Inférer rapidement la géométrie des collections

Jean-Louis Giavitto, Jean-Paul Sansonnet et Olivier Michel

LRI - U.A. 410 du CNRS,
Bât 490 Université de Paris Sud, F-91405 Orsay cedex
e-mail: giavitto@lri.lri.fr

Résumé : Nous présentons un système de typage et un algorithme efficace permettant d'inférer la structure des tableaux multidimensionnels dans un langage data-flow pour le parallélisme massif.

Mots-clefs : collections, géométrie d'une collection, analyse statique, inférence de type, langage déclaratif, programmation massivement parallèle.

1. Motivations

Un nouveau concept a été introduit dans certains langages de programmation parallèle, la **collection** [Blelloch, Sabot 90] [Sipelstein, Blelloch 91], afin d'exploiter les ressources de calcul des nouvelles architectures massivement parallèles (comme la CM5 [TMC 91]). Une collection correspond a un ensemble homogène et structuré de données manipulées comme un tout. Les éléments d'une collection sont distribués sur les processeurs, ce qui permet l'expression du *parallélisme de données* [Hillis, Steele 86].

L'expression du *parallélisme de contrôle* est habituellement réalisée à travers la notion de tâche. Ce concept, très utile pour gérer les activités concurrentes à gros grain, est mal adapté à l'expression d'un parallélisme implicite et à grain fin. C'est pourquoi on se tourne vers les langages déclaratifs dans lesquels il est facile de calculer automatiquement les dépendances entre opérations élémentaires. Cette approche n'est possible que dans la mesure où l'on développe des techniques de compilation efficace.

On est donc naturellement conduit à développer des langages data-flow permettant de manipuler des collections. *Otto e Mezzo* [Giavitto 91a] [Giavitto 91b], en abrégé : **8½**, est un exemple de cette approche qui permet à la fois l'expression du parallélisme de contrôle

et du parallélisme de données. Un programme dans ce langage correspond à un système d'équations :

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

à résoudre dans l'ensemble des « **streams de collections** » (un *stream* est une suite infinie de valeurs, Cf. [Ashcroft, Wadge 76] [Wadge, Ashcroft 86]). L'exécution d'un programme correspond à calculer séquentiellement, pour chaque indice d'un stream, les collections qui résolvent le système.

Dans cet article nous présentons un système de typage et un algorithme efficace pour le calcul de la géométrie des collections définies par certaines équations récursives. Nous adoptons ici le point de vue qu'une collection correspond à un tableau multidimensionnel. La géométrie d'une collection correspond alors à la description de la structure du tableau : nombre de dimensions et nombre d'éléments dans chaque dimension.

2. Collections

Pour des raisons de simplicité, nous oublions dans la suite l'aspect temporel d'un programme *Otto e Mezzo* : nous nous restreignons à des équations entre collections et celles-ci seront vues comme des tableaux multidimensionnels. Un tel tableau peut être considéré comme un « vecteur de vecteurs ». Un vecteur est un tableau mono-dimensionnel. L'imbrication des vecteurs est restreinte de manière à ce que l'on ne puisse construire que des vecteurs dont tous les éléments ont la même structure (propriété d'*homogénéité*).

La structure d'un vecteur est appelée, en 8 1/2, sa **géométrie** (ce qui correspond au *shape* de C* [TMC 88] ou à la *form* de MOA [Hains, Mullin 92]). Une géométrie est décrite par une liste constituée du nombre d'éléments dans chaque dimension de la collection (Fig. 1).

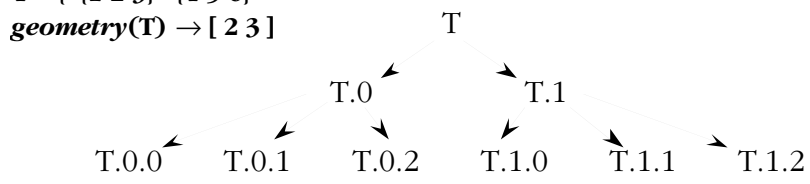

Figure 1: Exemple d'un vecteur imbriqué de dimension 2. Les accolades sont utilisées pour construire les collections, le point est un opérateur de projection. La géométrie d'un vecteur est spécifiée par une liste de nombres entre crochets.

Les opérations que nous considérons sur les collections sont un sous-ensemble de celles que l'on peut trouver dans APL ou *LISP. Nous les présentons informellement sans tenir compte des géométries. Les *opérations géométriques* modifient la structure d'une collection. La première opération est la *projection* qui permet d'accéder à un élément grâce à un index (qui doit être une constante entière) :

```
T = {a b c}
T.1 ⇒ b                    ("⇒" se lit "s'évalue en")
```

`T.1` est une collection contenant un seul élément (par convention, la notation scalaire « `b` » est équivalente dans ce contexte à « `{b}` », tableau à 1 élément). La valeur de ce point est la valeur du deuxième élément de T (l'indexation des points débute en 0). Les deux opérations géométriques suivantes, **`{}`** et **`#`**, permettent d'agréger et de concaténer deux collections :

```
U = {d e f}
{ T U } ⇒ { {a b c} {d e f} }    T # U ⇒ {a b c d e f}
```

Enfin, nous utiliserons une opération de troncature prenant en argument une géométrie, et qui s'applique sur chaque dimension d'un tissu :

```
T:[2] ⇒ {a b}      {T U}:[1 2] ⇒ { {a b} }
```

A ces opérations géométriques, il faut rajouter les opérations agissant sur les types scalaires (entier, réels, etc.), comme par exemple l'addition. La combinaison de ces opérations est réalisée de quatre manières différentes :

apply :	*f* (c_1, …, c_p)	application de fonction
alpha :	*f* ^ (c_1, …, c_p)	*map* LISP
beta :	*f* \ c	réduction
scan :	*f* \\ c	opération de préfixe

L'opérateur *apply* correspond au mécanisme standard d'application de fonction. L'opérateur *alpha* s'applique à une fonction entre scalaires et a pour résultat une fonction entre collections qui retourne la collection des résultats scalaires. La *beta-réduction* itère l'application d'une fonction dyadique et associative sur les éléments d'une collection (par exemple en utilisant l'addition, on obtient la somme de tous les éléments de l'ensemble ; en utilisant le minimum, on obtient l'élément minimal de la collection, etc.). L'opérateur de préfixe (ou *scan*) est similaire à la beta-réduction mais retourne une collection constituée des résultats partiels de la réduction (Cf. [Blelloch 89] pour un style de programmation basé sur les scans).

2.1 Un exemple

L'opérateur « `'` » permet de définir des vecteur dont la *i*-ème composante vaut *i-1* :

```
'5 ⇒ { 0 1 2 3 4 }
```

Nous utilisons cet opérateur pour écrire la définition d'une collection dont le *i*-ème élément vaut factorielle(*i*) :

```
index = '(n-1) + 1
fact = 1 # ((fact:[n-1] * index)
```

n est une constante entière. Le tissu `index` s'évalue en `{1 2 … n-1}`. Dans cette définition, la constante `1` qui apparaît le plus à droite correspond à une surcharge et doit en fait s'évaluer en un vecteur constant, dont chaque élément vaut 1 et dont la géométrie est `[n-1]` (Cf. la section suivante). La collection `fact` est définie récursivement. Si nous essayons d'évaluer la valeur de cette collection pour un élément *i*, on obtient :

```
fact.0    ⇒ (1 # ((fact:[n-1] * index)).0
          ⇒ {1}.0
          ⇒ 1
```

et pour `0<i≤n-1` :

```
fact.i    ⇒ ((fact:[n-1] * index)).(i - 1)
          ⇒ (fact:[n-1]).(i-1) * index.(i-1)
          ⇒ fact.(i-1) * ('(n-1) + 1).(i-1)
          ⇒ fact.(i-1) * ('(n-1).(i-1) + 1)
          ⇒ fact.(i-1) * i
```

ce qui correspond bien à la définition classique de la factorielle.

3. Typage des géométries

Les propriétés statiques (i.e. calculables avant l'exécution effective du programme) correspondent à une notion élargie du typage. Trois systèmes de types ont été développés pour 8½ afin de traiter les propriétés spatiales (la géométrie) et temporelles des « streams de collections ». Nous nous concentrons ici sur le système de typage utilisé pour calculer les géométries de chaque expression.

A priori, la géométrie d'une collection est explicitement définie à la PASCAL. Les géométries contraignent l'ensemble des expressions valides. Par exemple, il n'est pas possible d'additionner deux collections dont les géométries diffèrent :

```
R [10] = …     S [20] = …          T [10 2] = …

R + S ⇒ erreur: géométries incompatibles
R # T ⇒ erreur: le résultat n'a pas une géométrie homogène
```

Le système de type géométrique permet ces vérifications. Un autre avantage, considérable, est illustré par l'exemple suivant : l'opérateur "+" est un opérateur apparemment indifférent à la géométrie de ses arguments. Le calcul à la compilation de toutes les géométries permet de considérer ce polymorphisme apparent comme une *surcharge* (résolue à la compilation) et non comme un vrai opérateur polymorphe ad-hoc.

3.1 Inférer les géométries

Nous allons demander un peu plus à notre système de type. Tout d'abord, il serait désirable que la spécification de la géométrie dans la définition d'une collection soit

optionnelle. Notre seconde demande est de disposer de constantes polymorphes. Par exemple (introduit dans [Thatte 91]), l'anamorphose d'un vecteur s'écrit usuellement « a + b*V » où a et b sont des *constantes scalaires* et V un vecteur. Cette expression peut être écrite telle quelle en APL, mais en standard ML, en utilisant une liste V, la même expression se traduit par « map (op +)(distl (a, map(op *)(distl(b, V)))) » où distl est une primitive reprise de FP. Le gain en expressivité et le "confort" résultants des constantes polymorphes sont donc considérables.

La prise en compte de ces deux requêtes implique l'*inférence des géométries*. Nous rejetons une approche dynamique basée sur le calcul et la manipulation des géométries durant l'exécution. En effet, le calcul des géométries à la compilation fournit des informations très importantes pour la répartition des données et la gestion efficace du parallélisme.

3.2 Le système de type des géométries

Notre système de typage sera décrit en suivant le style de la "sémantique naturelle" [Kahn 87]. Les règles ci-dessous définissent les situations dans lesquelles la propriété « `e` a le type `H` dans l'environnement σ » est valide : ce qui se note « $\sigma \vdash$ `e:H` ». L'environnement σ assigne un type aux identificateurs libres dans `e`. Dans la suite, σ dénote un environnement de type ; `id` représente une variable et `t` une variable de type ; `e`, e_1, e_2 sont des expressions, `F`, `G`, `H` des expressions de types et `N`, `M` dénotent des entiers. Une géométrie est décrite par une liste d'entiers. N++G est la liste qui débute par N et se poursuit comme G (nous ne rentrons pas dans les détails techniques impliqués par l'identification des scalaires aux collections à un seul élément, Cf. [Hains, Mulin 92] pour une description plus précise de la représentation des géométries par des listes d'entiers).

Type d'un identificateur: $EG, id : G \vdash id : G$

Liaison: $\dfrac{EG, id : G \vdash e : G}{EG \vdash id = e : G}$ $\qquad$ $\dfrac{EG, id : G \vdash e : G}{EG \vdash id\,[G] = e : G}$

Arithmétique: $\dfrac{EG \vdash e : G \quad EG \vdash f : G}{EG \vdash e \text{ binop } f : G}$ $\qquad$ **Projection:** $\dfrac{EG \vdash e : G \quad 0 \le i < head(G)}{EG \vdash e\,.\,i : tail(G)}$

Application : $\dfrac{f(a, b) = e \quad EG, a : F, b : H \vdash e : G \quad EG \vdash g_1 : F, \; EG \vdash g_2 : H}{EG \vdash f(g_1, g_2) : G}$ $\qquad$ **Alpha:** $\dfrac{f(a, b) = e \quad EG, a : H, b : H \vdash e : G \quad EG \vdash g_1 : N{+}{+}H \quad EG \vdash g_2 : N{+}{+}H}{EG \vdash f\,\hat{}\,(g_1, g_2) : N{+}{+}G}$

« head » et « tail » calculent la tête et la queue d'une liste. Les clauses « `f (a, b) = e` » ou « $0 \le$ `i` < head(`G`) » ne sont pas des prémisses mais indiquent la condition d'application de la règle (i.e. f est une fonction de corps b et de paramètres formels a et b). La notation « `EG`,

`a : F` » dénote un environnement où la géométrie de `a` est `F` et où la les autres liaisons sont décrites par `EG`.

Réduction: $\dfrac{\begin{array}{c} f(a, b) = e \\ EG, a : F, b : F \vdash e : G \\ EG \vdash g : F \end{array}}{EG \vdash f \backslash g : G}$ **Scan :** $\dfrac{\begin{array}{c} f(a, b) = e \\ EG, a : F, b : F \vdash e : G \\ EG \vdash g : N{+}{+}F \end{array}}{EG \vdash f \backslash\backslash g : N{+}{+}G}$

Agrégation: $\dfrac{\begin{array}{c} EG \vdash e : N{+}{+}F \\ EG \vdash f : M{+}{+}F \end{array}}{EG \vdash e \# f : (N + M){+}{+}F}$ **Bl oc:** $\dfrac{EG \vdash e : G}{EG \vdash \{ e \} : 1{+}{+}G}$

4. Un algorithme pour l'inférence des types géométriques

L'implémentation de cet ensemble de règle n'est ni naturelle, ni efficace, en particuliers à cause des grandes difficultés créées par les définitions récursives. Aussi nous proposons une autre représentation du problème, qui va nous permettre de le résoudre très efficacement. Pour des raisons de place, nous n'exposerons pas le traitement des applications de fonction. Le lecteur intéressé pourra se reporter à [Giavitto 92].

L'idée est la suivante. Les règles d'inférences précédentes peuvent se dériver afin d'associer à chaque programme correct deux systèmes d'*équations linéaires*. Le premier système permet de calculer la dimension de chaque collection (Cf. fig. 2) Les notations utilisées sont les suivantes : une expression dénotée par E a pour dimension e et pour géométrie : $[e_1\ e_2\ \ldots e_e]$.

E = ID	e = id	E = {F, G, …}	$\begin{cases} e = f + 1 \\ e = g + 1 \\ \ldots \end{cases}$
E = F.n	e = f - 1		
E = F binop G	$\begin{cases} e = f \\ e = g \end{cases}$	E = F # G	$\begin{cases} e = f \\ e = g \end{cases}$

Figure 2: Équations entre les dimensions
(Les expressions sont en lettres capitales, la dimension correspondante en minuscule)

Une fois les dimensions de chaque tissu calculées (les équations étant linéaires, il existe un algorithme de résolution), on peut former les équations entres les cardinaux d'un tissu (le *i*-ème cardinal correspond au nombre d'éléments dans la *i*-ème dimension). Les équations entre cardinaux sont présentées en figure 3 et dérivent des règles de typages données à la section précédente.

Lors de la résolution du premier système d'équations entre dimensions, trois cas peuvent se produire: (1)*il n'y a pas de solution* : dans ce cas le programme est incorrectement typé ;

(2) *il y a une unique solution* : on peut poursuivre le typage des géométries en formant et en résolvant les équations entre cardinaux ; (3) *il y a une infinité de solutions* : le programme est ambiguë, le programmeur doit préciser la géométrie de certaines collections. Lors de la résolution du système d'équations entre cardinaux, il y a à nouveau trois cas possible : le système est correctement typé si et seulement si il existe une solution unique.

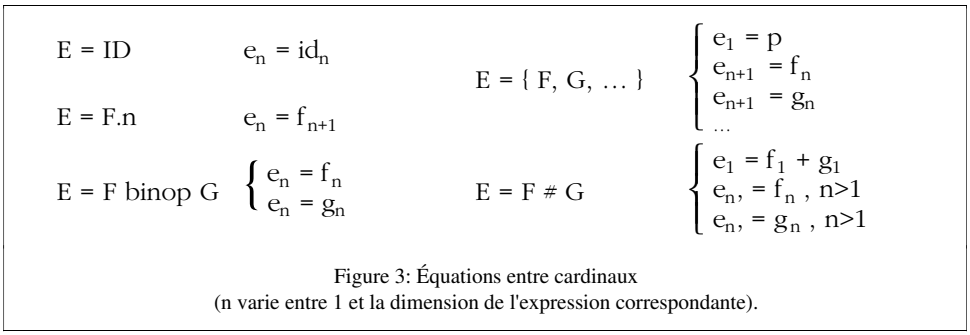

Figure 3: Équations entre cardinaux
(n varie entre 1 et la dimension de l'expression correspondante).

Le problème est de résoudre efficacement ces systèmes linéaires qui peuvent être grands mais qui sont très creux. Un examen de la forme des équations impliquées nous permet de traduire la résolution de ces systèmes en une descente dans un certain graphe associé à chaque système. Cette procédure de résolution est très efficace et on peut montrer que l'algorithme obtenu est en O(n^2) où n est le nombre de collections (Cf. [Giavitto 92]).

5. Conclusion

Le but de cet article est de proposer un mécanisme efficace d'inférence de la géométrie des tableaux dans un langage fonctionnel. Les expressions permises entre collections forment un sous-ensemble des expressions APL (par exemple) mais couvrent bien les opérateurs que l'on retrouve dans les langages massivement parallèles. De plus, notre algèbre de tableaux permet les définitions récursives. Plusieurs langages de collections existent, mais à notre connaissance, aucun permettant des géométries implicites avec une gestion statique (par exemple en FORTRAN90 les constantes sont polymorphes mais la géométrie de chaque tableau doit être explicite).

La gestion des géométries est effectuée à travers un typage statique. De ce point de vue, les constantes et les opérations prédéfinies sont surchargées et leur sémantique est déduite du contexte. Les règles de typage ont été données dans le style de la sémantique naturelle. Mais la vérification des types et leur inférence est réalisée plus efficacement à travers la détermination de la dimension et des cardinaux de chaque collection. Le point clé est que les opérations géométriques se traduisent par des combinaisons linéaires des dimensions et des cardinaux. Cette propriété ne peut plus être assurée quand les fonctions ne sont plus du premier ordre (excepté pour nos quatre modes d'applications de fonctions). Cependant les définitions récursives sont possibles, ainsi que nous l'avons vu.

[Thatte 91] propose une approche très différente du typage des géométries, à travers le concept de *coercition* (en anglais "coercion"). Le sous-typage est utilisé pour capturer la notion de "scaling" (l'alpha-extension en est un exemple). Un des points intéressants de cette approche est que les coercitions sont inférées comme une composition de quatre opérateurs originaires de FP : •, α, *distl* ,*distr* et *trans*. La résolution du typage demande des techniques sophistiquées et coûteuses mais peut prendre en compte des fonctions d'ordre supérieur. Cependant le langage de base ne contient pas de primitive d'alpha-extension explicite (ce qui serait incompatible avec les algorithmes développés), ce qui conduit à des coercitions non désirées (par exemple {{1 2} {3 4}} # {{5 6} {7 8}} est interprété comme {{1 2 5 6} {3 4 7 8}} et non comme {{1 2} {3 4} {5 6} {7 8}}).

6. Références

[Ashcroft, Wadge 76] E.A. Ashcroft and W. W. Wadge, *Lucid - A formal system for writing and proving programs*, SIAM Journal on Computing (3):336-354, September, 1976.

[Backus 78] J. Backus, *Can programming be liberated from the Von Neumann style ? A functionnal style and its algebra of programs*, Com. ACM Vol. 21, N° 8, pp 613-641, August 1978

[Blelloch 89] Guy E. Blelloch, *Scans primitive Parallel Operations*, IEEE Transactions on Computers, Vol. 38, N° 11, November 1989.

[Blelloch, Sabot 90] Guy E. Blelloch and Gary W. Sabot, *Compiling Collection-oriented Languages onto Massively Parallel Computers*, Journal of Parallel and Distributed Computing, 8, 119-134 (1990)

[Giavitto 91] Jean-Louis Giavitto, *A synchronous data-flow language for massively parallel computers*, Proc. of ParallelComputing'91, 3-6 September 1991, London.

[Giavitto 92] Jean-Louis Giavitto, *Typing Geometry of Homogeneous Collection*, Proc. of the 2th Int. Workshop on array, ATABLE-92, Montréal 29June-1 July 1992

[Hains, Mullin 92] Gaétan Hains, Lenore Mullin, *Parallel Functional Programming with Arrays*, internal report, U. Montréal.

[Hillis, Steele 86] W. Daniel Hillis, Guy L. Steele, *Data parallel Algorithms*, Communication of the ACM, Vol. 29, N° 12, December 1986

[Kahn 87] Gilles Kahn, *Natural Semantics*, in Proc. of the Symposium on Theoretical Aspect of Computer Science, Passau Germany, February 1987, also available as INRIA report N° 601, February 1987.

[Sipelstein, Blelloch 91] Jay Sipelstein, Guy Blelloch, *Collection-oriented Languages*, Proc. of the IEEE, Vol. 79, N° 4, Avril 1991, pp 504-523.

[Thatte 91] Satish Thatte, *A type system for implicit scaling*, Science of computer programming, 17 (1991), 217-245

[TMC 86] Thinking Machine Corporation, *The Essential *Lisp Manual*, Cambridge, 1986

[TMC 91] Thinking Machine Corporation, *The Connection-Machine CM5 technical Summary*, October 1991

[Wadge, Ashcroft 85] W. W. Wadge and E.A. Ashcroft, *Lucid, the Data flow Programming Language*, Academic Press U.K., 1985.