} = \mathtt{X} \lhd (\mathtt{f1} \circ \mathtt{f2})$ est raffiné par le programme `P'` ayant, à la place de cette équation, l'équation $\mathtt{Y} = (\mathtt{X} \lhd \mathtt{f1}) \lhd \mathtt{f2}$:

$sat(\mathtt{P})$:

$$
\begin{aligned}
p_1 \Rightarrow \quad & (p_2 \wedge (v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{f1} \circ \mathtt{f2}) \wedge (\sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}}) \wedge \\
& (dom(v_{\mathtt{X}}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge dom(v_{\mathtt{Y}}) \subseteq dom(\sigma_{\mathtt{Y}}) \wedge \\
& dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge img(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(v_{\mathtt{X}}))
\end{aligned}
$$

$sat(\mathtt{P}')$:

$$p_1 \Rightarrow \quad (p_2 \wedge (v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{f1} \circ \mathtt{f2}) \wedge (\sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}}) \wedge$$
$$(dom(v_{\mathtt{X}}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge dom(v_{\mathtt{Y}}) \subseteq dom(\sigma_{\mathtt{Y}}) \wedge$$
$$dom(\mathtt{f1}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge dom(\mathtt{f2}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge$$
$$img(\mathtt{f1}) \subseteq dom(v_{\mathtt{X}}) \wedge img(\mathtt{f2}) \subseteq dom(v_{\mathtt{X}} \circ \mathtt{f1}))$$

Nous devons donc montrer que :

$$(2) \begin{cases} img(\mathtt{f1}) \subseteq dom(v_{\mathtt{X}}) \\ img(\mathtt{f2}) \subseteq dom(v_{\mathtt{X}} \circ \mathtt{f1}) \end{cases} \Rightarrow \quad (1) \quad \{img(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(v_{\mathtt{X}})$$

De la définition de la composition : $img(\mathtt{f1} \circ \mathtt{f2}) \subseteq img(\mathtt{f1})$

donc (2) se réécrit : $\begin{cases} img(\mathtt{f1} \circ \mathtt{f2}) \subseteq img(\mathtt{f1}) \subseteq dom(v_{\mathtt{X}}) \\ img(\mathtt{f2}) \subseteq dom(v_{\mathtt{X}} \circ \mathtt{f1}) \end{cases}$

et on a (2) $\Rightarrow$ (1), et donc $sat(\mathtt{P'}) \Rightarrow sat(\mathtt{P})$.

– Pour la règle (7a), nous montrons la relation de raffinement entre l'énoncé `P`, ayant parmi ses post-équations $\mathtt{Y} = \mathtt{(f1} \circ \mathtt{f2)} :: \mathtt{X}$, et spécifié par $sat(\mathtt{P})$:

$$p_1 \Rightarrow \quad (p_2 \wedge (v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \wedge (\sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}} \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \wedge$$
$$dom(v_{\mathtt{X}}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge dom(v_{\mathtt{Y}}) \subseteq dom(\sigma_{\mathtt{Y}}) \wedge$$
$$dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f1} \circ \mathtt{f2}))$$

et l'énoncé `P'`, ayant parmi ses post-équations $\mathtt{Y} = \mathtt{f1} :: \mathtt{(f2} :: \mathtt{X)}$, spécifié par $sat(\mathtt{P}')$:

$$p_1 \Rightarrow \quad (p_2 \wedge (v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \wedge (\sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}} \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \wedge$$
$$dom(v_{\mathtt{X}}) \subseteq dom(\sigma_{\mathtt{X}}) \wedge dom(v_{\mathtt{Y}}) \subseteq dom(\sigma_{\mathtt{Y}}) \wedge$$
$$dom(v_{\mathtt{X}} \circ \mathtt{f2}^{-1}) \subseteq dom(\mathtt{f1}) \wedge dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f2}))$$

Il reste à montrer que :

$$(2) \begin{cases} dom(v_{\mathtt{X}} \circ \mathtt{f2}^{-1}) \subseteq dom(\mathtt{f1}) \\ dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f2}) \end{cases} \Rightarrow \quad (1) \quad \{dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f1} \circ \mathtt{f2})$$

Pour cela, nous utilisons la propriété suivante : si g est une fonction et f est une bijection, alors $dom(g \circ f^{-1}) = f(dom(g))$.

Comme $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f2})$ nous avons :

$$\begin{array}{ll} & dom(v_{\mathtt{X}} \circ \mathtt{f2}^{-1}) \subseteq dom(\mathtt{f1}) \\ \Rightarrow & \mathtt{f2}(dom(v_{\mathtt{x}})) \subseteq dom(\mathtt{f1}) \\ \Rightarrow & \mathtt{f2}^{-1}(\mathtt{f2}(dom(v_{\mathtt{x}}))) \subseteq \mathtt{f2}^{-1}(dom(\mathtt{f1})) \\ \Leftrightarrow & dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{f1} \circ \mathtt{f2}) \end{array}$$

Nous avons (2) $\Rightarrow$ (1), et donc $sat(\mathtt{P'}) \Rightarrow sat(\mathtt{P})$.

□

3.4 Un outil visuel pour la conception de programmes : VPEI

Il ressort de la présentation précédente que le placement géométrique des données est l'élément déterminant de la conception d'un programme. La définition de l'opération fonctionnelle impose en effet que pour calculer des valeurs, ces valeurs doivent se trouver en un même point du domaine discret. Ce choix doit avoir une influence favorable sur la performance des programmes, car il oblige le programmeur à expliciter les déplacements (les communications dans une machine parallèle) nécessaires avant de pouvoir effectuer un calcul. Ce choix est similaire à celui fait dans la plupart des langages data-parallèles, comme C*[54], CM-Fortan[55], MPL[38], Hyper-C[45], *etc*, qui imposent la localité des données aux processeurs virtuels qui les modifient. En revanche, cette démarche demande un effort supplémentaire au programmeur qui doit expliciter le placement. Pour cela, il nous a semblé utile de proposer un outil fournissant une aide graphique aux manipulations géométriques que doit effectuer l'utilisateur.

3.4.1 Présentation de l'outil

Un prototype d'outil nommé VPEI[1], a été conçu pour permettre la conception de programmes PEI par simple manipulations graphiques. L'utilisateur peut créer des champs de données dans $\mathbb{Z}$, $\mathbb{Z}^2$ ou $\mathbb{Z}^3$, et leur appliquer ensuite une opération fonctionnelle, une opération de routage ou de changement de base. Pour appliquer une opération, l'utilisateur doit entrer au clavier la fonction et, selon l'opération choisie, le logiciel permet de visualiser schématiquement les effets de cette opération en montrant l'aspect du champ de données résultant. Quand le champ obtenu correspond aux attentes, il peut être validé et devient un champ du programme sur lequel on peut appliquer de nouvelles opérations. Après validation, le texte de l'équation définissant ce nouveau champ est affiché dans une fenêtre texte. L'outil autorise également certaines définitions récursives de champs de données : après avoir défini un pas de la récurrence, pouvant être constitué de plusieurs opérations, il peut réitérer l'application de cette suite d'opérations et visualiser le champ obtenu à chaque étape.

La figure 3.2 montre le déroulement de la conception d'un programme de produit de matrices. Après avoir créé les matrices initiales, l'utilisateur applique les opérations pour placer les valeurs de ces matrices sur la face du dessus d'un cube (`A`), et la face latérale d'un cube (`B`). Les valeurs sont ensuite dupliquées à travers chacun des cubes dans la direction $(0, 1, 0)$ pour `A` et $(1, 0, 0)$ pour `B`. Les deux champs ainsi obtenus (`A'` et `B'`) sont superposés pour donner un champ `A_B`. Ce champ contient des séquences à deux éléments

1. Abréviation de visual PEI

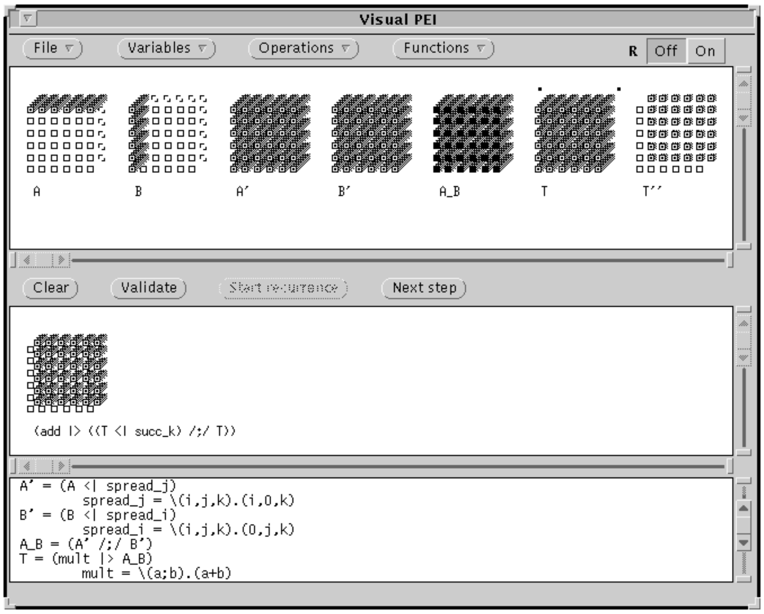

FIG. 3.2 - *Construction d'un programme avec* VPEI

symbolisés par des points «noircis». L'application du produit à tous les couples donne le champ `T`. On utilise ensuite une récurrence pour faire la somme de tous les produits. Un pas de récurrence est décrit comme la somme du champ `T` avec lui-même décalé d'une position dans la direction $(0, 0, 1)$. Cette étape est représentée dans la fenêtre du milieu dans la figure 3.2. Ensuite, l'utilisateur peut répéter l'étape de récurrence précédemment définie en appuyant sur un bouton, et voir à chaque étape l'aspect des champs.

Une fois l'ensemble des champs créés, les équations entre les champs ainsi que les définitions des fonctions utilisées peuvent être simplement rappelées pour obtenir le texte de programme PEI. Notons également qu'un champ intermédiaire inutile peut être supprimé après avoir substitué, dans les définitions des champs l'utilisant, son nom par sa définition.

3.4.2 Intérêts-Limitations

VPEI permet la construction pas-à-pas du programme avec un contrôle visuel sur l'aspect de chaque nouveau champ créé. A chaque fois qu'un nouveau champ est défini, le placement des valeurs est calculé sur un petit nombre de points à l'aide de la librairie

Polylib [63] et du logiciel MAPLE[11] pour les compositions et inversions de fonctions. Ce dessin d'un nouveau champ, même s'il est partiel, permet généralement de détecter immédiatement si l'opération effectuée donne le résultat prévu. Une limitation par rapport à l'expressivité du langage PEI réside dans l'obligation de construire des définitions explicites, c'est-à-dire assimilables à des affectations. On ne peut en effet appliquer une opération que sur un champ connu (précédemment défini) alors que PEI autorise des définitions implicites. Ce problème se révèle aussi gênant dans les définitions récursives.

VPEI ne permet de concevoir que de petits programmes PEI, le nombre de champs intermédiaires venant très vite alourdir les manipulations, et ce malgré la fonction de substitution décrite précédemment.

Finalement, l'expérience de ce premier prototype (environ 6000 lignes de langage C) nous fait penser que l'ajout d'une fenêtre interprète d'expressions PEI remédierait aux limitations évoquées. Cet interprète devrait être capable de représenter graphiquement une équation entrée au clavier.

3.5 Un outil pour le raffinement de programmes

L'outil PEI initialement développé par E. Violard dans le cadre de sa thèse est un outil de manipulation de programmes. L'ensemble des règles de raffinement et d'abstraction présentées en section 3.3 y sont implantées et on peut transformer un programme dans cet environnement par application des règles sur chacune des équations. Les seules modifications qu'on peut apporter au programme se font par ce biais. L'utilisation de cet outil est logiquement l'étape suivante d'écriture d'un programme, qui a pu être conçu initialement dans l'environnement VPEI par exemple.

L'environnement PEI apporte une aide à l'utilisateur pour la transformation des programmes, mais ne permet pas, pour l'instant de transformations complètement automatisées. Nous espérons pouvoir implanter ce type de transformation de programmes sur la base des travaux réalisés concernant les stratégies de transformation, présentés dans le chapitre 4.

En l'état actuel, la transformation d'un programme se fait pas-à-pas : l'utilisateur sélectionne une expression de champs de données quelconque dans le système d'équations, et peut choisir, par menu, le raffinement ou l'abstraction de cette expression. Le logiciel, s'il peut identifier cette expression à un membre de l'une des règles, remplace dans le programme l'expression initiale par l'expression raffinée ou abstraite. L'utilisateur peut également réécrire une expression de champs de données (par exemple pour arriver à une forme d'expression qu'il est possible de raffiner où d'abstraire) ; le système vérifie alors que la nouvelle expression est équivalente à la précédente.

Pour plus de détails sur ce logiciel, on pourra se référer à [56], dont les annexes décrivent de manière exhaustive les fonctionnalités de base du logiciel. Le mode de fonctionnement reste identique à la version développée initialement, bien que de nombreuses modifications aient été apportées. Notamment, un traducteur permettant d'associer à un programme PEI, une interprétation fonctionnelle (*cf.* section 2.3.5) sous la forme d'un programme CAML, à été ajouté à l'environnement. Le programme CAML généré peut être utilisé pour vérifier l'exactitude des résultats obtenus sur un jeu d'essai. Egalement, un contrôleur de type à été développé pour les programmes PEI : cet outil, utilisant le calculateur OMEGA[30], type chacune des expressions de champs de données d'un énoncé PEI, c'est-à-dire calcule le domaine des valeurs et le domaine de référence de chacune de ces expressions. Le contrôle porte alors sur la cohérence des expressions vis-à-vis de la définition des opérations. Nous verrons dans le chapitre 5, que les résultats produits par ce contrôleur de type peuvent être utilisés par d'autres outils.

Chapitre 4

Stratégies de transformations

Il a été montré dans [56, 46] que PEI permettait de transformer un programme en utilisant les résultats issus des techniques de parallélisation automatique : l'analyse des dépendances permet de déterminer un changement de base composé d'une fonction d'ordonnancement et de placement des calculs. Ce changement de base est appliqué aux deux membres de chacune des équations de l'énoncé PEI et on montre que l'application des règles qui ont pour effet d'instiller le changement de base au coeur des expressions, est confluente. Cette technique de transformation est maintenant classique et est disponible dans de nombreux formalismes, notamment ALPHA. Nous rappelons brièvement dans la première section ce type de transformation sur l'exemple de l'élimination de Gauss.

Dans la suite du chapitre, nous nous intéressons à d'autres applications du raffinement de programmes évoqué dans le chapitre précédent. Notamment, nous verrons que le raffinement permet de prendre en compte des relations fines entre programmes, que seule l'équivalence faible permet de décrire. Les stratégies que nous proposons ici sont constituées de l'application successive de plusieurs règles de raffinement, et constituent des schémas types de transformation d'une forme de programme vers une autre. Selon l'architecture de la machine parallèle, les primitives implantées sur le matériel ou dans le langage visé, certaines solutions sont connues pour être plus efficaces que d'autres. Nous montrons ici, par le biais de ces stratégies de transformation, comment passer d'une solution à une autre. Tout au long de l'exposé des stratégies, nous illustrerons les techniques décrites avec l'exemple du produit matrice-vecteur. Cet exposé des stratégies de transformation précise les idées présentées dans [22].

4.1 Transformation par changement de base

Le programme PEI suivant traduit l'algorithme d'élimination de Gauss. L'algorithme résout le système linéaire $Ax = b$ où A est une matrice de dimension $n \times n$ et b un vecteur de taille n. L'algorithme procède par une triangularisation de la matrice A pour ensuite effectuer une substitution par remontée et résoudre le système. Nous ne nous intéressons ici qu'à la phase de triangularisation. Le programme PEI suivant prend en entrée les champs `A` et `B` représentant respectivement la matrice et le vecteur. Nous plongeons ensuite ces champs dans un espace à trois dimensions (i, j, k) à travers les champs `X` et `Y`. Le calcul est alors spécifié par une définition récursive du champ `C`. La matrice triangularisée ainsi que le vecteur modifié se trouvent sur le plan $i = k$ et sont représentés par les champs `XN` et `YN`.

Si la version de ce programme doit être interprétée opérationnellement, en choisissant l'axe k comme direction d'ordonnancement, nous constatons que le programme utilise alors des opérations géométriques de type diffusion, comme par exemple la fonction `pivot`. Or, certaines machines n'autorisent pas ce type de communication et ne permettent que des communications d'un processeur vers un de ses voisins. Ce sont par exemple les hypothèses prises pour les réseaux systoliques. Cette contrainte de communication influe sur l'ordonnancement du programme, et nous cherchons une nouvelle écriture du programme pour en tenir compte. Le nouveau programme est obtenu par l'application d'un changement de base permettant d'obtenir une version systolique du programme `Gauss1`.

```
Gauss1 : (A,B) ↦ (XN,YN)
⎧ A = alignx :: X
⎪ B = aligny :: Y
⎨ C = el ▷ (X /&/ Y /&/ (C ◁ pre) /&/ (C ◁ cp) /&/ (C ◁ lp) /&/ (C ◁ piv))
⎪ XN = C ◁ last1
⎩ YN = C ◁ last2

alignx  =  λ(i,j,k)|(1≤i≤n & 1≤j≤n & k=0).(i,j)
aligny  =  λ(i,j,k)|(1≤i≤n & j=n+1 & k=0).(i)
pre     =  λ(i,j,k)|(k<i≤n & 1≤j≤n+1 & 0<k).(i,j,k-1)
piv     =  λ(i,j,k)|(k<i≤n & 1≤j≤n+1 & 0<k).(k,k,k-1)
cp      =  λ(i,j,k)|(k<i≤n & 1≤j≤n+1 & 0<k).(i,k,k-1)
lp      =  λ(i,j,k)|(k<i≤n & 1≤j≤n+1 & 0<k).(k,j,k-1)
el      =  id # λ(a;b;c;d).(a-b*c/d)
last1   =  λ(i,j,k)|(1≤i≤n & i≤j≤n & k=i-1).(i,j,k)
last2   =  λ(i,j,k)|(1≤i≤n & j=n+1 & k=i-1).(i,j,k)
```

L'analyse des dépendances selon les techniques systoliques classiques fournit un ordonnancement possible des calculs défini par la fonction $t(i,j,k) = i + j + k$, ainsi qu'un placement défini par $p(i,j,k) = (i,k)$. Nous pouvons expliciter ce choix de cadencement et de placement en transformant le programme précédent par l'application du changement de base `h` $=$ `λ(i,j,k).(i,k,i+j+k)` aux deux membres de certaines des équations de `Gauss1`. Sur ce type d'exemple, il est intéressant d'appliquer ce changement de base à toutes les équations, sauf celles qui explicitent un changement de dimension entres variables, en l'occurrence celles définissant `A` et `B`. Le résultat de la transformation est un programme dont les champs sont exprimés dans une nouvelle base, c'est-à-dire que les expressions fonction d'un champ `X` deviennent des expressions fonction de `h::X`.

Il est intéressant de noter que cette transformation peut être automatisée car le sous-ensemble des règles (1b), (3b), (3d), (7b), (11b) augmenté de la règle `Y` $=$ `X` $\longrightarrow$ `h::X` $=$ `h::X` est confluent, et toute expression fonction de `X` devient une expression de (`h::X`). Ce type de transformation a d'ailleurs été implanté et sera intégré prochainement dans l'environnement de transformation. Un nombre fini d'applications successives de ce sous-ensemble de règles[1] nous amène finalement à l'énoncé suivant, où les champs `h::C`, `h::XN` et `h::YN` ont été renommés `C`, `XN` et `YN` (en se servant de la propriété de raffinement 3 définie à partir de l'équivalence forte des champs de données).

```
Gauss2 : (A,B) ↦ (XN,YN)
⎧ A = alignx'::X
⎪ B = aligny'::Y
⎨ C = el ▷ (X/&/Y/&/(C ◁ pre') /&/(C ◁ cp') /&/(C ◁ lp') /&/(C ◁ piv'))
⎪ XN = C ◁ last1'
⎩ YN = C ◁ last2'

alignx'  =  λ(i,j,k)|(1≤i≤n & 1≤k-i-j≤n & j=1).(i,k-i-j)
aligny'  =  λ(i,j,k)|(1≤i≤n & k-i-j=n+1 & j=1).(i)
pre'     =  λ(i,j,k)|(j<i≤n & 2*j+i<k & k-i-j≤n+1 & 1≤j≤n).(i,j-1,k-1)
piv'     =  λ(i,j,k)|(j<i≤n & 2*j+i<k & k-i-j≤n+1 & 1≤j≤n).(j,j-1,3*j-1)
cp'      =  λ(i,j,k)|(j<i≤n & 2*j+i<k & k-i-j≤n+1 & 1≤j≤n).(i,j-1,i+2*j-1)
lp'      =  λ(i,j,k)|(j<i≤n & 2*j+i<k & k-i-j≤n+1 & 1≤j≤n).(j,j-1,j+k-i-1)
last1'   =  λ(i,j,k)|(1≤i≤n & i≤k-i-j≤n & j=i-1).(i,j,k)
last2'   =  λ(i,j,k)|(1≤i≤n & k-i-j=n+1 & j=i-1).(i,j,k)
```

1. Une exception est faite pour les équations exprimant un changement de dimension, mais ce cas particulier n'empêche pas l'automatisation.

4.2 Stratégie 1 : simplification des communications

La première stratégie de transformation que nous introduisons a pour but de simplifier les opérations de routage d'un programme. Ce type de simplification présente au moins deux intérêts. Il permet, quand la simplification est possible, de diminuer le volume des communications à effectuer. Egalement, cette stratégie de transformation permet de conclure sur l'équivalence de plusieurs programmes différant par leurs communications et nous pensons que cet aspect concerne particulièrement les algorithmes systoliques, c'est-à-dire les algorithmes parallèles qui alternent une phase de calcul avec une phase de communication.

La simplification est possible lorsque les routages sont uniformes avec la forme générale suivante : $f(i_1, \ldots, i_{m-1}, i_m) = (i_1 + a_1, \ldots, i_{m-1} + a_{m-1}, i_m - 1)$ (la variable i_m peut représenter la direction de temps par exemple). Bien que ces conditions semblent contraignantes, de nombreux algorithmes systoliques utilisent de tels routages.

4.2.1 Principe

Le principe de la simplification repose sur l'expression de la fonction utilisée par un routage, sous la forme d'une composition de fonctions. Dans un programme où un champ de données est défini par `Y` $=$ `X` $\lhd$ `f`, nous cherchons à réécrire la fonction `f` sous la forme `f` $=$ `g` $\circ$ `f'`, où `f'` est une fonction plus «simple» que `f`. Ensuite, nous cherchons une définition équivalente à celle de `Y`, utilisant uniquement le routage `f'`. Pour cela, il faut trouver la bijection `h` d'un changement de base, nous permettant d'obtenir une équation équivalente de la forme `Y'` $=$ `X` $\lhd$ `f'`, où `Y'` $=$ `h :: Y`. Il s'agit donc de trouver une représentation différente des champs ne nécessitant plus le routage des valeurs par la fonction `g`.

Nous nous intéressons plus particulièrement aux champs de données définis récursivement à l'aide de ces opérations de routage. Ces champs sont définis par une équation de la forme : `X` $=$ `X0 /&/ (X` $\lhd$ `f)`, et sont l'équivalent des variables uniformisées des langages d'équations récurrentes [48].

4.2.2 Conditions

La réécriture de l'équation exprimant le routage récursif nécessite, afin d'arriver à l'équation voulue, certaines conditions énoncées dans la proposition suivante :

Proposition 4 Soit une bijection `h` et une fonction `f` telle que $dom(\texttt{f}) \cup img(\texttt{f}) \subseteq dom(\texttt{h})$. Alors un énoncé ayant pour post-équation

$$\texttt{X} = \texttt{X0 /\&/ (X} \lhd \texttt{f)} \qquad (E1)$$

est raffiné par l'énoncé où $(E1)$ est substituée par

$$\mathtt{h::X} = \mathtt{h::X0/\&/\,(h::X)} \lhd \mathtt{h} \circ \mathtt{f} \circ \mathtt{h}^{-1} \qquad (E2)$$

si $dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h}) \wedge dom(\sigma_{\mathtt{X0}}) \subseteq dom(\mathtt{h})$.

Preuve. En utilisant les règles de réécriture et les hypothèses de la proposition :

$$\begin{array}{lcl} \mathtt{X} = \mathtt{X0/\&/\,(X} \lhd \mathtt{f)} & \xrightarrow{(I)} & \mathtt{h::X} = \mathtt{h::(X0/\&/\,(X} \lhd \mathtt{f))} \\ & \xrightarrow{(11b)} & \mathtt{h::X} = \mathtt{(h::X0)/\&/\,(h::(X} \lhd \mathtt{f))} \\ & \xrightarrow{(3b)} & \mathtt{h::X} = \mathtt{(h::X0)/\&/\,((h::X)} \lhd \mathtt{h} \circ \mathtt{f} \circ \mathtt{h}^{-1}) \end{array}$$

□

Ainsi toute bijection satisfaisant les conditions précédentes permet de transformer $(E1)$ en $(E2)$; nous donnons maintenant une condition suffisante pour l'existence d'une bijection particulière $\mathtt{h}$ telle que la décomposition $\mathtt{f} = \mathtt{g} \circ \mathtt{f'}$ permette de supprimer le routage $\mathtt{g}$.

Théorème 1 *Soit une équation de la forme $(E1)$ où*

$$\mathtt{f} = \lambda(i_1, \ldots, i_{m-1}, i_m) \mid (0{<}i_m{<}N).(i_1 + a_1, \ldots, i_{m-1} + a_{m-1}, i_m - 1)$$

et soit $\mathtt{g} = \lambda(i_1, \ldots, i_{m-1}, i_m) \mid (0{\leq}i_m{<}N).(i_1 + b_1, \ldots, i_{m-1} + b_{m-1}, i_m)$, *il existe alors une bijection* $\mathtt{h}$, *construite à partir de* $\mathtt{g}$, *telle que* $(E1) \longrightarrow (E2)$ *et* $\mathtt{f} = \mathtt{g} \circ \mathtt{f'}$, *avec* $\mathtt{f'} = \mathtt{h} \circ \mathtt{f} \circ \mathtt{h}^{-1}$.

Ce théorème nous permet donc de remplacer une définition de la forme $\mathtt{X} = \mathtt{X0/\&/\,(X} \lhd \mathtt{f)}$ par une définition de la forme $\mathtt{X'} = \mathtt{X0'/\&/\,(X'} \lhd \mathtt{f')}$, avec $\mathtt{X'}$ fortement équivalent à $\mathtt{X}$.

Preuve. Considérons l'ensemble des bijections h_i $(i \in [0 \ldots N{-}1])$ définies sur des domaines disjoints restreints à chaque valeur entière de i_m. Chaque h_i est obtenu par la composition de $\mathtt{g} = \lambda(i_1, \ldots, i_m) \mid (0{\leq}i_m{<}N).(i_1 + b_1, \ldots, i_m)$ avec elle-même i fois, c'est-à-dire $h_i = \lambda(i_1, \ldots, i_m)|(i_m = i).\mathtt{g}^i(i_1, \ldots, i_m)$. Nous avons donc :

$$\begin{array}{lcl} h_0 & = & \lambda(i_1, \ldots, i_m)|(i_m = 0).(i_1, \ldots, i_{m-1}, 0) \\ & = & \lambda(i_1, \ldots, i_m)|(i_m = 0).(i_1 + i_m b_1, \ldots, i_{m-1} + i_m b_{m-1}, i_m) \\ h_1 & = & \lambda(i_1, \ldots, i_m)|(i_m = 1).(i_1 + b_1, \ldots, i_{m-1} + b_{m-1}, 1) \\ & = & \lambda(i_1, \ldots, i_m)|(i_m = 1).(i_1 + i_m b_1, \ldots, i_{m-1} + i_m b_{m-1}, i_m) \\ h_2 & = & \lambda(i_1, \ldots, i_m)|(i_m = 2).(i_1 + 2b_1, \ldots, i_{m-1} + 2b_{m-1}, 2) \\ & = & \lambda(i_1, \ldots, i_m)|(i_m = 2).(i_1 + i_m b_1, \ldots, i_{m-1} + i_m b_{m-1}, i_m) \\ \vdots & & \end{array}$$

$$\begin{array}{rcl} h_{N-1} & = & \lambda(i_1,\ldots,i_m)|(i_m = N-1).(i_1 + (N-1)b_1,\ldots,i_{m-1} + (N-1)b_{m-1}, N-1) \\ & = & \lambda(i_1,\ldots,i_m)|(i_m = N-1).(i_1 + i_m b_1,\ldots,i_{m-1} + i_m b_{m-1}, i_m) \end{array}$$

Considérons la fonction $\mathtt{h}= h_0 \# h_1 \# \ldots \# h_{N-1}$. Par définition

$$\mathtt{h} = \lambda(i_1,\ldots,i_m) \mid (0{\le}i_m{<}N).(i_1 + i_m b_1,\ldots,i_{m-1} + i_m b_{m-1}, i_m)$$

Cette fonction est bijective, et a pour inverse

$$\mathtt{h}^{-1} = \lambda(i_1,\ldots,i_m) \mid (0{\le}i_m{<}N).(i_1 - i_m b_1,\ldots,i_{m-1} - i_m b_{m-1}, i_m)$$

Soit $\mathtt{f}' = \lambda(i_1,\ldots,i_m) \mid (0{<}i_m{<}N).(i_1+a_1-b_1,\ldots,i_{m-1}+a_{m-1}-b_{m-1}, i_m-1)$ la fonction qui vérifie $\mathtt{f} = \mathtt{g} \circ \mathtt{f'}$. Nous avons alors,

$$\begin{array}{l} \mathtt{h} \circ \mathtt{f} \circ \mathtt{h}^{-1} \\ = \mathtt{h} \circ \lambda(i_1,\ldots,i_{m-1},i_m) \mid (0{<}i_m{<}N).(i_1 - i_m b_1 + a_1,\ldots,i_{m-1} - i_m b_{m-1} + a_{m-1}, i_m - 1) \\ = \lambda(i_1,\ldots,i_{m-1},i_m) \mid (0{<}i_m{<}N). \\ \qquad (i_1 - i_m b_1 + a_1 + (i_m-1)b_1,\ldots,i_{m-1} - i_m b_{m-1} + a_{m-1} + (i_m-1)b_{m-1}, i_m - 1) \\ = \lambda(i_1,\ldots,i_{m-1},i_m) \mid (0{<}i_m{<}N).(i_1 + a_1 - b_1,\ldots,i_{m-1} + a_{m-1} - b_{m-1}, i_m - 1) \\ = \mathtt{f}' \end{array}$$

$\square$

4.2.3 Application au produit matrice-vecteur

Considérons une première version de programme du produit matrice-vecteur, calculant $p = A{\times}b$, où b est un vecteur de longueur n. Pour des raisons de lisibilité, nous ne nous intéressons pour l'instant qu'à la première partie de l'algorithme, c'est-à-dire la multiplication des coefficients, la somme de ces produits constituant une phase traitée ultérieurement. L'algorithme utilisé pour cette première phase est une version systolique : le vecteur b est initialement placé sur la diagonale de la matrice A. A chaque étape $k = 1 \ldots n$, nous pouvons effectuer n produits $p_{i,i} = A_{i,i} \times b_i$ sur la diagonale. Puis tous les éléments de A sont décalés sur la gauche, et les éléments de b sont décalés le long de la diagonale, vers le haut à gauche (les bords sont bouclés toroïdalement) avant l'étape suivante (voir figure 4.1, sur laquelle les flèches montrent le déplacement des valeurs du champ `A`). En représentant les calculs dans un cube formé des deux directions i et j de la matrice A et de la direction k, au bout de n étapes, nous avons l'ensemble des produits sur le plan $(i = j)$ du cube.
L'algorithme décrit se traduit par le progamme PEI suivant : `P` est défini comme étant la superposition des champs de données `A` et `B`, tous deux définis récursivement par les routages utilisant les fonctions `left` et `upleft` avec les valeurs initiales `A0` et `B0` définies à partir des données `M` et `V`.

$$\texttt{Matvect1 : (M,V)} \mapsto \texttt{P}$$
$$\begin{cases} \texttt{A0} = \texttt{cube} :: \texttt{A0} \\ \texttt{B0} = \texttt{cube} :: \texttt{B0} \\ \texttt{M} = \texttt{aligna} :: \texttt{A0} \\ \texttt{V} = \texttt{alignb} :: \texttt{B0} \\ \texttt{A} = \texttt{A0 /\&/ (A} \lhd \texttt{ left)} \\ \texttt{B} = \texttt{B0 /\&/ (B} \lhd \texttt{ upleft)} \\ \texttt{P} = \texttt{prod} \rhd \texttt{((A /\&/B)} \lhd \texttt{ diag)} \end{cases}$$

$$\begin{array}{lcl} \texttt{cube} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).(i,j,k)} \\ \texttt{aligna} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&k=0).(i,j)} \\ \texttt{alignb} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&i=j\&k=0).(i)} \\ \texttt{left} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).(i,(j+1) mod n,k-1)} \\ \texttt{upleft} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).((i+1) mod n,(j+1) mod n,k-1)} \\ \texttt{prod} & = & \lambda\texttt{(a;b).(a*b)} \\ \texttt{diag} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k}\leq\texttt{n\&i=j)} \end{array}$$

Décomposition des routages

Les routages `left` et `upleft` peuvent se décomposer de façon à faire apparaître une fonction de routage `g` dans les expressions :

$$\begin{array}{lcll} \texttt{left} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{k<n).(i,(j+1) mod n,k)} \circ & \\ & & \lambda\texttt{(i,j,k)}|\texttt{(0<k<n \& 0}\leq\texttt{i,j<n).(i,j,k-1)} & = \texttt{g} \circ \texttt{prev} \\ \texttt{upleft} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{k<n).(i,(j+1) mod n,k)} \circ & \\ & & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n).((i+1) mod n,j,k)} \circ & \\ & & \lambda\texttt{(i,j,k)}|\texttt{(0<k<n \& 0}\leq\texttt{i,j<n).(i,j,k-1)} & = \texttt{g} \circ \texttt{up} \circ \texttt{prev} \end{array}$$

Les noms donnés aux fonctions obtenues dans cette décomposition reflètent le fait que, déplacer les valeurs le long de la diagonale est équivalent à les déplacer vers la gauche, puis vers le haut. L'idée est donc de démontrer qu'un programme qui ne déplace aucune valeur de `A` tout en routant les valeurs de `B` vers le haut, est équivalent au programme précédent ; autrement dit, que nous pouvons simplifier les opérations de routage en enlevant le facteur commun `g` des deux fonctions.

Nous devons montrer qu'il existe un changement de base permettant d'obtenir, à partir du programme initial, un programme avec des routages simplifiés, c'est-à-dire que pour chaque équation de la forme $\texttt{X} = \texttt{X0 /\&/ (X} \lhd \texttt{ g} \circ \texttt{f')}$, il existe un changement de base, tel que lorsqu'il est appliqué à cette équation, elle se réécrit de la manière suivante : $\texttt{h} :: \texttt{X} = \texttt{(h} :: \texttt{X0) /\&/ ((h} :: \texttt{X)} \lhd \texttt{ f')}$.

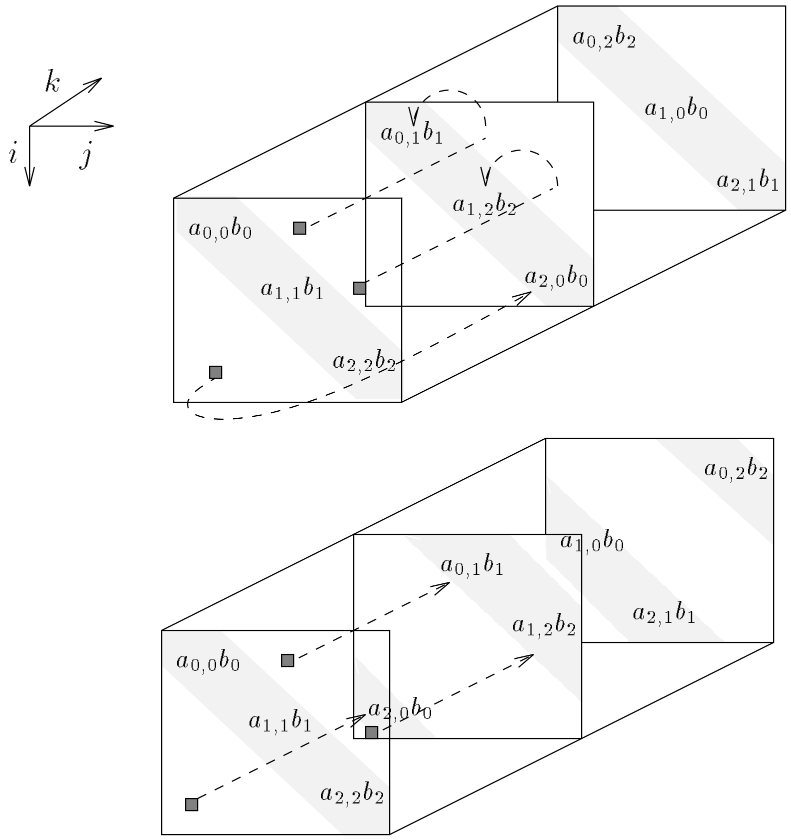

FIG. 4.1 - *Transformation de* `Matvect1` *en* `Matvect2` *pour* $n = 3$

Transformation de `Matvect1`

Nous pouvons maintenant appliquer la stratégie de transformation précédente pour obtenir de nouvelles définitions des champs `A` et `B`, dont les routages ne comportent plus le facteur `g`. Ensuite, il faut transformer chaque équation faisant référence à `A` et `B` pour tenir compte de leurs nouvelles définitions. La transformation est réalisée par le changement de base `h`. Dans notre exemple, nous déduisons du facteur

$$\mathtt{g} = \lambda\mathtt{(i,j,k)}|\mathtt{(0}\leq\mathtt{k<n).(i,(j+1)\ mod\ n,k)}$$

la bijection :

$$\mathtt{h} = \lambda\mathtt{(i,j,k)}|\mathtt{(0}\leq\mathtt{k<n).(i,(j+k)\ mod\ n,k)}$$

dont l'inverse est $\mathtt{h}^{-1} = \lambda$`(i,j,k)|(0≤k<n).(i,(j-k) mod n,k)`.

- Le théorème 1 nous permet de réécrire les équations définissant `A` et `B` car *img*(`left`) et *img*(`upleft`) sont inclus dans *dom*(`h`), et tous les champs ont leurs domaines de référence inclus dans *dom*(`h`). Nous avons

$$\begin{array}{lcl} \mathtt{A} = \mathtt{A0}\ \mathtt{/\&/}\ (\mathtt{A}\ \lhd\ \mathtt{left}) & \longrightarrow & \mathtt{h}::\mathtt{A} = (\mathtt{h}::\mathtt{A0})\mathtt{/\&/}((\mathtt{h}::\mathtt{A})\ \lhd\ \mathtt{prev}) \\ \mathtt{B} = \mathtt{B0}\ \mathtt{/\&/}\ (\mathtt{B}\ \lhd\ \mathtt{upleft}) & \longrightarrow & \mathtt{h}::\mathtt{B} = (\mathtt{h}::\mathtt{B0})\mathtt{/\&/}((\mathtt{h}::\mathtt{B})\ \lhd\ \mathtt{up}\circ\mathtt{prev}) \end{array}$$

- Nous devons maintenant exprimer les champs `M` et `V` qui dépendent de `A0` et `B0` respectivement, en fonction de (`h :: A0`) et (`h :: B0`) . Pour cela, nous introduisons ces dernières expressions dans les définitions : `M = aligna :: A0`, et `V = alignb :: B0` sous la forme suivante :

$$\begin{array}{l} \mathtt{M} = (\mathtt{aligna}\circ\mathtt{h}^{-1}\circ\mathtt{h})::\mathtt{A0} \\ \mathtt{V} = (\mathtt{alignb}\circ\mathtt{h}^{-1}\circ\mathtt{h})::\mathtt{B0} \end{array}$$

  L'application de la règle (7a) mène à une réécriture des alignements de `A0` et `B0` en :

$$\begin{array}{l} \mathtt{M} = \mathtt{aligna}\circ\mathtt{h}^{-1}::(\mathtt{h}::\mathtt{A0}) \\ \mathtt{V} = \mathtt{alignb}\circ\mathtt{h}^{-1}::(\mathtt{h}::\mathtt{B0}) \end{array}$$

  Le calcul de $\mathtt{aligna}\circ\mathtt{h}^{-1}$ donne λ`(i,j,k)|(0≤i,j<n & k=0).(i,j) = aligna` et celui de $\mathtt{alignb}\circ\mathtt{h}^{-1}$ donne λ`(i,j,k)|(0≤i,j<n & i=j & k=0).(i) = alignb`.

- Egalement, nous introduisons les nouvelles expressions `h :: A` et `h :: B` dans l'équation définissant `P`, en appliquant le changement de base aux deux membres de l'équation. L'équation `h :: P = h :: (prod ▷ ((A /&/ B) ◁ diag))` devient :

$$\begin{array}{ll} \xrightarrow{(1b)} & \mathtt{h}::\mathtt{P} = \mathtt{prod} \rhd (\mathtt{h}::((\mathtt{A}\ \mathtt{/\&/}\ \mathtt{B})\ \lhd\ \mathtt{diag})) \\ \xrightarrow{(3b)} & \mathtt{h}::\mathtt{P} = \mathtt{prod} \rhd ((\mathtt{h}::(\mathtt{A}\ \mathtt{/\&/}\ \mathtt{B}))\ \lhd\ \mathtt{diag'}) \\ & \text{car } dom(\mathtt{diag}) \cup img(\mathtt{diag}) \subseteq dom(\mathtt{h}) \text{ et} \\ & \text{où } \mathtt{diag'} = \mathtt{h}\circ\mathtt{diag}\circ\mathtt{h}^{-1} = \lambda\mathtt{(i,j,k)|(i=(j-k)\ mod\ n)} \\ \xrightarrow{(11b)} & \mathtt{h}::\mathtt{P} = \mathtt{prod} \rhd (((\mathtt{h}::\mathtt{A})\ \mathtt{/\&/}\ (\mathtt{h}::\mathtt{B}))\ \lhd\ \mathtt{diag'}) \end{array}$$

Toutes ces équations forment le programme `Matvect1'` :

$$\texttt{Matvect1'} : (\texttt{M},\texttt{V}) \mapsto \texttt{P}$$
$$\left\{\begin{array}{l}
\texttt{h::A0} = \texttt{cube::(h::A0)} \\
\texttt{h::B0} = \texttt{cube::(h::B0)} \\
\texttt{M} = \texttt{aligna::(h::A0)} \\
\texttt{V} = \texttt{alignb::(h::B0)} \\
\texttt{h::A} = \texttt{h::A0 /\&/ ((h::A)} \lhd \texttt{prev)} \\
\texttt{h::B} = \texttt{h::B0 /\&/ ((h::B)} \lhd \texttt{up} \circ \texttt{prev)} \\
\texttt{h::P} = \texttt{prod} \rhd \texttt{(((h::A) /\&/ (h::B))} \lhd \texttt{diag')}
\end{array}\right.$$
$$\begin{array}{lcl}
\texttt{prev} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n \& 0<k<n).(i,j,k-1)} \\
\texttt{up} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n).((i+1) mod n,j,k)} \\
\texttt{diag}' & = & \lambda\texttt{(i,j,k)}|\texttt{(i=(j-k) mod n)}
\end{array}$$

Ce programme peut être simplifié en utilisant la propriété 3, car tous les champs ont leur domaines de référence inclus dans $dom(\texttt{h})$ et on vérifie $img(\texttt{h}) \subseteq dom(\texttt{h})$. Egalement, les deux premières équations peuvent être supprimées car $\texttt{h} \circ \texttt{cube = h}$, et on vérifie $dom(v_{\texttt{A0}}) \subseteq dom(\texttt{h})$. Nous obtenons ainsi le programme `Matvect2`, raffinant `Matvect1`:

$$\texttt{Matvect2} : (\texttt{M},\texttt{V}) \mapsto \texttt{P}$$
$$\left\{\begin{array}{l}
\texttt{M} = \texttt{aligna::A0} \\
\texttt{V} = \texttt{alignb::B0} \\
\texttt{A} = \texttt{A0 /\&/ (A} \lhd \texttt{prev)} \\
\texttt{B} = \texttt{B0 /\&/ (B} \lhd \texttt{up} \circ \texttt{prev)} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A/\&/B)} \lhd \texttt{diag'}
\end{array}\right.$$
$$\begin{array}{lcl}
\texttt{prev} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n \& 0<k<n).(i,j,k-1)} \\
\texttt{up} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n).((i+1) mod n,j,k)}
\end{array}$$

En fait, en utilisant la même technique, on peut montrer que les programmes `Matvect1` et `Matvect2` sont équivalents.

4.2.4 Application au produit de matrices de Cannon

Un exemple similaire est celui de l'algorithme de multiplication de matrices de Cannon [8]. L'algorithme évite la réplication des coefficients en routant les valeurs nécessaires pour chacune des multiplications. L'encombrement mémoire induit par cet algorithme est de fait réduit [34]. Comme le montre la figure 4.2, après l'alignement initial des matrices, et le produit des coefficients appariés, le routage consiste en une rotation de droite à gauche des valeurs de la première matrice, et du bas vers le haut de celles de la deuxième matrice. L'algorithme peut s'écrire :

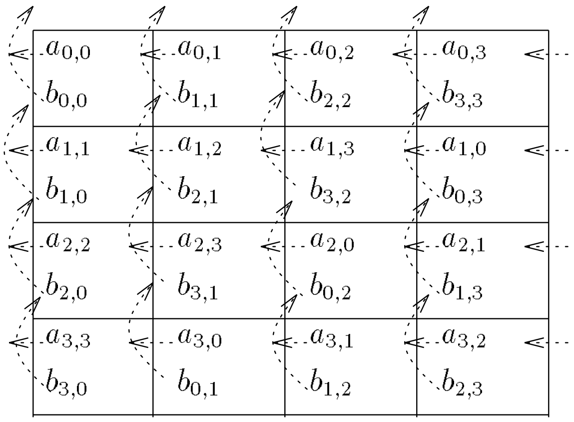

FIG. 4.2 - *Produit de matrices de Cannon* (n=4) *: alignement initial des matrices et communications*

```
Cannon : (MA,MB) ↦ C
  ⎧ MA = align1 :: A0
  ⎪ MB = align2 :: B0
  ⎨ A = A0 /&/ (A ◁ left)
  ⎪ B = B0 /&/ (B ◁ up)
  ⎪ P = prod ▷ (A /&/ B)
  ⎩ C = add ▷ (P /&/ C ◁ prev)
 align1  =  λ(i,j,k)|(0≤i≤n & 0≤j<n & k=0).(i,(i+j) mod n)
 align2  =  λ(i,j,k)|(0≤i,j≤n & k=0).((i+j) mod n,j)
 left    =  λ(i,j,k)|(0≤i<n & 0<k<n).(i,(j+1) mod n,k-1)
 up      =  λ(i,j,k)|(0≤i,j<n & 0<k<n).((i+1) mod n,j,k-1)
 prev    =  λ(i,j,k)|(0≤i,j<n & 0<k<n).(i,j,k-1)
 prod    =  λ(a;b).(a+b)
 add     =  id # λ(a;b).(a+b)
```

Nous pouvons là aussi, trouver des programmes équivalents. Par exemple, en réécrivant les fonctions de routages `left` et `up` de la manière suivante :

$$
\begin{array}{lcl}
\texttt{left} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{k<n).((i+1) mod n,j,k)} \circ \\
 & & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{j<n\&0<k<n).((i-1) mod n,(j+1) mod n,k-1)} \\
 & = & \texttt{g} \circ \texttt{downleft} \\
\texttt{up} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{k<n).((i+1) mod n,j,k)} \circ \\
 & & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).(i,j,k-1)} \\
 & = & \texttt{g} \circ \texttt{prev}
\end{array}
$$

La bijection $\texttt{h} = \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{k<n).((i+k) mod n,j,k)}$, obtenue à partir du facteur commun g est utilisée pour effectuer un changement de base et obtenir le programme suivant, selon le même procédé :

$$
\texttt{Cannon2 : (MA,MB)} \mapsto \texttt{C}
$$
$$
\left\{
\begin{array}{l}
\texttt{MA} = \texttt{align1::A0} \\
\texttt{MB} = \texttt{align2::B0} \\
\texttt{A} = \texttt{A0 /\&/(A} \lhd \texttt{downleft)} \\
\texttt{B} = \texttt{B0 /\&/(B} \lhd \texttt{prev)} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/B)} \\
\texttt{C} = \texttt{add} \rhd \texttt{(P /\&/C} \lhd \texttt{prev)}
\end{array}
\right.
$$
$$
\begin{array}{lcl}
\texttt{downleft} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{j<n\&0<k<n).((i-1) mod n,(j+1) mod n,k-1)} \\
\texttt{prev} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).(i,j,k-1)}
\end{array}
$$

Dans ce nouveau programme, les coefficients de la matrice A sont déplacés succesivement sur la ligne suivante et la colonne précédente, du plan suivant, tandis que les coefficients de B sont simplement déplacés sur le plan suivant.

4.3 Stratégie 2 : introduction d'une diffusion

Le programme du produit matrice-vecteur Matvect1, est un algorithme que nous avons qualifié de systolique car ses routages représentent des communications point-à-point. En supposant que l'on dispose d'une primitive de diffusion, caractéristique du modèle de programmation data-parallèle, ces communications peuvent être remplacées par cet opérateur de diffusion. Cela signifie qu'une définition récursive de champ de données (celle de B sur l'exemple) est transformée en une définition simple, faisant intervenir une opération de routage, elle-même définie récursivement. Une telle transformation repose sur le résultat suivant, qui décrit une autre stratégie de transformation de programmes :

Théorème 2 *Soit* g *une fonction définie récursivement par* g = i # (g ∘ s), *où* i *est l'identité définie sur un sous-domaine, et* s *est une fonction. L'équation* Y = X ◁ g *se réécrit alors en :* Y = X ◁ i /&/ Y ◁ s

Preuve. Par application des règles de raffinement, nous avons :

$$\begin{array}{ll} & \mathtt{Y} = \mathtt{X} \lhd \mathtt{g} \\ \xrightarrow{def.} & \mathtt{Y} = \mathtt{X} \lhd (\mathtt{i} \,\#\, (\mathtt{g} \circ \mathtt{s})) \\ \xrightarrow{(9a)} & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ \mathtt{X} \lhd (\mathtt{g} \circ \mathtt{s}) \\ \xrightarrow{(5a)} & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd \mathtt{g}) \lhd \mathtt{s} \end{array}$$

En réitérant le calcul :

$$\begin{array}{ll} & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd (\mathtt{i} \,\#\, (\mathtt{g} \circ \mathtt{s}))) \lhd \mathtt{s} \\ \longrightarrow & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd \mathtt{i}) \lhd \mathtt{s}\ /\&/\ (\mathtt{X} \lhd \mathtt{g} \circ \mathtt{s}) \lhd \mathtt{s} \\ \longrightarrow & \mathtt{Y} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd \mathtt{i}) \lhd \mathtt{s}\ /\&/\ ((\mathtt{X} \lhd \mathtt{i}) \lhd \mathtt{s}) \lhd \mathtt{s}\ /\&/\ .. \end{array}$$

Déplions maintenant la définition Y' = X ◁ i /&/ Y' ◁ s :

$$\begin{array}{ll} & \mathtt{Y'} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ \mathtt{Y'} \lhd \mathtt{s} \\ \xrightarrow{def} & \mathtt{Y'} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd \mathtt{i}) \lhd \mathtt{s}\ /\&/\ (\mathtt{Y'} \lhd \mathtt{s}) \lhd \mathtt{s} \\ \xrightarrow{def} & \mathtt{Y'} = \mathtt{X} \lhd \mathtt{i}\ /\&/\ (\mathtt{X} \lhd \mathtt{i}) \lhd \mathtt{s}\ /\&/\ ((\mathtt{X} \lhd \mathtt{i}) \lhd \mathtt{s}) \lhd \mathtt{s}\ /\&/\ .. \end{array}$$

Pour toute étape n de la récurrence, les n premiers termes des définitions de Y et Y' sont égaux, donc Y = Y'. Finalement nous avons : Y = X ◁ g ⟶ Y = (X ◁ i) /&/ (Y ◁ s)
□

Les définitions des champs de données A et B dans le programme Matvect2 sont semblables à l'expression raffinée du théorème 2. Intuitivement, on constate que le champ A, par exemple, est constitué des valeurs de A0 qui ont été successivement recopiées sur chacun des plans k de A. On déduit facilement de cette observation, une fonction qui exprime qu'une valeur placée au point (i, j, k) du champ A provient du point $(i, j, 0)$ du champ A0. Nous écrivons donc en Pei l'équation A = A0 ◁ spreada, où spreada = λ(i,j,k)|(0≤i,j,k<n).(i,j,0) est l'opération de routage traduisant cette diffusion. Après avoir défini B de manière similaire, nous trouvons le programme suivant, utilisant uniquement des routages de type diffusion.

$$\texttt{Matvect3 : (M,V)} \mapsto \texttt{P}$$
$$\left\{\begin{array}{l}\texttt{M} = \texttt{aligna}::\texttt{A0}\\ \texttt{V} = \texttt{alignb}::\texttt{B0}\\ \texttt{A} = \texttt{A0} \lhd \texttt{spreada}\\ \texttt{B} = \texttt{B0} \lhd \texttt{spreadb}\\ \texttt{P} = \texttt{prod} \rhd \texttt{((A/\&/ B)} \lhd \texttt{diag')}\end{array}\right.$$
$$\texttt{spreada} = \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).(i,j,0)}$$
$$\texttt{spreadb} = \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).((i+k) mod n,j,0)}$$

Il s'agit maintenant de vérifier que ce programme est raffiné par le précédent. Pour cela, nous appliquons le théorème 2.

Preuve. D'après l'énoncé du théorème 2, il faut montrer qu'on peut réécrire les fonctions `spreada` et `spreadb` sous une forme récursive identique à la définition de `g` dans le théorème. En effet, `spreada` et `spreadb` peuvent se réécrire de la manière suivante :

$$\begin{array}{rcl}
\texttt{spreada} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).(i,j,0)}\\
 & = & \lambda\texttt{(i,j,k)}|\texttt{(k=0)} \;\#\; \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).(i,j,0)}\\
 & = & \lambda\texttt{(i,j,k)}|\texttt{(k=0)} \;\#\; \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).(i,j,0)}\\
 & \circ & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).(i,j,k-1)}\\
 & = & \texttt{i} \;\#\; \texttt{spreada} \circ \texttt{prev}\\
\texttt{spreadb} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).((i+k) mod n,j,0)}\\
 & = & \lambda\texttt{(i,j,k)}|\texttt{(k=0).((i+k) mod n,j,0)} \;\#\\
 & & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).((i+k) mod n,j,0)}\\
 & = & \lambda\texttt{(i,j,k)}|\texttt{(k=0)} \;\#\\
 & & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j,k<n).((i+k) mod n,j,0)}\\
 & \circ & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&0<k<n).((i+1) mod n,j,k-1)}\\
 & = & \texttt{i} \;\#\; \texttt{spreadb} \circ \texttt{up} \circ \texttt{prev}
\end{array}$$

□

4.4 Représentation du problème en deux dimensions

Nous pouvons constater, d'après la définition du champ `P` du précédent programme, que ses valeurs sont au nombre de n^2, car les produits ne s'effectuent que sur le plan diagonal du cube. On peut donc souhaiter «compacter» les valeurs placées par ce champ dans un espace carré à deux dimensions, par exemple, comme le montre la figure 4.3. Ce compactage illustre une application de l'équivalence faible : nous ne nous intéressons plus à tous les points de notre domaine de référence, mais seulement au domaine des valeurs. Nous procédons en deux étapes pour montrer la relation entre le programme en deux dimensions et celui en trois dimensions : d'abord nous transformons le programme `Matvect3'` afin que les valeurs de tous ses champs soient placées sur le premier plan. Ensuite nous montrons que ce nouveau programme est un raffinement du programme dont les champs appartiennent à $\mathbb{Z}^2$.

4.4.1 Regroupement des valeurs sur le premier plan

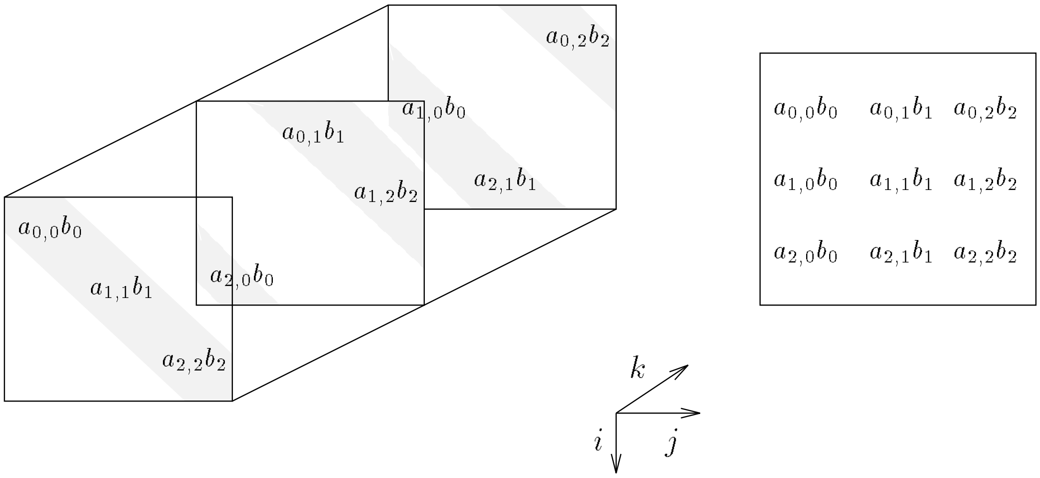

FIG. 4.3 - *Compactage des champs* `A` *et* `B` *dans un carré de* $\mathbb{Z}^2$

Pour mettre en évidence le placement des valeurs sur une partie seulement du domaine de référence, nous écrivons un programme plus abstrait, `Mavect3'`, dont les routages sur les champs `A`, `B` prennent en compte le placement des valeurs restreint au plan $i = (j - k)\ mod\ n$. Ce programme est le suivant :

$$\texttt{Matvect3'} : (\texttt{M,V}) \mapsto \texttt{P}$$
$$\begin{cases} \texttt{M} = \texttt{aligna} :: \texttt{A0} \\ \texttt{V} = \texttt{alignb} :: \texttt{B0} \\ \texttt{A'} = \texttt{A0} \lhd \texttt{spreada}' \\ \texttt{B'} = \texttt{B0} \lhd \texttt{spreadb}' \\ \texttt{P} = \texttt{prod} \rhd (\texttt{A'/\&/B'}) \end{cases}$$
$$\texttt{spreada}' = \lambda(\texttt{i,j,k})|(\texttt{0}\leq\texttt{i,j,k<n}\,\&\,\texttt{i=(j-k) mod n}).(\texttt{i,j,0})$$
$$\texttt{spreadb}' = \lambda(\texttt{i,j,k})|(\texttt{0}\leq\texttt{i,j,k<n}\,\&\,\texttt{i=(j-k) mod n}).(\texttt{j,j,0})$$

Le raffinement du programme précédent est obtenu en réécrivant les équations comportant les routages :

$$\begin{array}{ll} & \texttt{A'} = \texttt{A0} \lhd \texttt{spreada}' \\ \overset{(5a)}{\to} & \texttt{A'} = (\texttt{A0} \lhd \texttt{spreada}) \lhd \texttt{diag}' \\ & \text{car } \texttt{spreada} \circ \texttt{diag}' = \texttt{spreada}' \end{array}$$

$$\begin{array}{ll} & \texttt{B'} = \texttt{B0} \lhd \texttt{spreadb}' \\ \overset{(5a)}{\to} & \texttt{B'} = (\texttt{B0} \lhd \texttt{spreadb}) \lhd \texttt{diag}' \\ & \text{car } \texttt{spreadb} \circ \texttt{diag}' = \texttt{spreadb}' \end{array}$$

En renommant les expressions $\texttt{A0} \lhd \texttt{spreada}$ et $\texttt{B0} \lhd \texttt{spreadb}$, respectivement `A` et `B`, nous obtenons pour l'équation du champ `P` :

$$\begin{array}{ll} & \texttt{P} = \texttt{prod} \rhd (\texttt{A} \lhd \texttt{diag}' \ \texttt{/\&/B} \lhd \texttt{diag}') \\ \overset{(8b)}{\to} & \texttt{P} = \texttt{prod} \rhd ((\texttt{A /\&/B}) \lhd \texttt{diag}') \\ & \text{car } dom(\texttt{diag}') \subseteq dom(\sigma_{\texttt{A}}) \cap dom(\sigma_{\texttt{B}}) \wedge img(\texttt{diag}') \subseteq dom(v_{\texttt{A}}) \cap dom(v_{\texttt{B}}) \\ & \text{(de la dernière équation de } \texttt{Matvect3}) \end{array}$$

Nous obtenons ainsi le programme `Matvect3` par raffinement.

Nous pouvons maintenant expliciter le placement des valeurs sur le premier plan du cube. Nous constatons que les domaines de valeurs de `A0` et `A'`, de même que ceux de `B0` et `B'`, sont différents bien que leurs cardinaux soient égaux. Nous allons montrer que le programme `Matvect3"` qui suit, est raffiné par `Matvect3'`.

$$\texttt{Matvect3" : (M,V)} \mapsto \texttt{P}$$

$$\left\{\begin{array}{ll} \texttt{M} = \texttt{aligna} :: \texttt{A0} & (i) \\ \texttt{V} = \texttt{alignb} :: \texttt{B0} & (ii) \\ \texttt{A} = \texttt{A0} \lhd \texttt{front} & (iii) \\ \texttt{B} = \texttt{B0} \lhd \texttt{spreadb}'' & (iv) \\ \texttt{P} = \texttt{prod} \rhd (\texttt{A/\&/B}) & \end{array}\right.$$

$$\begin{array}{lcl} \texttt{front} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&k=0).(i,j,k)} \\ \texttt{spreadb}'' & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&k=0).(j,j,0)} \end{array}$$

Pour cela, nous transformons l'opération de routage sur le champ `B`, en utilisant la propriété 2 qui permet remplacer toutes les occurrences de `B0` par `h :: B0`, où la bijection `h`, égale à son inverse, est :

$$\begin{array}{rl} \texttt{h} = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i,j<n\&k=0).(i,j,(j-i) mod n)} \ \# \\ & \lambda\texttt{(i,j,k)}|\texttt{(i=(j-k) mod n\&k}\neq\texttt{0).(i,j,0)} \\ & \lambda\texttt{(i,j,k)}|\texttt{(i}\neq\texttt{(j-k) mod n\&k}\neq\texttt{0).(i,j,k)} \end{array}$$

Nous avons :

$$
\begin{array}{ll}
 & \mathtt{V} = \mathtt{alignb} :: \mathtt{B0} \\
\stackrel{(prop.2)}{\rightarrow} & \mathtt{V} = \mathtt{alignb} :: (\mathtt{h} :: \mathtt{B0}) \\
 & \text{car } dom(\sigma_{\mathtt{B0}}) \subseteq img(\mathtt{h}) \text{ de } (iv) \\
\stackrel{(7b)}{\rightarrow} & \mathtt{V} = (\mathtt{alignb} \circ \mathtt{h}) :: \mathtt{B0} \\
\rightarrow & \mathtt{V} = \mathtt{alignb} :: \mathtt{B0} \\
 & \text{car } \mathtt{alignb} \circ \mathtt{h} \texttt{ = } \mathtt{alignb}
\end{array}
$$

et

$$
\begin{array}{ll}
 & \mathtt{B} = (\mathtt{h} :: \mathtt{B0}) \lhd \mathtt{spreadb}'' \\
\stackrel{(3a)}{\rightarrow} & \mathtt{B} = \mathtt{h} :: (\mathtt{B0} \lhd \mathtt{spreab}') \\
 & \text{car } \mathtt{spreadb}' = \mathtt{h}^{-1} \circ \mathtt{spreadb}'' \circ \mathtt{h} \\
 & \text{et } dom(\mathtt{spreadb}'') \cup img(\mathtt{spreadb}'') \subseteq img(\mathtt{h}) \wedge dom(v_{\mathtt{B0}}) \subseteq dom(\mathtt{h}) \\
 & \text{car } dom(v_{\mathtt{B0}}) = dom(\mathtt{alignb}) \text{ des deuxièmes et quatrièmes équations de } \mathtt{Matvect3'}
\end{array}
$$

Pour le champ `A`, nous voulons introduire l'opération géométrique $\mathtt{spreada}'$ appliquée sur `A0`. Il s'agit là d'équivalence faible, et nous remplaçons l'expression $\mathtt{A} = \mathtt{A0} \lhd \mathtt{front}$, à l'aide de la propriété 4.(1), par la nouvelle expression :

$$\mathtt{h} :: ((\mathtt{A0} \lhd \mathtt{front}) \lhd \mathtt{spreada}')$$

puisque nous vérifions les conditions :

- `h` est une inverse gauche de $\mathtt{spreada}'^{-1}$
- $dom(\mathtt{front}) = img(\mathtt{spreada}')$
  car $img(\mathtt{spreada}') = img(\mathtt{front}) \subseteq dom(v_{\mathtt{A0}}) \subseteq dom(\mathtt{aligna}) = img(\mathtt{front})$ de (i) et (iii).
- $\mathtt{h}(img(\mathtt{spreada}')) \subseteq dom(\sigma_{\mathtt{A0}}) \subseteq img(\mathtt{h})$
  car $\mathtt{h}(img(\mathtt{spreada}')) = dom(\mathtt{spreada}') \subseteq dom(\mathtt{front}) \subseteq dom(\sigma_{\mathtt{A0}}) \subseteq img(\mathtt{h})$ de (iii).

L'équation définissant le champ `A` devient ensuite :

$$
\begin{array}{ll}
 & \mathtt{A} = \mathtt{h} :: ((\mathtt{A0} \lhd \mathtt{front}) \lhd \mathtt{spreada}') \\
\stackrel{(5b)}{\rightarrow} & \mathtt{A} = \mathtt{h} :: (\mathtt{A0} \lhd \mathtt{front} \circ \mathtt{spreada}') \\
 & \text{car } img(\mathtt{spreada}') \subseteq dom(\mathtt{front}) \subseteq dom(\sigma_{\mathtt{A0}}) \\
\rightarrow & \mathtt{A} = \mathtt{h} :: (\mathtt{A0} \lhd \mathtt{spreada}') \\
 & \text{car } \mathtt{front} \circ \mathtt{spreada}' = \mathtt{spreada}'
\end{array}
$$

L'équation définissant le champ P, en utilisant les nouvelles définitions de A et B, devient :

$$
\begin{array}{ll}
 & \mathtt{P} = \mathtt{prod} \rhd (\mathtt{h}::(\mathtt{A0} \lhd \mathtt{spreada'})\ \mathtt{/\&/}\ \mathtt{h}::(\mathtt{B0} \lhd \mathtt{spreadb'})) \\
\stackrel{(11a)}{\to} & \mathtt{P} = \mathtt{prod} \rhd (\mathtt{h}::(\mathtt{A0} \lhd \mathtt{spreada'}\ \mathtt{/\&/}\ \mathtt{B0} \lhd \mathtt{spreadb'})) \\
\stackrel{(1a)}{\to} & \mathtt{P} = \mathtt{h}::(\mathtt{prod} \rhd (\mathtt{A0} \lhd \mathtt{spreada'}\ \mathtt{/\&/}\ \mathtt{B0} \lhd \mathtt{spreadb'})) \\
\stackrel{(prop.2)}{\to} & \mathtt{h}::\mathtt{P} = \mathtt{h}::(\mathtt{prod} \rhd (\mathtt{A0} \lhd \mathtt{spreada'}\ \mathtt{/\&/}\ \mathtt{B0} \lhd \mathtt{spreadb'})) \\
 & \text{car } dom(\sigma_{\mathtt{P}}) \subseteq img(\mathtt{h}) \\
\stackrel{(II)}{\to} & \mathtt{P} = \mathtt{prod} \rhd (\mathtt{A0} \lhd \mathtt{spreada'}\ \mathtt{/\&/}\ \mathtt{B0} \lhd \mathtt{spreadb'}) \\
 & \text{car } dom(\upsilon_{\mathtt{P}}) \subseteq dom(\mathtt{h})
\end{array}
$$

Il suffit maintenant de nommer A l'expression $\mathtt{A0} \lhd \mathtt{spreada'}$ et B l'expression $\mathtt{B0} \lhd \mathtt{spreadb'}$, pour obtenir le programme Matvect3'.

4.4.2 Suppression d'une dimension

Nous pouvons maintenant montrer que le programme Matvect3" raffine le programme suivant, dont les champs sont exprimés dans $\mathbb{Z}^2$:

$$
\mathtt{Matvect4}\ :\ (\mathtt{M},\mathtt{V}) \mapsto \mathtt{P}
$$
$$
\left\{
\begin{array}{l}
\mathtt{A0} = \mathtt{matrix}::\mathtt{A0} \\
\mathtt{B0} = \mathtt{matrix}::\mathtt{B0} \\
\mathtt{M} = \mathtt{A0} \\
\mathtt{V} = \mathtt{align}::\mathtt{B0} \\
\mathtt{B} = \mathtt{B0} \lhd \mathtt{spread} \\
\mathtt{P} = \mathtt{prod} \rhd (\mathtt{A0/\&/B})
\end{array}
\right.
$$
$$
\begin{array}{lcl}
\mathtt{matrix} & = & \lambda(\mathtt{i},\mathtt{j})|(0 \leq \mathtt{i},\mathtt{j} < \mathtt{n}) \\
\mathtt{align} & = & \lambda(\mathtt{i},\mathtt{j})|(0 \leq \mathtt{i},\mathtt{j} < \mathtt{n}\ \&\ \mathtt{i} = \mathtt{j}).(\mathtt{i}) \\
\mathtt{spread} & = & \lambda(\mathtt{i},\mathtt{j})|(0 \leq \mathtt{i},\mathtt{j} < \mathtt{n}).(\mathtt{j},\mathtt{j})
\end{array}
$$

Pour le montrer, il suffit de substituer dans Matvect4 toutes les occurrences de A0, B0, B et P par aligna : : A0, aligna : : B0, aligna : : B et aligna : : P respectivement, à l'aide de la propriété 2 (on sait des deux premières équations de Matvect4 que $dom(\sigma_{\mathtt{A0}})$ et $dom(\sigma_{\mathtt{B0}}) \subseteq img(\mathtt{aligna})$, et on déduit des autres équations $dom(\sigma_{\mathtt{P}}) = dom(\sigma_{\mathtt{B}}) = dom(\sigma_{\mathtt{B0}})$). Nous avons :

$$\left\{\begin{array}{l}
\texttt{aligna::A0} = \texttt{matrix::(aligna::A0)} \\
\texttt{aligna::B0} = \texttt{matrix::(aligna::B0)} \\
\texttt{M} = \texttt{aligna::A0} \\
\texttt{V} = \texttt{align::(aligna::B0)} \\
\texttt{aligna::B} = \texttt{(aligna::B0)} \lhd \texttt{spread} \\
\texttt{aligna::P} = \texttt{prod} \rhd \texttt{(aligna::A0/\&/ aligna::B)}
\end{array}\right.$$

Comme `matrix` $\circ$ `aligna` = `aligna`, les deux premières équations sont toujours vraies (les contraintes $dom(v_{\texttt{A0}}) \subseteq dom(\texttt{aligna})$ et $dom(v_{\texttt{B0}}) \subseteq dom(\texttt{aligna})$ pouvant être déduites respectivement de la troisième et quatrième équation) et peuvent être supprimées. En effet après réécriture par la règle (7b), on obtient par exemple pour la première équation : `aligna::A0` = `aligna::A0`. La substitution nous donne immédiatement l'alignement de `M` par rapport à `A0` dans `Matvect3"` (troisième équation). De même, l'alignement de `V` par rapport à `B0` apparaît après l'application de la règle (7b), car `align` $\circ$ `aligna` `= alignb`. Le routage `spreadb`$''$ employé dans `Matvect3"` s'obtient par le raffinement de l'équation définissant `B` :

$$\begin{array}{ll}
 & \texttt{aligna::B} = \texttt{(aligna::B0)} \lhd \texttt{spread} \\
\stackrel{(3a)}{\to} & \texttt{aligna::B} = \texttt{aligna::(B0} \lhd \texttt{aligna}^{\texttt{-1}} \circ \texttt{spread} \circ \texttt{aligna)} \\
 & \text{car } dom(\texttt{spread}) \subseteq img(\texttt{spread}) \subseteq img(\texttt{aligna}) \\
 & \text{et } dom(v_{\texttt{B0}}) \subseteq dom(\texttt{aligna}) \text{ de l'équation définissant } \texttt{B} \\
\stackrel{def.}{\to} & \texttt{aligna::B} = \texttt{aligna::(B0} \lhd \texttt{spreadb)} \\
 & \text{car } \texttt{aligna}^{\texttt{-1}} \circ \texttt{spread} \circ \texttt{aligna} = \texttt{spreadb}'' \\
\stackrel{(II)}{\to} & \texttt{B} = \texttt{B0} \lhd \texttt{spreadb}'' \\
 & \text{car } dom(v_{\texttt{B}}) = img(\texttt{spread}) \subseteq dom(\texttt{aligna})
\end{array}$$

Pour obtenir l'équation (iii) de `Matvect3"` nous introduisons le champ intermédiaire `A` défini par `A` = `A0`, et nous renforçons les contraintes sur cette post-équation, qui peut se réécrire `A` = `A0` $\lhd$ `front` (propriété 6). Enfin en appliquant successivement les règles (11a) et (II) sur la dernière équation du précédent système, ($dom(v_{\texttt{P}}) \subseteq dom(\texttt{aligna})$) nous obtenons `Matvect3"`. Nous concluons donc que `Matvect4` est raffiné par `Matvect3"`.

4.5 Stratégie 3 : introduction d'une réduction

Nous avons jusqu'à présent raisonné sur la phase du produit des coefficients de l'algorithme du produit matrice-vecteur. Nous complétons maintenant l'algorithme par la phase de sommation de ces produits. Nous allons montrer comment passer d'une version utilisant des routages de voisins-à-voisins à une version profitant d'un opérateur de réduction. Introduire un tel opérateur est une stratégie courante de transformation de programmes, car il a été montré que la mise en oeuvre de ces opérateurs sur une machine parallèle a p processeurs, permet de résoudre des problèmes de taille n en un temps de l'ordre de $\frac{n}{p} + \log_2(p)$ [33].

La sommation des coefficients se trouvant sur chacune des lignes de `P` nécessite un routage des valeurs le long de chacune de ces lignes. Le programme suivant montre une façon possible d'effectuer la sommation des éléments de ligne :

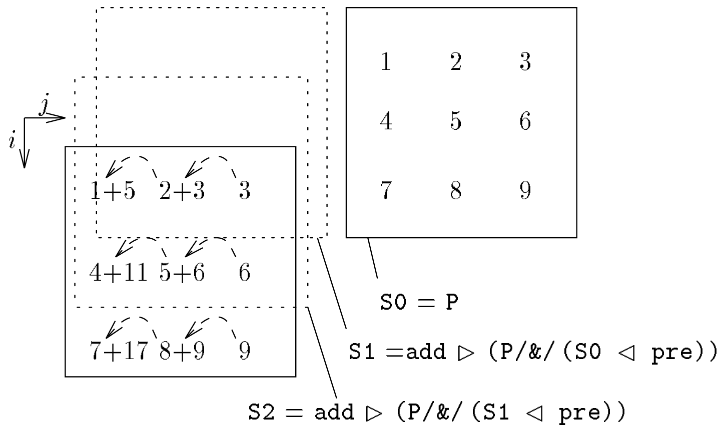

FIG. 4.4 - *Routage récursif des produits*

```
Matvect4 : (M,V) ↦ R
⎧ A0 = matrix::A0
⎪ B0 = matrix::B0
⎪ M = A0
⎪ V = align::B0
⎨ B = B0 ◁ spread
⎪ P = prod ▷ (A0/&/B)
⎪ S = add ▷ (P/&/S ◁ pre)
⎩ R = S ◁ col
```

$$\begin{array}{lcl}
\texttt{col} & = & \lambda\texttt{(i,j)}|\texttt{(j=0)} \\
\texttt{pre} & = & \lambda\texttt{(i,j)}|\texttt{(0}\leq\texttt{i<n \& 0}\leq\texttt{j<n-1).(i,j+1)} \\
\texttt{prod} & = & \lambda\texttt{(a;b).(a*b)} \\
\texttt{add} & = & \texttt{id} \# \lambda\texttt{(a;b).(a+b)}
\end{array}$$

La définition du champ S procède récursivement pour acheminer les produits : les paires de valeurs sont formées par superposition de la matrice P et d'elle-même déplacée par l'opération pre (cf. figure 4.4), puis les deux valeurs sont sommées pour n'en rendre qu'une.

Nous montrons à présent que la définition récursive du champ S peut s'abstraire en une définition non récursive, faisant apparaître une opération de réduction. Cette transformation est validée par l'application du théorème suivant :

Théorème 3 *Soit* f *une fonction définie par la récurrence* $\texttt{f = i} \# \texttt{(f} \circ \texttt{s}^{-1}\texttt{)}$, *où* i *est l'identité définie sur un sous-domaine, et* s *est bijective. L'équation* $\texttt{Y} = \texttt{f} \;; \triangleright \texttt{X}$ *se réécrit en la définition suivante, où le champ intermédiaire* T *est défini récursivement :*

$$\begin{array}{lcl}
\texttt{T} & = & \texttt{X /\&/ (T} \lhd \texttt{s)} \\
\texttt{Y} & = & \texttt{T} \lhd \texttt{i}
\end{array}$$

Preuve. En utilisant les règles de raffinement, nous pouvons écrire :

$$\begin{array}{ll}
 & \texttt{Y} = \texttt{f} \;; \triangleright \texttt{X} \\
\xrightarrow{def.} & \texttt{Y} = \texttt{(i} \# \texttt{f} \circ \texttt{s}^{-1}\texttt{)} \;; \triangleright \texttt{X} \\
\xrightarrow{(10)} & \texttt{Y} = \texttt{i} \;; \triangleright \texttt{X /\&/ (f} \circ \texttt{s}^{-1}\texttt{)} \;; \triangleright \texttt{X} \\
\xrightarrow{(6)} & \texttt{Y} = \texttt{i} \;; \triangleright \texttt{X /\&/ f} \;; \triangleright \texttt{(s}^{-1} \;; \triangleright \texttt{X)} \\
\xrightarrow{(prop1)} & \texttt{Y} = \texttt{X} \lhd \texttt{i /\&/ f} \;; \triangleright \texttt{(X} \lhd \texttt{s)}
\end{array}$$

En itérant le calcul :

$$\begin{array}{ll}
\longrightarrow & \texttt{Y} = \texttt{X} \lhd \texttt{i /\&/ (X} \lhd \texttt{s)} \lhd \texttt{i /\&/ f} \;; \triangleright \texttt{((X} \lhd \texttt{s)} \lhd \texttt{s)} \\
\xrightarrow{(8b)} & \texttt{Y} = \texttt{(X /\&/ X} \lhd \texttt{s /\&/ (X} \lhd \texttt{s)} \lhd \texttt{s /\&/ ...)} \lhd \texttt{i} \\
 & \text{car } dom(\texttt{i}) \subseteq dom(\sigma_{\texttt{X}}) \\
 & \text{et } img(\texttt{i}) = dom(\texttt{i}) \subseteq dom(v_{\texttt{X}}) \cap \ldots \cap dom(v_{\texttt{X}} \circ \texttt{s} \ldots \circ \texttt{s}) \\
\longrightarrow & \texttt{Y} = \texttt{T} \lhd \texttt{i} \quad \text{avec} \quad \texttt{T} = \texttt{X /\&/ (T} \lhd \texttt{s)}
\end{array}$$

□

Nous pouvons maintenant écrire le programme Pei suivant, qui abstrait le programme Matvect4 :

$$\texttt{Matvect5 : (M,V)} \mapsto \texttt{R}$$
$$\begin{cases} \texttt{A0} = \texttt{matrix::A0} \\ \texttt{B0} = \texttt{matrix::B0} \\ \texttt{M} = \texttt{A0} \\ \texttt{V} = \texttt{align::B0} \\ \texttt{B} = \texttt{B0} \lhd \texttt{spread} \\ \texttt{P} = \texttt{prod} \rhd \texttt{(A0/\&/B)} \\ \texttt{S} = \texttt{sum} \rhd \texttt{(red ;}\rhd \texttt{P)} \\ \texttt{R} = \texttt{S} \lhd \texttt{col} \end{cases}$$
$$\texttt{red} = \lambda\texttt{(i,j)|(0}\leq\texttt{i,j<n).(i,0)}$$
$$\texttt{sum} = \texttt{id} \ \# \ \lambda\texttt{(a;b).(a+sum(b))}$$

Preuve. Nous pouvons réécrire la fonction `red` et obtenir une définition semblable à celle de `f` dans le théorème précédent :

$$\begin{aligned} \texttt{red} &= \lambda\texttt{(i,j)|(0}\leq\texttt{i,j<n).(i,0)} \\ &= \lambda\texttt{(i,j)|(0}\leq\texttt{i<n\&j=0).(i,0)} \ \# \\ &\quad\ \lambda\texttt{(i,j)|(0}\leq\texttt{i<n\&0<j<n).(i,0)} \\ &= \texttt{col} \ \# \ \lambda\texttt{(i,j)|(0}\leq\texttt{i<n\& 0}\leq\texttt{j<n).(i,0)} \circ \\ &\quad\ \lambda\texttt{(i,j)|(0}\leq\texttt{i<n\&0<j<n).(i,j-1)} \\ &= \texttt{col} \ \# \ \texttt{red} \circ (\lambda\texttt{(i,j)|(0}\leq\texttt{i,j<n-1).(i,j+1)})^{-1} \\ &= \texttt{col} \ \# \ \texttt{red} \circ \texttt{pre}^{-1} \end{aligned}$$

Posons $\texttt{Y} = \texttt{red ;}\rhd \texttt{P}$. On déduit du thérorème précédent, un raffinement de la définition de `Y` sous la forme :

$$\begin{cases} \texttt{T} = \texttt{P/\&/(T} \lhd \texttt{pre)} \\ \texttt{Y} = \texttt{T} \lhd \texttt{col} \end{cases}$$

L'équation $\texttt{S} = \texttt{sum} \rhd \texttt{Y}$ de `Matvect5` devient donc $\texttt{S} = \texttt{sum} \rhd \texttt{(T} \lhd \texttt{col)}$. Cette définition de `S` peut se réécrire d'après la règle (2b), en $\texttt{S} = \texttt{(sum} \rhd \texttt{T)} \lhd \texttt{col}$.

Il nous reste à utiliser le fait que la somme `sum` d'une séquence de longueur quelconque peut se réécrire en fonction de la somme binaire `add`, de la façon suivante :

$$\texttt{sum = id} \ \# \ \lambda\texttt{(a;b).(add(a;sum(b)))}$$

Utilisons maintenant la proposition suivante :

Proposition 5 Soit une fonction récursive $\texttt{f = id} \ \# \ \lambda\texttt{(a;b).(a} \otimes \texttt{f(b))}$, où $\otimes$ est un opérateur binaire associatif. Alors :

$$\texttt{f} \rhd \texttt{(X1 /\&/X2)} = \texttt{f} \rhd \texttt{(f} \rhd \texttt{X1 /\&/f} \rhd \texttt{X2)}$$

Preuve. D'une part, nous avons $\mathtt{f} \circ \mathtt{f} = \mathtt{f}$ car pour une séquence x de longueur quelconque, $\mathtt{f}(x)$ rend une séquence à un élément, et l'image par `f` d'une séquence d'un élément est cet élément.
D'autre part, soient les champs $\mathtt{X1} = (v_1 : \sigma_1)$, $\mathtt{X2} = (v_2 : \sigma_2)$ et les points $z_1 \in dom(v_1)$ et $z_2 \in dom(v_2)$. Supposons que $v_1(z_1)$ est une séquence de longueur quelconque : $v_1(z_1) = \{a_1; \ldots; a_n\}$. De même $v_2(z_2) = \{b_1; \ldots; b_m\}$.

Les sémantiques respectives des deux expressions sont alors :

Pour $\mathtt{f} \rhd \mathtt{(X1\ /\&/\ X2)}$:

- sur le domaine $dom(v_1) \cap dom(v_2)$: $\begin{aligned}[t] &(\mathtt{f} \circ (v_1; v_2))(z_1; z_2) \\ &= \mathtt{f}(v_1(z_1); v_2(z_2)) \\ &= a_1 \otimes \ldots \otimes a_n \otimes b_1 \otimes \ldots \otimes b_m \end{aligned}$
- sur le domaine $dom(v_1) \backslash dom(v_2)$: $\mathtt{f}(v_1(z_1)) = a_1 \otimes \ldots \otimes a_n$
- sur le domaine $dom(v_2) \backslash dom(v_1)$: $\mathtt{f}(v_2(z_2)) = b_1 \otimes \ldots \otimes b_n$

Pour $\mathtt{f} \rhd \mathtt{(f} \rhd \mathtt{X1\ /\&/\ f} \rhd \mathtt{X2)}$:

- sur le domaine $dom(v_1) \cap dom(v_2)$: $\begin{aligned}[t] &(\mathtt{f} \circ ((\mathtt{f} \circ v_1);(\mathtt{f} \circ v_2)))(z_1; z_2) \\ &= (\mathtt{f} \circ v_1(z_1)) \otimes (\mathtt{f}(\mathtt{f} \circ v_2(z_2))) \\ &= (\mathtt{f} \circ v_1(z_1)) \otimes (\mathtt{f} \circ v_2(z_2)) \\ &= (a_1 \otimes \ldots \otimes a_n) \otimes (b_1 \otimes \ldots \otimes b_m) \end{aligned}$
- sur le domaine $dom(v_1) \backslash dom(v_2)$: $\mathtt{f}(\mathtt{f}(v_1(z_1))) = a_1 \otimes \ldots \otimes a_n$
- sur le domaine $dom(v_2) \backslash dom(v_1)$: $\mathtt{f}(\mathtt{f}(v_2(z_2))) = b_1 \otimes \ldots \otimes b_n$

Les deux expressions de champs de données sont donc égales si l'opérateur $\otimes$ est associatif.
□

La proposition 5 nous donne : $\begin{aligned}[t] \mathtt{sum} \rhd \mathtt{T} &= \mathtt{sum} \rhd \mathtt{(P\ /\&/\ T} \lhd \mathtt{pre)} \\ &= \mathtt{sum} \rhd \mathtt{(sum} \rhd \mathtt{P /\&/\ sum} \rhd \mathtt{(T} \lhd \mathtt{pre))} \end{aligned}$

Puisque le champ de données `P` n'est formé que de séquences d'un élément, on peut déduire : $\mathtt{sum} \rhd \mathtt{P} = \mathtt{P}$. De même, puisque $(\mathtt{P\ /\&/\ sum} \rhd \mathtt{(T} \lhd \mathtt{pre)})$ est formé de séquences d'au plus deux éléments, on a :

$$\mathtt{sum} \rhd \mathtt{T} \;=\; \mathtt{add} \rhd \mathtt{(P/\&/(sum} \rhd \mathtt{T)} \lhd \mathtt{pre)}$$

En renommant $\mathtt{sum} \rhd \mathtt{T}$ en `S'`, nous avons $\mathtt{S'} = \mathtt{add} \rhd \mathtt{(P\ /\&/\ S'} \lhd \mathtt{pre)}$

De plus, comme $\texttt{col} \circ \texttt{col} = \texttt{col}$, nous avons :

$$\begin{array}{rl} & \texttt{R} = \texttt{((sum} \rhd \texttt{T)} \lhd \texttt{col)} \lhd \texttt{col} \\ \longrightarrow & \texttt{R} = \texttt{(sum} \rhd \texttt{T)} \lhd \texttt{col} \\ \xrightarrow{def.} & \texttt{R} = \texttt{S'} \lhd \texttt{col} \end{array}$$

Au final, nous avons $\left\{ \begin{array}{l} \texttt{S'} = \texttt{add} \rhd \texttt{(P /\&/ S'} \lhd \texttt{pre)} \\ \texttt{R} = \texttt{S'} \lhd \texttt{col} \end{array} \right.$

Ces deux dernières équations sont présentes dans `Matvect4` et nous avons montré qu'elles sont un raffinement de l'équation définissant `R` de `Matvect5`. Les deux programmes ne différant que par ces équations, nous concluons que `Matvect5` abstrait `Matvect4`.
□

Note. La démonstration eût été plus simple si nous avions également utilisé la fonction `sum` dans les programmes précédents, fonction qui aurait été appliquée sur la séquence finale obtenue par superpositions successives. Cependant, nous aborderons dans le prochain chapitre la traduction d'équations PEI en code data-parallèle, et des opérations comme `sum` ne sont pas directement traduisibles dans ces langages en raison de l'absence du type liste.

Finalement, nous avons montré par abstractions successives, que le programme `Matvect1` est une version très particularisée, par son schéma de communication explicite, du programme abstrait `Matvect5`, proche de l'énoncé même du problème du produit matrice-vecteur.

4.6 Stratégie 4 : modification des alignements

4.6.1 Algorithme initial

Nous savons que le placement des données revêt une importance primordiale dans le domaine du parallélisme, étant donné le coût actuel élevé des communications. Nous montrons dans cette section une stratégie de transformation basée sur l'équivalence faible permettant de modifier le placement initial des données. L'exemple choisi est dérivé du produit matrice-vecteur : il s'agit en fait d'une variante de l'opération *Gaxpy* [23], qui est simplement un calcul de la forme : $c = Ax + y$, $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$. Nous nous intéressons ici au cas particulier $y = x$, cette opération étant fréquemment utilisée de manière itérative pour calculer $A(\ldots(A(Ax + x) + x)\ldots) + x$, par exemple dans la méthode du gradient conjugué.

La première version reprend le placement des données de `Matvect5`. Ce placement est parfois obligatoire quand on doit tenir compte de certaines restrictions liées à la machine où à un langage (par exemple, les diffusions doivent se faire selon les axes canoniques de la grille de processeurs virtuels sur les *CM-200* and *CM-5* de *Thinking Machines* [47]). Si cette contrainte disparaît, il existe alors un algorithme dont le placement des données diminue le nombre de communications nécessaires, et nous montrons dans cette section comment passer du premier algorithme au second.

Les placements des différents champs de données de l'algorithme initial sont représentés sur la figure 4.5(a). Comme dans `Matvect5`, le vecteur x (représenté par le champ `X`) est aligné sur la première ligne de la matrice (`B0`), puis recopié sur toutes les lignes de la matrice pour former `B`. Les produits (`P`) sont ensuite sommés le long des lignes pour obtenir Ax. Les sommes arbitrairement placées sur la dernière colonne forment `C0`. L'addition de Ax avec x nécessite la transposition de `C0`, donnant le champ `C1`. Finalement le résultat $Ax + x$ est placé sur la première ligne de la matrice (`C`). Cet algorithme s'écrit en PEI :

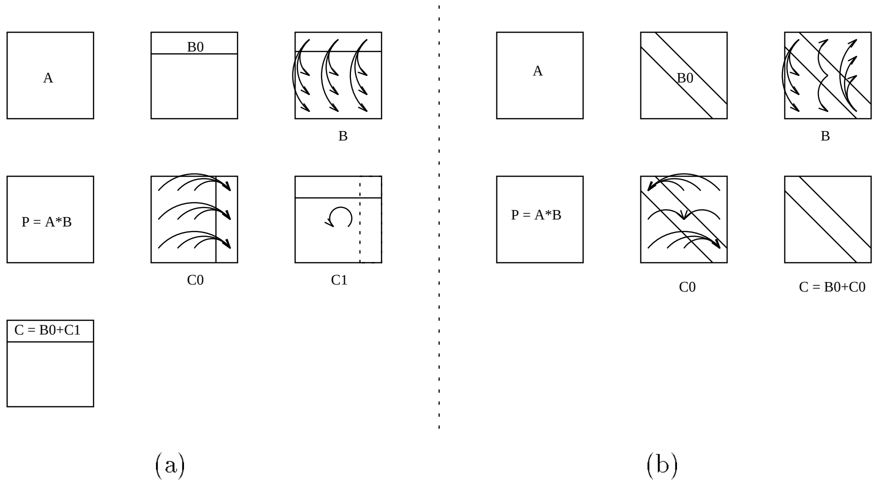

FIG. 4.5 - *(a) Algorithme initial (b) Algorithme amélioré*

$$
\texttt{Gaxpx1 : (A,X)} \mapsto \texttt{C}
\left\{
\begin{array}{l}
\texttt{A} = \texttt{matrix::A} \\
\texttt{X} = \texttt{align::B0} \\
\texttt{B} = \texttt{B0} \lhd \texttt{spread} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)} \\
\texttt{C0} = \texttt{sum} \rhd \texttt{(reduce ;}\rhd\texttt{ P)} \\
\texttt{C1} = \texttt{C0} \lhd \texttt{col2row} \\
\texttt{C} = \texttt{add} \rhd \texttt{(B0 /\&/ C1)}
\end{array}
\right.
$$

$$
\begin{array}{lcl}
\texttt{matrix} & = & \lambda\texttt{(i,j)}|\texttt{(0}\leq\texttt{i,j<n).(i,j)} \\
\texttt{align} & = & \lambda\texttt{(i,j)}|\texttt{(i=0\&0}\leq\texttt{j<n).(j)} \\
\texttt{spread} & = & \lambda\texttt{(i,j)}|\texttt{(0}\leq\texttt{i,j<n).(0,j)} \\
\texttt{reduce} & = & \lambda\texttt{(i,j)}|\texttt{(0}\leq\texttt{i,j<n).(i,n-1)} \\
\texttt{col2row} & = & \lambda\texttt{(i,j)}|\texttt{(i=0\&0}\leq\texttt{j<n).(j,n-1)} \\
\texttt{add} & = & \lambda\texttt{(a;b).(a+b)} \\
\texttt{prod} & = & \lambda\texttt{(a;b).(a*b)} \\
\texttt{sum} & = & \texttt{id} \,\#\, \lambda\texttt{(a;b).(a+sum b)}
\end{array}
$$

4.6.2 Algorithme avec placement amélioré

Si on s'autorise la diffusion des valeurs diagonales, nous pouvons alors concevoir une version plus efficace, comme esquissé sur la figure 4.5(b). En effet, les alignements de x et Ax sur la diagonale font que la phase de transposition devient inutile. Le gain engendré est important dès lors que l'opération est répétée de nombreuses fois.

Ré-alignments du vecteur

L'idée de la transformation repose sur le ré-alignement de `B0` et `C0`, que nous allons placer sur la diagonale, comme le suggère la figure 4.5. Cette modification du placement du vecteur va ensuite guider la reste de la transformation.

Nous changeons d'abord la définition de `B0` fonction de `X` en appliquant un changement de base (égal à son inverse), qui échange les valeurs placées sur la diagonale et celles placées sur la première ligne :

$$
\begin{array}{ll}
\texttt{row2diag} = & \lambda\texttt{(i,j)}|\texttt{(i=0\&0}\leq\texttt{j<n).(j,j)} \;\# \\
& \lambda\texttt{(i,j)}|\texttt{(i}\neq\texttt{0\&i=j\&0}\leq\texttt{j<n).(0,j)} \;\# \\
& \lambda\texttt{(i,j)}|\texttt{(i}\neq\texttt{0\&i}\neq\texttt{j\&0}\leq\texttt{i,j<n).(i,j)}
\end{array}
$$

Des définitions 10 et 9, nous obtenons un programme raffiné en remplaçant toutes les occurrences de `B0` par `(row2diag::B0)`:

$$\texttt{Gaxpx1'} : (\texttt{A},\texttt{X}) \mapsto \texttt{C}$$
$$\begin{cases} \texttt{A} = \texttt{matrix} :: \texttt{A} & \\ \texttt{X} = \texttt{align} :: (\texttt{row2diag} :: \texttt{B0}) & (i) \\ \texttt{B} = (\texttt{row2diag} :: \texttt{B0}) \lhd \texttt{spread} & (ii) \\ \texttt{P} = \texttt{prod} \rhd (\texttt{A} \ /\&/\ \texttt{B}) & \\ \texttt{C0} = \texttt{sum} \rhd (\texttt{reduce} \ ;\!\rhd\ \texttt{P}) & \\ \texttt{C1} = \texttt{C0} \lhd \texttt{col2row} & \\ \texttt{C} = \texttt{add} \rhd (\texttt{row2diag} :: \texttt{B0} \ /\&/\ \texttt{C1}) & \end{cases}$$

Le nouvel alignement de B0 par rapport à X s'obtient en appliquant les règles :

$$\begin{array}{rl} (i) \stackrel{(7b)}{\longrightarrow} & \texttt{X} = \texttt{align} \circ \texttt{row2diag} :: \texttt{B0} \\ \longrightarrow & \texttt{X} = \texttt{align2} :: \texttt{B0} \\ & \text{avec } \texttt{align2} = \texttt{align} \circ \texttt{row2diag} = \lambda\texttt{(i,j)}|\texttt{(i=j}\,\&\,\texttt{0}\leq\texttt{j<n).(j)} \end{array}$$

Intuitivement, on constate que le champ B a les mêmes valeurs dans les deux énoncés, mais que ces valeurs ont des origines différentes ; ceci est un cas typique d'équivalence faible. De ce fait, nous remplaçons le membre droit de l'équation (ii) par un champ faiblement équivalent selon les définitions 9 et 10 :

$$\texttt{B} = (\texttt{row2diag} :: ((\texttt{row2diag} :: \texttt{B0}) \ \lhd\ \texttt{spread})) \lhd \texttt{row2diag}$$

Nous simplifions maintenant cette expression pour obtenir une nouvelle expression du routage appliqué sur B0. Dans ce qui suit, i dénote l'identité dans $\mathbb{Z}^2$:

$$\begin{array}{rl} & \texttt{B} = (\texttt{row2diag} :: ((\texttt{row2diag} :: \texttt{B0}) \ \lhd\ \texttt{spread})) \lhd \texttt{row2diag} \\ \stackrel{(3a)}{\longrightarrow} & \texttt{B} = (\texttt{row2diag} :: (\texttt{row2diag} :: (\texttt{B0} \ \lhd\ \texttt{spread2}))) \lhd \texttt{row2diag} \\ & \text{car } dom(\texttt{spread}) \cup img(\texttt{spread}) \subseteq img(\texttt{row2diag}) \\ & \text{et } dom(v_{\texttt{B0}}) \subseteq dom(\texttt{row2diag}) \text{ de } (i), \\ & \text{et avec } \texttt{spread2} = \texttt{row2diag} \circ \texttt{spread} \circ \texttt{row2diag} = \lambda\texttt{(i,j)}|\texttt{(0}\leq\texttt{i,j<n).(j,j)} \\ \stackrel{(7b)}{\longrightarrow} & \texttt{B} = (\texttt{i}|_{dom(\texttt{row2diag})} :: (\texttt{B0} \ \lhd\ \texttt{spread2})) \lhd \texttt{row2diag} \\ \longrightarrow & \texttt{B} = (\texttt{B0} \ \lhd\ \texttt{spread2}) \lhd \texttt{row2diag} \\ & \text{simplification, car } dom(\texttt{spread2}) \subseteq dom(\texttt{row2diag}) \end{array}$$

$\overset{(5b)}{\longrightarrow}$ $\mathtt{B} = \mathtt{B0} \lhd \mathtt{spread2} \circ \mathtt{row2diag}$
car $img(\mathtt{row2diag}) \subseteq dom(\mathtt{spread2}) \subseteq dom(\sigma_{\mathtt{B0}})$

$\longrightarrow$ $\mathtt{B} = \mathtt{B0} \lhd \mathtt{spread2}$
définition de spread2, $\mathtt{spread2} = \mathtt{spread2} \circ \mathtt{row2diag}$

Les précédents calculs de raffinement nous amènent à :

$$
\mathtt{Gaxpx2 : (A,X)} \mapsto \mathtt{C}
\left\{
\begin{array}{l}
\mathtt{A} = \mathtt{matrix :: A} \\
\mathtt{X} = \mathtt{align2 :: B0} \\
\mathtt{B} = \mathtt{B0} \lhd \mathtt{spread2} \\
\mathtt{P} = \mathtt{prod} \rhd \mathtt{(A\ /\&/\ B)} \\
\mathtt{C0} = \mathtt{sum} \rhd \mathtt{(reduce}\ ; \rhd \mathtt{P)} \\
\mathtt{C1} = \mathtt{C0} \lhd \mathtt{col2row} \\
\mathtt{C} = \mathtt{add} \rhd \mathtt{(row2diag :: B0\ /\&/\ C1)}
\end{array}
\right.
$$

De la même manière, nous modifions l'alignement de Ax (C0) en utilisant le changement de base :

$$
\begin{array}{rl}
\mathtt{col2diag} = & \lambda\mathtt{(i,j)}|\mathtt{(j{=}n\text{-}1\ \&\ 0{\leq}i{<}n).(i,i)}\ \# \\
& \lambda\mathtt{(i,j)}|\mathtt{(j{\neq}n\text{-}1\ \&\ i{=}j\ \&\ 0{\leq}i{<}n).(i,n\text{-}1)}\ \# \\
& \lambda\mathtt{(i,j)}|\mathtt{(j{\neq}n\text{-}1\ \&\ i{\neq}j\ \&\ 0{\leq}i,j{<}n).(i,j)}
\end{array}
$$

Toujours en utilisant les définitions 9 et 10, nous pouvons raffiner Gaxpx2 en :

$$
\mathtt{Gaxpx2' : (A,X)} \mapsto \mathtt{C}
\left\{
\begin{array}{ll}
\mathtt{A} = \mathtt{matrix :: A} & \\
\mathtt{X} = \mathtt{align2 :: B0} & \\
\mathtt{B} = \mathtt{B0} \lhd \mathtt{spread2} & \\
\mathtt{P} = \mathtt{prod} \rhd \mathtt{(A\ /\&/\ B)} & \\
\mathtt{col2diag :: C0} = \mathtt{sum} \rhd \mathtt{(reduce}\ ; \rhd \mathtt{P)} & (iii) \\
\mathtt{C1} = \mathtt{(col2diag :: C0)} \lhd \mathtt{col2row} & (iv) \\
\mathtt{C} = \mathtt{add} \rhd \mathtt{(row2diag :: B0\ /\&/\ C1)} &
\end{array}
\right.
$$

Comme le montre la figure 4.5(b), les valeurs de P sont rassemblées sur la diagonale pour former C0. Les champs P et $\mathtt{(col2diag :: P)} \lhd \mathtt{col2diag}$ sont faiblement équivalents. Nous obtenons une nouvelle réduction en appliquant les étapes suivantes de raffinement à cette expression :

$$
\begin{array}{ll}
 & \texttt{col2diag::C0} = \texttt{sum} \rhd (\texttt{reduce} \ ;\rhd \ ((\texttt{col2diag::P}) \lhd \texttt{col2diag})) \\
\xrightarrow{(3a)} & \texttt{col2diag::C0} = \texttt{sum} \rhd (\texttt{reduce} \ ;\rhd \ (\texttt{col2diag::(P} \lhd \texttt{col2diag)})) \\
 & \text{car } dom(v_{\texttt{P}}) \subseteq dom(\texttt{cold2diag}) \\
\xrightarrow{(3c)} & \texttt{col2diag::C0} = \texttt{sum} \rhd (\texttt{col2diag::(reduce2} \ ;\rhd \ (\texttt{P} \lhd \texttt{col2diag)})) \\
 & \text{renommage : } \texttt{reduce2} = \texttt{col2diag} \circ \texttt{reduce} \circ \texttt{col2diag} \\
 & \qquad\qquad\qquad\qquad = \lambda\texttt{(i,j)}|(0 \le \texttt{i} < \texttt{n} \,\&\, 0 \le \texttt{j} < \texttt{n}).\texttt{(i,i)} \\
\xrightarrow{(1a)} & \texttt{col2diag::C0} = \texttt{col2diag::(sum} \rhd (\texttt{reduce2} \ ;\rhd \ (\texttt{P} \lhd \texttt{col2diag)})) \\
\xrightarrow{(II)} & \texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \ ;\rhd \ (\texttt{P} \lhd \texttt{col2diag})) \\
 & \text{car } dom(v_{\texttt{C0}}) \subseteq dom(\texttt{col2diag}) \text{ de l'équation } (iv)
\end{array}
$$

Les réécritures suivantes nous permettent de supprimer le routage `col2diag` de la dernière expression :

$$
\begin{array}{ll}
\longrightarrow & \texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \circ \texttt{col2diag} \ ;\rhd \ (\texttt{P} \lhd \texttt{col2diag})) \\
 & \text{car } \texttt{reduce2 = reduce2} \circ \texttt{col2diag} \\
\xrightarrow{(6,prop1)} & \texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \ ;\rhd \ ((\texttt{P} \lhd \texttt{col2diag}) \lhd \texttt{col2diag})) \\
 & \text{car } \texttt{col2diag} \text{ est égale à son inverse} \\
\xrightarrow{(5b)} & \texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \ ;\rhd \ (\texttt{P} \lhd \texttt{i}|_{dom(\texttt{col2diag})})) \\
 & \text{car } dom(\texttt{col2diag}) \subseteq dom(\sigma_{\texttt{P}}) \\
\longrightarrow & \texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \ ;\rhd \ \texttt{P}) \\
 & \text{simplification, car } \begin{array}{l} img(\texttt{col2diag}) \subseteq dom(v_{\texttt{P}}) \\ dom(\texttt{col2diag}) \subseteq dom(\sigma_{\texttt{P}}) \end{array}
\end{array}
$$

Nous avons donc :

$$
\texttt{Gaxpx3 : (A,X)} \mapsto \texttt{C}
$$
$$
\left\{
\begin{array}{l}
\texttt{A} = \texttt{matrix::A} \\
\texttt{X} = \texttt{align2::B0} \\
\texttt{B} = \texttt{B0} \lhd \texttt{spread2} \\
\texttt{P} = \texttt{prod} \rhd (\texttt{A /\&/ B}) \\
\texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \ ;\rhd \ \texttt{P}) \\
\texttt{C1} = (\texttt{col2diag::C0}) \lhd \texttt{col2row} \quad (vi) \\
\texttt{C} = \texttt{add} \rhd (\texttt{row2diag::B0/\&/C1})
\end{array}
\right.
$$

Suppression de la transposition

Intuitivement, nous observons que `C0` et `B0` sont tous les deux alignés sur la diagonale, et qu'ainsi la transposition définie par `col2row` est inutile. Comme `B0` est aligné par `row2diag`, nous devons introduire `row2diag` dans l'expression de `C0`. L'idée est de décomposer `col2diag` à l'aide de `row2diag`. Puisque `col2diag` ré-aligne `B0` sur la dernière colonne, nous devons compléter `row2diag` par un changement de base qui échange les valeurs placées sur la dernière colonne avec celles de la première ligne. Appelons `row2col` le changement de base :

$$\begin{aligned}\texttt{row2col} = \quad & \lambda\texttt{(i,j)}|\texttt{(j=n-1}\,\&\,\texttt{0}\leq\texttt{i<n).(0,i)}\ \#\\ & \lambda\texttt{(i,j)}|\texttt{(j}\neq\texttt{n-1}\,\&\,\texttt{i=0}\,\&\,\texttt{0}\leq\texttt{j<n).(j,n-1)}\ \#\\ & \lambda\texttt{(i,j)}|\texttt{(j}\neq\texttt{n-1}\,\&\,\texttt{i}\neq\texttt{0}\,\&\,\texttt{0}\leq\texttt{i,j<n).(i,j)}\end{aligned}$$

Comme `col2diag` et $\texttt{row2col} \circ \texttt{row2diag}$ sont égales sur le domaine de valeur de `C0`, les champs `col2diag :: C0` et $\texttt{row2col} \circ \texttt{row2diag :: C0}$ sont faiblement équivalents. De plus, $\texttt{row2col} \circ \texttt{row2diag :: C0}$ peut être réécrit `row2col :: (row2diag :: C0)` à l'aide de la règle (7a), et `col2row` est égal à `row2col` sur la première ligne (i.e. le domaine de valeurs de `row2diag :: C0`). Par conséquent, nous pouvons conclure que les champs de données `C1` et `row2diag :: C0` sont faiblement équivalents. Ceci nous permet d'obtenir le nouvel énoncé `Gaxpx3` en substituant la définition de `C1` par `row2diag :: C0` en utilisant l'implication modulo l'équivalence faible (définition 10):

$$\texttt{Gaxpx3' : (A,X)} \mapsto \texttt{C}$$
$$\left\{\begin{array}{l}\texttt{A} = \texttt{matrix :: A}\\ \texttt{X} = \texttt{align2 :: B0}\\ \texttt{B} = \texttt{B0} \lhd \texttt{spread2}\\ \texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)}\\ \texttt{C0} = \texttt{sum} \rhd \texttt{(reduce2 ;}\rhd\texttt{ P)}\\ \texttt{C} = \texttt{add} \rhd \texttt{(row2diag :: B0 /\&/ row2diag :: C0)}\end{array}\right.$$

L'énoncé final est obtenu en appliquant les règles (11a), (1a) sur la dernière équation, afin de factoriser le changement de base `row2diag` dans la partie droite. On peut alors simplifier l'équation en remplaçant l'unique occurrence de `C` par `row2diag :: C`, puis en appliquant la règle (II).

$$\begin{array}{ll}
\xrightarrow{(11a)} & \texttt{C} = \texttt{add} \rhd (\texttt{row2diag} :: (\texttt{B0 /\&/ C0})) \\
\xrightarrow{(1a)} & \texttt{C} = \texttt{row2diag} :: (\texttt{add} \rhd (\texttt{B0 /\&/ C0})) \\
\xrightarrow{prop2} & \texttt{row2diag} :: \texttt{C} = \texttt{row2diag} :: (\texttt{add} \rhd (\texttt{B0 /\&/ C0})) \\
 & \text{car } dom(\sigma_{\texttt{C}}) \subseteq img(\texttt{row2diag}) \\
\xrightarrow{(II)} & \texttt{C} = \texttt{add} \rhd (\texttt{B0 /\&/ C0}) \\
 & \text{car } dom(\sigma_{\texttt{C}}) \subseteq dom(\texttt{row2diag})
\end{array}$$

Nous obtenons finalement un programme raffiné, dont le vecteur est initialement aligné sur la diagonale :

$$\texttt{Gaxpx4 : (A,X)} \mapsto \texttt{C}$$

$$\left\{\begin{array}{l}
\texttt{A} = \texttt{matrix} :: \texttt{A} \\
\texttt{X} = \texttt{align2} :: \texttt{B0} \\
\texttt{B} = \texttt{B0} \lhd \texttt{spread2} \\
\texttt{P} = \texttt{prod} \rhd (\texttt{A /\&/ B}) \\
\texttt{C0} = \texttt{sum} \rhd (\texttt{reduce2} \; ;\!\rhd \; \texttt{P}) \\
\texttt{C} = \texttt{add} \rhd (\texttt{B0/\&/ C0})
\end{array}\right.$$

$$\begin{array}{lcl}
\texttt{align2} & = & \lambda(\texttt{i,j})|(\texttt{i=j} \,\&\, 0\le \texttt{j<n}).(\texttt{j}) \\
\texttt{spread2} & = & \lambda(\texttt{i,j})|(0\le \texttt{i,j<n}).(\texttt{j,j}) \\
\texttt{reduce2} & = & \lambda(\texttt{i,j})|(0\le \texttt{i,j<n}).(\texttt{i,i})
\end{array}$$

Chapitre 5

PEI et les langages data-parallèles

Nous avons montré dans [21] que le formalisme PEI et le modèle de programmation data-parallèle présentent de nombreuses similitudes. Les objets manipulés sont dans les deux cas des ensembles de valeurs : les champs de données en PEI et les variables parallèles dans les langages data-parallèles. La manipulation de ces objets passe par l'application d'opérations globales de trois types : la modification des valeurs (le calcul défini en PEI par l'opération fonctionnelle), le réarrangement des valeurs dans l'ensemble (les communications définies en PEI par l'opération géométrique) ou le placement d'un objet par rapport à un autre (l'opération de changement de base en PEI).

Il semble alors naturel de chercher à vérifier et préciser ces similitudes, et il nous a semblé que l'étude du passage d'un énoncé PEI à un programme data-parallèle permette de confirmer les liens existants, ou de mettre en avant les différences. Nous exposons donc dans ce chapitre les principes qui dirigent la traduction des programmes PEI vers les langages data-parallèles. Un premier prototype de traducteur à été réalisé pour générer du code *High Performance Fortran* (HPF) à partir de programmes PEI. Le choix de HPF comme langage cible a été motivé par la forte probabilité qu'il devienne une norme en matière de langages data-parallèles pour la programmation d'applications scientifiques.

Nous présentons rapidement le langage HPF, puis nous le comparons aux autres langages data-parallèles existants, en mettant en avant certains aspects sémantiques qui le distinguent de ces autres langages. Par la suite nous exposons les principes de la traduction de PEI vers HPF, notamment ceux implantés dans le prototype réalisé dans le cadre de ce travail. L'objectif premier de la réalisation de ce traducteur est de produire des codes data-parallèles exploitables par un compilateur commercial. Avec l'aide de cet outil, il est possible d'observer les conséquences des transformations PEI sur un code data-parallèle. Le traducteur autorise ainsi indirectement de faire de la transformation de programmes data-parallèles. Enfin, nous verrons qu'il est capable, moyennant des modifications mineures, de générer du code data-parallèle pour d'autres compilateurs existants.

5.1 Présentation d'HPF

Nous ne faisons ici qu'un exposé succinct et partiel du langage HPF, et nous encourageons le lecteur à consulter [32] pour une présentation exhaustive, ainsi que [52] pour avoir un aperçu des futures orientations de développement.

Le langage HPF est un langage *data-parallèle*. Les langages data-parallèles reprennent l'idée, issue du langage APL [29], de pouvoir manipuler directement non seulement des variables scalaires, mais aussi des collections de valeurs, habituellement des tableaux. Cette idée a été reprise dans Fortran90 à travers les expressions de tableaux. On peut ainsi exprimer l'addition de deux matrices `A` et `B` par la simple écriture `A+B`. La parallélisation de telles expressions semble alors naturelle : on peut imaginer que le compilateur distribue les valeurs des tableaux sur plusieurs processeurs et applique le même traitement en parallèle à chacune des valeurs. Les collections de valeurs peuvent alors être vues comme des *variables parallèles*. Le programmeur dispose d'un modèle de programmation proche de celui d'un langage séquentiel, en manipulant les objets parallèles comme des scalaires.

5.1.1 Opérations parallèles

Comme tous les langages data-parallèles, et notamment Fortran 90, un traitement parallèle en HPF est spécifié par l'application d'une opération *scalaire* sur un tableau. L'opération est alors effectuée sur tous (ou une partie) les éléments du tableau en parallèle. Si plusieurs tableaux sont impliqués dans l'opération, ils doivent être *conformes*, c'est-à-dire être de même dimension et avoir la même taille dans toutes les dimensions.

L'opération peut n'être appliquée que sur une section de tableau, notée comme en Fortran 90, pour chaque dimension, par un triplet `(`*borne inférieure* `:` *borne supérieure* `:` *pas* `)`, où *pas* est optionnel. Pour un tableau `T`, `T(`x`:`y`:`p`)` désigne les éléments de tableau `T(`i`)` tels que $x \leq i \leq y \wedge (i - x)\ mod\ p = 0$. Ainsi la section `A(2,3:10:3)` désigne les éléments de la ligne 2, colonnes 3, 6 et 9. Notons aussi que `T(:)` désigne tous les éléments du tableau unidimensionnel `T`.

L'instruction `forall`

En HPF, le programmeur spécifie l'application d'une instruction data-parallèle avec l'instruction `forall`, qui généralise l'affectation de sections de tableaux de Fortran 90.

`forall` (*triplet-spec*, ..., *triplet-spec*: *masque*) *affectation*

où *triplet-spec* est de la forme : *indice* = (*borne inférieure*: *borne supérieure*: *pas*) et *masque* est un prédicat (*masque* et *pas* sont optionnels).

La sémantique opérationnelle de l'instruction `forall` est la suivante :

1. énumération de l'ensemble des valeurs prises par chaque indice,
2. énumération de l'ensemble des processeurs virtuels valides, défini par le produit cartésien des ensembles des valeurs d'indice,
3. énumération de l'ensemble des processeurs virtuels actifs, défini par la restriction de l'ensemble des processeurs virtuels valides vérifiant *masque*,
4. évaluation de la partie droite de l'affectation par l'ensemble des processeurs virtuels actifs,
5. affectation du résultat à la partie gauche.

Par exemple, l'instruction

```
forall (i=2:3,j=2:4,j.GT.i) a(i-1,j)=a(i,j)
```

est exécutée de la manière suivante : l'ensemble des valeurs d'indices est évalué : $v_i = \{2{,}3\}$ et $v_j = \{2{,}3{,}4\}$. Puis l'ensemble des processeurs virtuels valides est évalué : $\{(2{,}2),(2{,}3),(2{,}4),(3{,}2),(3{,}3),(3{,}4)\}$ et enfin l'ensemble des processeurs virtuels actifs : $\{(2{,}3),(2{,}4),(3{,}4)\}$. Enfin les valeurs `a(2,3)`,`a(2,4)`,`a(3,4)` sont respectivement affectées à `a(1,3)`,`a(1,4)`,`a(2,4)`.

Le bloc `forall`

Il faut distinguer l'*instruction* `forall` du bloc `forall`. La première construction n'autorise qu'une affectation liée au `forall`. Dans le deuxième cas, plusieurs instructions peuvent former l'intérieur du `forall`, qui est terminé par `endforall`. Le bloc peut notamment contenir une imbrication de `forall`. Dans ce cas, l'ensemble des processeurs actifs est une restriction de l'ensemble des processeurs actifs défini par les `forall` englobants.

Cette construction nous sera utile par la suite car HPF impose une restriction importante : un indice apparaissant dans un $triplet - spec$ ne peut dépendre d'autres indices apparaissant dans un autre *triplet − spec* ([32], page 171). Il est donc nécessaire d'utiliser une imbrication de `forall` pour décrire des domaines qui ne dépendent pas que de bornes constantes. Par exemple, un domaine triangulaire sera décrit par :

```
forall (i=1:n)
    forall (j=1:i)
        a(i,j) = ....
    endforall
endforall
```

La boucle indépendante

Notons que l'on peut également spécifier un traitement parallèle en insérant un bloc d'instructions à l'intérieur d'une boucle `do ... enddo` précédée de la directive `$hpf! independent`. Le programmeur garantit alors au compilateur que les itérations sont indépendantes les unes des autres, permettant ainsi au compilateur de distribuer chacune des itérations sur des processeurs différents pour une exécution parallèle. Nous n'utiliserons pas ici ce type de boucle, qui se rapproche du modèle d'exécution MIMD, mais nous pouvons remarquer que l'instruction `forall` peut être remplacée par une imbrication parfaite de boucles `do` indépendantes. Par exemple, les sémantiques du `forall` et de l'imbrication de boucles indépendantes suivantes sont équivalentes.

```
forall (i=1:n,j=1:m)          $hpf! independent(i,j)
        A(i,j)=B(j,i)               do i=1,n
                                       do j=1,m
                                          A(i,j)=B(j,i)
                                       enddo
                                    enddo
```

5.1.2 Intrinsics

HPF enrichit le jeu des primitives de Fortran 90 propres au modèle data-parallèle, baptisées *intrinsics*. Ces primitives implantent notamment les réductions de tableaux. Elles permettent ainsi d'écrire des programmes plus portables tout en gagnant en efficacité, ces primitives pouvant être optimisées par le compilateur. Nous ne citerons ici que les primitives de la famille `XXX_SCATTER` que nous utiliserons par la suite pour traduire les réductions géométriques de PEI. Parmi les opérateurs de réduction disponibles (remplace `XXX`) on trouve `SUM`, `PRODUCT`, `MAXVAL`, *etc.* La primitive prend comme paramètres :

- un tableau a de dimension m qui contient les données à réduire,

- un tableau b de dimension n qui contient les valeurs initiales des réductions,
- un ensemble de tableaux d'indirection x_1 à x_n de même dimension que a.

Soit $\mathcal{A}$ l'ensemble des indices de a, et z un m-uplet $\in \mathbb{N}^m$. Un élément du tableau résultat calculé par la primitive de la famille `XXX_SCATTER` utilisant $\otimes$ comme opérateur de réduction est :

$$r(i_1, \ldots, i_n) = \bigotimes_{z \in \mathcal{A} | \forall k \in [1..n], x_k(z) = i_k} a(z) \qquad \otimes\, b(i_1, \ldots, i_n)$$

Appuyons nous sur un exemple de [27], page 90 pour expliquer de manière plus intuitive le fonctionnement de la primitive. Pour les tableaux :

$$\mathtt{a} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathtt{b} = \begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix}, \mathtt{x1} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 3 & 2 & 1 \end{bmatrix} \text{ et } \mathtt{x2} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

le résultat de `r = SUM_SCATTER(a,b,x1,x2)` est $\mathtt{r} = \begin{bmatrix} 14 & 6 & 0 \\ 8 & -5 & -6 \\ 0 & -8 & -9 \end{bmatrix}$

L'élément de tableau résultat $\mathtt{r}(1{,}2)$ est calculé comme suit : on cherche tous les couples $\mathtt{x1}(i,j)$,$\mathtt{x2}(i,j)$ tels que $\mathtt{x1}(i,j) = 1$ et $\mathtt{x2}(i,j) = 2$. A chaque fois que la condition est vérifiée, on somme la valeur $\mathtt{a}(i,j)$. Nous avons ici deux couples vérifiant les conditions, sélectionnant deux éléments dans `a` :

$\mathtt{x1}(1{,}2) = 1$	et	$\mathtt{x2}(1{,}2) = 2$	$\mathtt{a}(1{,}2) = 2$
$\mathtt{x1}(2{,}3) = 1$	et	$\mathtt{x2}(2{,}3) = 2$	$\mathtt{a}(2{,}3) = 6$

soit $\mathtt{r}(1{,}2) = 2 + 6 + \mathtt{b}(1{,}2) = 2 + 6 - 2 = 6$.

Des primitives de réduction moins générales implantent les opérations de type *parallel-prefix* du modèle data-parallèle. La réduction s'effectue selon un des axes canoniques du tableau. Ces primitives sont baptisées `XXX_SUFFIX` et `XXX_PREFIX`. L'exemple classique de la sommation de n nombres (voir page 15) s'écrit aisément avec cette primitive. Pour sommer le tableau `a` précédent selon les lignes, nous écrivons : `SUM_PREFIX(a, DIM = 2)` dont le résultat est le tableau :

$$\begin{bmatrix} 1 & 3 & 6 \\ 4 & 9 & 15 \\ 7 & 15 & 24 \end{bmatrix}$$

5.1.3 Alignement des données

HPF permet au programmeur d'insérer dans les programmes, des directives à l'intention du compilateur, concernant le placement des données sur les processeurs. Le placement influe de manière considérable sur le volume de communications inter-processeurs et donc sur les performances d'un programme. Le compilateur se décharge de cette tâche sur le programmeur car l'optimisation automatique du placement des données est un problème difficile [17, 43].

Le placement se fait en plusieurs étapes: les variables parallèles sont alignées entre elles, éventuellement à travers des tableaux fictifs appelés *templates*, à l'aide de la directive `align`. Les variables parallèles sont ensuite distribuées, avec la directive `distribute`, sur une machine abstraite constituée d'un tableau de processeurs virtuels (déclarée par la directive `processors`). Enfin, le tableau de processeurs virtuels est placé sur les processeurs physiques (le nombre de processeurs physiques est supposé inférieur ou égal au nombre de processeurs virtuels). Le programmeur ne peut influencer [1] que les deux premiers types de placements. Notons qu'à ces directives (statiques) correspondent des instructions (dynamiques) équivalentes qui sont `realign` et `redistribute`.

Alignement

L'alignement, à travers la directive `align`, précise les relations de placement des variables parallèles, les unes par rapport aux autres, ou par rapport à un template. Dans le cas où une variable parallèle est alignée sur une autre, le compilateur aligne implicitement la variable avec un template. Le programmeur a la garantie que deux éléments de variables parallèles alignés sur un même élément de template seront finalement placés sur un même processeur physique. La directive `align` offre de nombreuses possibilités d'alignements par le biais de fonctions affines de variables muettes représentant les dimensions, et du symbole `*`. Par exemple :

```
      real A(100,100), B(100,100), C(100,100), D(100), E(50),F(100,100)
!hpf$ template T(100,100)
!hpf$ align A(i,j) with T(i,j)          ! alignement a l'identite
!hpf$ align B(j,i) with T(i,j)          ! transposition
!hpf$ align C(i,j) with T(i+2,j)        ! translation
!hpf$ align D(i) with T(5,i)            ! positionnement
!hpf$ align E(i) with D(2*i-1)          ! eclatement regulier
!hpf$ align F(i,*) with T(i,1)          ! ecrasement de la deuxieme dimension
!hpf$ align E(i) with T(i,*)            ! duplication
```

1. Les directives de placements ne sont que des conseils donnés au compilateur.

Distribution

La distribution des éléments d'un template sur la machine virtuelle se fait, par dimension, par le biais des modes `block`, `block(`n`)`, `cyclic` ou `cyclic(`n`)`, ou n est le nombre d'éléments. Pour une machine `M` et un template `T` déclarés respectivement par :

```
!hpf$ processors M(n1)  ! n1 et n2 sont des parametres
!hpf$ template T(n2)    ! on suppose n2 > n1
```

Le mode `block` attribue des blocs de taille $t = \lceil \frac{\mathtt{n2}}{\mathtt{n1}} \rceil$ éléments contigus du template à chacun des processeurs virtuels : le premier processeur reçoit les éléments 1 à t, le deuxième $t + 1$ à $2t$, *etc.* Le mode `block(`n`)` permet de fixer la taille à $t = n$. Le mode `cyclic` attribue les éléments un à un aux processeurs, puis après avoir attribué `n1` éléments, recommence à partir du premier processeur jusqu'à avoir distribué tous les éléments. Par exemple :

```
!hpf$ processors M(2)
!hpf$ template T(5)
!hpf$ distribute T(block) onto M   ! M(1) a T(1,2,3) et M(2) a T(4,5)
!hpf$ distribute T(cyclic) onto M  ! M(1) a T(1,3,5) et M(2) a T(2,4)
```

5.2 Différence sémantique des indices

La plupart des langages data-parallèles ne permettent pas d'aligner les variables parallèles entres elles. Les variables parallèles sont directement placées sur des machines virtuelles, spécifiées par les mots-clés `shape` en C*[54] ou DPCE[53], ou `collection` en HyperC[45]. La déclaration d'une variable parallèle est réalisée en associant la variable avec une machine virtuelle. Par exemple en C* ou DPCE, les déclarations

```
shape[10][10]M;
int M:A,B;
```

ont pour effet d'allouer deux variables parallèles `A` et `B` sur la grille de processeurs virtuels `M`. Ces variables ont le même nombre de dimensions et la même taille que `M`, et sont alignées à l'identité avec `M`, c'est-à-dire qu'un élément $\mathtt{A}_{i,j}$ et $\mathtt{B}_{i,j}$ est alloué au processeur virtuel $\mathtt{M}_{i,j}$. Par conséquent, un calcul ne peut impliquer que des valeurs placées au mêmes indices de la machine virtuelle. Si ce n'est pas le cas, le programmeur doit explicitement donner l'instruction de déplacer ou recopier la valeur sur le processeur virtuel. Cette restriction (également présente dans Pei) possède l'avantage de fait prendre conscience au programmeur des communications induites par son algorithme. Le texte du programme contient donc des *instructions* de communications, c'est-à-dire des instructions qui "déforment" l'énoncé initial du problème, pour faire en sorte que les indices utilisés référencent les processeurs virtuels et non les occurrences initiales des variables.

HPF, en revanche, n'impose pas cette contrainte. Les indices des variables parallèles ne référencent pas les processeurs virtuels mais sont réellement les indices des tableaux. Par conséquent, le placement ne modifie pas la sémantique d'un programme. Pour un même texte de programme, mais dont les variables ne pas placées de la même manière, seul le temps d'exécution diffère. Ainsi, la figure 5.1 illustre cette différence sur une transposition de matrice, écrite en C* (à gauche) et le programme équivalent en HPF (à droite). En C*, les indices `i` et `j` représentent des numéros de processeurs virtuels, c'est-à-dire que les valeurs de `B` ne sont pas transposées ; en HPF au contraire, les indices sont les indices des matrices. Le programme HPF sans les directives de placement (lignes 1 à 3) produirait les mêmes résultats que le programme listé, mais avec une performance moindre si le compilateur ne place pas automatiquement les matrices comme indiqué.

```
shape mat[100][100];
float : mat i,j,A,B;

  i=pc_coord(0);
  j=pc_coord(1);
  [j][i] B = [i][j] A;
```

```
 real A(100,100), B(100,100)
!hpf$ processors M(100,100)
!hpf$ align A(i,j) with B(j,i)
!hpf$ distribute A(block) onto M

 forall(i=1:100,j=1:00) B(j,i) = A(i,j)
```

FIG. 5.1 - *Transposition de matrice en C* et en HPF*

5.3 Schéma d'ensemble du mécanisme de traduction

Un énoncé PEI quelconque n'est, de manière générale, pas directement traduisible en un code data-parallèle. Les caractéristiques de PEI empêchant cette traduction sont de deux sortes : l'aspect équationnel des énoncés et l'absence d'ordonnancement opérationnel pour évaluer la solution d'une équation. Par conséquent, certaines transformations *sémantiques*, utilisant les règles énoncées dans les chapitres 3 et 4, doivent être préalablement effectuées pour obtenir un énoncé raffiné sous une forme *traduisible*.

> **Equations** : Nous ne pouvons traduire que des équations assimilables à des affectations, c'est-à-dire celles dont au moins un des membres n'est composé que d'un identificateur de champ. Par exemple, nous ne savons pas traduire l'équation `f :: X` = `g :: Y`. Il est donc nécessaire de faire apparaître uniquement `X` ou `Y` dans l'un des membres, à l'aide des règles exposées en section 3.3. En l'occurrence, on peut faire

apparaître `X` uniquement en membre gauche si `f` est inversible, et en appliquant $\mathtt{f}^{-1}$ sur les deux membres (règles (I) et (7b) à condition que les domaines de référence de `X` et de `Y` soient inclus dans $dom(\mathtt{h})$), puis en calculant le nouveau changement de base $\mathtt{f}^{-1} \circ \mathtt{g}$ appliqué à `Y`, ce qui donne l'équation $\mathtt{X} = (\mathtt{f}^{-1} \circ \mathtt{g}) :: \mathtt{Y}$. Dans d'autres cas, la traduction est impossible : par exemple, aucune règle de raffinement n'existe actuellement pour réécrire l'équation $\mathtt{X} \lhd \mathtt{f} = \mathtt{Y} \lhd \mathtt{g}$ de manière à faire apparaître `X` ou `Y` seul sur l'un des membres, dans le cas où `f` et `g` sont des fonctions non inversibles. Par conséquent, l'énoncé contenant une telle équation ne pourra être traduit.

Cadencement des équations récurrentes : nous ne traduisons que les équations récurrentes pour lesquelles nous pouvons déterminer un cadencement des calculs dans le temps. Ce problème est largement abordé dans la littérature concernant la parallélisation automatique, et nous savons qu'il consiste à trouver des fonctions de cadencement et de placement. En PEI, cette transformation se traduit par un changement de base appliqué à l'énoncé. Un exemple simple est celui du processus de diffusion discrète de la figure 5.2.

```
do i = 1, n
      do j = 1, n
            y(i, j) = f(y(i − 1, j), y(i, j − 1))
      enddo
enddo
```

FIG. 5.2 - *Processus de diffusion discrète*

qui s'écrit en PEI, si `X` est le champ contenant les valeurs initiales :

$$\begin{array}{ll}\mathtt{Y} = \mathtt{f} \rhd (\mathtt{X}\ /\&/\ (\mathtt{Y} \lhd \mathtt{f1})\ /\&/\ (\mathtt{Y} \lhd \mathtt{f2})) & (1)\\ \mathtt{f1} = \lambda(\mathtt{i,j})|(1 \leq \mathtt{i} \leq \mathtt{n}\ \&\ 1 \leq \mathtt{j} \leq \mathtt{n}).(\mathtt{i-1,j}) & \\ \mathtt{f2} = \lambda(\mathtt{i,j})|(1 \leq \mathtt{i} \leq \mathtt{n}\ \&\ 1 \leq \mathtt{j} \leq \mathtt{n}).(\mathtt{i,j-1}) & \end{array}$$

Nous savons qu'il existe dans ce cas, une transformation affine temps $\times$ espace permettant de paralléliser la récurrence. En l'occurrence, cette transformation peut être :

$$\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

où l'indice t représente le temps et p les processeurs. En PEI, cette transformation est réalisée par l'application de changement de base `h` $= \lambda$`(t,p).(t+p,t)` sur l'équation (1). Après application des règles, nous obtenons :

$$\begin{array}{ll} \texttt{Y'} = \texttt{f} \triangleright \texttt{(X' /\&/ (Y'} \triangleleft \texttt{g1) /\&/ (Y'} \triangleleft \texttt{g2))} & (2) \\ \texttt{g1} = \lambda\texttt{(t,p)|(1}\leq\texttt{p}\leq\texttt{n\& 1}\leq\texttt{t-p}\leq\texttt{n).(t-1,p-1)} & \\ \texttt{g2} = \lambda\texttt{(t,p)|(1}\leq\texttt{p}\leq\texttt{n\& 1}\leq\texttt{t-p}\leq\texttt{n).(t-1,p)} & \end{array}$$

où $\texttt{g1} = \texttt{h} \circ \texttt{f1} \circ \texttt{h}^{-1}$ et $\texttt{g2} = \texttt{h} \circ \texttt{f2} \circ \texttt{h}^{-1}$. Une dimension définissant les fonctions géométriques `g1` et `g2` comme des translations, nous verrons que cette transformation nous permet de traduire la récurrence par une boucle séquentielle sur t et une boucle parallèle sur p.

Notons que dans cet exemple, une seule direction représente le temps, sous la forme d'une translation présente dans toutes fonctions géométriques. Notre traducteur actuel ne reconnaît que cette forme de cadencement linéaire, mais d'autres techniques de cadencement pourraient être incluses, par exemple la technique de cadencement multi-dimensionnel exposée dans [16]. De nouvelles formes d'équations pourraient alors être traduites : entre autres, l'équation (1) pourrait être traduite par deux boucles séquentielles, donnant le programme Fortran de la figure 5.2.

Après avoir transformé l'énoncé de manière à respecter ces principes, nous pouvons entamer un processus de transformation de nature *syntaxique*, que nous décrivons maintenant.

5.4 Le traducteur `pei2hpf`

La traduction nécessite que les énoncés PEI acceptés en entrée aient une forme bien précise, que nous appelons *forme traduisible*. Généralement, les énoncés transformés par le calcul de raffinement PEI pour obtenir des équations assimilables à des affectations, et des équations récurrentes ordonnancées, ne sont pas sous cette forme traduisible. Cependant, si elle existe, la mise sous forme traduisible d'un tel énoncé est un processus automatisable de transformation syntaxique : il est réalisé dans notre prototype par un pré-processeur baptisé `normhpf`. Dans la suite, nous donnons d'abord la forme traduisible choisie, ainsi que les raisons qui nous ont amené à ce choix, puis nous montrons les transformations syntaxiques opérées par `normhpf`.

5.4.1 Forme traduisible

Le traducteur accepte en entrée un sous-ensembles des programmes PEI, constitué des équations ayant la forme suivante :

Forme traduisible
1. Opération géométrique : $\mathtt{Y} = \mathtt{X} \lhd \mathtt{g}$ 2. Opération fonctionnelle : $\mathtt{Y} = \mathtt{f} \rhd (\mathtt{X}_1$ /&/ ... /&/ $\mathtt{X}_n)$ 3. Opération de changement de base : $\mathtt{Y} = \mathtt{h} :: \mathtt{X}$ 4. Propriété de champ de données : $\mathtt{Y} = \mathtt{h} :: \mathtt{Y}$ 5. Réduction géométrique : $\mathtt{Y} = \mathtt{fr} \rhd (\mathtt{gr}\ ; \rhd\ \mathtt{X})$ 6. Définition récursive : $\mathtt{Y} = \mathtt{fc} \rhd (\mathtt{X}$ /&/ $(\mathtt{Y} \lhd \mathtt{g}_1)$ /&/ ... /&/ $(\mathtt{Y} \lhd \mathtt{g}_n))$

Contraintes
– Dans 5. la définition de fr doit être de la forme récursive pré-définie $\mathtt{fr} = \mathtt{id} \# \lambda(\mathtt{a;b}).(\mathtt{a} \otimes \mathtt{fr(b)})$, où $\otimes$ est un opérateur associatif, commutatif avec élément neutre. Dans le prototype actuel, seuls + et * sont acceptés. – Dans 6. fc doit être de la forme $\mathtt{fc} = \mathtt{id}\ \# \lambda(\mathtt{a;b;\ldots;z}).(\mathtt{a} \odot_1 \mathtt{b} \odot_2 \ldots \odot_n \mathtt{z})$, où les $\odot_i$ sont des opérateurs quelconques sur des scalaires.

Nous dirons qu'un énoncé PEI est *traduisible* si toutes ses équations sont sous l'une des formes décrites dans le tableau *Forme traduisible* et si elles respectent les conditions énoncées dans le tableau *Contraintes*.

Plusieurs raisons nous ont amené à ne considérer que cet ensemble restreint d'expressions. Elles nous permettent d'associer à chacune d'entre elles, un ensemble d'instructions HPF, sans avoir besoin d'informations supplémentaires contextuelles (tirées du programme). Par ailleurs, la traduction d'expressions de champs de données plus compliquées n'est pas toujours possible. Examinons les problèmes liés à certaines expressions PEI :

Opération géométrique : On se limite à la traduction d'équations de la forme $\mathtt{Y} = \mathtt{X} \lhd \mathtt{g}$. Considérons en effet, un champ de données défini par $\mathtt{Y} = (\mathtt{X} \lhd \mathtt{g}) \lhd \mathtt{f}$. La traduction de cette expression supposerait de la raffiner en une expression de la forme $\mathtt{Y} = \mathtt{X} \lhd (\mathtt{g} \circ \mathtt{f})$, en vérifiant les conditions de la règle (5b) dans l'environnement PEI, et de calculer la composition de fonctions $\mathtt{g} \circ \mathtt{f}$ car HPF ne permet pas d'exprimer la composition de fonctions agissant sur des indices de tableau. On est donc conduit à exprimer ces deux opérations géométriques successives en décomposant l'expression initiale en deux équations $\mathtt{Y} = \mathtt{X}' \lhd \mathtt{f}$ et $\mathtt{X}' = \mathtt{X} \lhd \mathtt{g}$. C'est ce que fera la phase de normalisation présentée plus loin.

Opération fonctionnelle : Nous imposons que tous les champs correspondant aux arguments de la fonction utilisée par l'opération fonctionnelle soient présents dans l'équation. Considérons par exemple la fonction `f` $= \lambda$`(a;b).(a-(b*a))` et l'équation `Y` $=$ `f` $\rhd$ `(X1/&/ X2)`. Sous cette forme, il est très simple de substituer aux arguments `a` et `b` les champs `X1` et `X2` respectivement. Dans le cas contraire, par exemple `Y` $=$ `f` $\rhd$ `X`, il est nécessaire de consulter dans le reste du programme la définition de `X`.

Opération inverse géométrique : L'opération de réduction géométrique de Pei pose le problème du non-déterminisme des solutions obtenues. Un énoncé constitué de la seule équation `Y` $=$ `f` $;\rhd$ `X` n'est pas un programme Pei, si la fonction `f` n'est pas injective car le champ solution `Y` n'est pas unique (les séquences de valeurs ont un ordre quelconque). Or, le langage HPF ne permet ce type d'opération que si la communication est accompagnée d'un opérateur de réduction, à l'aide des *intrinsics* de la famille `XXX_SCATTER`. C'est pourquoi nous avons choisi de ne traduire que les réductions Pei de la forme 5.

5.4.2 Normalisation avant traduction

Comme nous l'avons dit en introduction, certains énoncés peuvent être transformés syntaxiquement en des énoncés traduisibles à l'aide d'un pré-processeur agissant comme un filtre[2] avant la traduction proprement dite. Ce pré-processeur nommé **`normhpf`** est entre autres chargé d'introduire les définitions de champs de données intermédiaires nécessaires, et plus généralement de transformer l'énoncé initial en énoncé traduisible par de simples ajustements syntaxiques. Par exemple, une équation non-récursive de la forme $\mathtt{Y} = e_1(e_2(\ldots e_n(\mathtt{X})\ldots))$, où e_i est une simple opération géométrique, fonctionnelle, ou un changement de base, est remplacée par l'ensemble d'équations :

$$\begin{cases} \mathtt{Y} = e_1(\mathtt{Y_1}) \\ \mathtt{Y_1} = e_2(\mathtt{Y_2}) \\ \ldots \\ \mathtt{Y_n} = e_n(\mathtt{X}) \end{cases}$$

L'énoncé Pei obtenu après ces transformations est équivalent à l'énoncé initial, comme nous l'avons vu en section 3.2. La figure 5.3 illustre l'ensemble des transformations que le pré-processeur peut effectuer pour rendre traduisible un énoncé :

2. au sens Unix du terme.

Sample : (A,B,C) $\mapsto$ Z	Sample : (A,B,C) $\mapsto$ Z
$\begin{cases} \texttt{W} = \texttt{X /\&/ Y} \\ \texttt{Z} = \texttt{fc1} \rhd \texttt{W} \\ \texttt{X} = \texttt{fb1::((A} \lhd \texttt{fg1)} \lhd \texttt{fg2)} \\ \texttt{Y} = \texttt{fc3} \rhd \texttt{(fc2} \rhd \texttt{(B /\&/ C) /\&/ Y} \lhd \texttt{fg3)} \end{cases}$	$\begin{cases} \texttt{Z} = \texttt{fc1} \rhd \texttt{(X /\&/ Y)} \\ \texttt{X} = \texttt{fb1::X'} \\ \texttt{X'} = \texttt{X''} \lhd \texttt{fg2} \\ \texttt{X''} = \texttt{A} \lhd \texttt{fg1} \\ \texttt{Y1} = \texttt{fc2} \rhd \texttt{(B /\&/ C)} \\ \texttt{Y} = \texttt{fc3} \rhd \texttt{(Y1 /\&/ Y} \lhd \texttt{fg3)} \end{cases}$

FIG. 5.3 - *Transformations syntaxiques réalisées par* `normhpf`

5.4.3 Ordonnancement avant traduction

Pour produire un code data-parallèle, nous devons également ordonner les équations du programme PEI sous forme traduisible. Nous utilisons pour cela l'ordre induit par les définitions de champs de données : nous disons qu'un champ `X` dépend causalement d'un champ `Y` si `X` la définition de `X` contient au moins une occurrence de `Y`. A partir des champs en entrée, nous pouvons construire un graphe dont les noeuds sont les champs de données du programme, et les arcs représentent la relation «dépend causalement de». L'énoncé PEI est traduisible si ce graphe est sans circuit. Dans ce cas, dans le programme impératif, tout champ `X` dépendant causalement d'un champ `Y` devra apparaître après `Y` dans l'ordre des instructions. Cette relation sur l'ensemble des champs d'un tel graphe étant d'ordre partiel, nous pouvons choisir pour deux champs non comparables un ordre d'apparition quelconque. De plus, pour pouvoir traduire les équations en affectations, le champ dépendant causalement d'une expression ne peut être qu'un identificateur[3].

Ainsi, le programme de droite de la figure 5.3 est ordonnancé comme le montre la figure 5.4. L'ordre d'apparition des équations dans le système sera celui des instructions du programme impératif.

Cette méthode d'ordonnancement est simple car elle tire partie de l'ordonnancement propre au modèle data-parallèle : comme nous l'avons déjà vu, le fil d'exécution des opérations parallèles est en effet séquentiel. Ici, les champs de données ou variables parallèles sont considérées dans leur entièreté. Plus précisément, cela signifie que l'ordonnancement effectué implique qu'une variable parallèle est complètement définie par une instruction, c'est-à-dire que toutes ses valeurs sont évaluées lors de cette instruction. Par opposition, nous verrons en section 5.10.2 qu'un ordonnancement issu des techniques de parallélisation permet d'évaluer certains sous-ensembles de valeurs d'une variable parallèle à des instants distincts.

3. Une exception est faite actuellement pour les alignements déduits d'un changement de base

$$
\texttt{Sample : (A,B,C)} \mapsto \texttt{Z}
$$
$$
\begin{cases}
\texttt{X''} = \texttt{A} \lhd \texttt{fg1} \\
\texttt{Y1} = \texttt{fc2} \rhd \texttt{(B /\&/ C)} \\
\texttt{Y} = \texttt{fc3} \rhd \texttt{(Y1 /\&/ Y} \lhd \texttt{fg3)} \\
\texttt{X'} = \texttt{X''} \lhd \texttt{fg2} \\
\texttt{X} = \texttt{fb1 :: X'} \\
\texttt{Z} = \texttt{fc1} \rhd \texttt{(X /\&/ Y)}
\end{cases}
$$

FIG. 5.4 - *Tri des équations selon l'ordre des dépendances causales*

Pratiquement dans le prototype, l'ordonnancement est réalisé par un autre filtre baptisé `ord`, appliqué après `normhpf`.

5.4.4 Synopsis du fonctionnement du traducteur

Lorsqu'elles aboutissent à une forme traduisible, des transformations précédentes résulte un programme PEI dont la structure est semblable à celle d'un programme parallèle. Le travail restant à réaliser se réduit maintenant à la traduction des équations une à une, et à l'inférence des dimensions des variables parallèles utilisées. Le mécanisme de la traduction peut être résumé de la manière suivante :

1. Analyse lexicale et syntaxique du programme PEI et construction d'un arbre de syntaxe abstraite (AST).

2. Parcours de l'AST pour stocker les informations présentes dans l'équation dans un format de données intermédiaire. On vérifie à ce moment que les informations retirées des équations pourront être exploitées pour la production de code.

3. Passe supplémentaire sur l'ensemble des équations pour inférer des informations globales concernant les domaines des champs de données. On complète alors les informations du format intermédiaire.

4. Passage des informations à un formateur de texte qui exploite les informations du format intermédiaire pour produire le code HPF.

5.4.5 Traduction HPF des équations PEI

Nous nous intéressons maintenant aux instructions HPF associées à chacune des équations traduisibles.

Opération géométrique

Soit l'équation de la forme 1.:

$$\mathtt{Y} = \mathtt{X} \lhd \mathtt{g}$$

Rappelons que g est de la forme $\lambda(z) \mid dom(\mathtt{g}).\mathtt{g}(z)$ et que le champ Y est $(v_{\mathtt{X}} \circ \mathtt{g} : \sigma_{\mathtt{X}})$. L'instruction `forall` d'HPF autorise une traduction directe de l'opération géométrique. Nous traduisons l'équation précédente par :

$$\mathtt{forall}\ (z \in dom(\mathtt{g}))\ \mathtt{Y}(z)\ \mathtt{=}\ \mathtt{X(g(}z\mathtt{))}$$

Si le domaine de g n'est pas précisé dans la définition de la fonction, le domaine utilisé est $dom(\sigma_{\mathtt{X}})$. De plus, la contrainte $img(\mathtt{g}) \subseteq dom(v_{\mathtt{X}})$ garantit que les indices $\mathtt{g}(z)$ sont valides pour le tableau X. Par exemple, la traduction de l'équation

$$\begin{array}{rl} & \mathtt{Y} = \mathtt{X} \lhd \mathtt{right} \\ \text{avec} & \mathtt{right} = \lambda\mathtt{(i)}|\mathtt{(1<i\leq 100).(i-1)} \end{array}$$

est `forall (i=2:100) Y(i) = X(i-1)`.

Opération fonctionnelle

Soit l'équation de la forme 2. :

$$\mathtt{Y} = \mathtt{f} \rhd (\mathtt{X}_1\ \mathtt{/\&/} \ldots \mathtt{/\&/}\ \mathtt{X}_n)$$

Cette équation est traduite par l'instruction HPF :

$$\mathtt{forall}\ (z \in dom(\sigma_{\mathtt{X}})\ ,\ z \in dom(\mathtt{f}))\ \mathtt{Y}(z)\ \mathtt{=}\ \mathtt{f(X}_1(z),\ldots,\mathtt{X}_n(z))$$

Notons que nous supposons ici $dom(\sigma_{\mathtt{X}}) = dom(\sigma_{\mathtt{X}_1}) = \ldots = dom(\sigma_{\mathtt{X}_n})$. Cette contrainte actuelle peut être levée dans une version future du traducteur.
La restriction de l'ensemble des processeurs virtuels valides à $dom(\mathtt{f})$ est représenté par la partie *masque* de l'instruction `forall`. Nous traduirons par exemple, l'équation :

$$\begin{array}{l} \mathtt{Y} = \mathtt{divide} \rhd \mathtt{(X1/\&/X2)} \\ \mathtt{divide} = \lambda\mathtt{(a;b)}|\mathtt{(b\neq 0).(a/b)} \end{array}$$

en $\mathtt{forall}\ (z \in dom(\sigma_{\mathtt{X2}})\mathtt{,X2(}z\mathtt{).NEQ.0)}\ \mathtt{Y}(z)\ \mathtt{=}\ \mathtt{X1(}z\mathtt{)/X2(}z\mathtt{)}$

Propriété de champ de données

L'équation $\mathtt{X} = \mathtt{h} :: \mathtt{X}$ exprime que l'application de la fonction identité sur le domaine des valeurs et référence de X ne modifie pas X. Dans ce type d'équation, la fonction utilisée dans le changement de base ne peut être autre que l'identité. En effet, de par la définition du changement de base, nous devons vérifier les égalités : $v_{\mathtt{X}} = v_{\mathtt{X}} \circ \mathtt{h}^{-1}$ et $\sigma_{\mathtt{X}} = \sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}$. Nous pouvons déduire de cette équation les dimensions de la variable X utilisée dans le programme HPF, car les valeurs du champ X sont placées dans $dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h})$. Cette forme d'équation n'est donc pas directement traduite sous forme d'instructions HPF, mais utilisée pour dimensionner le tableau utilisé pour la traduction du champ X. Cette information est utilisée conjointement avec les changements de base liant différents champs de données.

Opération de changement de base

Une équation de la forme $\mathtt{Y} = \mathtt{h} :: \mathtt{X}$ indique que toute valeur placée au point z dans le champ Y est aussi la valeur placée au point $\mathtt{h}(z)$ dans le champ X. On peut imaginer deux manières de traduire cette équation.

La première consiste à déterminer quel tableau est calculé le premier lors de l'exécution du programme (supposons que ce soit X), et à recopier les valeurs $\mathtt{X}(z)$ en $\mathtt{Y}(\mathtt{h}(z))$. Cette façon de procéder implique des communications lors de l'exécution, mais est l'unique possibilité de traduction dans des langages data-parallèles n'offrant pas la possibilité d'alignement.

L'autre choix de traduction considère que l'équation indique comment les tableaux sont placés les uns par rapport aux autres sur la machine virtuelle. Nous avons vu que HPF offre la possibilité de traduire cette notion avec la directive `align`. Ce choix est préférable dans la mesure où il n'induit pas de communication à l'exécution. Avec cette méthode nous choisissons d'aligner les tableaux X et Y relativement à un template commun T. Schématiquement, les alignements correspondants à l'équation précédente sont :

```
align X(z) with T(z)
align Y(z) with T(h(z))
```

Le template doit pouvoir contenir les différents tableaux du programme. Pour cela, ses bornes sont calculées de façon à définir l'enveloppe parallélépipèdique de $dom(\mathtt{h}) \cup img(\mathtt{h})$. En supposant que les autres équations du programme PEI nous apportent des informations sur les bornes des tableaux, et le type *type* des valeurs stockées, nous pouvons alors produire les déclarations des tableaux et du template.

Supposons qu'on infère du programme que les tableaux X et Y sont de taille $[1 \ldots n] \times$

$[1 \ldots n]$ alors les déclarations des tableaux et du template sont :

$$\mathit{type}\ \mathtt{X(n)},\ \mathit{type}\ \mathtt{Y(h(1):h(}n\mathtt{))}$$
$$\mathtt{template\ T(}dom(\mathtt{h}) \cup img(\mathtt{h})\mathtt{)}$$

Enfin, à l'exécution, il reste à recopier les valeurs du tableau déjà calculé dans le nouveau afin de respecter l'égalité des valeurs spécifiée par l'équation PEI (contrairement au premier mode de traduction, cette copie n'engendre pas de communication du fait des alignements réalisés). Illustrons simplement ce procédé avec l'exemple d'un décalage entre tableaux. Considérons les équations suivantes, où X est un champ en entrée :

$$\begin{array}{ll} & \mathtt{X} = \mathtt{matrix} :: \mathtt{X} \\ & \mathtt{Y} = \mathtt{transl} :: \mathtt{X} \\ \text{avec} & \mathtt{transl} = \lambda\mathtt{(i,j)}|\mathtt{(1}\leq\mathtt{i,j}\leq\mathtt{n).(i+1,j)} \\ \text{et} & \mathtt{matrix} = \lambda\mathtt{(i,j)}|\mathtt{(1}\leq\mathtt{i,j}\leq\mathtt{n).(i,j)} \end{array}$$

La première équation, comme nous l'avons vu précédemment, contraint les domaines de référence et de valeurs de X au domaine de $\mathtt{matrix}^{-1}$, c'est-à-dire que ces domaines sont inclus dans $\{(i,j) \mid 1\leq i,j\leq n\}$. La seconde équation, toujours selon la définition du changement de base, contraint $dom(v_{\mathtt{Y}})$ et $dom(\sigma_{\mathtt{Y}})$: par exemple $v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{h}^{-1} = v_{\mathtt{X}} \circ \lambda\mathtt{(i,j)}|\mathtt{(2}\leq\mathtt{i}\leq\mathtt{n+1}\,\&\,\mathtt{1}\leq\mathtt{j}\leq\mathtt{n).(i-1,j)}$, qui implique $dom(v_{\mathtt{Y}}) \subseteq \{(i,j) \mid 2\leq i\leq n+1 \wedge 1\leq j\leq n\}$. Nous allons donc traduire ces équation par :

```
type X(1:n), type Y(2:n+1)
template T(n+1,n)
align X(i,j) with T(i,j)
align Y(i,j) with T(i+1,j)
forall (i=1:n,j=1:n) Y(i+1,j) = X(i,j)
```

Restrictions :

L'expressivité de la directive `align` n'étant pas aussi grande que celles des fonctions apparaissant dans le changement de base, nous sommes amenés à considérer plusieurs restrictions. La première est liée à la forme affine des alignements HPF. Par conséquent la fonction de changement de base doit être de la forme :

$$\mathtt{h} = \lambda(i_1,\ldots,i_m)\,.\,(f_1(i_1,\ldots,i_m),\ldots,f_n(i_1,\ldots,i_m))$$

où les f_i sont des fonctions affines.

Une autre contrainte survient quand, pour $\mathtt{h} : \mathbb{Z}^m \mapsto \mathbb{Z}^n$, nous avons $m \neq n$. Dans ce cas, la syntaxe de la directive nous oblige à aligner le tableau ayant le moins de dimensions sur celui ayant le plus de dimensions. Nous distinguons donc deux cas pour la traduction :

- si $m > n$ alors `align X(`z`) with T(`z`)`
  `align Y(`dom`(h`$^{-1}$`)) with T(`img`(h`$^{-1}$`))`
  avec $z \in \mathbb{Z}^m$.
- si $m < n$ alors `align X(`dom`(h)) with T(`img`(h))`
  `align Y(`dom`(h`$^{-1}$`)) with T(`img`(h`$^{-1}$`))`

Ainsi, l'alignement d'un vecteur sur la première colonne d'un vecteur est souvent défini par l'équation `Y` $=$ `h :: X`. Le changement de base `h` est alors λ`(i,j)|(j=0).(i)` si `Y` est le vecteur, ou bien son inverse λ`(i).(i,0)` si `X` est le vecteur. Si `Y` est le vecteur nous produisons les directives :

```
align X(i,j) with T(i,j)
align Y(i) with T(i,0)
```

Dans ce cas, la seule contrainte implicite de l'équation est que les valeurs du vecteur sont celles de la première colonne de la matrice ($v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \lambda$`(i).(i,0)`).

Par contre, si `X` est le vecteur, le tableau `Y` ne peut avoir de valeurs que sur sa première colonne ($dom(v_{\mathtt{Y}}) = dom(\sigma_{\mathtt{Y}})$ et $dom(\sigma_{\mathtt{Y}}) \subseteq \{(i,j) \mid j{=}0\}$). Par conséquent les directives générées sont :

```
align X(i) with T(i,0)
align Y(i,0) with T(i,0)
```

Le traducteur affiche d'ailleurs un message d'avertissement dans ce dernier cas de figure, car l'utilisation de la première colonne seulement d'une matrice est inhabituelle.

Réduction géométrique combinée avec une opération fonctionnelle

Seules les fonctions `XXX_SCATTER` de réduction de tableaux d'HPF (voir section 5.1.2) permettent d'exprimer les réductions PEI en toute généralité. Soit l'équation PEI :

$$\begin{array}{rl} & \mathtt{Y} = \mathtt{fr} \rhd \mathtt{(gr\ ;} \rhd \mathtt{X)} \\ \text{où} & \mathtt{fr} = \mathtt{id} \# \lambda\mathtt{(a;b).(a} \otimes \mathtt{fr(b))} \end{array}$$

et soit $e_{\otimes}$ l'élément neutre de $\otimes$. Pour traduire cette équation nous avons besoin de construire les tableaux d'indirections x_1 à x_n utilisés par HPF. Pour cela, nous définissons la fonction $index$ qui sélectionne la $k^{ième}$ dimension d'un m-uplet :

$$\begin{array}{rrcl} index: & \mathbb{N}^m \times [1 \ldots m] & \mapsto & \mathbb{N} \\ & ((i_1, .., i_m), k) & \rightarrow & i_k \end{array}$$

Les tableaux $x_k, \forall k \in [1 \ldots n]$ sont construits par :

$$\forall(i_1, \ldots, i_m) \in dom(\sigma_{\mathtt{X}}), x_k(i_1, \ldots, i_m) = index(\mathtt{gr}(i_1, \ldots, i_m), k)$$

Ainsi, nous construisons une application de l'ensemble des $(i_1, \ldots, i_m) \in dom(\sigma_{\mathtt{X}})$ dans l'ensemble des éléments $(x_1(i_1, \ldots, i_m), \ldots, x_n(i_1, \ldots, i_m))$, comme spécifié dans [27], p.89.

Notons que nous sommes ici dans le cas particulier où $n = m$ puisque les réductions géométriques de PEI sont définies par des fonctions de $\mathbb{Z}^m \mapsto \mathbb{Z}^m$.

La traduction de l'équation précédente est alors :

$$\mathtt{Y} = \otimes\mathtt{\_SCATTER(X,}e_{\otimes}\mathtt{,}x_1\mathtt{,}\ldots\mathtt{,}x_n\mathtt{)}$$

Nous traduisons par exemple l'équation

$$\begin{array}{ll} & \mathtt{Y} = \mathtt{sum} \rhd \mathtt{(red\ ;}\rhd\mathtt{\ X)} \\ \text{avec} & \mathtt{sum} = \mathtt{id} \# \lambda\mathtt{(a;b).(a+sum(b))} \\ & \mathtt{red} = \lambda\mathtt{(i,j)|(1}\leq\mathtt{i,j}\leq\mathtt{3).(i,i)} \end{array}$$

par

$$\mathtt{Y} = \mathtt{SUM\_SCATTER(X,0,} \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}\mathtt{)}$$

Une amélioration consisterait à examiner les cas particuliers de réductions selon les axes canoniques du domaine de référence, afin d'utiliser les primitives plus simples XXX_SUFFIX ou XXX_PREFIX.

Définition récursive

Considérons l'équation suivante :

$$\mathtt{Y} = \mathtt{fc} \rhd \mathtt{(X\ /\&/(Y} \lhd \mathtt{g}_1\mathtt{)/\&/\ldots\ /\&/(Y} \lhd \mathtt{g}_n\mathtt{))}$$

Trouver la solution Y d'une telle équation revient donc à résoudre une équation de point fixe. Une traduction naïve consiste à calculer une suite de champs Y dont le dernier est solution de l'équation. C'est la solution retenue dans [21], où une boucle séquentielle sur un indice supplémentaire représente la suite des champs Y calculés.

Une optimisation consiste à confondre cette boucle avec la ou les directions représentant le temps, selon que l'on utilise des cadencements mono où multi-dimensionnels. Rappelons en effet qu'une telle équation n'est traduisible que si elle définit un cadencement des calculs : les termes successifs de la récurrence doivent alors être calculés selon ce cadencement. Ceci se traduit de manière générale, par un bloc constitué, par autant de boucles séquentielles imbriquées qu'il y a de dimensions de temps, et à l'intérieur de ces boucles, par une opération parallèle `forall` sur les dimensions non temporelles. L'imbrication des boucles suppose que l'indice d'une boucle soit indépendant des indices des boucles internes. Si ce n'est pas le cas, on résout classiquement ce problème par l'algorithme d'élimination de Fourier.

La traduction est constituée de deux blocs d'instructions. Le premier effectue l'initialisation de `Y` avec les valeurs de `X` utilisées au premier instant de calcul, tandis que le deuxième est une boucle séquentielle énumérant les différents instants de calcul.

Dans l'équation précédente, les fonctions $\mathtt{g}_k, k \in [1 \ldots n]$ définies de $\mathbb{Z}^m$ dans $\mathbb{Z}^m$ peuvent s'écrire

$$\mathtt{g}_k = (f_{k1}(i_1, \ldots i_m), \ldots, f_{km}(i_1, \ldots i_m))$$

PEI autorise l'utilisation, dans une même définition récursive, de champs ayant des domaines de valeurs différents. La conséquence est qu'il est difficile de déterminer quelles valeurs, provenant des différents champs superposés, sont utilisées à chacune des étapes de la récurrence. Le problème se pose notamment pour la traduction HPF : l'opérateur `fc` est défini en PEI (comme dans tout langage fonctionnel) sur une liste de longueur quelconque, alors qu'en HPF, nous devrons distinguer les cas pour lesquels l'opérateur a un nombre différent d'arguments. Le problème se résume à décomposer l'union des domaines de valeurs utilisés, en sous-domaines sur lesquels le nombre de valeurs superposées est constant. Un tel algorithme de décomposition a par exemple été implanté pour un problème similaire dans une version récente de la librairie Polylib[63]. Nous ne nous intéressons pour l'instant qu'à deux cas particuliers correspondants à la grande majorité des problèmes rencontrés dans la pratique :

- Nous considérons le cas où le domaine des valeurs du champ initial est disjoint de celui des opérations géométriques appliquées récursivement. C'est le cas de l'élimination de Gauss par exemple, où les valeurs du champ initial sont uniquement utilisées pour définir le premier pas de la récurrence. Dans ce cas l'opérateur dans la récurrence ne porte pas sur les valeurs du champ initial.

- Le deuxième cas considéré est celui où les valeurs du champ initial sont utilisées tout au long de la récurrence. Un exemple typique est la somme préfixe d'un vecteur de nombres. Dans ce cas l'opérateur utilise à toute étape de la récurrence une valeur du vecteur initial.

Dans le cas particulier d'un cadencement affine mono-dimensionnel, supposons que la dimension i_1 soit la dimension temporelle. Les fonctions f_{k1} représentent alors les dépendances temporelles pour tout $k \in [1 \ldots n]$. Ces fonctions sont soit de la forme

$$f(i_1, \ldots, i_m) = f'(i_1) = i_1 - c$$

où c est une constante positive, soit de la forme

$$f_{k1}(i_1, \ldots, i_m) = exp$$

où exp a une valeur indépendante de i_1, mais est inférieure à i_1 pour tout i_1 dans $dom(f_{k1})$.

Pour nous situer dans un cas simple (évitant la décomposition des domaines évoquée précédemment), supposons que la dépendance temporelle est $f'(i_1) = i_1 - c$ pour toutes les fonctions $\mathtt{g}_k$. Nous cherchons les bornes inférieures et supérieures de la boucle séquentielle, que nous appelons $i_{1_{min}}$ et $i_{1_{max}}$ respectivement.

L'initialisation de certaines valeurs de Y avec celles du champ initial X est nécessaire pour amorcer la récurrence. Les éléments de tableau de Y devant être initialisés sont ceux référencés à l'instant $i_1 = i_{1_{min}}$. Soit $t_0 = f'(i_{1_{min}})$ l'instant précédant le premier pas de calcul de la récurrence. Pour tout $k \in [1 \ldots n]$, nous désignons par $\mathcal{D}_k$ l'ensemble des indices référencés par une dépendance $\mathtt{g}_k$ à l'instant $i_{1_{min}}$:

$$\mathcal{D}_k = (t_0, f_{k2}(i_{1_{min}}, i_2, \ldots, i_m), \ldots, f_{km}(i_{1_{min}}, i_2, \ldots, i_m))$$

L'ensemble des indices référencés à l'instant $i_1 = i_{1_{min}}$ est alors $\mathcal{D} = \mathcal{D}_1 \cup \ldots \cup \mathcal{D}_n$. Tous les éléments de Y appartenant à $\mathcal{D}$ doivent donc être initialisés avec les valeurs correspondantes du champ initial X.

Dans la suite de la récurrence, nous considérons les deux cas évoqués précédemment : le champ initial X ne doit pas apparaître dans l'instruction si il n'est utilisé que par le premier calcul de la récurrence, ce qui se traduit par la condition $dom(v_{\mathtt{X}}) \cap (dom(\mathtt{g}_1) \cup \ldots \cup dom(\mathtt{g}_n)) = \{\}$. Dans le deuxième cas de figure qui se traduit par $(dom(\mathtt{g}_1) \cup \ldots \cup dom(\mathtt{g}_n)) \subset dom(v_{\mathtt{X}})$, il apparaît dans l'instruction. Dans les instructions HPF suivantes donnant la forme générale de la traduction, le champ X est donc présent ou pas selon le cas.

Notons que la restriction dont nous avons parlé en section 5.1.1 nous oblige à utiliser des blocs `forall` pour un schéma général de traduction. Nous supposons dans ce qui suit que l'ordre d'imbrication des `forall` est tel qu'un indice dépend uniquement des indices apparaissant dans les `forall` englobants.

```
c initialisation
    forall (i_1 = f'(i_1_min))
       forall (i_2 = D_2)
       ...
          forall (i_m = D_m)
    &        Y(i_1, i_2, ..., i_m) = X(i_1, i_2, ..., i_m)
       ...
       endforall
    endforall
! recurrence
    do i_1 = i_1_min, i_1_max
       forall (i_2 = dom(f_12), ..., i_m = dom(f_n2) ∩ dom(fc))
       ...
          forall (i_m = dom(f_1m) ∪ ... ∪ dom(f_nm))
    &        Y(i_1, i_2, ..., i_m) =
    &        fc(X(i_1, i_2, ..., i_m),Y(g_1(i_1, i_2, ..., i_m)), ..., Y(g_n(i_1, i_2, ..., i_m)))
          ...
       endforall
    enddo
```

Nous pouvons illustrer simplement ce schéma de traduction avec l'exemple de la diffusion discrète présenté sur la figure 5.2. De l'équation (2), page 86, pour laquelle on a trouvé un cadencement, nous devons construire une boucle séquentielle ayant pour indice `t`, et donc faire de `t` une variable indépendante. Ici, les expressions de `t` et de `p` deviennent :

$$\left\{ \begin{array}{l} 1 \le \mathtt{t} - \mathtt{p} \le n \\ 1 \le \mathtt{p} \le n \end{array} \right. \quad \text{c'est-à-dire} \quad \left\{ \begin{array}{l} 2 \le \mathtt{t} \le 2n \\ max(1, \mathtt{t} - n) \le \mathtt{p} \le min(n, \mathtt{t} - 1) \end{array} \right.$$

L'instant initial est donné par la borne inférieure de l'intervalle des valeurs de l'indice `t`, c'est-à-dire `t`=2, et l'instant précédent est $t_0 = 1$. Au premier pas de la récurrence on fait référence aux indices (t_0, `p-1`) et (t_0, `p`) pour $max(1, t_0) \le \mathtt{p} \le min(n, t_0))$, soit $\mathtt{p} = 1$. Pour les deux fonctions `g1` et `g2`, on doit donc initialiser la variable `Y` aux indices $\{(1,1)\} \cup \{(1,0)\}$. Si, du reste de l'énoncé, nous déduisons que $dom(v_{\mathtt{X}}) \cap (dom(\mathtt{g1}) \cup dom(\mathtt{g2})) = \{\}$ alors l'instruction interne à la boucle sera `Y'(t,p) = f(Y'(t-1,p-1),Y'(t-1,p))`. Si, par contre $(dom(\mathtt{g1}) \cup dom(\mathtt{g2})) \subset dom(v_{\mathtt{X}}))$, alors l'équation (2) est traduite par :

```
 forall (t=1,p=0:1) Y'(t,p) = X(t,p)
 do t = 2,2*n
   forall (p=max(1,t-n):min(n,t-1))
&      Y'(t,p) = f(X'(t,p),Y'(t-1,p-1),Y'(t-1,p))
 enddo
```

5.5 Exemple 1 : élimination de Gauss

Nous reprenons ici l'algorithme présenté sous la forme du programme PEI de la section 4.1. Il nous permet notamment d'appliquer la méthode de traduction des récurrences exposée précédemment. Pour cet exemple, nous constatons que l'union des domaines de valeurs des champs initiaux `X` et `Y` est disjonte de l'union des domaines des routages utilisés dans la récurrence. La récurrence n'utilise donc des valeurs initiales qu'au premier pas de calcul, et seul le champ défini récursivement apparaît dans le calcul.

Dans un but de concision, nous avons fusionné les `forall` sur les dimensions `i` et `j`, ces deux indices étant indépendants l'un par rapport à l'autre.

```
       program Gauss
       real C(1:n,1:n+1,0:n), XN(1:n,1:n,1:n), YN(1:n,n+1:n+1,1:n)
       real X(1:n,1:n,0:0), Y(1:n,n+1:n+1,0:0)
       real A(1:n,1:n), B(1:n)

!hpf$  template T_C(1:n,1:n+1,0:n)
!hpf$  align XN(i,j,k),YN(i,j,k), C(i,j,k) WITH T_C(i,j,k)
!hpf$  align A(i,j) WITH T_C(i,j,0)
!fpf$  align B(i) WITH T_C(i,n+1,0)

       forall (i=1:n,j=1:n) X(i,j,0)=A(i,j)
       forall (i=1:n) Y(i,n+1,0)=B(i)

c initialisation recurrence
       forall (k=0,i=1:n,j=1:n) C(i,j,k) = X(i,j,k)
       forall (k=0,i=1:n,j=n+1) C(i,j,k) = Y(i,j,k)
c recurrence
       do k=1,n-1
          forall (i=k:n,j=k:n)
       &      C(i,j,k) = C(i,j,k-1)-C(i,k,k-1)*C(k,j,k-1)/C(k,k,k-1)
       enddo
```

```
      forall (i=1:n,j=1:n,k=1:n,k .EQ. i-1) XN(i,j,k) = C(i,j,k)
      forall (i=1:n,j .EQ. n+1 .AND. k .EQ. i-1) YN(i,j,k) = C(i,j,k)
      end
```

5.6 Exemple 2 : *Gaxpx*

Nous traduisons ici les programmes Gaxpx1 et Gaxpx4 de la section 4.6. Ces programmes illustrent plus particulièrement la traduction des réductions géométriques. Notons que nous avons instancié le paramètre de structure du problème ($n = 5$) car les tableaux nécessaires à SUM_SCATTER ne doivent contenir que des constantes. La premier programme est la traduction de l'algorithme initial, qui place le vecteur X sur la première ligne de la matrice, et nécessite ensuite de transposer le vecteur résulat C0 pour l'additionner avec le vecteur original X.

```
      program Gaxpx1
      real A(0:4,0:4), B(0:4,0:4), P(0:4,0:4)
      real C0(0:4,0:4), C1(0:4,0:4), C(0:4,0:4)
      real X(0:4),B0(0:4,0:4)
!hpf$ template T_B0(0:4,0:4)

!hpf$ align A(i,j), B(i,j), P(i,j), C1(i,j), C0(i,j), C(i,j) with T_B0(i,j)
!hpf$ align X(j) with T_B0(0,j)

      forall (j=0:4) B0(0,j)=X(j)
      forall (i=0:4,j=0:4) B(i,j) = B0(0,j)
      forall (i=0:4,j=0:4,true) P(i,j) = (A(i,j)*B(i,j))
      C0 = SUM_SCATTER(P,0,
      &    ([[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4],
      &       [0,1,2,3,4],[0,1,2,3,4]]),
      &    ([[4,4,4,4,4],[4,4,4,4,4],[4,4,4,4,4],
      &       [4,4,4,4,4],[4,4,4,4,4]]))
      forall (j=0:4,i .EQ. 0) C1(i,j) = C0(j,4)
      forall (i=0:4,j=0:4,true) C(i,j) = B0(i,j)+C1(i,j)
      end
```

Dans ce deuxième programme, on constate que la phase de transposition a disparu (avant dernier forall de Gaxpx1), ainsi que le tableau intermédiaire C1. Des conclusions de la section 4.6, nous savons que les vecteurs résultats de ces deux programmes sont les mêmes et sont placés sur la première ligne dans Gaxpx1 et sur la diagonale dans Gaxpx2.

```
       real A(0:4,0:4), B(0:4,0:4), P(0:4,0:4)
       real C0(0:4,0:4), C(0:4,0:4)
       real X(0:4),B0(0:4,0:4)
!hpf$  template T_B0(0:4,0:4)

!hpf$  align A(i,j), B(i,j), P(i,j), C0(i,j), C(i,j) with T_B0(i,j)
!hpf$  align X(j) with T_B0(0,j)

       forall (j=0:4) B0(j,j)=X(j)
       forall (i=0:4,j=0:4) B(i,j) = B0(j,j)
       forall (i=0:4,j=0:4,true) P(i,j) = (A(i,j)*B(i,j))
       C0 = SUM_SCATTER(P,0,
       &    ([[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4],
       &       [0,1,2,3,4],[0,1,2,3,4]]),
       &    ([[0,1,2,3,4],[0,1,2,3,4],[0,1,2,3,4],
       &       [0,1,2,3,4],[0,1,2,3,4]]))
       forall (i=0:4,j=0:4,true) C(i,j) = B0(i,j)+C0(i,j)
       end
```

5.7 Adaptation à d'autres langages data-parallèles

Le schéma de traduction présenté produit des programmes HPF d'un style très particulier, dont la sémantique des indices de tableau est la même que celles des programmes Pei : les indices désignent les points d'un domaine de référence commun aux champs du programme. Ce domaine de référence représente la machine virtuelle utilisée par le programme data-parallèle. Nous avons vu que cette sémantique était celle de la plupart des langages data-parallèles (C*[54], MPL[38], Hyper-C[45], DPCE[53], *etc*), et par conséquent le code produit peut facilement être adapté à n'importe quel langage data-parallèle autorisant la déclaration de machines virtuelles. Les modifications à apporter concernent principalement le remplacement des alignements par des instructions de communication, comme nous l'avons évoqué dans la traduction des changements de base Pei. Egalement, la réduction géométrique de Pei nécessite une primitive de réduction très générale dans le langage cible, et seul HPF à notre connaissance propose un opérateur assez général pour exprimer toutes ces réductions.

5.8 Perspectives

Nous donnons dans cette section quelques pistes pour améliorer la traduction présentée précédemment. Cette amélioration consiste à profiter des alignements de HPF pour abstraire les expressions de tableaux de leur signification géométrique par rapport au référentiel représentant la machine virtuelle.

Nous pouvons maintenant schématiser l'utilisation typique des champs de données dans les énoncés PEI : à partir des champs en entrée, d'autres champs sont définis pour représenter le placement sur le référentiel des valeurs en entrée. Ces définitions utilisent des opérations géométriques et des changements de base (alignement, duplication) afin d'amener les valeurs en tous les points du référentiel où auront lieu des calculs. Nous appelons cette dernière catégorie de champ, les *champs placés*. Dans notre schéma de traduction précédent, ce sont les champs placés qui apparaissent dans les calculs, et par conséquent les indices des tableaux traduisant ces champs désignent des processeurs virtuels, comme nous l'avons souligné en section 5.2. L'idée est donc de substituer dans le programme les champs placés par les champs en entrée munis de l'alignement correspondant. Ainsi, le programme HPF résultat de la traduction ne comporte plus les "déformations" dues au placement dont nous avons parlé. Pour cela, nous devons améliorer la génération des directives d'alignement de façon à prendre en compte les duplications, ainsi que la traduction des équations récurrentes.

5.8.1 Traduction des diffusions par des alignements

Les alignements HPF que nous avons pris en compte jusqu'à présent sont uniquement déduits des changements de base de l'énoncé PEI, et représentent par conséquent des bijections entre la représentation des valeurs de tableau et la représentation des valeurs sur un template. Cependant, la directive `align` nous permet de prendre en compte également des duplications de valeurs. Déterminer quand une valeur intiale est dupliquée revient à identifier dans l'énoncé PEI, le motif :

$$\begin{aligned} \mathtt{X} &= \mathtt{h} :: \mathtt{Y} \\ \mathtt{Z} &= \mathtt{Y} \lhd \mathtt{g} \end{aligned}$$

où $\mathtt{X}$ est un champ de données en entrée, $\mathtt{h}\colon \mathbb{Z}^n \mapsto \mathbb{Z}^p$, et $\mathtt{g} : \mathbb{Z}^n \mapsto \mathbb{Z}^n$ est une fonction non injective, avec $img(\mathtt{g}) = dom(\mathtt{h})$. Supposons les champs $\mathtt{X}$, $\mathtt{Y}$ et $\mathtt{Z}$ respectivement égaux aux paires de fonctions $(v_{\mathtt{X}} : \sigma_{\mathtt{X}})$, $(v_{\mathtt{Y}} : \sigma_{\mathtt{Y}})$ et $(v_{\mathtt{Z}} : \sigma_{\mathtt{Z}})$. De la définition des opérations, les deux équations précédentes nous donnent :

$$\begin{aligned} (v_{\mathtt{X}} : \sigma_{\mathtt{X}}) &= (v_{\mathtt{Y}} \circ \mathtt{h}^{-1} : \sigma_{\mathtt{Y}} \circ \mathtt{h}^{-1}) \\ (v_{\mathtt{Z}} : \sigma_{\mathtt{Z}}) &= (v_{\mathtt{Y}} \circ \mathtt{g} : \sigma_{\mathtt{Y}}) \end{aligned}$$

Nous cherchons maintenant la relation entre les valeurs du champ entrée X et les valeurs du champ Z. Nous avons $v_{\mathtt{X}} = v_{\mathtt{Y}} \circ \mathtt{h}^{-1} \Rightarrow v_{\mathtt{X}} \circ \mathtt{h} = v_{\mathtt{Y}} \circ \mathtt{h}^{-1} \circ \mathtt{h}$ soit $v_{\mathtt{Y}}|_{dom(h)} = v_{\mathtt{X}} \circ \mathtt{h}$. Comme $dom(v_{\mathtt{Y}}) \subseteq dom(\mathtt{h})$, $v_{\mathtt{Y}} = v_{\mathtt{X}} \circ \mathtt{h}$. Alors $v_{\mathtt{Z}} = v_{\mathtt{X}} \circ \mathtt{h} \circ \mathtt{g}$. Cette relation nous donne le placement des valeurs initiales sur le champ Z. Par exemple, le motif d'équations :

$$\begin{aligned}&\mathtt{X} = \lambda\mathtt{(i,j)|(1\leq i\leq n\,\&\,j=i).(i)::Y}\\&\mathtt{Z} = \mathtt{Y} \lhd \lambda\mathtt{(i,j)|(1\leq j\leq n\,\&\,1\leq j\leq n).(j,j)}\end{aligned}$$

donne $v_{\mathtt{Z}} = v_{\mathtt{X}} \circ \lambda\mathtt{(i,j).(j)}$, ce qui signifie intuitivement, que les valeurs de coordonnées (i,j) de Z «viennent» des points de coordonnées j de X, ce qui peut se traduire par la directive **align X(j) with Z(*,j)**. En alignant ces deux tableaux sur un template commun, et en renommant les indices, nous avons :

```
align X(i)   with T(*,i)
align Z(i,j) with T(i,j)
```

5.8.2 Substitution des champs placés par les champs entrée

Il est important de noter que l'alignement de X sur le template est permanent, c'est-à-dire que toute modification d'un élément X(i) durant l'exécution est reportée sur tous ses éléments dupliqués ([32], page 118). Ainsi les instructions

`X(1) = v`   et   `forall (i=1:n,j=1) Z(i,j) = v`

réalisent des affectations (respectivement aux variables X et Y) par le même ensemble de processeurs virtuels.

Cette caractéristique permet, dans certaines conditions, de remplacer une traduction faisant intervenir la variable Z, par une traduction où n'intervient que la variable X. La condition, pour réaliser une telle substitution, dans le cas d'une duplication, est que toute opération faisant référence aux valeurs de Z, le fait pour l'ensemble des valeurs dupliquées. Par contre, si dans l'énoncé PEI, une expression fait référence à une partie seulement des valeurs sur la première dimension (la dimension dupliquée par l'alignement de X) du champ Z, alors nous ne pouvons exprimer cette expression en utilisant X à la place de Z. Par exemple, si on trouve dans le programme une expression comme $\mathtt{A} = \mathtt{Z} \lhd \lambda\mathtt{(i,j)|(i=1\,\&\,j=1)}$, on ne pourra pas exprimer A en fonction d'une variable X alignée comme ci-dessus.

5.8.3 Optimisation des traductions des récurrences

Nos programmes traduits en HPF sont des programmes à assignation unique, et il est souvent reproché à ce modèle son coût mémoire prohibitif. Des travaux importants ont été menés dans ce domaine [9, 40, 62, 35] mais nous nous intéressons ici à un cas particulier d'optimisation mémoire.

En partant du constat que seul le résultat final d'une définition récursive intéresse généralement le programmeur, il vient naturellement l'idée, en cas de récurrence de profondeur un, de substituer les écritures mémoires sur la dimension temporelle par plusieurs écritures dans une cellule mémoire unique. Nous pouvons déterminer que seul le résultat final de la récurrence est significatif par la présence d'une opération de géométrique sélectionnant ces valeurs finales. Il faut naturellement s'assurer qu'aucune autre définition n'utilise une valeur du champ à un instant particulier. Dans ce cas, nous remplaçons la dimension temporelle par un scalaire. Dans l'exemple `PrefixSum` (page 15) de la somme de n nombres, notre traduction devient :

```
A = R(1)
do i=2,n
   A = R(i) + A
enddo
```

Nous n'avons plus les sommes partielles comme dans le version traduite précédemment, mais uniquement la somme finale des valeurs du vecteur `R`.

5.8.4 Exemple

Le programme de la figure 5.6 est une version du produit matriciel vu en section 3.4, utilisant n^3 processeurs virtuels pour deux matrices A et B ($n \times n$). Le programme aligne les matrices A et B dans un référentiel à trois dimensions : A est alignée sur la face latérale d'un champ cubique, et B sur la face du dessus du cube commme le montre la figure 5.5.

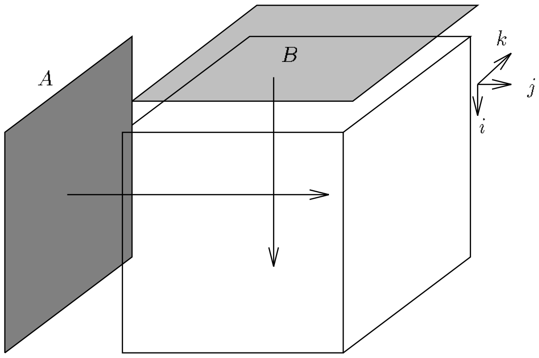

FIG. 5.5 - *Produit matriciel sur un cube de processeur virtuels*

Le champ `A'` est ensuite défini comme la diffusion de la matrice A selon les lignes du cube, et `B'` comme la diffusion de B selon les colonnes du même cube. Les deux diffusions permettent de calculer l'ensemble des n^3 produits en une opération fonctionnelle. Enfin, les produits sont sommés le long de la troisième dimension, et le résulat `R` est le dernier plan de cette dimension.

$$
\texttt{MatMult : (A,B)} \mapsto \texttt{R}
$$

$$
\left\{
\begin{array}{l}
\texttt{A} = \texttt{align1}::\texttt{A0}\\
\texttt{B} = \texttt{align2}::\texttt{B0}\\
\texttt{A0} = \texttt{dim}::\texttt{A}\\
\texttt{B0} = \texttt{dim}::\texttt{B}\\
\texttt{A'} = \texttt{A0} \lhd \texttt{spread1}\\
\texttt{B'} = \texttt{B0} \lhd \texttt{spread2}\\
\texttt{P} = \texttt{mult} \rhd \texttt{(A' /\&/ B')}\\
\texttt{C} = \texttt{add} \rhd \texttt{(P /\&/ C} \lhd \texttt{prev)}\\
\texttt{R} = \texttt{C} \lhd \texttt{last}
\end{array}
\right.
$$

$$
\begin{array}{lcl}
\texttt{align1} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i<n}\,\&\,\texttt{j=0}\,\&\,\texttt{0}\leq\texttt{k<n).(i,k)}\\
\texttt{align2} & = & \lambda\texttt{(i,j,k)}|\texttt{(i=0}\,\&\,\texttt{0}\leq\texttt{j<n}\,\&\,\texttt{0}\leq\texttt{k<n).(k,j)}\\
\texttt{dim} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i<n}\,\&\,\texttt{0}\leq\texttt{j<n}\,\&\,\texttt{0}\leq\texttt{k<n).(i,j,k)}\\
\texttt{spread1} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i<n}\,\&\,\texttt{0}\leq\texttt{j<n}\,\&\,\texttt{0}\leq\texttt{k<n).(i,0,k)}\\
\texttt{spread2} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i<n}\,\&\,\texttt{0}\leq\texttt{j<n}\,\&\,\texttt{0}\leq\texttt{k<n).(0,j,k)}\\
\texttt{last} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i<n}\,\&\,\texttt{0}\leq\texttt{j<n}\,\&\,\texttt{k=n-1)}\\
\texttt{prev} & = & \lambda\texttt{(i,j,k)}|\texttt{(0}\leq\texttt{i<n}\,\&\,\texttt{0}\leq\texttt{j<n}\,\&\,\texttt{1}\leq\texttt{k<n).(i,j,k-1)}\\
\texttt{mult} & = & \lambda\texttt{(a;b).(a*b)}\\
\texttt{add} & = & \texttt{id}\ \#\ \lambda\texttt{(a;b).(a+b)}
\end{array}
$$

FIG. 5.6 - *Programme* PEI *de produit matriciel*

Nous identifions dans ce programme deux motifs de diffusion, qui sont les paires d'équations :

$$
\begin{array}{l}
\texttt{A} = \texttt{align1}::\texttt{A0}\\
\texttt{A'} = \texttt{A0} \lhd \texttt{spread1}
\end{array}
\quad \text{et} \quad
\begin{array}{l}
\texttt{B} = \texttt{align2}::\texttt{B0}\\
\texttt{B'} = \texttt{B0} \lhd \texttt{spread2}
\end{array}
$$

Les valeurs des champs placés `A'` et `B'` sont donc $v_{\texttt{A'}} = v_{\texttt{A}} \circ \texttt{align1} \circ \texttt{spread1} = \lambda\texttt{(i,j,k).(i,k)}$ et $v_{\texttt{B'}} = v_{\texttt{B}} \circ \texttt{align2} \circ \texttt{spread2} = \lambda\texttt{(i,j,k).(k,j)}$ respectivement. Nous avons vu que ces diffusions peuvent être traduites par les directives d'alignements :

```
align A(i,k) with A'(i,*,k)
align B(k,j) with B'(*,j,k)
```

Etant donné les alignements spécifiés et le fait que toutes les références aux champs `A'` et `B'` utilisent la totalité des valeurs de ces champs, nous pouvons substituer dans le programme HPF, les références `A'(i,j,k)` par `A(i,k)` et `B'(i,j,k)` par `B(k,j)`.

La somme des produits, définie par le champ `C` sous une forme récursive, nous amène à chercher un cadencement pour la traduction. Une fonction de temps candidate est naturellement la dépendance sur l'axe k. La boucle séquentielle représentant le temps peut par conséquent utiliser `k` comme indice. Après substitution de `P` par sa définition, la traduction du calcul des produits selon ce procédé est :

```
forall (i=0:n-1,j=0:n-1,k=0) C(i,j,k) = A(i,k)*B(k,j)
do k=1,n-1
    forall (i=0:n-1,j=0:n-1) C(i,j,k) = C(i,j,k-1) + A(i,k) * B(k,j)
enddo
```

De plus, nous sommes dans le cas favorable où l'assignation unique sur `C` peut être levée facilement. La seule autre définition faisant référence à une occurrence de `C(i,j,k)` étant la sélection `last` du dernier plan de la dimension k nous pouvons tranformer le tableau tri-dimensionnel `C` en en tableau bi-dimensionnel et supprimer la dernière équation. Le problème se pose de savoir comment doit être aligné ce tableau bi-dimensionel sur le cube de processeurs virtuels. Pour restituer les communications de voisins à voisins présentes dans la version à assignation unique, utilisée pour la somme des produits, une possiblité est de placer le tableau `C` à chaque itération de la boucle `do`, sur le plan correspondant à la valeur de l'indice `k`. Cet alignement dynamique du tableau s'exprimerait alors par `realign C(i,j) with T(i,j,k)`. Cependant, le coût en temps du réalignement est probablement important et il semble qu'un alignement statique arbitraire sur l'un des plans de la troisième dimension est préférable. Nous choisissons ainsi d'aligner `C` sur le plan `k`$=n-1$, et notre traduction est finalement :

```
        real A(0:n-1,0:n-1), B(0:n-1,0:n-1), C(0:n-1,0:n-1)
!hpf$   template T(0:n-1,0:n-1,0:n-1)
!hpf$   align A(i,k) with T(i,*,k)
!hpf$   align B(k,j) with T(*,j,k)
!hpf$   align C(i,j) with T(i,j,n-1)

        forall (i=0:n-1,j=0:n-1,k=0) C(i,j) = A(i,k) * B(k,j)
        do k=1,n-1
            forall(i=0:n-1,j=0:n-1) C(i,j) = C(i,j) + A(i,k) * B(k,j)
        enddo
```

Ce texte de programme est bien sûr beaucoup plus intuitif que la version utilisant les champs placés.

5.9 Environnement logiciel autour de PEI

La figure 5.7 présente les outils déjà développés ou en cours de développement (traits pleins), où les projets de développements (traits hachurés) autour du formalisme PEI. Nous avons déjà présenté en section 3.5 et 3.4, le module de transformation des énoncés PEI, et celui de conception graphique VPEI. Le module d'interprétation fonctionnelle est décrit dans [57] et implante l'interprétation fonctionnelle des champs de données telle que nous l'avons définie en section 2.3.5. La traduction du programme PEI en un code CAML nous permet indirectement d'exécuter le programme PEI, et ce module pourrait être intégré dans l'environnement graphique afin de visualiser quelles sont, et où sont les valeurs dans les champs, en fonction de champs en entrée.

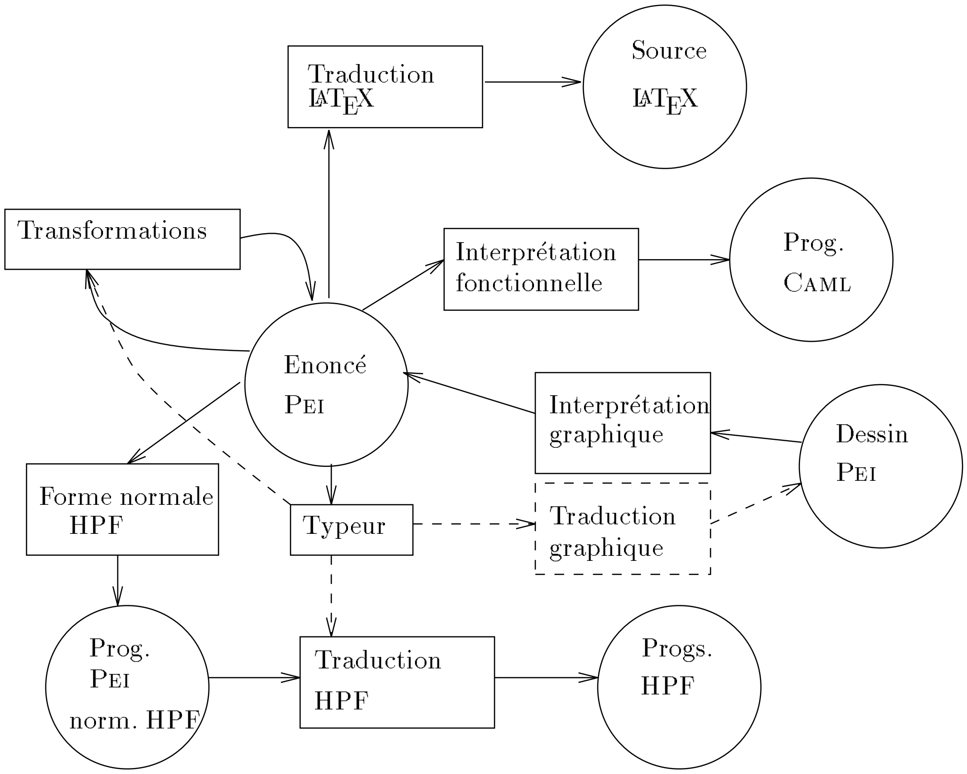

FIG. 5.7 - *Environnement logiciel autour de* PEI

Les développements en cours sur le traducteur `pei2hpf` ont pour objectif immédiat d'intégrer les informations rendues par le contrôleur de type. Ce contrôleur de type infère les domaines de valeurs et de référence des champs à partir des définitions de l'énoncé et vérifie que le programme est bien typé, c'est-à-dire qu'aucune définition n'est en contradiction avec une autre. Les calculs ensemblistes nécessaires pour ce type de calcul sont réalisés à l'aide de la librairie OMEGA[30]. Les informations retournées par ce module devraient permettre la traduction de programmes dont les domaines sont moins simplistes que ceux actuellement acceptés.

5.10 Travaux relatifs

5.10.1 Contexte

De nombreux projets de recherche visent à produire du code parallèle à partir de spécifications plus faciles à appréhender par l'utilisateur.

La voie focalisant le plus d'efforts et la plus ambitieuse est naturellement la compilation des programmes séquentiels existants en programmes parallèles, avec de nombreux projets comme SUIF[1], PIPS[28], PAF[18], LooPo[15], *etc.* Cependant, les compilateurs de ce type ont montré jusqu'à présent des limitations de deux sortes. La première est la limite du *modèle polytope*[36] et concerne tous les travaux menés sur les techniques de parallélisation. La deuxième est spécifique au choix d'accepter en entrée des spécifications formulées dans un langage séquentiel (souvent Fortran 77) dont la sémantique est très compliquée.

Avec l'apparition du modèle data-parallèle est apparue l'idée de scinder la compilation en deux phases : transformer le code séquentiel en code data-parallèle, puis compiler le code data-parallèle en code à parallélisme de contrôle, proche du modèle d'exécution actuel des machines. Le compilateur Bouclettes[6] appartient par exemple à la première catégorie. Dans la deuxième catégorie, de nombreux outils ont été développés, tant par des universités que par des entreprises commerciales, parmi lesquels pghpf (Portland Group), HPFC[13], Adaptor[7], *etc.*

Un autre choix consiste à utiliser comme moyen de spécification un langage doté d'une sémantique claire. Le formalisme des équations récurrentes fait partie de ce type de spécification, et a été choisi pour l'environnement Opera[37] par exemple. D'autres projets ont adopté ou inventé des formalismes pouvant bénéficier des techniques développées dans le domaine du génie logiciel. Dans cette optique, plusieurs voies ont été explorées. Parmi les idées plus répandues, on trouve les projets visant à extraire le parallélisme de spécifications données sous forme fonctionnelle, comme dans SISAL[41] par exemple, ou d'enrichir un langage de type fonctionnel avec des objets parallèles comme dans NESL[3] ou DP-Haskell[25].

Une voie intermédiaire consiste à considérer d'abord les aspects géométriques relatifs au modèle de programmation parallèle, à définir des opérations sur ces objets. C'est le choix fait dans les langages ALPHA, CRYSTAL ou PEI. Par exemple, ALPHA est un langage fonctionnel permettant de spécifier un problème sous la forme d'équations récurrentes, portant sur des variables représentant des domaines polyédriques.

Ces trois projets ont en commun d'avoir défini un formalisme autorisant les transformations de programmes. Ces transformations peuvent modifier notamment les informations géométriques du programme.

5.10.2 Comparaison avec ALPHA

Les travaux menés à l'IRISA sur le langage ALPHA sont proches de ceux entrepris autour de PEI.

Le langage ALPHA

ALPHA est un langage conçu[39] à l'origine pour la programmation d'architectures parallèles synchrones, en permettant d'exprimer un problème sous la forme d'un système d'équations récurrentes affines sur des domaines polyédriques. Un programme ALPHA est constitué de variables en entrée et en sortie, et d'un système d'équations de la forme $X = exp$, où X est une variable déclarée sur un domaine polyédrique $Dom(X)$, appelé domaine spatial, et exp est une fonction définie sur $Dom(X)$.

Les expressions sont formées de variables et d'*opérateurs*. Les opérateurs sont de deux sortes :

- les opérateurs immmobiles généralisent la notion usuelle d'opérateur aux domaines spatiaux. Ils permettent l'application d'opérateurs arithmétiques ou logiques à une collection de valeurs. L'opérateur immobile est semblable à l'opération fonctionnelle de PEI.
- les opérateurs spatiaux :
  - l'**opérateur de restriction** permet de restreindre le domaine de définition d'une expression. Le domaine de l'expression E définie par $E = D' : X$ (X est restreinte au domaine D') est $Dom(E) = D' \cap Dom(X)$.
  - l'**opérateur case**, généralement utilisé en conjonction avec l'opérateur de restriction, permet de définir différemment une variables sur des sous-domaines disjoints. Une expression E définie par cas est notée $E =$ **case** $exp_1 ; \ldots ; exp_n$ **esac**; et $Dom(E) = Dom(exp_1) \cup \ldots \cup Dom(exp_n)$.
  - l'**opérateur de dépendance** exprime une fonction entre domaines spatiaux. L'expression $E.(dep)$ dénote la composition de E et dep, où dep, appelée *fonction de dépendance*, est une fonction affine dont l'image est le domaine de E. Cette dépendance peut aussi s'exprimer à l'aide d'une notation équivalente de

type tableau : ainsi les expressions $E' = E.(z \to f(z))$ et $E'[z] = E[f(z)]$, $z \in Dom(f)$, sont équivalentes. L'opérateur de dépendance est semblable à l'opération géométrique de PEI.

- l'**opérateur de réduction** est semblable à la réduction géométrique de PEI. Etant donné une expression E et f une fonction affine définie sur $Dom(E)$, l'expression **reduce**$(\oplus, f, E)$ a pour valeur au point z, $z \in img(f)$, la valeur $\oplus_{z' \in Dom(E) | f(z') = z} E_{z'}$, où $E_{z'}$ dénote la valeur de E au point z'.

- Transformations. Les transformations du langage ALPHA s'expriment sous la forme de règles de réécritures du programme. Parmi les règles de transformation les plus utilisées, on trouve le changement de base. Un changement de base F sur une variable A définie sur $Dom(A)$ est possible si il existe une fonction G inverse de F sur $Dom(A)$. La réécriture du programme se fait de la manière suivante :

```
A:     Dom(A)
let
       A = ... A ...;
       ... = ... A ...;
tel;
```

$\to$

```
A:     Dom(A).F
let
       A = (... A.F ...).G;
       ... = ... A.F ...;
tel;
```

Différences entre PEI et ALPHA

Les programmes ALPHA sont fortement typés, chaque variable étant explicitement déclarées sur un domaine (éventuellement paramétré). Nous avons vu qu'en PEI les domaines de valeurs ou de référence des champs de donnés pouvaient être non-bornés. Nous laissons au contrôleur de type la charge d'inférer les informations manquantes sur ces domaines, et de vérifier leurs validités.

Outre les limites sur la forme des domaines et les fonctions de dépendances, la différence majeure entre ces deux langages sont les contraintes imposées pour spécifier l'égalité de deux expressions. En PEI, les contraintes sont proches de celles des langages à parallélisme de données dont la sémantique des indices représente des processeurs virtuels. On retrouve par exemple la notion de conformité présente dans ces langages : une opération fonctionnelle où géométrique en PEI ne peut être appliquée que sur des champs définis sur même espace et dont un des domaines de référence inclut ceux des autres. Cette contrainte oblige le programmeur à aligner les champs du programme à l'aide de l'opération de changement de base, qui est une opération à part entière du langage PEI. Les informations d'alignements sont donc données dès la phase initiale de conception. Les transformations donnant des programmes fortement équivalents se font également par changement de base.

Cette opération étant la seule à modifier la fonction σ d'un champ de données, qui exprime la façon dont les valeurs seront placées en fin de compte sur les processeurs virtuels, les alignements spécifiés dans l'énoncé original sont conservés tout au long de ce type de transformation. En ALPHA, par contre aucune contrainte de ce type n'est imposée : l'alignement est l'étape ultime de transformation, et consiste à décider que certains indices représentent les dimensions spatiales de l'architecture virtuelle.

Développements

Une des directions de recherche commune à ces deux langages, est la traduction de programmes exprimés dans ces formalismes, vers les principaux modèles de programmation existants.

Un traducteur qualifié de naïf [61] a été conçu pour évaluer les variables résultats d'un programme ALPHA. Les programmes ALPHA peuvent être vus comme des programmes fonctionnels, et l'évaluation des différentes valeurs d'une variable peut se faire selon les dépendances causales entre les valeurs des variables. Ce principe est le même que celui implanté dans le module d'interprétation fonctionnelle de PEI, où chacune des valeurs d'un champ de sortie est une fonction des valeurs de champs en entrée. Une optimisation réalisée est la mémorisation des valeurs déjà calculées afin d'éviter les ré-évaluations. Un autre problème rencontré est le parcours, au moyen d'un nid de boucles, du domaine polyèdrique d'une variable résultat pour demander l'évaluation des valeurs placées en chacun des points entiers du polyèdre. Dans [50], il est proposé un autre ordonnancement pour l'exécution du programme séquentiel, que celui induit par les dépendances causales. L'idée est d'utiliser les techniques d'ordonnancement et de placement utilisées en parallélisation automatique. L'ordonnancement fournit un ordre d'évaluation des valeurs cohérent, tout en évitant les changements de contexte coûteux qu'implique l'évaluation fonctionnelle du traducteur naïf. Notons que le schéma de traduction procède, de manière analogue à celui exposé dans le chapitre précédent, par des mises sous formes normales des programmes ALPHA par l'application confluente de règles de réécriture.

La précédente méthode est étendue [49] pour la génération de programmes data-parallèles impératifs. Le même principe d'ordonnancement est conservé. Par comparaison, la méthode exposée pour notre traducteur utilise un ordonnancement hybride : nous utilisons l'ordre induit des dépendances causales pour les équations simples, et nous calculons un cadencement pour les définitions récursives quand aucune dimension ne peut représenter le temps de manière évidente.

Un problème supplémentaire est l'alignement des données sur la grille de processeurs virtuels, de façon à minimiser le coût des communications. Il est par conséquent nécessaire de trouver un "bon" changement de base pour l'alignement des variables. A la différence d'un programme ALPHA, un programme PEI comporte les informations d'alignement des champs. Ce sont ces alignements, repérables par l'application d'un changement de base, dont nous nous servons dans la traduction des programmes PEI en HPF. L'optique d'ALPHA peut s'avérer très intéressante dès lors que des outils de placement automatique seraient efficaces. Dans ces conditions, on pourrait cacher la phase d'alignement à l'utilisateur. Cependant, nous avons déjà souligné la difficulté du problème : Darte et Robert [14] ont notamment montré que le problème de l'alignement sur une grille de processeurs virtuels, des calculs et des communications de manière à minimiser les communications, est NP-complet. Ce problème nécessite donc de recourir à des heuristiques, qui peuvent être fondées sur le volume des communications induites par un alignement [51].

Chapitre 6

Conclusion

Il est classique de considérer qu'un programme est une fonction sur des objets typés : cette fonction définit *ce que calcule* le programme. D'un point de vue plus opérationnel, on doit en outre définir *comment* ces calculs sont organisés; et lorsque l'implantation suppose l'utilisation d'une machine parallèle, on définit seulement un ordre partiel, qui peut (ou doit, selon les modèles d'éxécution) être complété en disant *où* sont opérés ces calculs et *où* sont placées les données impliquées.

Cette dernière approche conduit naturellement à l'introduction d'une certaine "géométrie" dans l'expression des programmes, complémentaire de l'aspect fonctionnel habituel. C'est ce qui est à la base des concepts de PEI.

PEI a été conçu à une époque où peu de langages incluaient cette notion de géométrie comme caractéristique à part entière du langage. Les formalismes comparables, autorisant également le raffinement et la transformation de programmes, comme ALPHA[39], CRYSTAL[12] ou UNITY[10] par exemple, n'intègrent pas, à ce point, les contraintes géométriques sur le placement des valeurs présentes dans PEI. Nous avons souligné dans la comparaison avec le langage ALPHA, que cette différence modifiait l'approche de la conception d'un programme : l'écriture d'un programme PEI impose en effet de concevoir *a priori* la manière dont les données sont alignées, à l'aide de l'opération de changement de base.

Notre thèse est que PEI permet de modéliser les concepts des langages à parallélisme de données dont la sémantique des indices des variables est celle des processeurs virtuels. Citons par exemple C*[54], MPL[38], HYPERC[45], et DPCE[53]. Le langage $\mathcal{L}$[4] qui modélise cette approche intègre une certaine forme de géométrie, limitée au minimum, c'est-à-dire à la définition de vecteurs et à la référence à l'indice courant des processeurs virtuels associés. De fait, dans ces langages, l'aspect géométrique est omniprésent par la notion même de variables parallèles : le programmeur doit explicitement aligner, par des communications, les variables de son programme de façon à ce que tous les calculs ne s'effectuent que sur des valeurs locales à chaque processeur virtuel. Nous pensons donc que beaucoup de techniques formelles développées dans PEI peuvent être appliquées aux langages à parallélisme de données, et sont complémentaires d'autres travaux théoriques, comme ceux sur la preuve de programmes data-parallèles [5].

Cependant, préalablement à l'application des techniques PEI aux langages de programmation existants, il était nécessaire de poursuivre le travail d'investigation sur la définition du formalisme lui-même. Nous avons présenté dans le premier chapitre la définition actuelle du formalisme. Nous avons ensuite étendu le calcul de raffinement par rapport à sa définition originale, de façon à pouvoir statuer sur des relations plus fines entre programmes, que nous appelons *équivalence faible*. Le troisième chapitre traite des stratégies de transformations d'énoncés PEI. Ces stratégies sont une application directe du calcul de raffinement, et montrent comment utiliser les règles de raffinement pour représenter des transformations courantes. Notamment, on est souvent amené à substituer des communications de voisin à voisin par des diffusions (ou vice-versa), ou par des opérations de réductions lorsqu'un opérateur est appliqué sur les valeurs communiquées. Les stratégies, s'appuyant sur le calcul de raffinement, devraient à terme, permettre de passer immédiatement d'une version à l'autre, et montrer la relation entre les deux versions. Deux autres stratégies sont également développées dans ce chapitre. La première concerne la modification du schéma de communication d'un programme, et la deuxième, la modification des alignements de données. Cette dernière transformation illustre un cas d'équivalence faible entre programmes. Il est probable que l'étude de cas pratiques plus nombreux pourraient nous inspirer d'autres idées de transformations, que nous pourrons tenter de généraliser en stratégies.

La pratique montre que la manipulation d'un formalisme comme PEI, devient très vite complexe et fastidieuse, en raison des nombreuses contraintes qu'impose la précision du calcul de raffinement. Pour aider le programmeur, les outils de développements se montrent rapidement indispensables. En PEI, ces outils de base sont : un environnement de transformations permettant d'appliquer les règles de raffinement, un traducteur de programmes PEI en programmes CAML basé sur l'interprétation fonctionnelle, et un outil de conception graphique baptisé VPEI. Ce dernier outil développé en début de thèse, visait à montrer l'intérêt d'une représentation graphique des variables parallèles et de leurs alignements, lors de la conception du programme.

Enfin, pour amorcer l'application des techniques PEI aux langages data-parallèles existants, il nous a semblé intéressant de réaliser un traducteur de PEI vers le langage *High*

Performance Fortran[27]. Ce traducteur, présenté dans le chapitre 5, a deux objectifs.

L'objectif premier est la compréhension des problèmes que pose le passage d'un formalisme équationnel comme PEI à un langage data-parallèle impératif comme HPF. Ce travail nous a amené à isoler un sous-ensemble des énoncés PEI, qui sont les programmes PEI traduisibles. Nous montrons que de nombreuses transformations sémantiques peuvent être effectuées dans l'environnement de transformation, pour ne laisser au traducteur qu'un travail de transformation syntaxique. Nous soulignons ici les améliorations à apporter au traducteur, notamment dans la traduction des équations récurrentes : la généralité des domaines potentiellement utilisés dans une définition récursive, pose encore de nombreux problèmes. Par contre, il est remarquable que, dans le processus de traduction, les opérations du formalisme sont traduites de manière très directe, notamment grâce à l'opération de changement de base, exprimée par un alignement en HPF. Le deuxième objectif, une fois le traducteur fiabilisé, est de faire indirectement de la transformation de programmes HPF. On pourrait en effet observer précisément l'impact des transformations d'énoncés PEI sur leurs traductions HPF.

Ces travaux sur l'utilisation du formalisme PEI font apparaître que la tâche à accomplir pour intégrer ces techniques dans un environnement réel de développement est encore immense. D'une part, il faut intégrer et adapter aux outils développés, les résultats théoriques existants. Les stratégies de transformations par exemple, doivent devenir des fonctionnalités à part entière de l'environnement de transformations. D'autre part, d'autres idées, comme le test d'équivalence forte entre programmes, présenté en annexe B, pourraient également venir enrichir l'aide qu'apporte ce type d'outil. D'autres travaux restent à mener pour utiliser la possibilité qu'offre HPF d'exprimer avec les indices des variables parallèles, non plus des processeurs virtuels, mais des éléments de tableaux. Nous avons abordé cet aspect dans le dernier chapitre, et nous espérons pouvoir améliorer la traduction afin de produire des textes de programmes utilisant cette notation plus abstraite. D'un programme PEI dont les variables sont alignées de manière à rendre les communications locales, nous obtiendrions alors un programme HPF dont les alignements sont presque cachés à l'utilisateur, n'apparaissant que sous forme de directives à l'extérieur des instructions de traitements.

Bibliographie

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and C.W. Tseng. The suif compiler for scalable parallel machines. In *Proceedings of the seventh SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[2] R. J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988.

[3] Guy E. Blelloch, Siddharta Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[4] Luc Bougé. *The Data Parallel Programming Model : a Semantic Perspective*, volume LNCS 1132, pages 4–26. Springer Verlag, May 1996.

[5] Luc Bougé. On the semantics of languages for massively parallel simd architectures. In *PARLE'91, LNCS*, pages 166–183. Springer-Verlag, 1991.

[6] Pierre Boulet. *Outils pour la parallélisation automatique*. PhD thesis, École Normale Supérieure de Lyon, Janvier 1996.

[7] Thomas Brandes. Compiling data-parallel programs to message passing programs for massively parallel mimd systems. In K. M. Decker and R. M. Rehmann, editors, *Working Conference on Massively Parallel Programming Models : suitability, Realization, and Performance*, September 1993.

[8] L. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, 1969.

[9] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Université de Rennes I, February 1993.

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design :* A foundation. Addison Wesley, 1988.

[11] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Michael B. Monagan, and Stephen M. Watt. *MAPLE Reference Manual.* University of Waterloo, Waterloo Maple Software, 1989.

[12] Marina Chen, Young il Choo, and Jingke Li. *Parallel Functional Languages and Compilers*, chapter Chapter 7. Frontier Series. ACM Press, 1991.

[13] Fabien Coelho. *Contributions to High Performance Fortran Compilation.* PhD thesis, Ecole des Mines de Paris, Octobre 1996.

[14] Alain Darte and Yves Robert. A graph theoretic approach to the alignement problem. Technical report, Ecole Normale Supérieure de Lyon, LIP, July 1993. RR93-20.

[15] Nills Ellmenreich, Peter Faber, Martin Griebl, Robert Günz, Harald Keimer, Wolfgang Meisl, Sabine Wetzel, Christian Wieninger, and Alexander Wüst. Loop parallelization in the polytope model. Technical report, University of Passau, April 1996.

[16] Paul Feautrier. Some efficient solution to the affine scheduling problem, part ii, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[17] Paul Feautrier. Towards automatic distribution. *Parallel processing letters*, 4(3):233–244, 1994.

[18] Paul Feautrier. Compiling for massively parallel architectures; a perspective. *Microprogramming and Microprocessors*, 1995. To appear.

[19] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transaction on Computer*, 21(9):948–960, 1972.

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's guide and Tutorial.* The MIT Press.

[21] Stéphane Genaud. Transformations d'énoncés PEI. *Technique et science informatiques*, 15(5):601–618, 1996.

[22] Stéphane Genaud, Eric Violard, and Guy-René Perrin. Transformations techniques in PEI. In Peter Magnusson, Seif Haridi, and Ali Khayri, editors, *LNCS*, volume 966, pages 131–142, Stockholm, Sweden, August 1995. EURO-PAR95 Parallel Processing, Springer-Verlag.

[23] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The John Hopkins University Press, 2nd edition, 1989.

[24] Martin Griebl and Jean-François Collard. Generation of synchronous code for automatic parallelization of `while` loops. In Peter Magnusson, Seif Haridi, and Ali Khayri, editors, *LNCS*, volume 966, Stockholm, Sweden, August 1995. EURO-PAR95 Parallel Processing, Springer-Verlag.

[25] Jonathan M. D. Hill. *Data-parallel lazy functional programming.* PhD thesis, Queen Mary and Westfield College, Department of Computer Science, Mile End Road, London, E1 4NS, UK, September 1994.

[26] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[27] HPF Forum. *High Performance Fortran Language Specification*, 1.0 edition, May 3 1993.

[28] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : An overview of the pips project. In *International Workshop on compilers for Parallel Computers*, pages 199–212, December 1990.

[29] Kenneth E. Iverson. *A Programming Language.* John Wiley and Sons, Inc., 1962.

[30] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library - Version 1.00*, April 1996. Interface Guide.

[31] E. Knapp. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Transactions on Programming Languages and Systems*, 12(2):203–223, April 1990.

[32] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, and Mary E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[33] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on computers*, 34(10):965–968, October 1985.

[34] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing, design and analysis of algorithms.* The Benjamin/Cummings Publishing Company, 1994.

[35] Vincent Lefebvre. Gestion de la mémoire dans les programmes parallélisés. Technical Report RR 96/17, Lab PRiSM - Université de Versailles, August 1996.

[36] Christian Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93, LNCS 715*, pages 398–416. Springer-Verlag, June 1993.

[37] V. Loechner and C. Mongenet. A toolbox for affine recurrence equations parallelization. In B. Hertzberger and G. Serazzi, editors, *Int. Conf. on High-Performance Computing and Networking*, pages 263–268. Springer Verlag, May 1995. Milan, Italy.

[38] MasPar Computer Corporation. *MasPar MP-1 MPL Programing Manuals*, December 1991.

[39] Christophe Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes I, 1989.

[40] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, 1993.

[41] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and iterations in a single assignement language: Reference Manual version 1.2.* Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[42] Message Passing Interface Forum. *MPI: A message-passing Interface Standard*, June 1995.

[43] Catherine Mongenet. Mappings for communications minimization using distribution and alignment. In Lubomir Bic, Wim Böhm, Paraskevas Evripidiou, and Jean-Luc Gaudiot, editors, *Proceedings of IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 185–193, Limassol, Cyprus, June 1995. ACM Press.

[44] Carroll Morgan. *Programming from specifications.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1990.

[45] Nicolas Paris. *HyperC documentation.* Hyperparallel Technologies, June 1993.

[46] Guy-René Perrin. Parallel programming with PEI. In Christian Lengauer, editor, *Dagsthul Seminar on Loop Parallelization*, April 1996.

[47] Serge Petiton and Nahid Emad. *A data parallel scientific computation introduction*, volume LNCS 1132, pages 45–62. Springer Verlag, May 1996.

[48] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing*, 1:95–113, 1989.

[49] Patrice Quinton, Sanjay Radjopadhye, and Doran Wilde. On deriving data parallel code from a functional program. In *Proceedings of the 9th International Symposium on Parallel Processing, IPPS'95*, pages 766–773, Santa Barbara, CA, April 1995. IEEE Computer Society Press.

[50] Patrice Quinton, Doran Wilde, and Sanjay Rajopadhye. Deriving imperative code from functional programs. In *Functional Programming Languages and Computer Architectures, FCPA'95*, pages 36–44. ACM Press, June 1995.

[51] Sanjay Rajopadhye. Analysis of affine communications specifications. In *IEEE Symposium on Parallel and Distributed Processing*, pages 530–537. IEEE Computer Society Press, December 1993.

[52] Robert S. Schreiber. *An introduction to HPF*, volume LNCS 1132, pages 27–44. Springer Verlag, May 1996.

[53] DPCE Subcommitte. Data-parallel c extensions. technical report 1.6 x3j11/94-080. Technical report, Numerical C Extensions Group of X3J11, 1994.

[54] Thinking Machines Corporation. *C* Programming Guide*, November 1990.

[55] Thinking Machines Corporation. *CM Fortran Reference Manual*, December 1992.

[56] Eric Violard. *Une théorie unificatrice pour la construction de programmes parallèles par des techniques de transformations.* PhD thesis, Université de Franche-Comté, Octobre 1992.

[57] Eric Violard. Data-parallelism versus functional programming : the contribution of PEI. Technical report, ICPS, January 1995. Publication interne 95-03.

[58] Eric Violard and Guy-René Perrin. PEI : a single unifying model to design parallel programs. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE 93, LNCS*, volume 694, pages 500–516. Springer-Verlag, June 1993.

[59] Eric Violard and Guy-René Perrin. Reduction in PEI. In Bruno Buchberger and Jens Volkert, editors, *LNCS*, volume 854, pages 112–123. CONPAR 94, Springer-Verlag, September 1994.

[60] Pierre Weis and Xavier Leroy. *Le langage* CAML. Interéditions - iia, 1993.

[61] Doran Wilde and Sanjay Rajopadhye. The naive execution of affine recurrence equations. In Peter Cappello, Catherine Mongenet, Guy-René Perrin, Patrice Quinton, and Yves Robert, editors, *Application Specific Array Processors,ASAP'95*, pages 1–12. IEEE Computer Society Press, August 1995.

[62] Doran Wilde and Sanjay Rajopadhye. Memory reuse analysis in the polyhedral model. In Luc Bougé, Pierre Fraignaud, Anne Mignotte, and Yves Robert, editors, *EURO-PAR'96 Parallel Processing*, volume I, pages 389–397. Springer-Verlag, LNCS 1123, 1996.

[63] Doran K. Wilde. A library for doing polyhedral operations. Technical report, IRISA, December 1993. PI 785.

Annexe A

Preuves des propriétés

Preuve. (propriété 1, page 23) Si $\mathtt{Y}$ est une expression fonction de $\mathtt{X}$, d'après la définition des opérations, il existe une bijection β telle que :

$$\sigma_{\mathtt{Y}} = \sigma_{\mathtt{X}} \circ \beta^{-1} \wedge dom(v_{\mathtt{X}}) \subseteq dom(\beta)$$

Montrons alors qu'il existe une bijection g telle que la paire de bijections $(g \circ \beta, g)$ vérifie les trois propositions (a), (b) et (c) :

- (a) s'écrit $\sigma_{\mathtt{X}} \circ \beta^{-1} \circ g^{-1} = \sigma_{\mathtt{Y}} \circ g^{-1}$, qui est vérifiée quelquesoit g.
- (b) s'écrit $dom(v_{\mathtt{X}}) \subseteq dom(g \circ \beta)$. D'après $dom(v_{\mathtt{X}}) \subseteq dom(\beta)$, on peut conclure que (b) est vérifiée pour toute bijection g telle que $img(\beta) \subseteq dom(g)$.
- (c) s'écrit $dom(v_{\mathtt{Y}}) \subseteq dom(g)$.

On peut donc conclure que toute bijection g telle que $img(\beta) \cup dom(v_{\mathtt{Y}}) \subseteq dom(g)$ satisfait les conditions.
□

Preuve.(propriété 2, page 31) Si $dom(\sigma_{\mathtt{X}}) \subseteq img(\mathtt{h})$, alors le champ de données $(\mathtt{h}^{-1} :: \mathtt{X})$ est correctement défini, car $dom(v_{\mathtt{X}}) \subseteq dom(\sigma_{\mathtt{X}}) \subseteq img(\mathtt{h})$. En le renommant $\mathtt{X}'$, nous avons alors :

$\mathtt{X}$
$= \mathtt{h} \circ \mathtt{h}^{-1} :: \mathtt{X}$ $\quad (dom(\sigma_{\mathtt{X}}) \subseteq img(\mathtt{h}))$
$= \mathtt{h} :: (\mathtt{h}^{-1} :: \mathtt{X})$
$= \mathtt{h} :: \mathtt{X}'$

Les champs $\mathtt{X}$ et $\mathtt{X}'$ sont fortement équivalents car $dom(\sigma_{\mathtt{X}'}) = dom(\sigma_{\mathtt{X}} \circ \mathtt{h}) \subseteq dom(\mathtt{h})$.
□

Preuve.(propriété 3, 32) Nous utilisons la définition 10. Soit $\mathtt{X'} = \mathtt{h} \circ \mathtt{h} :: \mathtt{X}$.

- $\mathtt{X}'$ est correctement défini. En effet, $\mathtt{X}'$ est correctement défini ssi $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h} \circ \mathtt{h})$. Par définition de la composition de fonctions $dom(\mathtt{h} \circ \mathtt{h}) = dom(\mathtt{h}) \cap \mathtt{h}^{-1}(dom(\mathtt{h}))$. Si $img(\mathtt{h}) \subseteq dom(\mathtt{h})$ alors $dom(\mathtt{h}) \cap \mathtt{h}^{-1}(dom(\mathtt{h})) = dom(\mathtt{h}) \cap \mathtt{h}^{-1}(img(\mathtt{h})) = dom(\mathtt{h})$. Nous avons bien $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})$ car nous avons $dom(v_{\mathtt{X}}) \subseteq dom(\sigma_{\mathtt{X}})$ ($\mathtt{X}$ est correctement défini) et $dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h})$ (hypothèse).

- $\mathtt{X}'$ est fortement équivalent à $\mathtt{h} :: \mathtt{X}$. En effet $\mathtt{X}'$ est fortement équivalent à $\mathtt{h} :: \mathtt{X}$ ssi $dom(\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) \subseteq dom(\mathtt{h})$. Nous avons bien $dom(\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) \subseteq dom(\mathtt{h}^{-1})$ et $img(\mathtt{h}) \subseteq dom(\mathtt{h})$ (hypothèse) impliquent $dom(\sigma_{\mathtt{X}} \circ \mathtt{h}^{-1}) \subseteq dom(\mathtt{h})$.

- Soit $e(\mathtt{X}') = \mathtt{h}^{-1} :: \mathtt{X'}$. Cette expression est correctement définie car $dom(v_{\mathtt{X}'}) \subseteq dom(\mathtt{h}^{-1})$, c'est-à-dire $dom(v_{\mathtt{X}} \circ \mathtt{h}^{-1} \circ \mathtt{h}^{-1}) \subseteq dom(\mathtt{h}^{-1})$ qui est vrai.

  Nous devons montrer $e(\mathtt{X}') = \mathtt{h} :: \mathtt{X}$. Nous avons bien
  $$\begin{array}{lcll} e(\mathtt{X}') & = & \mathtt{h}^{-1} :: \mathtt{X'} & \\ & = & \mathtt{h}^{-1} :: (\mathtt{h} \circ \mathtt{h} :: \mathtt{X}) & \text{(déf.)} \\ & = & \mathtt{h}^{-1} \circ \mathtt{h} \circ \mathtt{h} :: \mathtt{X} & \text{(7a),(7b)} \\ & = & \mathtt{h} :: \mathtt{X} & \end{array}$$

  D'autre part
  $$\begin{array}{lcll} e(\mathtt{h} :: \mathtt{X}) & = & \mathtt{h}^{-1} :: (\mathtt{h} :: \mathtt{X}) & \\ & = & \mathtt{h}^{-1} \circ \mathtt{h} :: \mathtt{X} & \\ & = & \mathtt{X} & (\text{car } dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h}) \text{ et } dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h})) \end{array}$$

  □

Preuve.(propriété 4, page 32)

(1) Soit $\mathtt{E}'$ le champ $(\mathtt{h}^{-1} :: \mathtt{E}) \lhd \mathtt{g}$. Supposons $\mathtt{h}(dom(\mathtt{g})) \subseteq dom(\sigma_{\mathtt{E}}) \subseteq img(\mathtt{h})$ et $dom(v_{\mathtt{E}}) = dom(\mathtt{g}) = \mathtt{h}(img(\mathtt{g}))$. Alors, $\mathtt{E}'$ est bien défini car $dom(v_{\mathtt{E}}) \subseteq img(\mathtt{h})$ (car $dom(v_{\mathtt{E}}) \subseteq dom(\sigma_{\mathtt{E}})$), $dom(\mathtt{g}) \subseteq \mathtt{h}^{-1}(dom(\sigma_{\mathtt{E}}))$ (car $\mathtt{h}(dom(\mathtt{g})) \subseteq dom(\sigma_{\mathtt{E}})$) et $img(\mathtt{g}) \subseteq \mathtt{h}^{-1}(dom(v_{\mathtt{E}}))$ (car $dom(v_{\mathtt{E}}) = dom(\mathtt{g})$).

Nous avons alors :

$$
\begin{array}{ll}
\mathtt{E} & \\
= \mathtt{h} \circ \mathtt{h}^{-1} :: \mathtt{E} & (\text{car } dom(\sigma_{\mathtt{E}}) \subseteq img(\mathtt{h})) \\
= \mathtt{h} :: (\mathtt{h}^{-1} :: \mathtt{E}) & \\
= \mathtt{h} :: ((\mathtt{h}^{-1} :: \mathtt{E}) \lhd \mathtt{g} \circ \mathtt{g}^{-1}) & (\text{car } dom(v_{\mathtt{E}}) = dom(\mathtt{g})) \\
= \mathtt{h} :: (((\mathtt{h}^{-1} :: \mathtt{E}) \lhd \mathtt{g}) \lhd \mathtt{g}^{-1}) & (\text{car } \mathtt{h}(dom(\mathtt{g})) \subseteq dom(\sigma_{\mathtt{E}})) \\
= \mathtt{h} :: (\mathtt{E}' \lhd \mathtt{g}^{-1}) &
\end{array}
$$

De la caractérisation de l'équivalence faible, E et E' sont faiblement équivalents car $dom(v_{\mathtt{E}'}) = dom(\mathtt{g}) = img(\mathtt{g}^{-1})$. Donc, d'après la définition 10, nous concluons $\mathtt{P} \sqsubseteq \mathtt{P}\{S[\mathtt{E} \backslash \mathtt{h} :: (\mathtt{E} \lhd \mathtt{g}^{-1})]\}$.

(2) Soit $\mathtt{E}'$ le champ $\mathtt{h}^{-1} :: (\mathtt{E} \lhd \mathtt{g})$. Supposons $dom(v_{\mathtt{E}}) = img(\mathtt{g})$ et $dom(\mathtt{g}) \subseteq dom(\sigma_{\mathtt{E}}) \subseteq img(\mathtt{h})$. Alors, $\mathtt{E}'$ est correctement défini car $dom(\mathtt{g}) \subseteq img(\mathtt{h})$ (h est une inverse droite de g), $dom(\mathtt{g}) \subseteq dom(\sigma_{\mathtt{E}})$ et $img(\mathtt{g}) \subseteq dom(v_{\mathtt{E}})$.

Nous avons alors :

$$
\begin{array}{ll}
\mathtt{E} & \\
= \mathtt{E} \lhd \mathtt{g} \circ \mathtt{g}^{-1} & (\text{car } dom(v_{\mathtt{E}}) = img(\mathtt{g})) \\
= (\mathtt{E} \lhd \mathtt{g}) \lhd \mathtt{g}^{-1} & (\text{car } dom(\mathtt{g}) \subseteq dom(\sigma_{\mathtt{E}})) \\
= (\mathtt{h} \circ \mathtt{h}^{-1} :: (\mathtt{E} \lhd \mathtt{g})) \lhd \mathtt{g}^{-1} & (\text{car } dom(\sigma_{\mathtt{E}}) \subseteq img(\mathtt{h})) \\
= (\mathtt{h} :: (\mathtt{h}^{-1} :: (\mathtt{E} \lhd \mathtt{g}))) \lhd \mathtt{g}^{-1} & \\
= (\mathtt{h} :: \mathtt{E}') \lhd \mathtt{g}^{-1} &
\end{array}
$$

De la caractérisation de l'équivalence faible, E et E' sont faiblement équivalents puisque $\mathtt{h}(dom(v_{\mathtt{E}'})) = \mathtt{h}(\mathtt{h}^{-1}(dom(\mathtt{g}))) = dom(\mathtt{g}) = img(\mathtt{g}^{-1})$.
Donc, $\mathtt{P} \sqsubseteq \mathtt{P}\{S[\mathtt{E} \backslash (\mathtt{h} :: \mathtt{E}) \lhd \mathtt{g}^{-1}]\}$ d'après la définition 10.

□

Annexe B

Test d'équivalence forte

Nous avons vu que la transformation de programmes est souvent fondée sur l'application de changement de base. Sans être spécifique à PEI, mais au contraire pouvant être fertile plus généralement en construction de programmes (par exemple de programmes HPF correspondants à des programmes PEI fortement équivalents), cette constatation peut mener à des pratiques intéressantes par "analogies" entre progammes, pour décider que deux programmes sont équivalents, ou bien valider une intuition qu'un programme donné calcule bien ce qu'on pense.

Nous exposons ici une solution permettant de déterminer, étant donné deux systèmes d'équations récurrentes ayant une structure équivalente (seules les fonctions de dépendances diffèrent) si l'un peut être obtenu de l'autre par un changement de base global sur le programme. Nous ne considérons ici que le cas d'un changement de base exprimé sous la forme d'une fonction affine, ainsi que des dépendances affines dans l'énoncé. Dans ces conditions, le problème revient à résoudre un système linéaire, pour lequel nous utilisons MAPLE[11].

Bien que cette idée ne soit qu'une ébauche de réflexion sur des tests d'équivalence plus généraux, cette fonctionnalité pourrait être implantée assez facilement dans l'environnement de transformations PEI.

Le formalisme utilisé pour représenter ces équations importe peu, et nous utilisons ALPHA dans cet exemple. Etant donné les deux programmes `Cholesky1` et `Cholesky2` suivants, nous cherchons à savoir si ils sont équivalents. Une condition suffisante est qu'il existe un changement de base permettant de passer de l'un à l'autre.

Nous dirons que l'énoncé `Cholesky2` peut être obtenu de `Cholesky1`, si il existe une bijection h telle que, pour toute dépendance f_i de `Cholesky1`, $f_i = h^{-1} \circ g_i \circ h$, où g_i est la dépendance correspondante dans `Cholesky2`. De plus, pour vérifier l'équivalence forte, une même relation doit exister entre les domaines des variables ($v = v' \circ h^{-1}$).

```
system Cholesky1 : { N | N>=1 }
               ( a : { i,j | 1<=i<=j; 1<=j<=N } of integer )
       returns ( l : { i,j | 1<=i<=N; 1<=j<=i } of integer );

var
   L  : { i,j,k | j<=i<=N; -1<=k<=j-1; 1<=j } of integer;

let
   L[i,j,k]  = case
               { | i=j; k=i-1; 1<=i<=N } : sqrt(L[i,j,k-1]);
               { | 2<=i<=N; 1<=j<=i-1; k=j-1 } : L[i,j,k-1] / L[j,j,j-1] ;
               { | 2<=i<=N; 2<=j<=i; 0<=k<=j-2 } : L[i,j,k-1] -
                L[i,k+1,k] * L[j,k+1,k] ;
               { | 1<=i<=N; 1<=j<=i; k=-1 } : a[j,i];
               esac;
   l[i,j] = L[i,j,j-1];
tel;
```

```
system Cholesky2 :{N | 1<=N}
               (a : {i,j | 1<=i<=j; j<=N} of integer)
       returns (l : {i,j | j<=i<=N; 1<=j} of integer);
var
  L : {i,j,k | 2j+k<=i<=j+k+N; (k+1,1)<=j; -1<=k} of integer;
let
  L[i,j,k] =
      case
        {| i=3k+2; j=k+1; 0<=k<=N-1} : sqrt(L[i-1,j,k-1]);
        {| 3k+3<=i<=2k+N+1; j=k+1; 0<=k} : L[i-1,j,k-1] / L[3j-1,j,j-1];
        {| 2j+k<=i<=j+k+N; k+2<=j; 0<=k} :
           L[i-1,j,k-1] - L[i-j+k+1,k+1,k] * L[j+2k+1,k+1,k];
        {| 2j-1<=i<=j+N-1; 1<=j; k=-1} : a[j,i-j-k];
      esac;
  l[i,j] = L[i+2j-1,j,j-1];
tel;
```

Nous pouvons exprimer le problème sous forme linéaire. Nous écrivons la dépendance affine f d'un n-uplet z sous la forme $Fz + t_f$, où F est une matrice constante et t_f un vecteur constant. La bijection que nous cherchons s'écrit alors $h(z) = Hz + t_h$ et son inverse $h^{-1}(z) = H^{-1}z - H^{-1}t_h$.

Vérifier qu'il existe un changement de base entre deux dépendances f et g consiste alors à vérifier l'égalité :

$$\begin{array}{rcl} Fz + t_f & = & H^{-1}(G(Hz + t_h) + t_g) - H^{-1}t_h \\ & = & H^{-1}GHz + H^{-1}Gt_h + H^{-1}t_g - H^{-1}t_h \end{array}$$

c'est-à-dire, vérifier d'une part :

$$F = H^{-1}GH$$

et d'autre part :

$$t_f = H^{-1}Gt_h + H^{-1}t_g - H^{-1}t_h$$

Le système à résoudre est consitué d'équations de ce type pour chacun des couples de dépendances f_i, g_i. La résolution est effectuée par MAPLE qui tente de déterminer les valeurs des inconnues. Sur cet exemple de Cholesky, le système a une solution qui est :

$$H = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ et } t_h = \begin{pmatrix} 3x \\ x \\ x \end{pmatrix}$$

Le fait que le vecteur de translation t_h ne soit pas complètement déterminé provient du fait que nous ne prenons pas en compte les domaines des variables. On peut vérifier que ces domaines vérifient la relation et conclure que ces deux programmes sont fortement équivalents.

Table des figures