RenPar'7,
Actes des 7^{es} Rencontres
Francophones du Parallélisme,
PIP–FPMs Mons, Belgique, 30 mai–2 juin 1995
Dekeyser, Libert et Manneback Editeurs

Techniques de transformation d'énoncés PEI pour la production de programmes data-parallèles

Stéphane Genaud
Université Louis Pasteur, Strasbourg I
Laboratoire ICPS, Pôle API, Bd S. Brant, F-67400 Illkirch
genaud@icps.u-strasbg.fr, http://opera.u-strasbg.fr/~genaud

1. Introduction

Nous présentons le formalisme PEI [2] et nous montrons qu'il représente un outil adéquat pour la spécification et la transformation de programmes data-parallèles. Le formalisme repose sur un modèle géométrique auquel est associée une sémantique bien définie, et les transformations basées sur un calcul de raffinement permettent de mettre en évidence l'équivalence de programmes. Les opérations de PEI permettant d'exprimer les concepts du modèle de programmation data-parallèle [3], nous pouvons transformer les programmes jusqu'à atteindre une version utilisant des primitives implantées efficacement.
L'article rappelle brièvement la définition du formalisme, puis décrit le calcul de raffinement régissant les transformations. Suit un exemple de transformation du programme produit matrice-vecteur : une première version utilise des communications explicitées par le programmeur pour faire la somme des produits, tandis que la version tranformée utilise un opérateur de réduction. Nous donnons pour ces deux versions un texte HPF [4] pour montrer la traduction possible dans un langage data-parallèle.

2. Définition du formalisme PEI

2.1. Spécifications et programmes

Un problème est spécifié en PEI par une relation entre des *multi-ensembles* de valeurs, appelés *entrées* et *sorties*. Ces valeurs sont représentées sur un domaine géométrique discret pour former un *champ de données*. Une spécification PEI est un système d'équations non-orientées, chacune définissant l'égalité d'expressions de champs de données. Nous dirons qu'une spécification est un *programme* si le système d'équations définit une fonction, c'est-à-dire qu'il a au plus une solution. Illustrons les opérations du langage avec l'exemple simple de l'addition d'une matrice et de sa transposée :

$$\mathtt{MATSUM:A} \mapsto \mathtt{C}$$
$$\begin{cases} \mathtt{A} = \mathtt{matrix} :: \mathtt{A} \\ \mathtt{T} = \mathtt{A} \lhd \mathtt{transp} \\ \mathtt{C} = \mathtt{add} \rhd (\mathtt{A}\ /\&/\ \mathtt{T}) \end{cases}$$
$$\begin{array}{lcl} \mathtt{matrix} & = & \lambda(\mathtt{i,j})\,|(\mathtt{0<=i,j<n}).(\mathtt{i,j}) \\ \mathtt{transp} & = & \lambda(\mathtt{i,j}).(\mathtt{j,i}) \\ \mathtt{add} & = & \lambda(\mathtt{a}\cdot\mathtt{b}).(\mathtt{a+b}) \end{array}$$

- la première équation est une pré-condition sur le champ de données `A`. Elle exprime la propriété d'invariance de `A` lorsqu'on lui applique le *changement de base* (noté `::`) `matrix`. Ce changement de base est l'identité sur le domaine carré `(0<=i,j<n)`.
- la deuxième équation définit `T` par une *opération de routage* (notée $\lhd$) appliquée sur `A`. Le routage exprime des dépendances sur le domaine géométrique : 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`.
- La troisième équation définit le champ de données en sortie `C`. La partie droite définit la *superposition* de `A` et `T` (notation `/&/`), *i.e.* l'ensemble des couples $(\mathtt{a}\cdot\mathtt{b})$, avec $\mathtt{a} \in \mathtt{A}$ et $\mathtt{b} \in \mathtt{T}$. L'*opération fonctionnelle* (notée $\rhd$) `add` est appliquée sur tous les éléments de ce champ.

Enfin, l'opération de routage, quelquesoit la fonction, a une opération inverse en PEI [1]. Cette opération appelée la *réduction géométrique* (notée $\mathtt{g} \cdot \rhd \mathtt{X}$) place en les points de coordonnées z, les séquences de valeurs de X situées en tous les points y ($y \in dom(\mathtt{g})$), tels que $\mathtt{g}(y) = z$.

Proposition 1. *Le champ de données* $\mathtt{g} \cdot\triangleright \mathtt{X}$ *est égal à* $\mathtt{X} \triangleleft \mathtt{inv(g)}$ *ssi* g *est bijective.*

2.2. Objets du langage

Les seuls objets manipulés en Pei sont les champs de données et les fonctions qui permettent d'écrire les opérations appliquées aux champs de données. La notation des fonctions en Pei est dérivée du lambda-calcul. Ainsi une fonction f de domaine $dom(\mathtt{f})$ =`{x |P(x)}` est notée λx $|$(P(x)).f(x). Par commodité, nous écrivons λx.f(x) lorsque nous avons λx $|$(true).f(x), et λx $|$(P(x)) pour λx $|$(P(x)).x. Enfin, une fonction f définie sur des sous-domaines disjoints est notée f1 + f2, et le domaine d'une fonction composée $\mathtt{f} \circ \mathtt{g}$ est $\{\mathtt{x} \in dom(\mathtt{g}) \mid \mathtt{g(x)} \in dom(\mathtt{f})\}$.

2.3. Sémantique

Un champ de données X est une paire de fonctions $(v : \sigma)$ où $v : Z^n \mapsto V$ est une représentation géométrique d'un multi-ensemble de valeurs V (telle que l'on puisse distinguer chacune des valeurs par des coordonnées entières), et σ est une bijection telle que $dom(v) \subset dom(\sigma)$. En plus de la *superposition*, l'opération interne associative notée /&/ qui forme des séquences à partir des valeurs de ses arguments, la sémantique des trois opérations externes sur le champ X= $(v : \sigma)$ est définie par :

- soit f une fonction de $dom(v) \mapsto W$, l'opération fonctionnelle $\mathtt{Y = f} \triangleright \mathtt{X}$ définit Y comme $(f \circ v : \sigma)$.
- soit g une fonction partielle de $Z^n \mapsto dom(v)$, alors l'opération de routage $\mathtt{Y = X} \triangleleft \mathtt{g}$ définit Y comme étant $(v \circ g : \sigma)$.
- Soit h une bijection de $dom(v) \mapsto Z^p$, le changement de base Y = h :: X définit Y comme étant $(v \circ h^{-1} : \sigma \circ h^{-1})$.

Le changement de base nous permet d'introduire l'équivalence de champs de données : pour un champ de données X et un changement de base h, nous dirons que X et h::X sont équivalents car les deux représentations sont celles d'un même multi-ensemble de valeurs.

3. Transformation de programmes

3.1. Raffinement

Le raffinement est un procédé classique de construction ou de tranformation de programmes [5], qui consiste à obtenir un programme satisfaisant toute spécification que satisfaisait le programme initial. Un tel procédé est généralement défini à partir de règles élémentaires. En Pei, ces règles sont les suivantes, où la notation $S \sqsubseteq S'$ signifie que S' raffine S et $S \equiv S'$ signifie que S et S' se raffinent mutuellement.

$\mathtt{Y=}\ f1 \triangleright (f2 \triangleright \mathtt{X})$	$\equiv$	$\mathtt{Y=}\ (f1 \circ f2) \triangleright \mathtt{X}$	(1)	$\mathtt{Y=}\ h :: (f \triangleright \mathtt{X})$	$\sqsubseteq$	$\mathtt{Y=}\ f \triangleright (h :: \mathtt{X})$	(7)
$\mathtt{Y=}\ \mathtt{X} \triangleleft (g1 \circ g2)$	$\equiv$	$\mathtt{Y=}\ (\mathtt{X} \triangleleft g1) \triangleleft g2$	(2)	$\mathtt{Y=}\ f \triangleright (h :: \mathtt{X})$	$\sqsubseteq$	$\mathtt{Y=}\ h :: (f \triangleright \mathtt{X})$	(8)
$\mathtt{Y=}\ \mathtt{X} \triangleleft (g1 + g2)$	$\equiv$	$\mathtt{Y=}\ (\mathtt{X} \triangleleft g1)\ \mathtt{/\&/}\ (\mathtt{X} \triangleleft g2)$	(3)	$\mathtt{Y=}\ (\mathtt{X1/\&/X2}) \triangleleft g$	$\equiv$	$\mathtt{Y=}\ (\mathtt{X1} \triangleleft g)\ \mathtt{/\&/}\ (\mathtt{X2} \triangleleft g)$	(9)
$\mathtt{Y=}\ (h1 \circ h2) :: \mathtt{X}$	$\equiv$	$\mathtt{Y=}\ h1 :: (h2 :: \mathtt{X})$	(4)	$\mathtt{Y=}\ h :: (\mathtt{X1/\&/X2})$	$\equiv$	$\mathtt{Y=}\ (h :: \mathtt{X1})\ \mathtt{/\&/}\ (h :: \mathtt{X2})$	(10)
$\mathtt{Y=}\ f \triangleright (\mathtt{X} \triangleleft g)$	$\sqsubseteq$	$\mathtt{Y=}\ (f \triangleright \mathtt{X}) \triangleleft g$	(5)	$\mathtt{Y=}\ (g1 + g2) \cdot\triangleright \mathtt{X}$	$\sqsubseteq$	$\mathtt{Y=}\ g1 \cdot\triangleright \mathtt{X}\ \mathtt{/\&/}\ g2 \cdot\triangleright \mathtt{X}$	(11)
$\mathtt{Y=}\ (h :: \mathtt{X}) \triangleleft g$	$\equiv$	$\mathtt{Y=}\ h :: (\mathtt{X} \triangleleft inv(h) \circ g \circ h)$	(6)	$\mathtt{Y=}\ (g1 \circ g2) \cdot\triangleright \mathtt{X}$	$\sqsubseteq$	$\mathtt{Y=}\ g1 \cdot\triangleright (g2 \cdot\triangleright \mathtt{X})$	(12)

La règle (3) est vraie si les fonctions g1 et g2 sont définies sur des domaines disjoints, la règle (4) dans le cas particulier où h1 et h2 sont des bijections, et les règles (6) et (7) sont vraies si h est bijective sur le domaine de X.

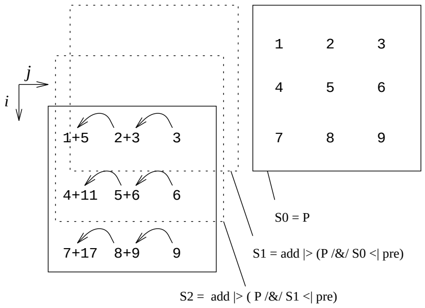
3.2. Stratégies de transformations

Nous montrons dans l'exemple suivant du produit matrice-vecteur comment remplacer des communications explicites lors de la sommation de n nombres, par un opérateur de communications plus abstrait qui est la réduction. Le premier programme est le suivant : à partir du vecteur V en entrée, le champ de données B est défini dans Z^2 comme étant égal au vecteur V sur la ligne $i = 0$. Les valeurs de B en cette ligne sont ensuite répliquées sur toutes ses lignes. La superposition de la matrice A et de B forme alors les couples nécessaires pour le produit des éléments. La seconde phase consiste à rassembler les produits et à les sommer pour obtenir le vecteur résultat. 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` (figure 1), puis les deux valeurs sont sommées pour n'en rendre qu'une.

$$
\texttt{MATVECT1:(A,V)} \mapsto \texttt{R}
\left\{
\begin{array}{l}
\texttt{V} = \texttt{align ::\ (B'} \lhd \texttt{line)} \\
\texttt{B} = \texttt{B'} \lhd \texttt{spread} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)} \\
\texttt{S} = \texttt{add} \rhd \texttt{(P /\&/ S} \lhd \texttt{pre)} \\
\texttt{R} = \texttt{S} \lhd \texttt{col}
\end{array}
\right.
$$

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

Figure 1. *Routage récursif des produits*



Pour traduire l'énoncé dans un langage data-parallèle nous devons trouver un mode opératoire permettant de calculer la solution de ce système d'équations. Nous devons vérifier pour cela que l'équation récurrente définissant `S` construit un ensemble bien-fondé. L'opération $\texttt{S} \lhd \texttt{pre}$ appliquée récursivement construit une suite monotone bornée (par le prédicat) et asssure cette propriété. En HPF nous aurions :

```
!HPF$ TEMPLATE T(n,n)
!HPF$ ALIGN A(i,j) WITH T(i,j)
!HPF$ ALIGN B(i,j) WITH T(i,j)
!HPF$ ALIGN V(j) WITH B(*,j)
      FORALL (i=0:n-1,j=0:n-1) P(i,j) = A(i,j) * B(i,j)
      FORALL (i=0:n-1,j=0:n-1) S(i,j)=P(i,j)
      DO k=n-1,0,-1
          FORALL (i=0:n-1,j=0:n-1,(j .LT. k)) S(i,j) = P(i,j) + S(i,j+1)
      ENDDO
      FORALL (i=0:n-1) R(i) = S(i,0)
```

Montrons à présent que la définition récursive du champ `S` peut s'abstraire en une définition non récursive.

Théorème 1 *Soit f une fonction définie par la récurrence $f = i + f \circ s^{-1}$, où i est une restriction de l'identité et s est bijective. L'équation* $\texttt{Y} = f \cdot \rhd \texttt{X}$ *est alors raffinée par la définition suivante, où le champ intermédiaire* `T` *est défini récursivement :* $\texttt{T} = \texttt{X /\&/ T} \lhd s$ *et* $\texttt{Y} = \texttt{T} \lhd i$

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

$$
\begin{array}{lcll}
\texttt{Y} = f \cdot \rhd \texttt{X} & \equiv & \texttt{Y} = (i + f \circ s^{-1}) \cdot \rhd \texttt{X} & (def.) \\
& \sqsubseteq & \texttt{Y} = i \cdot \rhd \texttt{X /\&/ } (f \circ s^{-1}) \cdot \rhd \texttt{X} & (11) \\
& \sqsubseteq & \texttt{Y} = i \cdot \rhd \texttt{X /\&/ } f \cdot \rhd (s^{-1} \cdot \rhd \texttt{X}) & (12) \\
& \equiv & \texttt{Y} = \texttt{X} \lhd i \texttt{ /\&/ } f \cdot \rhd (\texttt{X} \lhd s) & (prop.1) \\
& \sqsubseteq & \texttt{Y} = \texttt{X} \lhd i \texttt{ /\&/ } (\texttt{X} \lhd s) \lhd i \texttt{ /\&/ } f \cdot \rhd ((\texttt{X} \lhd s) \lhd s) & (\text{def.}[\texttt{X}\backslash(\texttt{X} \lhd \texttt{s})]) \\
& \equiv & \texttt{Y} = (\texttt{X /\&/ X} \lhd s \texttt{ /\&/ } (\texttt{X} \lhd s) \lhd s \texttt{ /\&/ } ...) \lhd i & (9) \\
& \equiv & \texttt{Y} = \texttt{T} \lhd i \ \text{ et } \texttt{T} = \texttt{X /\&/T} \lhd \texttt{s} &
\end{array}
$$

Nous pouvons maintenant écrire le programme suivant qui est raffiné par `MATVECT1` (il *abstrait* `MATVECT1`):

$$
\texttt{MATVECT2:(A,V)} \mapsto \texttt{R}
\left\{
\begin{array}{l}
\texttt{V} = \texttt{align ::\ (B'} \lhd \texttt{line)} \\
\texttt{B} = \texttt{B'} \lhd \texttt{spread} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)} \\
\texttt{S} = \texttt{sum} \rhd \texttt{(red} \cdot \rhd \texttt{P)} \\
\texttt{R} = \texttt{S} \lhd \texttt{col}
\end{array}
\right.
$$

$$
\begin{array}{l}
\texttt{red} = \lambda\texttt{(i,j)}\,|\texttt{(0<=i,j<n).(i,0)} \\
\texttt{sum} = \texttt{id} + \lambda\texttt{(a}\cdot\texttt{b).(a+sum(b))}
\end{array}
$$

Preuve. En réécrivant la fonction `red` comme suit :

$$\begin{array}{lcl}
\texttt{red} & = & \lambda(\texttt{i,j})\,|(\texttt{0<=i<n \& j=0}).(\texttt{i,0}) \;+\; \lambda(\texttt{i,j})\,|(\texttt{0<=i<n \& 0<j<n }).(\texttt{i,0}) \\
 & = & \texttt{col + } \lambda(\texttt{i,j})\,|(\texttt{0<=i<n \& 0<=j<n }).(\texttt{i,0})\texttt{ o } \lambda(\texttt{i,j})\,|(\texttt{0<=i<n \& 0<j<n }).(\texttt{i,j-1}) \\
 & = & \texttt{col + red o inv}(\lambda(\texttt{i,j})\,|(\texttt{0<=i,j<n-1}).(\texttt{i,j+1})) = \texttt{col + red o inv(pre)}
\end{array}$$

Soit `Y = red` $\cdot \rhd$ `P`, alors on déduit du thérorème 1 :

$$\left\{\begin{array}{l} \texttt{T} = \texttt{P/\&/T} \lhd \texttt{pre} \\ \texttt{Y} = \texttt{T} \lhd \texttt{col} \end{array}\right. \quad \text{et de } \texttt{MATVECT2}\text{, nous avons aussi} \quad \left\{\begin{array}{l} \texttt{S} = \texttt{sum} \rhd \texttt{Y} \\ \texttt{R} = \texttt{S} \lhd \texttt{col} \end{array}\right.$$

D'où : $\left\{\begin{array}{lcl} \texttt{S} & = & \texttt{sum} \rhd (\texttt{T} \lhd \texttt{col}) = (\texttt{sum} \rhd \texttt{T}) \lhd \texttt{col} \\ \texttt{R} & = & ((\texttt{sum} \rhd \texttt{T}) \lhd \texttt{col}) \lhd \texttt{col} = (\texttt{sum} \rhd \texttt{T}) \lhd \texttt{col} \end{array}\right.$

Il nous reste à montrer que la somme binaire `add` appliquée à une paire de valeurs, peut se généraliser en la somme `sum` d'une séquence de longueur quelconque. Pour cela, nous réécrivons la fonction `sum` en $\texttt{sum} = \texttt{id + } \lambda(\texttt{a}\cdot\texttt{b}).(\texttt{add}(\texttt{a}\cdot\texttt{sum}(\texttt{b})))$ et nous utilisons la proposition suivante :

Proposition 2. *Soit une fonction récursive* $\texttt{f = id + } \lambda(\texttt{a}\cdot\texttt{b})(\texttt{a} \otimes \texttt{f(b)})$*, où* $\otimes$ *est un opérateur binaire associatif. Alors,* $\texttt{f} \rhd (\texttt{X1 /\&/ X2}) \equiv \texttt{f} \rhd (\texttt{f} \rhd \texttt{X1 /\&/f} \rhd \texttt{X2})$

Alors $\texttt{sum} \rhd \texttt{T} = \texttt{sum} \rhd (\texttt{P /\&/T} \lhd \texttt{pre}) = \texttt{sum} \rhd (\texttt{sum} \rhd \texttt{P/\&/ sum} \rhd (\texttt{T} \lhd \texttt{pre}))$. Comme `P` n'a que des séquences d'un élément, $\texttt{sum} \rhd \texttt{P} = \texttt{P}$, et $(\texttt{P/\&/ sum} \rhd (\texttt{T} \lhd \texttt{pre}))$ a au plus deux éléments, on a $\texttt{sum} \rhd \texttt{T} = \texttt{add} \rhd \texttt{T} = \texttt{add} \rhd (\texttt{P /\&/ (sum} \rhd \texttt{T)} \lhd \texttt{pre})$.

En renommant $\texttt{sum} \rhd \texttt{T}$ en `S'` nous avons $\left\{\begin{array}{l} \texttt{S}' = \texttt{add} \rhd (\texttt{P /\&/ S'} \lhd \texttt{pre}) \\ \texttt{R} = \texttt{S'} \lhd \texttt{col} \end{array}\right.$

□

Nous obtenons ainsi un programme utilisant l'opérateur de réduction disponible dans le modèle data-parallèle. Le choix du chemin pour la réduction est cette fois-ci laissé au compilateur, sous l'hypothèse qu'il utilisera le chemin optimal pour une machine donnée.

```
!HPF$ TEMPLATE T(n,n)
!HPF$ ALIGN A(i,j) WITH T(i,j)
!HPF$ ALIGN B(i,j) WITH T(i,j)
!HPF$ ALIGN V(j) WITH B(*,j)
      FORALL (i=0:n-1,j=0:n-1) P(i,j) = A(i,j) * B(i,j)
      FORALL (i=0:n-1,j=0:n-1) S(i,0) = SUM(P(i,j))
      FORALL (i=0:n-1)  R(i) = S(i,0)
```

4. Conclusion

Nous avons donc montré l'intérêt de PEI pour valider de manière formelle des transformations de programmes pour atteindre l'énoncé désiré. Les transformations peuvent donc servir de manière générale, à obtenir un programme utilisant des primitives dont on sait qu'elles sont implantées de manière efficace dans le langage cible. Pour mener à bien l'écriture et la transformation de ces programmes, un environnement permettant d'appliquer interactivement les règles de transformations (PEI), ainsi qu'un outil de construction graphique de ces programmes (V.PEI) sont disponibles. Nos travaux en cours visent à traduire des programmes opérationnels PEI dans un langage data-parallèle comme HPF.

Bibliographie

1. E. Violard and G.-R. Perrin, *Reduction in* PEI., *CONPAR'94*, LNCS 854:112-123,1994.
2. E. Violard, *A mathematical theory and its environment for parallel programming.*,*PPL*, 4,1994.
3. L. Bougé et al., *Langages à parallélisme de données*,Technique et science informatiques, 12(6), Septembre 1993.
4. *High Performance Fortran Language Specification*, version 1.0 edition, May 1993.
5. C. Morgan, *Programming from specifications*, C.A.R. Hoare, Prentice Hall Ed., N.J., 1990.