RAIRO. INFORMATIQUE THÉORIQUE ET APPLICATIONS

P. JOUVELOT
P. FEAUTRIER

Parallélisation sémantique

RAIRO. Informatique théorique et applications, tome 24, n° 2 (1990), p. 131-159

<http://www.numdam.org/item?id=ITA_1990__24_2_131_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 24, n° 2, 1990, p. 131 à 159)

PARALLÉLISATION SÉMANTIQUE (*)

par P. JOUVELOT (1) et P. FEAUTRIER (2)

Communiqué par J.-E. PIN

Résumé. – *La parallélisation de programmes séquentiels est un des moyens permettant de profiter facilement des avantages architecturaux offerts sur les nouvelles machines parallèles. Nous proposons d'améliorer cette technique en introduisant la notion de* parallélisation sémantique. *Notre principe consiste à voir les transformations de programmes introduites par la parallélisation comme définissant des sémantiques dénotationnelles non standards du langage de programmation. Nous montrons comment utiliser ce concept pour détecter, dans un langage impératif simplifié ALL, des instructions complexes parallélisables, prendre en compte certains programmes avec indirections et reconnaître des réductions. Notre approche permet ainsi de paralléliser des programmes qui ne pouvaient être traités avec les techniques existantes. De plus, d'un point de vue théorique, en utilisant les acquis de la théorie des domaines et la notion d'interprétation abstraite, nous donnons des preuves de correction de ces transformations vis-à-vis de la sémantique standard du langage ALL. Une des retombées indirectes de notre approche découle de ce que la donnée d'une spécification dénotationnelle définit également un prototype exécutable de cette spécification. Nous avons ainsi implémenté une maquette de paralléliseur sémantique en utilisant le langage fonctionnel ML. Des exemples d'exécution sont donnés.*

Abstract. – *Detecting parallelism in sequential programs is one of the techniques that allow one to efficiently use the new supercomputers. Our purpose is to extend the usefulness of this method by using concepts from denotational semantics. The transformations that are used to construct parallel programs are seen as non-standard semantics of the source programming language. This concept is used to detect complex parallel features in a simple imperative language, to deal in a limited way with indirect indexing and to detect reductions. Programs that were left sequential by current techniques are now parallelized. Furthermore, by using the results of domain theory and the concept of abstract interpretation, it is possible to give correctness proofs for our transformations. Another by-product comes from the fact that a denotational specification is tantamount to a high level program. We have been able to implement a prototype of a semantical parallelizer in the ML functional language and to apply it to several examples.*

(*) Reçu novembre 1987, version finale en avril 1989. Ce travail de recherche a été effectué au Laboratoire M.A.S.I.

(1) C.A.I., École des Mines de Paris, 60, boulevard Saint-Michel, 75272 Paris, France et L.C.S., Massachusetts Institute of Technology, 545, Technology Square, Cambridge, MA 02139, U.S.A..

(2) Laboratoire M.A.S.I., Université Paris-VI, 4, place Jussieu, 75252 Paris Cedex 05, France.

Informatique théorique et Applications/Theoretical Informatics and Applications
0988-3754/90/02 131 29/$4.90/

1. INTRODUCTION

Deux approches logicielles ont été, jusqu'à présent, proposées pour utiliser pleinement les nouvelles architectures parallèles de machines.

La première consiste à concevoir de nouveaux langages de programmation dans lesquels le parallélisme est explicitement déclaré par le programmeur. Ainsi, on trouve, par exemple; Actus [29], VectorC [24] ou Fortran8X pour ce qui concerne la programmation impérative; *Lisp [36] ou MultiScheme [26] dans une optique fonctionnelle; ou Parlog [5] pour la programmation logique.

La seconde approche du parallélisme essaye de compiler les langages séquentiels classiques en détectant automatiquement le parallélisme (implicite) présent dans les programmes. Cette orientation a été essentiellement (mais voir également FX-87 [13] pour une tentative dans le domaine fonctionnel) adoptée pour Fortran à la suite des travaux de Kuck [23] et Kennedy [2] et a conduit à l'émergence de compilateurs *vectoriseurs* industriels pour des machines de la classe du Cray-1.

Cette approche est usuellement justifiée par, d'une part, l'existence d'un énorme stock de programmes séquentiels et, d'autre part, la rareté des spécialistes capables de concevoir directement leurs applications en parallèle. Elle est donc ordinairement vue comme une solution temporaire. Nous pensons, au contraire, que les méthodes développées pour la parallélisation automatique ont de fortes chances de se retrouver dans les compilateurs des futurs langages spécialisés. En effet, vue la grande variété des architectures parallèles, il paraît illusoire de demander à un programmeur d'exhiber tout le parallélisme contenu dans son programme. Lui demander de faire un choix revient à lui faire intégrer dans son code les caractéristiques de la machine cible, ce qui va à l'encontre de toute l'évolution de la programmation moderne. Il paraît plus raisonnable de laisser ce travail à la charge d'un compilateur, qui devra pour cela utiliser des méthodes du type de celles que nous présentons ici.

Notre conviction est que pour atteindre un tel objectif une approche très fine du programme source est nécessaire; nous proposons pour cela d'introduire la notion de *parallélisation sémantique.*

- Notre premier objectif est d'extraire automatiquement, au moment de la compilation, des *informations sémantiques* à partir du texte du programme. Nous les utiliserons pour implémenter des techniques sophistiquées de détection de constructions parallélisables à l'intérieur d'un programme séquentiel. Il sera alors possible de paralléliser des fragments de programmes qui n'étaient pas détectés comme parallélisables avec les méthodes plus classiques.

• Pour exprimer ce processus complexe d'extraction d'informations (*i. e.*, compréhension) de programmes, nous avons besoin d'un cadre théorique puissant et riche. Notre second principe sera d'utiliser des spécifications inspirées des techniques *dénotationnelles*. Nous considérerons la parallélisation de programmes comme une interprétation non standard du langage de programmation. A l'aide de la théorie des domaines et du formalisme de l'interprétation abstraite, nous serons alors à même de prouver la correction de nos méthodes de parallélisation par rapport à la sémantique standard du langage [20]. A noter que cette notion de correction des transformations de programmes est généralement peu abordée par les chercheurs travaillant dans le domaine de la parallélisation; nous proposons donc ici une méthode possible pour aborder ce point important.

• Le dernier aspect de notre approche est plus pragmatique. Il est bien connu qu'une spécification dénotationnelle d'un langage de programmation donne, gratuitement, un interprète *exécutable* de ce langage. En utilisant un langage fonctionnel d'ordre supérieur (en pratique, ML), nous avons implémenté directement à partir de nos spécifications certaines de nos techniques de parallélisation dans un prototype de *paralléliseur sémantique*.

Le reste de cet article est divisé en quatre sections.

La section 2 présente informellement le langage-objet ALL (*i. e.*, celui sur lequel les transformations seront appliquées) et donne un exemple complet de programme traité par notre maquette de paralléliseur, motivant ainsi la notion de parallélisation sémantique.

La section 3 introduit les notations et principes qui sont à la base de notre approche. Nous donnons la syntaxe et la sémantique formelles de notre langage-objet. Nous redéfinissons les Conditions de Bernstein [3], qui sont à la base de toutes les techniques de parallélisation, en utilisant notre nouveau point de vue. Nous présentons le cadre de l'interprétation abstraite dénotationnelle que nous utiliserons pour décrire nos méthodes de parallélisation. Enfin, nous discutons les problèmes pratiques que pose l'exécution directe de nos spécifications dénotationnelles de parallélisation.

Dans la section 4, nous donnons trois exemples détaillés d'interprétation abstraite de notre langage. Ces interprétations nous permettront d'améliorer sensiblement le processus de parallélisation grâce aux informations qu'elles synthétisent quant au comportement dynamique du programme à paralléliser. En pratique, nous introduisons les notions de *prédicats*, de *t-prédicats* et d'*expressions symboliques*, montrons comment elles permettent d'approcher plus finement le comportement dynamique d'un programme et explicitons le mécanisme de parallélisation qui en découle.

La dernière section présente notre implémentation d'un prototype de paralléliseur sémantique à l'aide du langage ML et en discute les problèmes pratiques.

2. MOTIVATIONS

2.1. Le langage ALL

Notre langage-objet ALL ([3]) [19] est un langage impératif séquentiel classique. Sa syntaxe est analogue à celle de Pascal, mais le caractère statique de la gestion mémoire le rapproche plutôt de Fortran. Nous ne nous intéresserons ici qu'à la parallélisation de boucles for à la Fortran ou de séquences d'instructions.

Outre les constructions séquentielles classiques, ALL possède trois constructions parallèles simples; elles ne sont pas supposées être utilisées par le programmeur, mais introduites par le paralléliseur sémantique, après détection des portions de programme parallélisables. Pour définir notre modèle de machine parallèle, nous utilisons le schéma standard du *paraordinateur* [30] avec un nombre infini de processeurs et une mémoire globale partagée de taille infinie.

- $|[\mathbf{C}_1; \ldots; \mathbf{C}_m]|$ évalue en parallèle les m commandes $\mathbf{C}_i$;
- **for // I to E do C** exécute les diverses itérations **C** de la boucle en parallèle pour **I** variant de **1** à **E**; **I** est supposé être une variable locale propre à chaque processeur (située dans une mémoire privée, par exemple);
- $\mathbf{I}_1 := \mathbf{for\_red}\ \mathbf{I}_2\ \mathbf{to}\ \mathbf{E}_1\ \mathbf{eval}\ \mathbf{E}_2$ réalise une réduction au sens APL du terme. Du point de vue du résultat final, un **for_red** est équivalent à :

$$\mathbf{for}\ \mathbf{I}_2\ \mathbf{to}\ \mathbf{E}_1\ \mathbf{do}\ \mathbf{I}_1 := \mathbf{E}_2$$

mais l'opération représentée par $\mathbf{E}_2$ est garantie associative, ce qui permet une implémentation parallèle efficace en $\mathcal{O}(\log \mathbf{E}_1)$. $\mathbf{I}_2$ est une variable locale. Par exemple,

$$[\mathbf{s} := \mathbf{0};\ \mathbf{s} := \mathbf{for\_red\ i\ to\ 10\ eval\ (s + t[i])};]$$

retourne dans **s** la somme des dix premiers éléments du tableau **t**.

([3]) A Little Language.

Bien évidemment, il existe d'autres constructions parallèles (e. g., DO-ACROSS sur l'Alliant FX8 [35]). Néanmoins, comme elles ne sont pas encore très répandues, nous avons décidé de limiter notre étude aux primitives aisément implémentables à l'aide des classiques FORK et JOIN.

Notre objectif est d'accroître le degré de parallélisme des programmes ALL. Avec nos hypothèses, cela revient à remplacer des instructions séquentielles par leur version parallèle. Bien évidemment, nous ne devons opérer cette transformation que si la sémantique séquentielle du programme initial est préservée. Les *Conditions de Bernstein* [3] nous donnent une condition suffisante permettant de le garantir; informellement, deux instructions peuvent être exécutées en parallèle si il n'y a pas de zone mémoire que l'une écrit et que l'autre lit ou écrit.

2.2. Un exemple développé

Avant de fournir une présentation complète de notre méthode de parallélisation sémantique, nous allons l'introduire par un exemple obtenu sur notre maquette de paralléliseur en ML. La fonction **parallel_pp** prend en entrée un texte de programme ALL (entre guillemets) et retourne une chaîne représentant sa version parallélisée :

```
# parallel-pp "
  array t[100]; array u[100];
  var i; var n; var m;
  read n;
  s := 0;
  if n > 10 then
    n := n + 1;
  else
    n := 10;
  for i to 10 do
    [
    t[i] := t[i + n];
    t[i + n + 20] := 0;
    u[2 * i + 51] := u[2 * i] + 50;
    s := s + t[i + n];
    ]";;
  array t[100]; array u[100];
  var i; var n; var m;
  |[
```

```
    read n;
    s := 0;
  ]|
  if (n) > (10) then
    n := (n) + (1);
  else
    n := 10;
  |[
    s := for_red i to 10 eval (s + t[(i) + (n)]);
    [
    for// i to 10 do
      |[
        t[i] := t[(i) + (n)];
        t[((i) + (n)) + (20)] := 0;
        u[((2) * (i)) + (51)] := u[((2) * (i)) + (50)]
      ]|
    i := 11;
    ]
  ]|
: string
```

Cette exécution illustre deux des trois aspects que nous aborderons par la suite.

Pour paralléliser la boucle sur **i** en préservant la sémantique séquentielle du programme, il est nécessaire de se rendre compte que les valeurs possibles de **n** en entrée de boucle sont toujours supérieures à 10. En effet, dans ce cas, il n'y a pas de conflit possible entre les éléments du tableau **t** accédés en lecture et écriture par la première instruction de la boucle. Les paralléliseurs actuels ne sont pas à même de traiter ce type de programme; ceci nécessite de la part du paralléliseur une « connaissance » approfondie du programme à paralléliser. Nous verrons comment la notion de *prédicats* permet de lever cette limitation et nous généraliserons ce concept au cas des prédicats sur tableaux, les *t-prédicats* (ce dernier point n'est pas apparent sur cet exemple — voir ci-dessous). Sur cet exemple, notre paralléliseur a obtenu cette information automatiquement par analyse du texte du programme précédant la boucle (par propagation de prédicats faisant intervenir **n**).

La propriété sur **n** n'est pas suffisante pour paralléliser la boucle dès que l'on prend en compte les autres instructions du corps de boucle. En effet, une autre raison pour laquelle la boucle sur **i** n'est pas parallélisée par les techniques actuelles provient du calcul de **s**. Pour se rendre compte que le

calcul de **s** correspond à une réduction sur le tableau **t** (**s** est la somme d'un sous-tableau de **t**) et peut donc être sortie de la boucle, il est à la fois nécessaire de détecter la présence d'une opération associative sur **s** et de prouver que les éléments de **t** concernés ne sont pas modifiés par d'autres instructions de la boucle. Nous verrons comment ceci peut être obtenu à l'aide d'une *évaluation symbolique* que nous décrirons.

3. PRINCIPES

Cette section introduit formellement les concepts qui sont à la base de la méthode de parallélisation sémantique.

3.1. Notations

Nous utilisons les notions classiques utilisées dans la théorie standard des domaines (*voir* par exemple [15]) que nous supposons connue. Les notations définies ci-dessous sont inspirées du mangage ML. Les domaines sémantiques seront écrits en *italique* (e.g., *Bool*, *Int*) tandis que les domaines syntaxiques utilisent la même police en romain gras (**e.g., Exp, Ide**). Le produit cartésien de deux domaines D_1 et D_2 est noté $D_1 \# D_2$; il contient des paires (x_1, x_2) d'éléments x_i de D_i. Cette notation est trivialement étendue aux listes $[d_1; \ldots; d_m] \in D^*$ d'éléments de D où @ représente l'opérateur de concaténation. Le domaine des fonctions continues ayant D_1 comme ensemble de départ et D_2 comme image est noté $D_1 \to D_2$; nous noterons $f\,[y/x]$ la fonction en tout point égale à f sauf en x où elle retourne y. Nos fonctions seront généralement currifiées et nous utilisons $[\![\ ]\!]$ pour distinguer les arguments qui sont des objets de la syntaxe abstraite du langage. Pour simplifier nos formules, nous ne précisons pas les opérateurs de projection et de plongement sur une somme jointe $D_1 + D_2$ de domaines D_1 et D_2; si nécessaire, ils peuvent être facilement reconstruits par une simple comparaison de types.

3.2. Le langage ALL

ALL [19] est un langage impératif séquentiel à la Pascal ou Fortran. Nous ne nous intéresserons ici qu'à la parallélisation (4) de boucles **for** à la Fortran ou de séquence d'instructions. Néanmoins, nous pourrons donner

(4) Ou vectorisation, qui est vue ici comme une version restreinte de la notion de parallélisation.

certaines de nos spécifications à l'aide de la boucle **while** si cela simplifie leur présentation. Toute boucle **for** peut être trivialement remplacée par une boucle **while**.

La syntaxe abstraite du langage ALL est donnée ci-dessous dans une notation BNF classique (sa syntaxe concrète n'est pas précisée car elle est triviale) :

P : : = D; C
D : : = var I | array I[K] | D_1; D_2
E : : = K | I | O1 E | E_1 O2 E_2 | I[E]
C : : = nop | I: = E | I[E_1]: = E_2 | for I to E do C | read I
while E do C | [C_1; ... ; C_m] | if E then C_1 else C_2
| [C_1; ... ; C_m] | | for //I to E do C | I_1: = for_red I_2 to E_1 eval E_2

où **K** est un élément du domaine **Con** des constantes, **I** est un identificateur du domaine **Ide**, **O1** (resp. **O2**) est un opérateur unaire (resp. binaire) et E_i représente une expression (**Exp**); un programme **P** est alors une paire formée d'une déclaration **D** et d'une commande **C** (**Com**). Nous ne préciserons pas ici ces classiques domaines primitifs. Il est important de noter que la borne inférieure de **I** dans la boucle **for** est implicitement **1**. De la même manière, les indices d'un tableau (que nous limitons, sans perte de généralité, à une seule dimension) sont supposés évoluer entre **1** et la valeur de la constante **K** utilisée dans sa déclaration.

En suivant [15], il est extrêmement facile de donner une sémantique dénotationnelle directe du sous-langage séquentiel de ALL (nous reviendrons sur le problème de la sémantique des constructions parallèles). Une version écrite en ML est présentée dans [19]. A partir de maintenant, nous supposerons donc que nous avons deux fonctions sémantiques E et C de type :

$$E: \quad \mathbf{Exp} \rightarrow \mathit{Store} \rightarrow \mathit{Value}$$

$$C: \quad \mathbf{Com} \rightarrow \mathit{Store} \rightarrow \mathit{Store}$$

où $\mathit{Store} = \mathbf{Ide} \rightarrow (\mathit{Value} + \mathit{Value}^*)$ associe à chaque identificateur, soit une valeur (dans le domaine *Value* que nous limiterons, sans perte de généralité, à $\mathit{Bool} + \mathit{Int}$), soit une liste de valeurs. Cette définition inhabituelle facilite le traitement des tableaux (en évitant d'introduire la notion d'environnement, qui n'est d'ailleurs pas réellement nécessaire puisque ALL n'autorise ni récursion, ni allocation dynamique de mémoire) et des entrées/sorties (une instruction **read** accède une liste de valeurs associée à un identificateur « système »).

3.3. Conditions de Bernstein

Les *Conditions de Bernstein* [3] définissent une condition suffisante pour évaluer concurremment (en préservant la sémantique séquentielle) deux commandes. Nous allons les définir formellement en utilisant la sémantique standard de ALL.

Soit L, de type :

$$L: \quad \mathbf{Com} \rightarrow \mathit{Store} \rightarrow (\mathbf{Exp} \# \mathit{Store})^*$$

la fonction d'ordre supérieur qui, pour une commande et un état-mémoire initial, retourne une liste de paires, appelées *locations*. Informellement, si nous avions défini l'implantation des objets de ALL en mémoire (au moyen par exemple d'un environnement), une location contiendrait toute l'information nécessaire pour calculer les adresses référencées par une commande. Le premier composant d'une location est une expression **E** *atomique*, *i.e.* une expression de la forme **I** ou $\mathbf{I}[\mathbf{E}_1]$. Le second est la valeur de l'état-mémoire utilisé au moment de l'évaluation de **E**. La fonction L fournit les locations utilisées en lecture par une commande. On a, par exemple :

$$L[\![\mathbf{I} := \mathbf{J} + \mathbf{T}[\mathbf{2}]]\!]\, s = [\mathbf{J}, s; \mathbf{T}[\mathbf{2}], s].$$

De la même manière, on peut définir une fonctionnelle M qui retourne les locations modifiées en écriture par une commande. Par exemple :

$$M[\![\mathbf{T}[\mathbf{I}] := \mathbf{2}]\!]\, s = [\mathbf{T}[\mathbf{I}], s].$$

Comme précédemment, l'état-mémoire s sera utilisé pour obtenir l' « adresse-mémoire » d'une expression atomique qui contient des éléments de tableaux.

Il est trivial de donner des spécifications précises de L et M, par définition récursive sur la structure de **Com**. Ces spécifications utilisent des objets qui ne sont connus qu'au moment de l'exécution d'une commande (*i.e.*, l'état-mémoire) et ne peuvent donc pas être utilisées telles quelles dans notre paralléliseur sémantique. Par contre, elles vont nous servir à définir précisément les conditions qui permettent la parallélisation de deux commandes.

Nous les explicitons ici en utilisant notre propre formalisme :

CONDITION 1 (Bernstein) : *Deux commandes* $\mathbf{C}_1$ *et* $\mathbf{C}_2$ *sont exécutables en parallèle à partir des états-mémoire* s_1 *et* s_2 *si les seuls conflits possibles sont entre* $L[\![\mathbf{C}_1]\!]\, s_1$ *et* $L[\![\mathbf{C}_2]\!]\, s_2$, i. e., *si :*

$$\neg\, (\mathit{conflit}\,(l_1, m_2) \vee \mathit{conflit}\,(m_1, l_2) \vee \mathit{conflit}\,(m_1, m_2))$$

où $l_i = L[\![\mathbf{C_i}]\!]s_i$ *et* $m_i = M[\![\mathbf{C_i}]\!]s_i$.

La relation de conflit entre locations est définie par induction sur la structure de **Exp** par :

$$\textit{conflit}((\mathbf{I}, s_1), (\mathbf{J}, s_2)) = (\mathbf{I} = \mathbf{J})$$

$$\textit{conflit}((\mathbf{I}, s_1), (\mathbf{J}[\mathbf{E_2}], s_2)) = \textit{faux}$$

$$\textit{conflit}((\mathbf{I}[\mathbf{E_1}], s_1), (\mathbf{J}[\mathbf{E_2}], s_2)) = (\mathbf{I} = \mathbf{J} \wedge E[\![\mathbf{E_1}]\!]s_1 = E[\![\mathbf{E_2}]\!]s_2)$$

et peut être naturellement étendue pour s'appliquer à des listes de locations.

Dans l'optique où nous nous sommes placés, ces conditions doivent être prises comme un postulat de base, que l'on pourrait d'ailleurs justifier par des considérations heuristiques telles qu'on les trouve dans l'article original de Bernstein. Pour aller au-delà, il faudrait définir formellement la sémantique de $|[\mathbf{C_1}; \mathbf{C_2}]|$ et montrer que les Conditions de Bernstein suffisent pour en prouver l'équivalence avec celle de $[\mathbf{C_1}; \mathbf{C_2}]$. Cet objectif devra attendre l'élaboration d'une sémantique dénotationnelle satisfaisante pour les programmes parallèles, ce qui est en dehors du sujet de cet article.

La vérification des Conditions de Bernstein est impossible car elles dépendent des états-mémoire s_1 et s_2 réels en entrée des instructions $\mathbf{C_1}$ et $\mathbf{C_2}$ que l'on cherche à paralléliser. Si on ne sait rien sur s_1 et s_2, il sera presque toujours possible de trouver des cas de conflit, donc de s'interdire toute parallélisation. Or, s_1 et s_2 ne sont pas quelconque; ils doivent être naturellement engendrés par la suite des calculs du programme objet. Nous montrerons plus loin comment formaliser cette restriction, par exemple au moyen d'assertions (4.1.2). Le but de la parallélisation sémantique est d'essayer d'approximer les valeurs des états-mémoire utilisés au cours des recherches de conflit entre commandes. Pour définir formellement cette notion d'approximation, nous allons utiliser la notion d'interprétation abstraite.

3.4. Interprétation abstraite

Une *interprétation abstraite* [8] définit une sémantique *non standard* (*i.e.*, différente de celle utilisée habituellement pour définir l'évaluation d'une commande par un ordinateur) d'un langage de programmation. Par exemple, on pourrait associer à chaque identificateur d'un programme une information approchée sur le signe (positif, négatif ou inconnu) de sa valeur en tout point d'un programme; on parle alors d'une sémantique *attribuée*. On pourrait également obtenir, comme le fait [9], une précondition logique pour toute commande d'un programme; il s'agirait alors d'une sémantique *relationnelle*.

Une sémantique *directe* abstraite d'un langage de programmation est une fonction de type :

$$A: \quad \mathbf{Com} \to \mathbf{D}_a \to \mathbf{D}_a$$

où D_a est appelé domaine *abstrait*. La relation qui lie A et la sémantique standard C du langage est donnée par une fonction dite de *concrétisation* γ qui applique D_a dans l'ensemble des parties $\mathscr{P}(Store)$; pour un élément abstrait donné d_a, $\gamma(d_a)$ représente l'ensemble des états-mémoire qui sont compatibles avec d_a.

Pour qu'une interprétation abstraite directe A soit correcte par rapport à la sémantique standard C, une condition doit être établie :

CONDITION 2 (Correction) : *A est correcte par rapport à C si et seulement si, pour toute commande $\mathbf{C}_1$ et élément abstrait d_a,*

$$\{C[\![\mathbf{C}_1]\!]\, s / s \in \gamma(d_a)\} \subseteqq \gamma(A[\![\mathbf{C}_1]\!]\, d_a).$$

La définition d'une interprétation abstraite pour un langage de programmation donné peut être faite de différentes manières : par exemple, [8] utilise une approche opérationnelle, tandis que [27] introduit une vue dénotationnelle. Nous allons utiliser cette seconde technique pour obtenir un cadre général et unique de spécification de parallélisation. De plus, nous serons à même de facilement implémenter nos spécifications.

3.5. Spécifications exécutables

En vue d'utiliser nos interprétations dénotationnelles abstraites comme des spécifications exécutables, nous devons insister sur deux points. Premièrement, nos domaines abstraits doivent être *syntaxiques* (au sens de [31]) pour être effectivement implémentables sur machine. Deuxièmement, nous devons prendre soin d'éliminer (*i.e.*, approximer) les points fixes qui vont apparaître dans nos spécifications, puisque notre processus de parallélisation doit être fait au moment de la compilation. Cela sera un aspect important du problème dont nous discuterons plus loin.

En pratique, nous avons utilisé ML [14] comme langage d'exécution de spécifications car il autorise la manipulation de fonctions d'ordre supérieur et les considère comme des « citoyens de première classe ». Ce sont exactement les caractéristiques nécessaires à un langage pour qu'il soit aisément utilisable pour écrire des spécifications dénotationnelles. De plus, le système de typage automatique offert par ML est un « plus » majeur quand il s'agit d'écrire

des expressions complexes utilisant des fonctions largement currifiées [17] : il assure une détection des erreurs extrêmement efficace.

4. PARALLÉLISATION SÉMANTIQUE

Nous allons proposer trois exemples d'interprétation abstraite du langage ALL, chacune ciblée vers un objectif différent :

- Le premier considère ALL comme un transformeur de *prédicats* (sémantique relationnelle) et utilise ceux-ci pour analyser des conflits entre différentes itérations de boucle, même quand des variables non-indices de boucle sont utilisées.
- Le second est basé sur une nouvelle notion, les *t-prédicats* (sémantique attribuée), et l'utilise pour traiter les cas des boucles dans lesquelles des indirections sur tableaux sont effectuées (par exemple, dans le traitement de matrices creuses).
- Le dernier exemple est légèrement différent et utilise une *évaluation symbolique* de ALL (sémantique attribuée) pour détecter des réductions à l'intérieur de boucles.

Nous allons proposer différentes techniques pour approximer les états-mémoire et donner les nouvelles Conditions de Bernstein adaptées à chacune de ces approximations. Le processus de parallélisation, qui est lui aussi une interprétation abstraite de ALL, est alors une conséquence immédiate des Conditions de Bernstein, que nous préciserons dans chaque cas.

4.1. Parallélisation par prédicats

Nous proposons de collecter des informations sur les variables utilisées par le programmeur en vue d'être à même de paralléliser le genre de programmes symbolisé par l'exemple suivant :

```
for i to 1000 do
    t[i+n] := 2 * t[i];
```

où nous avons besoin de prouver que $\mathbf{n} >= 1000$ est vérifié en entrée de boucle si nous voulons être sûrs qu'il n'y a pas de conflit dans le corps de boucle. La technique classique pour traiter ce type de problème est d'utiliser le *constant folding* [1] pour essayer d'obtenir la valeur de **n** au moment de la compilation. Notre approche est plus puissante puisqu'elle est capable de fonctionner même en présence d'informations « floues » sur l'estimation de

la valeur de **n**. [37] a proposé une technique similaire; notre méthode est néanmoins présentée dans un cadre plus formel et sémantiquement motivé.

4.1.1. *Spécification*

Nous définissons le domaine syntaxique **Pred** des *prédicats* comme la restriction du domaine des expressions booléennes aux seuls termes linéaires (*i. e.*, sans multiplication, si ce n'est par des constantes). Ce nouveau domaine n'est pas bien-fondé (il y a, en théorie, des prédicats de longueur infinie [32]). La concrétisation d'un prédicat **p** est donnée par :

$$\gamma(\mathbf{p}) = \{s \in Store / s \vdash \mathbf{p}\}$$

où $s \vdash \mathbf{p}$ est *vrai* s'il existe une conjonction dans la forme disjonctive normale de **p** qui est *vrai* dans l'état-mémoire *s*.

Nous définissons une sémantique de transformeur de prédicats C_p pour ALL par induction sur la structure de **Com** : $C_p[\![\mathbf{C}]\!]\,\mathbf{p}$ retourne un prédicat qui est valide en sortie de **C** si **p** est valide en entrée de **C**. Par exemple, si l'on considère la commande d'affectation simple, on a :

$$C_p[\![\mathbf{I} := \mathbf{E}]\!]\mathbf{p} = (\mathbf{p}\{\mathbf{I} \leftarrow \mathbf{I_0}\} \textbf{ and } \mathbf{I} = \mathbf{E}\{\mathbf{I} \leftarrow \mathbf{I_0}\})$$

où $\{\mathbf{I} \leftarrow \mathbf{I_0}\}$ représente une substitution syntaxique (ici appliquée sur le prédicat **p** et l'expression **E**) et $\mathbf{I_0}$ est un nouvel identificateur, implicitement quantifié existentiellement, qui dénote l' « ancienne » valeur de **I**. On reconnaît là la technique utilisée par [12] pour exprimer la plus forte post-condition valide après une affectation. En pratique, nous imposons également une condition syntaxique de *contexte*; **E** doit être une expression entière linéaire si nous voulons rester dans le domaine de l'arithmétique linéaire en nombres entiers. Ceci est obligatoire si nous voulons maintenir la propriété de décidabilité (*i. e.*, éviter le dixième problème d'Hilbert).

Un autre cas intéressant est le traitement de la boucle **while** :

$$C_p[\![\textbf{while E do C}]\!] = \mu(\lambda \mathrm{C_p}\,\mathbf{p}.(\mathbf{p} \textbf{ and } \neg\mathbf{E}) \textbf{ or } \mathrm{C_p}(C_p[\![\mathbf{C}]\!](\mathbf{p} \textbf{ and } \mathbf{E})))$$

où μ est le combinateur de plus petit point fixe. La post-assertion générée en sortie de boucle, dont l'existence est garantie par le théorème du point-fixe de Tarski, peut être vue comme un objet syntaxique infini [32]. Nous expliquerons plus loin comment approximer ce prédicat.

La condition générale de correction présentée en 3.4 devient ici :

CONDITION 3 (Prédicat) : *Pour toute commande* **C**, *prédicat* **p** *et état-mémoire s*,

$$s \vdash \mathbf{p} \quad \Rightarrow \quad C[\![\mathbf{C}]\!]\, s \vdash C_p[\![\mathbf{C}]\!]\, \mathbf{p}.$$

On la démontre par induction sur la structure de **Com** [21]. En vue d'utiliser la règle d'induction de Scott pour prouver la correction sémantique de l'équation correspondant à la commande **while**, nous devons munir le domaine **Pred** d'un ordre partiel plus riche que le seul ordre syntaxique, en pratique :

$$\mathbf{p}_1 < \mathbf{p}_2 \equiv \forall s \in \mathit{Store}, s \vdash \mathbf{p}_1 \Rightarrow s \vdash \mathbf{p}_2$$

modulo la relation d'équivalence $\sim$:

$$\mathbf{p}_1 \sim \mathbf{p}_2 \equiv \mathbf{p}_1 < \mathbf{p}_2 \wedge \mathbf{p}_2 < \mathbf{p}_1.$$

4.1.2. *Détection des conflits*

Nous utilisons tout d'abord ce transformeur de prédicats C_p pour étiqueter de manière naturelle chaque nœud de l'arbre abstrait d'un programme (identifié par un nombre de Dewey appelé *niveau*, élément d'un domaine **Niv** que nous ne spécifierons pas ici) avec sa pré-condition, sachant que la racine de l'arbre sera étiquetée par **true**. Cette association entre **Niv** et **Pred** définit une fonction d'*assertion*, élément du domaine **Assert** :

$$\mathbf{Assert} = \mathbf{Niv} \rightarrow \mathbf{Pred}.$$

Si un prédicat est un moyen d'approximer un état-mémoire (*Store*), alors un élément de $\mathbf{Reg} \equiv \mathbf{Exp} \# \mathbf{Pred}$ (où les expressions sont atomiques), appelée *région* [37], « approxime » une location.

Nous définissons alors les fonctions L et M de type :

$$L, M: \quad \mathbf{Com} \rightarrow \mathbf{Assert} \rightarrow \mathbf{Niv} \rightarrow \mathbf{Reg}^*$$

qui, pour une commande donnée **C** de niveau **n** et une fonction d'assertion **a**, retourne la liste des régions accédées en lecture (resp. écriture). Si A, de type $\mathbf{Exp} \rightarrow \mathbf{Pred} \rightarrow \mathbf{Reg}^*$, permet d'obtenir la liste des régions accédées en lecture par une expression, on a, par exemple :

$$L[\![\mathbf{I} := \mathbf{E}]\!]\, \mathbf{an} = A[\![\mathbf{E}]\!]\, (\mathbf{an})$$
$$M[\![\mathbf{I}[\mathbf{E}_1] := \mathbf{E}_2]\!]\, \mathbf{an} = [\mathbf{I}[\mathbf{E}_1], \mathbf{an}]$$

$$L[\![\textbf{if E then } \mathbf{C_1} \textbf{ else } \mathbf{C_2}]\!]\,\mathbf{an} = A[\![\mathbf{E}]\!](\mathbf{an}) \,@\, L[\![\mathbf{C_1}]\!]\,\mathbf{an_1} \,@\, L[\![\mathbf{C_2}]\!]\,\mathbf{an_2}$$

où $\mathbf{n_i}$ est l'indice de Dewey du i-ième fils de la commande de niveau $\mathbf{n}$.

Les Conditions de Bernstein peuvent alors être redéfinies par :

CONDITION 4 (Prédicat) : *Les commandes* $\mathbf{C_1}$ *(de niveau* $\mathbf{n_1}$*) et* $\mathbf{C_2}$ *(*$\mathbf{n_2}$*) d'un programme* **P** *sont parallélisables par rapport à la fonction d'assertion* **a** *si les seuls conflits sont entre régions de* $L[\![\mathbf{C_1}]\!]\,\mathbf{an_1}$ *et* $L[\![\mathbf{C_2}]\!]\,\mathbf{an_2}$ *où, dans la définition de la relation de conflit, la dernière clause est remplacée par :*

conflit $((\mathbf{I}[\mathbf{E_1}], \mathbf{p_1}), (\mathbf{J}[\mathbf{E_2}], \mathbf{p_2}))$

$$= \exists s_1, s_2, (\mathbf{I} = \mathbf{J} \wedge s_1 \vdash \mathbf{p_1} \wedge s_2 \vdash \mathbf{p_2} \wedge E[\![\mathbf{E_1}]\!]\,s_1 = E[\![\mathbf{E_2}]\!]\,s_2)$$

où la fonction d'assertion **a** de **P** peut être calculée très simplement à partir du transformateur de prédicats C_p (*voir* [21]). La vérification de la condition de conflit peut être vue comme le problème de décider la satisfiabilité d'une formule implicitement existentiellement quantifiée dans l'arithmétique entière de Presburger. Informellement, après avoir testé l'égalité syntaxique de **I** et **J**, on teste si la formule « $\mathbf{E_1} = \mathbf{E_2} \wedge \mathbf{p_1} \wedge \mathbf{p_2}$ » a une solution, ce qui est un problème décidable [22]. En pratique, et pour des raisons d'efficacité, nous nous limiterons à l'algorithme approché de Fourier [11] qui permet de décider dans l'ensemble Q des rationnels. Il est important de noter que ceci est une approximation conservative; nous risquons de perdre des cas où la parallélisation est possible, mais, en aucun cas, de décider de paralléliser là où cela serait incorrect. Pour améliorer cette technique approchée, nous utiliserons le classique résultat d'Euclide sur les équations diophantiennes; puisque $\mathbf{E_1} = \mathbf{E_2}$ peut être réécrit sous la forme $\sum_i \mathbf{a_i}\,\mathbf{x_i} = \mathbf{b}$, nous vérifierons également que le plus grand commun diviseur des $\mathbf{a_i}$ divise **b**.

Dans un paralléliseur réaliste, il est important d'utiliser un algorithme aussi précis que possible. On peut, par exemple, exploiter les méthodes de la Programmation Linéaire en Nombres Entiers, comme dans [34].

4.1.3. *Parallélisation*

La parallélisation d'ALL basée sur les prédicats est alors une opération simple. La boucle **for I to E do C** est transformée en **for// I to E do P** (où **P** est la version parallélisée du corps de boucle **C**) s'il n'y a pas de conflit entre les régions utilisées dans deux itérations différentes du corps de boucle **C**.

Le traitement de $[\mathbf{C_1}; \ldots; \mathbf{C_m}]$ est plus complexe. Une heuristique consiste à simplement trier topologiquement les m commandes parallélisées

$[\mathbf{P}_1; \ldots; \mathbf{P}_m]$, partiellement ordonnées par la relation de conflit. Ceci ne donne pas toujours un calcul optimal si leurs temps d'exécution sont différents (auquel cas, le problème d'ordonnancement est NP-complet). Nous ne détaillerons pas plus ce point; une présentation plus avancée peut être trouvée dans [21].

4.1.4. *Retour sur l'exemple*

Dans l'exemple de la section 2, le prédicat valide en entrée de boucle est

$$\textbf{true and } \mathbf{s=0} \textbf{ and } ((\mathbf{n_0 > 10} \textbf{ and } \mathbf{n=n_0+1}) \textbf{ or } (\mathbf{n_0 <= 10} \textbf{ and } \mathbf{n=10})).$$

Il est construit automatiquement par le transformeur de prédicats C_p opérant sur le programme objet à partir du prédicat initial **true**.

Les tests de conflit entre locations lues et modifiées par les différentes instructions à l'intérieur de la boucle seront donc limités aux état-mémoires qui vérifient ce prédicat. Par exemple, ceci explique que notre paralléliseur soit à même de détecter l'absence de conflit entre $\mathbf{t[i]}$ et $\mathbf{t[i+n]}$, conséquence du fait que le système ci-dessous, défini par la Condition 4:

$$\begin{cases} \mathbf{i_1 = i_2 + n} \\ \textbf{true and } \mathbf{s=0} \textbf{ and } ((\mathbf{n_0 > 10} \textbf{ and } \mathbf{n=n_0+1}) \textbf{ or } (\mathbf{n_0 <= 10} \textbf{ and } \mathbf{n=10})) \\ \mathbf{1 <= i_1} \textbf{ and } \mathbf{i_1 <= 10} \\ \mathbf{1 <= i_2} \textbf{ and } \mathbf{i_2 <= 10} \end{cases}$$

n'a pas de solution.

4.2. Parallélisation par t-prédicats

Le cadre précédent est incapable de traiter l'exemple suivant :

```
for i to n do
    t[u[i]] := 0;
```

qui utilise des indices de tableaux non linéaires. Pourtant, ce type d'opération est fréquent dans les calculs basés sur des structures de données creuses, que ce soit en Recherche Opérationnelle sur des graphes faiblement couplés ou en Analyse Numérique pour résoudre des problèmes d'Éléments Finis.

Nous pourrions essayer une extension naturelle des prédicats aux tableaux avec, par exemple pour l'affectation de tableau :

$$C_p[\![\mathbf{I[E_1] := E_2}]\!]\,\mathbf{p} = (\mathbf{p}\{\mathbf{I} \leftarrow \mathbf{I_0}\} \textbf{ and } \mathbf{I[E_1}\{\mathbf{I} \leftarrow \mathbf{I_0}\}] = \mathbf{E_2}\{\mathbf{I} \leftarrow \mathbf{I_0}\})$$

où $\mathbf{I_0}$ est un identificateur de type tableau, implicitement quantifié existentiellement. Malheureusement, ceci est sans espoir car l'arithmétique de Presburger quantifiée avec tableaux est indécidable [33]. De plus, cette formule ne précise pas le fait que le tableau **I** est inchangé si ce n'est à l'indice $\mathbf{E_1}$.

Pourtant, pour les programmes scientifiques pour lesquels les accès indirects sont généralement « statiques », des informations approchées peuvent être suffisantes. Nous proposons donc une approche beaucoup plus simple, basée sur des *t-prédicats* (pour « prédicat sur tableau ») associés à chaque tableau **I**; ils donnent, pour une valeur d'indice *k*, une estimation conservative de « **I[k]** ».

4.2.1. *Spécification*

Nous définissons hiérarchiquement le domaine **Decor** par :

$$\mathbf{Form} = \mathbf{Pred} + \{\exists, \forall\} \# \mathbf{Ide} \# \mathbf{Form}$$

$$\mathbf{Ens} \equiv \mathbf{Ide} \# \mathbf{Form}$$

$$\mathbf{TPred} \equiv \mathbf{Ide} \# \mathbf{Ens}$$

$$\mathbf{Decor} \equiv \mathbf{Ide} \rightarrow \mathbf{TPred}.$$

Nous adopterons les notations suivantes. Un t-prédicat (élément du domaine **TPred**) sera représenté par $\lambda \mathbf{x} . \{\mathbf{z/f(x, z)}\}$ où $\{\mathbf{z/f(x, z)}\}$ est un *ensemble* et **f(x, z)** une *forme* qui est soit un prédicat, soit une sous-forme quantifiée. Un *décor* associe chaque tableau à son t-prédicat. Informellement, une forme est un prédicat éventuellement partiellement quantifié, un ensemble représente l'ensemble des valeurs qui satisfont à une forme donnée et un t-prédicat est une fonction qui associe un ensemble à une valeur.

A titre d'exemple, le t-prédicat associé au tableau **u** après l'exécution de la boucle :

```
for i to 100 do [
        u[2 * i] = 2 * i + 1;
        ]
```

sera donné par :

$$\lambda \mathbf{x} . \{\mathbf{z}/(\exists \mathbf{i} . \mathbf{x} = 2 * \mathbf{i} \text{ and } 1 <= \mathbf{i} \text{ and } \mathbf{i} <= 100) \rightarrow (\exists \mathbf{i} . \mathbf{z} = 2 * \mathbf{i} + 1 \text{ and } \mathbf{x} = 2 * \mathbf{i} \text{ and } 1 <= \mathbf{i} \text{ and } \mathbf{i} <= 100), \mathbf{f_u(x, z)}\}$$

si son t-prédicat en entrée est $\lambda \mathbf{x} . \{\mathbf{z}/\mathbf{f_u}(\mathbf{x}, \mathbf{z})\}$. Nous utilisons la notation classique :

$$p \rightarrow q,\ r = (p \wedge q) \vee (\neg p \wedge r).$$

La sémantique $\varepsilon(\mathbf{t})s$ d'un t-prédicat **t** est une fonction qui associe le domaine des entiers *Int* et sa partition $\mathscr{P}(Int)$; elle permet d'obtenir une approximation des valeurs possibles pour chaque élément de tableau. Nous définissons de la même manière ε sur le domaine **Form** (resp. **Ens**) qui a *Bool* [resp. $\mathscr{P}(Int)$] pour image :

$$\varepsilon(\lambda \mathbf{x} . \{\mathbf{z}/\mathbf{f}(\mathbf{x}, \mathbf{z})\})s = \lambda u . \varepsilon(\{\mathbf{z}/\mathbf{f}(\mathbf{x}, \mathbf{z})\})s[u/\mathbf{x}]$$

$$\varepsilon(\{\mathbf{z}/\mathbf{f}(\mathbf{x}, \mathbf{z})\})s = \{u/\varepsilon(\mathbf{f}(\mathbf{x}, \mathbf{z}))s[u/\mathbf{z}]\}$$

$$\varepsilon(\exists \mathbf{x} . \mathbf{f}(\mathbf{x}))s = \exists u . \varepsilon(\mathbf{f}(\mathbf{x}))s[u/\mathbf{x}]$$

$$\varepsilon(\forall \mathbf{x} . \mathbf{f}(\mathbf{x}))s = \forall u . \varepsilon(\mathbf{f}(\mathbf{x}))s[u/\mathbf{x}]$$

$$\varepsilon(\mathbf{p})s = s \vdash \mathbf{p}$$

où nous supposons que chaque formule quantifiée est *fermée* (*i.e.*, il n'y a pas de variable non quantifiée *libre*), pour des raisons sur lesquelles nous reviendrons. Nous utilisons la même λ notation pour désigner des termes syntaxiques (dans **TPred**) et des fonctions (dans $Int \rightarrow \mathscr{P}(Int)$); le contexte permet toujours de lever aisément l'ambiguïté entre les deux concepts.

La concrétisation $\gamma(\mathbf{d})$ d'un décor **d** est donnée par :

$$\gamma(\mathbf{d}) = \{s \in Store / \forall \mathbf{K} \in \mathbf{Con},\ \forall \mathbf{I} \in \mathbf{Ide},\ E[\![\mathbf{I}[\mathbf{K}]]\!]s \in \varepsilon(\mathbf{dI})s(K[\![\mathbf{K}]\!])\}$$

où $\mathbf{K} : \mathbf{Con} \rightarrow Value$ est la sémantique standard des constantes (e. g., $K[\![\mathbf{2}]\!] = 2$). Informellement, la concrétisation d'un décor est un ensemble d'états-mémoire tel que, pour tout tableau **I** et index **K**, « **I**[**K**] » est un membre de l'approximation « **dIK** ».

Nous pouvons alors définir une sémantique de transformeur de t-prédicats C_t de ALL :

$$C_t : \quad \mathbf{Com} \rightarrow \mathbf{Assert} \rightarrow \mathbf{Niv} \rightarrow \mathbf{Decor} \rightarrow \mathbf{Decor}$$

où **Assert** et **Niv** sont des attributs uniquement hérités. Par exemple, en ce qui concerne l'affectation de tableau, nous souhaiterons alors avoir quelque chose comme (1) ci-dessous :

$$C_t[\![\mathbf{I}[\mathbf{E_1}] := \mathbf{E_2}]\!]\ \mathbf{and} =$$

$$\mathbf{d}[\lambda \mathbf{x} . \{\mathbf{z}/(\exists \mathbf{v_1} . \mathbf{f_1}(\mathbf{x})) \rightarrow (\exists \mathbf{v_{12}} . \mathbf{f_1}(\mathbf{x}) \ \mathbf{and}\ \mathbf{f_2}(\mathbf{z})),\ \mathbf{f_1}(\mathbf{x}, \mathbf{z})\}/\mathbf{I}]$$

avec $E_t [\![\mathbf{E_i}]\!] \mathbf{d\,(an)} = \{\mathbf{z_i/f_i\,(z_i)}\}$

et $\mathbf{dI} = \lambda \mathbf{x}\,.\,\{\mathbf{z/f_1(x,\ z)}\}$

où $\mathbf{v_1}$ (resp. $\mathbf{v_{12}}$) est la liste des variables libres de $\mathbf{f_1}$ (resp. $\mathbf{f_1}$ and $\mathbf{f_2}$) moins **x** (resp. **x** et **z**). Nous avons utilisé ici, la notation plus intuitive **f(z)** au lieu de $\mathbf{f(x)}\{\mathbf{x} \leftarrow \mathbf{z}\}$ pour désigner une substitution syntaxique; il est important de noter également que cette opération peut entrainer des α-conversions si **f** est quantifié. Informellement, seule la valeur de **I** est changée dans le décor d; à tout indice x, la valeur de « **I[x]** » est soit son ancienne valeur (si x n'appartient pas à l'ensemble des valeurs possibles de $\mathbf{E_1}$ obtenu par la fonction E_t), soit donnée par $\mathbf{E_2}$ (en restreignant dans le choix des valeurs possibles celles qui sont compatibles avec $\mathbf{E_1}$).

Dans l'équation (1), E_t, de type **Exp → Decor → Pred → Ens**, retourne l'ensemble des valeurs possibles d'une expression dans des décor et prédicat donnés. Par exemple :

$$E_t [\![\mathbf{I}]\!]\,\mathbf{dp} = \{\mathbf{z/z = I\ and\ p}\}$$

$$E_t [\![\mathbf{E_1\ \ O2\ \ E_2}]\!]\,\mathbf{dp} = \{\mathbf{z/z = z_1\ O_2\ z_2\ and\ f_1(z_1)\ and\ f_2(z_2)}\}$$

où $E_t [\![\mathbf{E_i}]\!]\,\mathbf{dp} = \{\mathbf{z_i/f_i(z_i)}\}$. Il est important de noter que, par construction, les formes utilisées dans la définition de C_t sont fermées. Ceci est obligatoire pour éviter des captures de noms qui pourraient apparaître dans des programmes comme;

```
t[n] := 1;
n := n + 1;
t[n] := 2;
```

où la variable **n** utilisée dans la troisième instruction doit être clairement distinguée de celle utilisée dans la première affectation de **t**. Ce découplage est réalisé par la quantification complète du t-prédicat associé à **t**.

Il ne reste qu'une dernière précaution à prendre. La règle (1) n'est correcte que si $\mathbf{E_1}$ vérifie une condition de *contrainte;* nous devons avoir une parfaite connaissance de l'influence de l'affectation dans le domaine de **I**. Plus formellement, une expression **E** est contrainte dans un décor **d** par un prédicat **p** si et seulement si il existe un unique z tel que

$$\exists s,\ \varepsilon(\mathbf{f(z)})\, s[z/\mathbf{z}]$$

soit vrai, où l'on suppose que $E_t [\![\mathbf{E}]\!]\,\mathbf{dp} = \{\mathbf{z/f(z)}\}$. Si l'expression $\mathbf{E_1}$ utilisée dans $\mathbf{I[E_1] := E_2}$ n'est pas contrainte, nous pourrions décider de paralléliser

une autre commande de manière incorrecte (*voir* [21] pour une discussion plus élaborée de ce problème).

La condition générale de correction présentée en 3.4 devient ici :

CONDITION 5 (Décor) : *Pour toute commande* **C** *de niveau* **n**, *tableau* **I**, *index* **K**, *état-mémoire s et décor* **d**,

$$E[\![\mathbf{I}[\mathbf{K}]]\!]\, s \in \varepsilon(\mathbf{dI})\, sk \quad \Rightarrow \quad E[\![\mathbf{I}[\mathbf{K}]]\!]\, s' \in \varepsilon(\mathbf{d'}\,\mathbf{I'})\, s'\, k$$

où $k = K[\![\mathbf{K}]\!]$, $s' = C[\![\mathbf{C}]\!]\, s$ *et* $\mathrm{d}' = C_t[\![\mathbf{C}]\!]$ **and**.

Informellement, la propriété que **dI** approxime conservativement **I** pour tout index k est invariante par exécution standard de ALL. La preuve de ce théorème se fait par induction sur **Com**, en utilisant la condition de contrainte et le lemme suivant :

LEMME : *Si* $E[\![\mathbf{I}[\mathbf{K}]]\!]\, s \in \varepsilon(\mathbf{dI})\, sk$, *alors, pour tout prédicat* **p** *et expression* **E**, *on a :*

$$s \vdash \mathbf{p} \quad \Rightarrow \quad E[\![\mathbf{E}]\!]\, s \in \varepsilon(E_t[\![\mathbf{E}]\!]\, \mathbf{dp})\, s.$$

Enfin, il est important de noter que la condition de contrainte peut être facilement réécrite comme une expression logique de l'arithmétique de Presburger. Ceci permet de préserver la décidabilité de notre technique.

4.2.2. *Détection des conflits*

En utilisant les notions introduites précédemment, il est possible d'étendre notre approximation des états-mémoire; il s'agit maintenant d'une paire (**p**, **d**) où le prédicat **p** (resp. le décor **d**) approxime les valeurs des variables simples (resp. des tableaux). Les définitions de région et fonction d'assertion sont modifiées de la même manière et la dernière clause définissant la relation de conflit utilisée pour analyser les Conditions de Bernstein devient :

$$\begin{aligned} &\mathit{conflit}((\mathbf{I}[\mathbf{E}_1],\ \mathbf{p}_1,\ \mathbf{d}_1),\ (\mathbf{J}[\mathbf{E}_2],\ \mathbf{p}_2,\ \mathbf{d}_2)) \\ &\qquad = \exists s_1, s_2,\ (\mathbf{I} = \mathbf{J} \wedge s_1 \vdash \mathbf{p}_1 \wedge s_2 \vdash \mathbf{p}_2 \wedge \\ &\qquad\qquad \varepsilon(E_t[\![\mathbf{E}_1]\!]\, \mathbf{d}_1\, \mathbf{p}_1)\, s_1 \cap \varepsilon(E_t[\![\mathbf{E}_2]\!]\, \mathbf{d}_2\, \mathbf{p}_2)\, s_2 \neq \varnothing). \end{aligned}$$

Puisque le test d'intersection est équivalent à décider la satisfiabilité de la conjonction des formes définissant les ensembles, nous restons dans l'arithmétique de Presburger quantifiée qui reste décidable [22]. Par exemple, nous pourrions utiliser l'algorithme de décision de [6].

La technique à appliquer pour paralléliser un programme ALL est la même que celle présentée dans le cas de la parallélisation par prédicats.

4.3. Parallélisation par évaluation symbolique

Le fait qu'une variable scalaire soit modifiée dans le corps d'une boucle ne signifie pas automatiquement que cette boucle ne puisse pas être parallélisée. Par exemple, dans :

```
s := 0;
for i to 1000 do
    s := s + a[i];
```

il est possible d'évaluer de manière efficace la boucle calculant la somme des 1 000 premiers éléments du tableau **a**. En effet, l'opération réalisée sur **s** est une *réduction* (*i. e.*, itération d'une opération associative) susceptible d'être implémentée sous forme arborescente sur une machine parallèle; si l'opération est de plus commutative, elle peut être exécutable efficacement sur une machine vectorielle. L'approche sémantique permet de détecter ce type d'opération. Nous introduisons la notion d'*évaluation symbolique* permettant d'analyser le type d'opération réalisée par une variable après l'exécution d'une séquence d'instructions ALL (e. g., un corps de boucle); si cette opération est caractéristique d'une réduction, alors une parallélisation peut être effectuée par l'introduction d'une commande **for_red** [18].

4.3.1. *Spécification*

Nous définissons hiérarchiquement une sémantique non standard pour des expressions symboliques (éléments du domaine **Svalue**) :

$$\textbf{Svalue} = (\textbf{Senv} \# \textbf{Exp})^*$$

$$\textbf{Senv} = \textbf{Svalue}^*$$

$$\textbf{Sstore} \equiv \textbf{Ide} \rightarrow \textbf{Svalue}$$

$$\textbf{State} \equiv \textbf{Sstore} \# \textbf{Ide}^*$$

où un *état symbolique* élément de **Sstate** (approximant un état-mémoire) est caractérisé par deux composantes : d'une part, une fonction appelée *mémoire symbolique*, élément de **Sstore**, qui associe à tout identificateur une *expression* ou *valeur* symbolique (**Svalue**); et, d'autre part, une liste d'identificateurs qui définit les valeurs (symboliques) initiales ([5]). Une valeur symbolique est représentée par une liste de paires formées d'un *environnement symbolique*

([5]) Elles ne sont en fait utilisées que pour prouver la correction de notre évaluation symbolique.

(**Senv**) et d'une expression. Un environnement symbolique est une conjonction d'expressions dénotant un chemin dans l'arbre des (éventuelles) instructions conditionnelles du corps de boucle.

La définition de la sémantique ν_e d'une valeur symbolique **v** est donnée par une double définition récursive :

$$\begin{aligned} \nu_e(\mathbf{v})\, s &= E[\![\mathbf{E_i}]\!]\, s \\ &\quad \text{si } \mathbf{v} \equiv [\mathbf{r_1}, \mathbf{e_1}; \ldots; \mathbf{r_n}, \mathbf{e_n}] \\ &\quad \text{et } \nu_r(\mathbf{r_i})\, s = vrai \\ \nu_r([\ \ ])\, s &= vrai \\ \nu_r(\mathbf{r}) &= \bigwedge_{\mathbf{r_i} \in \mathbf{r}} \nu_e(\mathbf{r_i})\, s \end{aligned}$$

où l'on montre [21] que le choix de i dans la définition de ν_e est non ambigu.

La concrétisation $\gamma(\mathbf{s})$ d'un état-symbolique **s** est donnée par :

$$\gamma(\mathbf{s}, \mathbf{l}) = \{s \in Store / \exists\, v \in Value^*,\ \forall\, \mathbf{I} \in \mathbf{Ide},\ E[\![\mathbf{I}]\!]\, s = \nu_e(\mathbf{sI})\, s\,[v/\mathbf{l}]\}$$

où la notation $f[y/x]$ est trivialement étendue au cas où x et y sont des listes de même longueur. Informellement, à tout état symbolique (**s**, **l**), on associe l'ensemble des états-mémoire qui, pour au moins une valeur de l'état initial représenté par **l**, rendent pour valeur de **I** la valuation de l'expression symbolique liée à **I** dans **s**.

Nous pouvons alors définir une sémantique de transformeur d'état-symbolique C_s de ALL :

$$C_s\colon \quad \mathbf{Com} \to \mathbf{Senv} \to \mathbf{Sstate} \to \mathbf{Sstate}.$$

Par exemple, en ce qui concerne l'affectation d'une variable simple, on obtient :

$$C_s[\![\mathbf{I} := \mathbf{E}]\!]\, \mathbf{r}\, (\mathbf{s}, \mathbf{l}) = \mathbf{s}\,[E_s[\![\mathbf{E}]\!]\, \mathbf{rs}/\mathbf{I}], \mathbf{l}$$

où E_s, de type **Com** → **Senv** → **Sstore** → **Svalue**, est l'évaluateur symbolique d'expressions dans une mémoire symbolique donnée. Il est intéressant de noter combien cette définition est proche de celle que l'on obtient pour définir la sémantique standard de l'instruction d'affectation; ceci est une justification supplémentaire du choix de nos domaines.

E_s est défini également par induction, cette fois sur **Exp**. Par exemple, on a :

$$E_s[\![\mathbf{I}]\!]\,\mathbf{rs} = [\mathbf{r} @ \mathbf{r}_1, \mathbf{e}_1; \ldots; \mathbf{r} @ \mathbf{r}_n, \mathbf{e}_n]$$

si $\mathbf{sI} = [\mathbf{r}_1, \mathbf{e}_1; \ldots; \mathbf{r}_n, \mathbf{e}_n]$. Il suffit simplement d'ajouter (ce qui revient à opérer un « et » logique) la valeur de l'environnement courant à toutes les valeurs symboliques possibles de **sI**.

La condition générale de correction présentée en 3.4 devient ici :

CONDITION 6 (Symbolique) : *Pour tout état-mémoire s, commande* **C** *et état symbolique* (**s**, **l**),

$$\exists v, \forall \mathbf{I}, E[\![\mathbf{I}]\!]\, s = \nu_e(\mathbf{sI})\, s[v/\mathbf{l}]$$

$$\Rightarrow$$

$$\exists w, \forall \mathbf{I}, E[\![\mathbf{I}]\!](C[\![\mathbf{C}]\!]\, s) = \nu_e(fst(C_s[\![\mathbf{C}]\!][\ \ ](\mathbf{s}, \mathbf{l}))\,\mathbf{I})\, s[w/\mathbf{l}]$$

où $fst(x, y) \equiv x$. La preuve se fait par induction sur la structure de **Com** [21].

4.3.2. *Parallélisation des réductions*

Dans cette nouvelle application de la notion de parallélisation sémantique, le problème n'est plus tellement dans la détection des conflits (puisque nous savons qu'ils existent, par définition de ce qu'est une réduction), mais dans la détection des réductions.

Le problème est d'analyser chaque expression symbolique pour vérifier que l'opération correspondante est associative. Cela peut être fait, soit par reconnaissance de formes, soit par analyse fonctionnelle. En pratique, nous avons seulement cherché à détecter les opérations constantes ainsi que celles qui donnent des fonctions constantes après application d'une fonction *inverse*. Par exemple, l'inverse de + est −, et celui de * est /.

Une fois qu'une opération de réduction est détectée, il suffit d'éliminer toute référence à cette variable dans le corps de boucle et de rajouter l'instruction **for_red** correspondante.

4.3.3. *Retour sur l'exemple*

Dans l'exemple de la section 2, on a calculé la valeur symbolique de **s** en fin de boucle par évaluation du corps à partir d'un état symbolique initial dont la mémoire lie tout identificateur à la liste vide :

$$[[], \mathbf{s} + \mathbf{t}[\mathbf{i} + \mathbf{n}]].$$

En utilisant l'inverse de l'addition (*i. e.*, la soustraction), on obtient l'expression **t[i+n]** dont on vérifie qu'elle n'est jamais modifiée par le corps de boucle; à noter que cette vérification nécessite l'utilisation de prédicats sur **n**. Le calcul de **s** correspond donc bien à une réduction, ce qui justifie la génération de la commande **for_red**.

5. IMPLÉMENTATION

Nous arrivons maintenant à l'aspect pratique de notre approche : l'exécution directe des spécifications dénotationnelles.

5.1. Implémentation des spécifications dénotationnelles

Nous avons implémenté nos spécifications à l'aide du langage ML [14]. Ceci nous permet d'obtenir, sans effort supplémentaire, une maquette de paralléliseur sémantique à partir des définitions précédentes; c'est un avantage indéniable de l'approche proposée.

Montrons sur un exemple l' « isomorphisme » entre les notations dénotationnelles et la syntaxe de ML. Nous avons choisi ici l'extrait de notre programme ML qui spécifie le transformeur C_p et, plus particulièrement, les définitions qui ont été données précédemment :

```
letrec Cp : Com → Pred → Pred = fun
        (I assign E). \p :Pred.
            let old_I = alloc_id( ) in
            let old_e = subst_exp (I, old_I) E in
            and_pred (subst_pred (I, old_I) p)
                     (exp_to_pred (binopr (ide_to_exp I,
                                           binop_bool 'eq',
                                           old_e))) |
    (if_then_else (E1, C1, C2)). \p : Pred.
            let true_pred = exp_to_pred E1 and
              false_pred = exp_to_pred (unopr (unop-log 'not', E1)) in
            or_pred(Cp C1 (and_pred true_pred c))
                    (Cp C2 (and_pred false_pred c)) ...
```

où nous avons supposé, par simplification, que les expressions étaient toujours linéaires. Dans un but de documentation, nous avons indiqué les types de **Cp** et **p**; ceci n'est en fait pas obligatoire car le système d'inférence de type de ML est à même de les déduire automatiquement [4]. Dans le programme ci-dessus, les **subst**_fonctions réalisent les substitutions syntaxiques sur les objets dont le type est précisé.

Nous avons très souvent utilisé la notion de *type abstrait de données* présente dans ML (par exemple, **and_pred** et **exp_to_pred** sont tous deux des constructeurs sur des éléments du domaine **Pred**, implémenté comme un type abstrait) et celle de *type concret de données* pour décrire les domaines syntaxiques [par exemple, **Com** où **(I assign E)** représente un *modèle*].

5.2. Traitement des points-fixes

Pour être à même d'implémenter nos spécifications, nous devons maintenir deux invariants.

Premièrement, nous ne devons utiliser que des domaines syntaxiques; eux seuls sont représentables en machine. Comme cela peut être aisément vérifié, tous les domaines définis dans cet article pour exprimer les transformations de ALL sont syntaxiques (ou isomorphes à un domaine syntaxique).

Le second point (beaucoup plus limitant) est que nous devons éliminer les points-fixes introduits par les boucles **while** et **for**. Malheureusement, calculer un point-fixe est en général indécidable [10], donc nous devons les approximer :

- Le traitement des points-fixes introduit par la détection des réductions est simple; il suffit de se limiter à un seul niveau de corps de boucle, auquel cas, aucun point-fixe n'est introduit.

- Pour le cas des prédicats, nous pourrions utiliser l'approche de [9]. En pratique, pour valider rapidement notre paralléliseur, nous avons utilisé une méthode beaucoup plus simple, mais moins puissante; le transformeur de prédicat associé à une boucle **for** simplifie le prédicat de pré-condition par toute variable (si ce n'est pas une réduction) qui est modifiée dans le corps de boucle et ajoute au prédicat restant l'assertion correspondant à l'indice de boucle :

```
(for (I, E, C)). \ p : Pred.
    let E_out = binopr (E, binop_int '+', e_one) and
        ide_1 = mapfilter (\(ide_to_exp I, _).I)
                          (M C assert_true init_niv) in
        and_pred (exp_to_pred (binopr (ide_to_exp I,
                                       binop_bool 'eq',
                                       E_out)))
                 (simplify p (I.ide_l))|
```

où **E_out** correspond à la valeur **E+1** de l'indice de boucle **I** en sortie de boucle et **ide_l** contient la liste des variables scalaires modifiées par le corps de boucle (qu'on obtient en calculant les régions modifiées par **C** et en filtrant celles dont le premier élément est un identificateur). Puisque que nous

n'utilisons que les identificateurs simples modifiés par **C**, il suffit d'utiliser la fonction **M** (qui retourne la liste des régions modifiées) avec des arguments quelconques. On retourne un prédicat qui exprime que **I** vaut **E_out** et que toute information sur les variables modifiées est perdue; **simplify** simplifie le prédicat par la liste d'identificateurs passée comme second argument (cette opération est une application directe de l'algorithme de Fourier [37]).

- Dans le cas des t-prédicats, nous avons décidé d'étendre cette heuristique; nous perdons toute information sur un tableau si ses mise-à-jour dans un corps de boucle utilisent des expressions atomiques (*i.e.*, régions) qui sont également modifiées dans la boucle.

Ce qui est important avec ces différentes approximations est qu'il est possible de prouver leur validité par rapport à la sémantique standard de ALL [21].

5.3. Paralléliseur sémantique

Le paralléliseur sémantique complet correspond à 3 500 lignes de code ML, plus 400 lignes écrites en FranzLisp (pour des raisons d'efficacité) et 140 en MLYacc (une version du compilateur de compilateur Yacc [16] écrite pour ML) pour l'analyseur syntaxique de ALL.

Dans l'exemple donné en début d'article, **parallel_pp** prend comme argument un arbre de syntaxe abstraite, le parallélise et imprime le résultat. L'arbre abstrait est automatiquement construit par l'analyseur MLYacc qui est appelé par ML dès que ce dernier rencontre un guillemet. L'impression est assurée par un pretty-printer qui peut également être considéré comme une autre interprétation non standard de ALL. La parallélisation de la boucle **for** et de son corps n'est possible que grâce à l'élaboration automatique d'un prédicat sur les valeurs possibles de **n**. A notre connaissance, notre paralléliseur sémantique est le seul outil automatique capable de traiter ce type de programme.

L'intégration des t-prédicats n'a pas été complètement achevée, bien que les composants principaux aient été séparément implémentés :

- La première raison en est pratique. La version de ML que nous avons utilisée ([6]) ne supporte pas de vraie compilation séparée. La gestion et maintenance de notre paralléliseur de plus de 3 500 lignes était donc très gourmande en temps; la compilation d'un programme ML est une opération très longue

([6]) ML V6.1 développée par l'Université de Cambridge, l'I.N.R.I.A. et le L.I.T.P. [7].

(des versions successives en FranzLisp, C et assembleur sont successivement générées par le compilateur ML). Heureusement, l'introduction de modules est en cours [25].

• La seconde raison est plus sérieuse car théorique. Bien que l'arithmétique de Presburger complète soit décidable, l'algorithme de décision de Cooper a une complexité énorme. Bien que cela soit, à notre connaissance, le meilleur algorithme pour ce problème, [28] a montré que sa borne-supérieure non déterministe était $2^{2^{pn}}$. Il est clair que des heuristiques plus abordables sont nécessaires (bien que le problème général soit NP-difficile).

6. CONCLUSION

Le travail que nous venons de présenter peut être envisagé de plusieurs points de vue. Il s'agit tout d'abord d'une application en vraie grandeur des méthodes de la sémantique dénotationnelle à un problème non trivial [20]. Dans cette optique, nous pensons avoir montré toute la puissance structurante des notions de sémantique non standard et d'interprétation abstraite. L'intérêt essentiel de ces méthodes est de permettre des preuves. Nous n'en avons pas exhibées dans cet article, mais le lecteur intéressé pourra se reporter à [21]. Il faut bien dire que ces preuves n'offrent pas un grand intérêt du point de vue mathématique : il s'agit essentiellement d'analyses cas par cas. Inversement, ce côté trivial laisse espérer une possibilité d'automatisation ou, à tout le moins, de vérification automatique, ce qui constituerait le plus puissant des outils de mise au point (*voir*, par exemple, l'approche LCF [14]). Il faut bien dire d'ailleurs que, de ce point de vue, le vérificateur de type de ML rend déjà des services inappréciables.

Nous avons ensuite montré qu'il était parfaitement faisable de recueillir automatiquement des renseignements sémantiques sur des programmes non triviaux. Sur ce point, qui constitue le centre de notre travail, nous avons exploité plusieurs notations, dont celles des assertions et de l'exécution symbolique. Nous avons également posé, avec la notion de t-prédicat, les prémisses de l'analyse des accès non-linéaires aux tableaux. Il y a encore beaucoup à faire sur ce sujet, qui recoupe les problèmes posés par l'emploi des pointeurs.

La réalisation d'un paralléliseur n'était pas l'objet central de ce travail. Nous avons cependant montré que nos méthodes permettaient de traiter des cas où la parallélisation repose sur des propriétés mathématiques des opérateurs utilisés. Nous sortons ainsi du cadre usuel des schémas de programme pour pénétrer dans le domaine des programmes interprétés.

Ce travail pose évidemment autant de questions qu'il en résoud. En amont, le problème d'une sémantique formelle du parallélisme reste ouvert. En aval, un outil d'aide à la construction et à la vérification des preuves serait extrêmement utile. De même, un outil aidant au passage d'une écriture ML à un programme dans un langage plus efficace serait le bienvenu. Au plan de la sémantique, d'autres types d'instructions devraient être analysés. Nous avons déjà parlé des accès non linéaires aux tableaux. Un autre domaine est celui des appels de procédures, y compris les récursions.

Il faut noter enfin que les méthodes que nous présentons sont susceptibles de nombreuses applications en dehors de la parallélisation automatique. Il faut citer entre autres l'optimisation et la vérification de programmes (en particulier de la légitimité des accès aux tableaux).

BIBLIOGRAPHIE

1. A. Aho, R. Sethi et J. D. Ullman, *Compilers*, Addison-Wesley, 1986.
2. J. R. Allen et K. Kennedy, *Automatic Loop Interchange*, A.C.M. SIGPLAN Notices, vol. 19, juin 1984, p. 233-246.
3. A. J. Bernstein, *Analysis of Programs for Parallel Processing*, I.E.E.E. Trans. on Elec. Comp., vol. 15, octobre 1966, p. 757-763.
4. L. Cardelli, *Basic Polymorphic Typechecking*, Polymorphism Newsletter 1, vol. II, Bell Labs, janvier 1985.
5. K. Clark et S. Gregory, *PARLOG: Parallel Programming in Logic*, A.C.M. Trans. on Prog. Lang. and Systems, vol. 8, janvier 1986, p. 1-49.
6. D. C. Cooper, *Theorem Proving in Arithmetic without Multiplication*, Machine Intelligence 7, 1972, p. 91-99.
7. G. Cousineau, *The ML Handbook*, draft I.N.R.I.A., mai 1985.
8. P. Cousot et R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximations of Fixpoints*, Proc. of the A.C.M. Conf. on Principles of Prog. Lang., janvier 1977, p. 238-252.
9. P. Cousot et N. Halbwachs, *Automatic Discovery of Linear Restraints among Variables of a Program*, Proc. of A.C.M. Conf. on Principles of Prog. Lang., janvier 1978, p. 84-96.
10. P. Cousot, *Méthodes Itératives de Construction de Points Fixes d'Opérateurs Monotones sur un Treillis : Analyse Sémantique de Programmes*, Thèse d'État, U.S.M.G., Grenoble, 1978.
11. R. J. Duffin, *On Fourier's Analysis of Linear Inequality Systems*, Mathematical Systems, vol. 1, North Holland, 1974.
12. R. W. Floyd, *Assigning Meanings to Programs*, 19 Symp. in Applied Math., American Math. Soc., 1967.
13. D. K. Gifford, P. Jouvelot, J. M. Lucassen et M. A. Sheldon, *FX-87 Reference Manual*, M.I.T./L.C.S. Tech. Rep. 407, septembre 1987.
14. M. J. C. Gordon et R. Milner, *Edinburgh LCF*, Lect. Note in Comp. Sci., n° 78, Springer Verlag, 1979.

15. M. J. C. GORDON, *The Denotational Description of Programming Languages*, Springer Verlag, 1979.

16. J. C. JOHNSON, *YACC: Yet Another Compiler Compiler*, Bells Labs, juillet 1978.

17. P. JOUVELOT, *ML : Un Langage de Maquettage*, Journées d'étude « Nouveaux Langages pour le Génie Logiciel », A.F.C.E.T., 1985.

18. P. JOUVELOT, *Parallelization by Semantic Detection of Reductions*, ESOP86, Lect. Note in Comp. Sci., n° 213, p. 223-236, Springer Verlag, mars 1986.

19. P. JOUVELOT, *Designing New Languages and New Language Manipulation Systems using ML*, A.C.M. SIGPLAN Notices, vol. 21, août 1986, p. 40-52.

20. P. JOUVELOT, *Semantic Parallelization: A Practical Exercise in Abstract Interpretation*, Proc. of the A.C.M. Conf. on Principles of Prog. Lang., janvier 1987.

21. P. JOUVELOT, *Parallélisation Sémantique : Une Approche Dénotationnelle Non-Standard pour la Parallélisation de Programmes Séquentiels*, Thèse de l'Université Paris-VI, Rapport M.A.S.I. 174, février 1987.

22. G. KREISEL et J. L. KRIVINE, *Éléments de Logique Mathématique*, Dunod, 1967.

23. D. J. KUCK, *The Structure of Computers and Computations*, John Wiley and Sons, 1977.

24. K. C. LI, *A Note on the Vector C Language*, A.C.M. SIGPLAN Notices, vol. 21, janvier 1986, p. 49-57.

25. D. MACQUEEN, in *Standard ML*, Edinburgh Univ. Int. Rep. ECS-LFCS-86-2, mars 1986.

26. J. S. MILLER, *MultiScheme*, M.I.T. Ph. D. thesis, juin 1987.

27. F. NIELSON, *Program Transformations in a Denotational Setting*, A.C.M. Trans. on Prog. Lang. and Systems, vol. 7, juillet 1985, p. 359-379.

28. D. C. OPPEN, *A* $2^{2^{2^{pn}}}$ *Upper Bound on the Complexity of Presburger Arithmetics*, J.C.S.S., vol. 16, 1978, p. 323-332.

29. R. H. PERROTT, *A Language for Array and Vector Processors*, A.C.M. Trans. on Prog. Lang. and Systems, vol. 1, octobre 1979, p. 177-195.

30. J. T. SCHWARTZ, *Ultracomputers*, A.C.M. Trans. on Prog. Lang. and Systems, vol. 2, octobre 1980, p. 484-521.

31. D. SCOTT, *The Lattice of Flow Diagrams*, Symp. on Semantics of Algorithmic Lang., Springer Verlag, 1972, p. 311-366.

32. J. E. STOY, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M.I.T. Press, 1977.

33. N. SUZUKI et D. JEFFERSON, *Verification Decidability of Presburger Array Programs*, Proc. of the Conf. on Theo. Comp. Sci., Waterloo, 1977, p. 202-212.

34. N. TAWBI, A. DUMAY et P. FEAUTRIER, *PAF : Un Paralléliseur Automatique pour FORTRAN*, Rapport M.A.S.I. 185, 1987.

35. J. A. TEST, *Multiprocessor Management in the Concentrix Operating System*, USENIX Conf., 1986.

36. Thinking Machines Corp., *The Essential *Lisp Manual*, T.M.C. Tech. Rep. 86.15, avril 1986.

37. R. TRIOLET, *Contribution à la parallélisation automatique de programmes FORTRAN comportant des appels de procédures*, Thèse de Docteur-Ingénieur, Université Paris-VI, décembre 1984.