Correction séquentielle de programmes parallèles dans le modèle asynchrone et mémoire partagée

Thierry Salset

Correction séquentielle de programmes parallèles dans le modèle asynchrone et mémoire partagée

Thierry SALSET

3 juillet 1997

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Le besoin en matière de puissance de calcul est sans cesse plus important. Les machines parallèles représentent un espoir d'accroissement de la vitesse à un rythme bien plus élevé que celui apporté par l'amélioration technologique des processeurs. Au lieu d'accélérer une seule machine, répliquez-la et faites que chacune coopère à la résolution d'une partie du problème pour résoudre la totalité du problème.

Pour exploiter les capacités d'une machine parallèle, le programmeur a le choix entre des langages à parallélisme explicite contenant des constructions parallèles et des langages à parallélisme implicite pour lesquels revient au compilateur ou à l'environnement d'exécution la tâche de trouver quelles sont les sections de code parallélisables. Par exemple, pour les langages fonctionnels qui travaillent avec des expressions plutôt qu'avec des instructions dont l'ordre d'évaluation n'est pas entièrement spécifié dans le source du programme, il est relativement facile à un compilateur d'exploiter cette opportunité en demandant l'évaluation parallèle de plusieurs expressions.

Dans le domaine du calcul scientifique où Fortran est beaucoup utilisé les compilateurs-paralléliseurs tentent d'extraire le maximum de parallélisme en éliminant la séquentialité superflue tout en conservant la sémantique du programme initial. Ils s'intéressent, pour cela, aux boucles `DO` qui représentent une source potentielle importante de parallélisme. Le code qu'ils produisent contient de nouvelles directives de compilation qui font appel à une bibliothèque de fonctions spécialisées propres à chaque constructeur de supercalculateurs, ce qui le rend peu lisible et non portable d'une architecture de machine à une autre. Le degré de parallélisme engendré n'est pas toujours très élevé ; il peut être amélioré, dans le cas d'un outil interactif, par l'intervention de l'utilisateur mais cela suppose une connaissance du programme source et quelques compétences en programmation parallèle. Malgré tout, il offre la possibilité à des non-spécialistes de porter des programmes existants presque sans effort et de rentabiliser immédiatement l'investissement matériel.

Mais pour obtenir les meilleures performances possibles, le programmeur doit directement utiliser un langage possédant des constructions parallèles explicites. Il va rencontrer de nouvelles sources d'erreurs ou de non terminaison : non-déterminisme de l'exécution, inter-blocage de deux processus et de nouveaux facteurs de limitation des performances liés à la gestion des processus concurrents (activation, dés-activation) et aux communications de données entre ces processus. La nature de ces problèmes va dépendre de l'architecture matérielle utilisée et du modèle de parallélisme imposé par le langage.

Il est donc important qu'il dispose d'outils logiciels capables de vérifier statiquement (à la compilation) certaines propriétés sémantiques et de lui indiquer le plus clairement possible la présence d'incohérences comme le ferait un vérificateur de type.

Un groupe de constructeurs de super-calculateurs a créé en 1989 le Parallel Computing Forum (PCF) avec l'intention de standardiser la syntaxe et surtout la sémantique des extensions parallèles de Fortran présentes dans leurs systèmes. Un comité de normalisation ANSI, dénommé X3H5, est issu de ce forum. Il s'est fixé pour objectif d'établir un modèle sémantique de programmation parallèle [29, 26] indépendant de tout langage et de l'appliquer à la définition d'extensions de langages impératifs, C et Fortran [40, 27] pour le moment.

Nous nous intéressons ici à un langage que nous appellerons Fortran X3H5, obtenu en ajoutant les constructions parallèles X3H5 à Fortran 77.

Introduction du parallélisme	PARALLEL DO *contrôle* *bloc* END PARALLEL DO PARALLEL SECTIONS *sections* END PARALLEL SECTIONS
Nouveaux types	EVENT ORDINAL
Synchronisation	CLEAR(*event/ordinal*) POST(*event/ordinal*) WAIT(*event/ordinal*)

TAB. 1.1 – *Constructions parallèles X3H5*

Les modèles de machines parallèles actuelles peuvent être classés dans deux catégories : SIMD (Single Instruction [stream] Multiple Data [stream]) ou MIMD (Multiple Instruction [stream] Multiple Data [stream]) selon la terminologie introduite par Flynn. Une machine est dite SIMD si elle ne possède qu'une seule unité de contrôle ; il peut s'agir d'unités vectorielles

qui sont adjointes à un processeur scalaire ou d'un tableau de processeurs relié à un ordinateur traditionnel qui distribue la même instruction à tous les processeurs qui l'exécutent de manière synchrone sur des données différentes. Une machine est dite MIMD si les unités de calcul sont capables d'opérer de façon indépendante et asynchrone les unes des autres et sur des données différentes ; c'est le cas d'un multi-processeur ou d'un groupe de machines indépendantes connectées entre elles par un réseau. Notons que pour ces deux catégories de machines, la mémoire physique peut être distribuée sur chaque processeur, partagée ou organisée selon une hiérarchie mélangeant les deux types.

Le modèle de programmation décrit le modèle de parallélisme tel qu'il est vu par le programmeur. C'est un modèle abstrait, indépendant du modèle d'architecture parallèle. Divers modèles existent ; les plus intéressants sont : le modèle dit à parallélisme de donnée (synchrone ou mono-thread et espace d'adressage global) dont se réclame le langage Fortran HPF [12], le modèle multi-thread (asynchrone) et mémoire partagée retenu par le comité X3H5 et le modèle multi-thread et passages de messages représenté notamment par les bibliothèques PVM [34, 14] et MPI [13]. Le premier et le dernier modèle cités représentent deux extrêmes du point de vue de la difficulté de programmation pour un utilisateur habitué à la programmation séquentielle.

Le parallélisme de données est le modèle le plus simple car le contrôle reste séquentiel. Il peut tirer partie d'une grande variété d'architectures de machines parallèles SIMD ou MIMD, mais la classe d'algorithmes pour laquelle il est efficace est restreinte.

Avec le modèle distribué, le programmeur a en charge la coordination et l'échange des données via l'envoi et la réception explicites de messages. C'est potentiellement le modèle le plus efficace mais la mise au point du programme peut être une tâche difficile pour des applications complexes particulièrement celles où l'échange des données doit s'effectuer selon un schéma de communication irrégulier et dynamique ou à un niveau de granularité fin.

La proposition de norme X3H5 décrit un modèle de parallélisme multi-thread et à mémoire partagée. Ce modèle de programmation est naturellement celui des multi-processeurs à mémoire réelle partagée, machines très répandues dotées d'un petit nombre de processeurs (typiquement quelques unités). Mais, des travaux de recherche récents permettent de l'implanter efficacement sur des architectures distribuées à passage de messages sous la forme d'une bibliothèque logicielle [3, 22]. Ces nouvelles fonctionnalités donnent au programmeur, disposant d'un réseau de stations de travail, l'illusion d'avoir une machine à mémoire partagée globale pour un coût supplémentaire nul. Cela devrait contribuer à populariser ce modèle de programmation.

Une des difficultés les plus importantes que l'on rencontre dans le développement de programmes avec ce modèle de parallélisme asynchrone et mémoire partagée est due à l'accès désordonné et concurrent de plusieurs pro-

cessus à la mémoire partagée, ce qui rend les résultats imprévisibles. Pour y remédier il faut insérer dans le programme des instructions de synchronisation.

Dans le programme ci-dessous, l'instruction `PARALLEL DO` indique la possibilité que chaque instance du corps de la boucle soit attribuée à un processus différent et exécutée par lui dès qu'il le peut. L'ordre d'exécution de chaque instance du corps de la boucle n'est plus fixé comme en séquentiel. L'état mémoire à la sortie de la boucle est donc indéterminé. Le programme est dit « non standard conforming» dans la proposition X3H5.

Par exemple, intéressons-nous à l'emplacement mémoire $A(4)$. Pour $I = 4$ il est lu par b mais quelle valeur y a t-il ? Ce peut être la valeur calculée par a pour $I = 1$ ou bien la valeur présente avant l'entrée dans la boucle ou encore quelque chose d'indéterminé si les opérations élémentaires dont se composent l'instruction d'écriture $a(I = 1)$ et l'instruction de lecture $b(I = 4)$ sont exécutées de manière entrelacée. On dit qu'il y a un conflit mémoire entre les deux instructions a et b.

```
    PARALLEL DO I=1,N
b:    ... = A(I)
a:    A(I+3) = ...
    END PARALLEL DO
```

Pour rendre ce programme déterministe, il faut rajouter des synchronisations. C'est le rôle joué dans cet exemple par les instructions `POST` et `WAIT` appliquées à des variables d'événements $E(I)$. La sémantique de ces opérations requiert qu'un `WAIT` ne peut être franchi que si un `POST` du même événement a été exécuté auparavant. On a désormais, dans notre exemple, le schéma de précédence suivant où $\rightsquigarrow$ représente une relation de précédence induite par une synchronisation :

$$a(I = 1) \rightarrow p(I = 1) \rightsquigarrow w(I = 4) \rightarrow b(I = 4)$$

Ainsi, une séquentialité partielle entre différentes instances du corps de la boucle parallèle a été rétablie.

Pour ce travail, nous avons choisi de nous restreindre à un sous-ensemble des extensions parallèles proposées par X3H5 qui comprend les boucles parallèles, les sections de code parallèles et les instructions de synchronisation par événements. Un programme utilisant ces constructions possède une version séquentielle obtenue en « déparallélisant » les boucles et en remplaçant les instructions de synchronisation par des `CONTINUE`. Pour ce qui est de Fortran lui-même nous nous sommes placés dans un cadre mono-procédural et avons retenu les instructions d'affectation, le `IF` structuré et les boucles `DO`. L'élimination du `GOTO` assure la terminaison en version séquentielle de tous les programmes. A part ça, ces restrictions n'enlèvent rien à la généralité des résultats obtenus.

```
    DO I=1,3                              DO I=1,3
      POST E(I)                             CONTINUE
    END DO                                END DO
    PARALLEL DO I=1,N                     DO I=1,N
w:    WAIT E(I)                             CONTINUE
b:    ... = A(I)                            ... = A(I)
a:    A(I+3) = ...                          A(I+3) = ...
p:    POST E(I+3)                           CONTINUE
    END PARALLEL DO                       END DO
```

FIG. 1.1 – *Un programme parallèle et sa version séquentielle*

Notre but est de prouver qu'un programme parallèle a le même comportement que sa version séquentielle. Moyennant certaines hypothèses, il est prouvé dans [7] que cette propriété résulte de la préservation des dépendances, définies sur la version séquentielle, par le flot de contrôle et les synchronisations. Cette notion de «correction séquentielle» peut sembler restrictive car elle ne s'applique pas aux programmes non déterministes conçus comme tels. Elle est néanmoins justifiée pour la programmation d'applications numériques. Dans ce domaine, l'habitude est plutôt de paralléliser des programmes séquentiels et l'emploi d'algorithmes non déterministes est rare.

La condition essentielle de préservation d'une dépendance est une relation d'implication entre la relation de dépendance et une relation de précédence. La relation de dépendance Dep est celle classiquement utilisée en parallélisation automatique ; elle relie deux instructions a et b si a est exécutée avant b dans la version séquentielle et s'il existe une région de la mémoire auxquelles elles accèdent, l'une au moins en écriture. Dans l'exemple de la figure 1.1, la relation de dépendance entre l'instruction a et l'instruction b s'exprime de la façon suivante :

$$\mathrm{Dep}_{a,b}(x,y) = (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x+3 = y)$$

la variable x (resp. y) étant associée à l'instruction a (resp. b) qui se situent dans un nid de boucles de profondeur 1.

De manière analogue, nous définissons une relation de synchronisation Sync entre une instruction `POST` et une instruction `WAIT` qui est vraie si ces deux instructions accèdent au même événement dans un ordre quelconque. Dans l'exemple, la relation de synchronisation entre p et w est :

$$\mathrm{Sync}_{p,w}(x,y) = (x+3 = y)$$

la variable x (resp. y) étant associée ici à l'instruction p (resp. w).

À chaque itération correspond une nouvelle *instance* de chaque instruction présente dans le corps d'une boucle. La relation de précédence Pre est

définie à partir d'un graphe de flot dont les sommets sont des instances d'instructions et dont les arcs représentent la réunion de la relation de précédence Pre^0 due au contrôle et d'une relation de précédence induite par le mécanisme d'attente résultant de l'exécution de couples `POST`/`WAIT` sur le même événement. Toujours dans notre exemple, la formule de précédence de contrôle entre a et p est :

$$\mathrm{Pre}^0_{a,p}(x, y) = (x = y)$$

Nous représentons sur la figure 1.2 un extrait du graphe de flot du programme de la figure 1.1 pour quatre itérations successives.

L'égalité des événements ne suffit pas à assurer la précédence de synchronisation entre une instance π d'un `POST` et une instance ω d'un `WAIT`, il faut aussi que π et ω soient exécutées, donc nous avons seulement :

$$\mathrm{Exe}^{\mathsf{s}}(\pi) \wedge \mathrm{Sync}(\pi, \omega) \wedge \mathrm{Exe}^{\mathsf{s}}(\omega) \Rightarrow \mathrm{Pre}(\pi, \omega)$$

où $\mathrm{Exe}^{\mathsf{s}}$ est un prédicat d'exécution défini sur la version séquentielle. Une instance d'instruction α est exécutée avant une instance d'instruction β s'il existe un chemin dans le graphe de flot allant de α à β, c'est à dire $\mathrm{Pre}(\alpha, \beta)$ vrai.

Comme les sommets du graphe sont des instances d'instructions, leur ensemble est infini bien qu'un nombre fini d'itérations soit réellement exécuté et que nous ne considérions que des chemins finis.

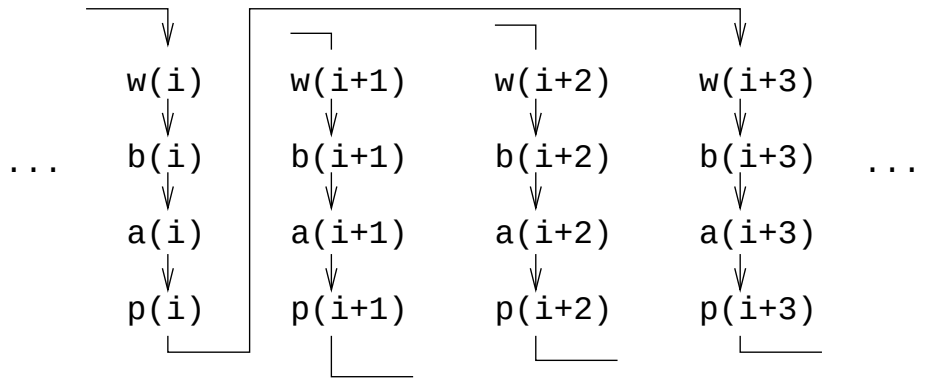

FIG. 1.2 – *Graphe de flot*

Ce graphe de flot infini n'est pas exploitable par un algorithme. Nous le transformons, par projection, en un graphe fini, dont les sommets sont des instructions du programme au sens d'éléments syntaxiques, et dont les arcs sont étiquetés par des formules de logique du premier ordre ; la projection consiste à associer à chaque instance l'instruction correspondante. La figure

1.3 montre le graphe de précédence résultat de la projection du graphe de flot de la figure 1.2.

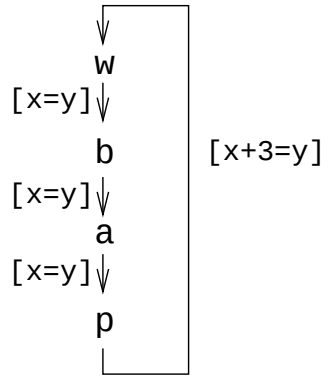

FIG. 1.3 – *Graphe de précédence*

Un découpage du programme en blocs permet d'agréger les parties du programme à l'intérieur desquelles la précédence est seulement due au contrôle. Le graphe de blocs du programme de la figure 1.1 est donné figure 1.4. Les arcs de ce graphe sont également étiquetés par des formules. Ces blocs sont plus généraux que les blocs de base [1] dans la mesure où ils peuvent contenir des instructions structurées quelconques mais sans synchronisations.

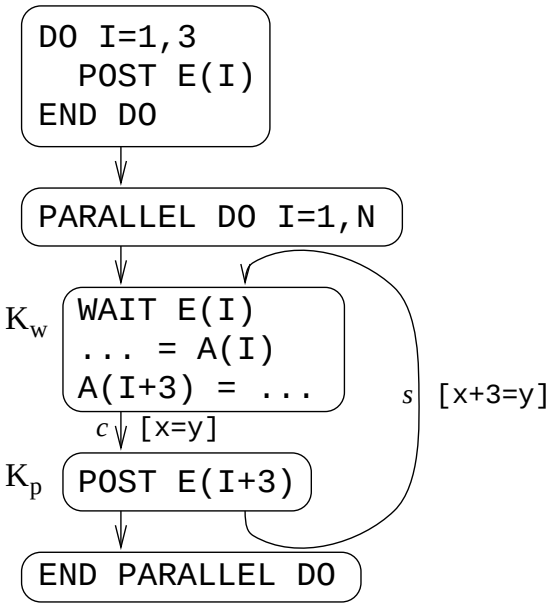

FIG. 1.4 – *Graphe de blocs*

Sur ce graphe de blocs, nous calculons une approximation de la formule de précédence par énumération de chemins de longueurs croissantes entre l'instruction source de la dépendance et l'instruction cible. Dans le cas où les expressions du programme sont linéaires, un algorithme tente de prouver que la formule de correction, qui est alors une formule de Presburger, est une tautologie. Il est fait appel pour la réduction des systèmes au test Omega [31] conçu à l'Université du Maryland. L'algorithme manipule des systèmes d'équations et d'inéquations sous forme symbolique. Il ne fournit pas seulement un résultat binaire (oui/non), mais un ensemble réduit de contraintes qui peut être interprété comme une condition de validité à respecter.

Dans l'exemple, les instructions a et b sont contenues dans un même bloc K_w, il y a un arc de contrôle c de K_w à K_p et un arc de synchronisation s de K_p à K_w avec la formule $\mathrm{Sync}_{p,w}(x,y) = (x+3=y)$; les chemins de K_w à lui-même sont de la forme $cs(cs)^*$ où la notation * signifie la répétition d'un chemin 0 ou n fois. Si nous commençons par le chemin le plus court, cs, la formule de précédence $\mathrm{Pre}_{cs}(x,y)$ associée à ce chemin est $\mathrm{Pre}^0_{a,p}(x,z_1) \wedge \mathrm{Exe}_p(z_1) \wedge \mathrm{Sync}_{p,w}(z_1,z_2) \wedge \mathrm{Exe}_w(z_2) \wedge \mathrm{Pre}^0_{w,b}(z_2,y)$. La formule de correction à prouver est :

$$\forall x \forall y \exists z_1 \exists z_2 \, (\mathrm{Dep}_{a,b}(x,y) \wedge \mathrm{Exe}^{\mathsf{s}}_a(x) \wedge \mathrm{Exe}^{\mathsf{s}}_b(y) \Rightarrow \mathrm{Pre}_s(x,y,z_1,z_2))$$

soit encore :

$$\begin{array}{l} \forall x \forall y \exists z_1 \exists z_2 \\ ((1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x+3=y) \\ \Rightarrow \\ (x = z_1) \wedge (1 \leqslant z_1 \leqslant N) \wedge (z_1+3=z_2) \wedge (1 \leqslant z_2 \leqslant N) \wedge (z_2 = y)) \end{array}$$

Ce problème est transmis au test Omega, qui élimine les variables z_1 et z_2, puis par l'algorithme dit du *gist* prouve que l'implication est vraie. En ce qui concerne la dépendance entre a et b, ceci prouve la correction séquentielle du programme.

Un prototype logiciel a été développé avec le générateur d'environnements Centaur qui fournit des formalismes de spécifications syntaxique et sémantique facilitant la conception d'outils spécifiques à un langage donné et leur intégration sous une même interface. Ce prototype intègre, sous un même environnement, un analyseur syntaxique, un « pretty-printer » et un vérificateur de type fonctionnant sur l'ensemble de la syntaxe Fortran 77 étendu aux propositions d'X3H5. L'algorithme de vérification de la correction séquentielle, présenté ici, a été intégré à cet environnement. Il est écrit en Lisp et utilise les fonctionnalités de Centaur pour la manipulation de l'arbre de syntaxe abstraite. La réalisation de ce prototype logiciel montre la faisabilité et l'intérêt de notre démarche.

De nombreux travaux de recherche portent sur l'analyse statique des programmes parallèles dans le cadre du modèle mémoire partagée et parallélisme de contrôle. Ce peut être pour résoudre des problèmes de compilation et d'optimisation du code ou pour faire du debugging statique comme dans notre cas.

En optimisation, [25, 17] s'intéressent au passage du modèle mémoire partagée au modèle mémoire distribuée. Le maintien de la cohérence mémoire est obtenu par l'échange de données entre les processus. Ces communications peuvent être optimisées par une analyse du flot de données dans les segments de code parallèle.

En vérification, la détection des conflits d'accès à la mémoire a été étudiée par D. Callahan, K. Kennedy et J. Subhlok [6]. Ils s'intéressent aux programmes parallèles contenant des instructions explicites de synchronisation par événements et introduisent un graphe de flot étendu aux programmes parallèles pour le calcul des précédences.

L'analyse des conflits mémoire dans les programmes séquentiels a fait l'objet de nombreux travaux de recherche pour la parallélisation automatique de programmes [2, 37, 4, 11, 28, 39, 41, 9]. Nous utilisons certaines des méthodes développées dans ce cadre mais dans un contexte différent.

La thèse que nous soutenons ici est que la correction séquentielle des programmes parallèles est pertinente pour ce langage, ceci pour trois raisons : l'orientation du langage vers le calcul numérique, la fondation théorique des résultats utilisés et le caractère opérationnelle de la vérification.

La pratique des numériciens est d'utiliser des machines parallèles pour tenter d'accélérer l'exécution d'algorithmes bien connus optimisés pour les machines séquentielles plutôt que de concevoir de nouveaux algorithmes pour un modèle de parallélisme donné.

En apparence, la condition principale de correction Dep $\Rightarrow$ Pre coule de source et n'a pas besoin de fondements théoriques. En fait, l'énoncé précis des hypothèses du théorème d'équivalence sémantique ainsi que sa preuve dans le cadre d'un langage de programmation comme celui que nous étudions, ne sont pas triviaux. Le travail de recherche de G. Caplain le montre bien.

Enfin, nous apportons une méthode de preuve opérationnelle sur des cas réels qui peut aider à la détection d'erreurs de programmation difficiles à repérer. La résolution exacte des systèmes de contraintes arithmétiques et leur manipulation formelle permettent de donner des conditions suffisantes sur les paramètres du programme pour que la correction séquentielle soit assurée.

Plan

Le deuxième chapitre présente la syntaxe du langage que nous étudions et nous décrivons informellement son modèle d'exécution. Nous parlons de

la notion de programme « non standard conforming ». Nous expliquons ce que nous entendons par programme correct.

Le chapitre 3 est consacré à la définition d'espace d'itération, d'instance d'instruction, de dépendances de données et de relation de synchronisation.

Au chapitre 4, nous donnons la définition de la précédence dans nos programmes parallèles qui fait intervenir la précédence due au contrôle et la précédence due aux synchronisations. La précédence de contrôle est explicitée. Nous énonçons le théorème principal de correction.

Le chapitre 5 présente un algorithme de vérification. Nous montrons comment nous calculons le prédicat de précédences à l'aide d'une décomposition par blocs du programme et comment sont résolus les systèmes d'équations.

Au chapitre 6 nous donnons quelques détails d'implémentation du prototype logiciel et présentons l'algorithme de construction du graphe de blocs.

Le chapitre 7 présente les résultats obtenus avec le système sur plusieurs exemples. Nous montrons les possibilités qu'apporte le traitement symbolique des formules mais aussi les limitations de l'algorithme employé.

Au chapitre 8, nous présentons une méthode générale pour exprimer un ensemble de chemins dans un graphe. Dans les cas où nous pouvons associer une formule de Presburger à cette expression de chemin, on est ramené au problème précédent de la preuve d'une implication. Nous donnons un exemple d'utilisation de cette méthode pour traiter un cas qui faisait échouer la méthode précédente.

Chapitre 2

Fortran X3H5 : syntaxe et modèle d'exécution

La spécification de notre langage, que nous présentons ici, s'inspire du modèle proposé par le standard X3H5.

Nous commençons par décrire la syntaxe du langage et les restrictions que nous avons faites puis nous expliquons informellement le modèle d'exécution d'un programme et les problèmes qui peuvent en résulter.

Ce que nous donnons ici, en ce qui concerne le modèle d'exécution, n'est qu'une interprétation de ce qui est proposé dans le projet de norme X3H5.

2.1 Syntaxe

Le langage que nous étudions est un sous-ensemble de Fortran 77 étendu avec un sous-ensemble des constructions de gestion du parallélisme du projet de norme ANSI X3H5. Nous avons retenu les instructions `PARALLEL DO` et `PARALLEL SECTIONS` qui combinent exécution parallèle et distribution du travail, et les instructions de synchronisation opérant sur des variables de type `EVENT` ou `ORDINAL`. Nous avons écarté l'instruction `PARALLEL` qui introduit une région parallèle et les instructions `PDO` et `PSECTIONS` qui utilisées à l'intérieur d'une région parallèle se comportent comme `PARALLEL DO` et `PARALLEL SECTIONS` mais permettent la réutilisation des processus. Nous excluons les mécanismes de synchronisation par exclusion mutuelle (sections critiques et verrous), et les instructions modifiant la portée des identificateurs : `SCOMMON` et `NEW`.

Notre sous-ensemble de X3H5 autorise la déclaration explicite de variables *privées* au début d'un `PARALLEL DO` ou d'une `PARALLEL SECTIONS`. Nous décrirons plus loin comment tenir compte de ces déclarations pour le calcul des dépendances.

Le sous-ensemble séquentiel comprend l'affectation, le `IF` structuré, la boucle `DO` et tous les types de données scalaires ou tableaux ; nous ex-

cluons les `GOTO`, les `COMMON`, les appels de procédure et les instructions `DATA`, `EQUIVALENCE` et `SAVE`.

2.1.1 Syntaxe abstraite

La syntaxe abstraite de notre langage est donnée table 2.1. Les objets non terminaux y figurent dans une police « télétype », les terminaux représentant un ensemble d'entités lexicales sont en caractères normaux et les littéraux sont en italique.

La présence du symbole * après une entité syntaxique signifie zéro ou n occurences de l'entité. Par exemple, `section`* est une suite de longueur éventuellement nulle de sections. On choisit de représenter ces suites de longueur variable par des listes, ce qui nous permet d'utiliser les notations habituelles sur les listes :

- $[s_1, s_2, \ldots, s_n]$ pour figurer la liste en extension;
- $x :: l$ pour distinguer l'élément de tête x du reste de la liste l;
- $l_1 @ l_2$ pour décomposer la liste en une liste préfixe l_1 et une liste suffixe l_2.

Nous utiliserons une notation « pointée » pour désigner un fils particulier d'un sommet de l'arbre de syntaxe abstraite d'un programme : si p est une `psections` alors p.`section`* est la liste de ses sections.

On définit *op* la fonction qui à un arbre de syntaxe abstraite fait correspondre sa racine qui est un opérateur syntaxique. Les opérateurs sont les nœuds terminaux et non-terminaux du langage. Par exemple, identifier et `assign` sont des opérateurs mais `stmt` qui désigne un groupe de descendants possibles d'un opérateur donné n'est pas un opérateur.

Différents types d'expression ont été distingués : expression d'indices (`iexp`), booléenne (`bexp`) et générale (`exp`). Leur syntaxe n'est pas fournie pour ne pas surcharger la table.

2.1.2 Restrictions syntaxiques

Dans un programme écrit selon cette syntaxe, nous distinguons trois types de variables : les *paramètres*, les *variables d'indice de boucle* et les *références*.

Nous appelons paramètres d'un programme toute variable initialisée au lancement du programme et plus jamais modifiée par la suite. Une *instance de programme* est obtenue à partir d'un programme en affectant des valeurs constantes aux paramètres.

Une variable d'indice (catégorie `index` dans la table 2.1) est définie dans un en-tête de boucles `DO` ou `PARALLEL DO` et est utilisée comme compteur d'itérations.

```
program     →  declaration* body
declaration →  name dimension* type
dimension   →  integer
type        →  integer | real | boolean | event
body        →  stmt*
stmt        →  assign | if | do | pdo | psections | epost | ewait | clear
               | opost | owait | set | continue
assign      →  lhs exp
lhs         →  name iexp*
name        →  identifier
iexp        →  ...
exp         →  ...
if          →  bexp thenbody elsebody
thenbody    →  stmt*
elsebody    →  stmt*
bexp        →  ...
do          →  head body
head        →  index lb ub
index       →  name
lb          →  iexp
ub          →  iexp

pdo         →  head declaration* body
psections   →  declaration* section*
section     →  sectionhead body
sectionhead →  name sectionwait
sectionwait →  name*
epost       →  lhs
ewait       →  lhs
clear       →  lhs
opost       →  lhs iexp
owait       →  lhs iexp
set         →  lhs iexp
```

TAB. 2.1 – *Syntaxe abstraite du langage*

Nous appelons référence, un identificateur muni d'une liste d'expressions éventuellement vide (catégorie `lhs` dans la table 2.1), qui n'est ni un paramètre ni une variable d'indice. Une référence désigne un emplacement mémoire.

Nous imposons les restrictions suivantes sur la syntaxe :

1. les bornes d'une boucle (`lb`, `ub` dans la table 2.1) séquentielle ou parallèle ne dépendent que des indices des boucles englobantes et de paramètres du programme

2. la variable d'indice d'une boucle n'est utilisée qu'en lecture dans le corps de la boucle et n'est pas définie au-delà de la fin de la boucle

3. les expressions d'indice des tableaux ne dépendent que des indices des boucles englobantes et de paramètres du programme

4. les expressions apparaissant dans les instructions sur les ordinaux, `opost`, `owait` et `set`, ne contiennent pas de variables autres que des variables d'indice et des paramètres

Les restrictions 1 et 2 permettent de définir l'espace d'itération d'une instruction et la notion d'instance (Cf. 3.1). Les restrictions 3 et 4 permettent d'affirmer que tout emplacement mémoire modifié ou lu par une instruction ne dépend que des paramètres du programme et de l'instance d'instruction et non de l'environnement mémoire tout entier ; c'est essentiel pour pouvoir définir statiquement les formules arithmétiques (pour les dépendances, les synchronisations, etc ...) dont nous aurons besoin par la suite. Dans la syntaxe abstraite de notre langage, nous avons fait appartenir les expressions concernées par ces restrictions à la catégorie syntaxique `iexp` plutôt qu'à la catégorie plus générale `exp`. Une « `iexp` » ne peut pas contenir de références alors qu'une « `exp` » peut contenir des variables de tous types.

Tous les programmes courants ne vérifient pas ces restrictions, même si elles définissent un style de programmation assez recommandable. Par contre, aucune restriction n'est faite pour le moment sur les expressions booléennes des `IF` ; elles peuvent donc contenir des variables quelconques (c'est-à-dire des références dans le sens vu plus haut), ce qui est normal pour des programmes courants. Dans les segments de programme contenant des synchronisations, nous serons cependant amenés à imposer d'autres restrictions.

Les expressions constituant les bornes des boucles sont évaluées une fois, à l'entrée dans la boucle, comme cela est spécifié par la norme Fortran 77, et non réévaluées à chaque itération.

Les boucles sont *normalisées*, leur incrément est de 1.

De nouvelles restrictions seront introduites plus loin, telle la linéarité des expressions d'indice et de bornes de boucle exigée par les algorithmes que nous employons. Il faut toutefois noter que ces restrictions ne sont pas plus

sévères que celles habituellement imposées dans le domaine de la parallélisation automatique.

2.2 Modèle sémantique

2.2.1 Modèle de programmation

Le standard décrit un modèle abstrait de système de calcul parallèle dans lequel plusieurs unités de calcul peuvent être impliquées dans l'exécution d'un programme, chaque unité exécutant un flot d'instructions indépendant. C'est un modèle *asynchrone* à *parallélisme de contrôle* et *mémoire partagée*. D'autres modèles de parallélisme existent et parmi eux, le modèle à *parallélisme de données* souvent retenu pour programmer les machines à architecture massivement parallèle. Nous passons en revue brièvement ces différents modèles.

Parallélisme de données

Dans le modèle à parallélisme de données, le contrôle reste séquentiel. Le parallélisme est situé au niveau des types de données et de leur manipulation.

Ce modèle a été développé initialement pour exploiter des machines dotées d'un grand nombre de processeurs de faible puissance munis d'une mémoire locale de petite taille. Tous les processeurs recoivent le programme instruction par instruction d'un contrôleur global et l'exécutent sur des données différentes. Ce mode d'exécution parallèle synchrone est qualifié de SIMD (Single Instruction Stream Multiple Data).

Maintenant, il existe des compilateurs qui traduisent des programmes écrit dans ce modèle vers des machines à contrôle distribué capables d'exécuter leur propre programme sur leurs propres données. Ce mode de fonctionnement est dit MIMD (Multiple Instruction Stream Multiple Data).

Le langage HPF (High Performance Fortran) [12] utilise ce modèle de programmation. Le compilateur prend en charge le processus de distribution des données sur les processeurs. Il peut être guidé par des directives incluses dans le programme. Cette approche quasi-automatique de la parallélisation est valable pour des algorithmes qui ont un schéma de communication calculable statiquement.

Parallélisme de contrôle

Au contraire, dans le modèle à parallélisme de contrôle, le parallélisme est situé au niveau des structures de contrôle. Il s'agit alors de spécifier les instructions qui doivent être exécutées en parallèle. Les données sont manipulées comme en séquentiel. Le parallélisme réel n'est obtenu que sur des machines MIMD ou multi-ordinateurs.

L'unité d'exécution séquentielle de base est le processus. Le parallélisme provient de la possibilité de mettre en œuvre plusieurs processus pour l'exécution de certains segments de code. Chaque processus a un compteur d'instructions qui lui est propre. Aucune hypothèse n'étant faite sur la vitesse relative des processus, ils évoluent de manière asynchrone. La seule ressource globale est le medium de communication : une mémoire partagée ou un réseau d'inter-connexion par lequel des messages transitent. C'est par son intermédiaire que les processus peuvent se synchroniser.

On distingue deux sous-classes à ce modèle de parallélisme suivant le modèle de communication inter-processus : mémoire partagée ou passage de messages.

Mémoire distribuée et passage de messages Si la mémoire est distribuée, à chaque processus P_i est associé une mémoire M_i adressable uniquement par P_i. Ce modèle a été formalisé par Hoare : c'est le modèle CSP (Communicating Sequential Processes) [20] associé à un langage, Occam, et à une machine parallèle, le réseau de Transputers. Ce modèle permet de définir des processus à l'aide d'opérations générales (la composition séquentielle, la composition parallèle, l'alternative gardée, la conditionnelle) à partir de processus élémentaires (l'affectation, l'envoi de message, la réception de message, et `SKIP` qui ne fait rien).

En CSP, la communication inter-processus se fait par échanges de messages et rendez-vous ; les instructions de base sont l'émission et la réception bloquantes de valeurs le long d'un canal donné.

$$\ldots \quad C!v \quad \ldots \parallel \ldots \quad C?x \quad \ldots$$

v est la valeur émise, x est la variable de réception, et C est un canal.

Toutefois, bien qu'il ait fait l'objet d'une description formelle rigoureuse, le langage Occam n'a pas connu un grand succès auprès des utilisateurs. Des environnements de passage de messages tels que PVM (Parallel Virtual Machine) [14] et MPI (Message Passing Interface) [13] sont devenus des standards de fait pour la programmation des multi-ordinateurs et des réseaux de stations de travail en raison de leur grande portabilité. Ils se présentent sous la forme d'une bibliothèque de fonctions C de relativement bas niveau pour la création et la destruction de processus, l'envoi et la réception de messages, le conditionnement et l'extraction des données composant les messages etc... La coordination des processus est obtenue à travers l'envoi explicite de messages dont seule la réception est bloquante.

Mémoire partagée La communication entre les processus se fait grâce à un espace d'adressage partagé réel ou simulé par logiciel. C'est une communication implicite : il n'y a pas d'instruction particulière de transmission et de réception d'informations. Les informations sont « transmises » lorsqu'un

processus écrit une valeur dans la mémoire partagée, et qu'un autre processus vient ensuite la lire. Ce mécanisme d'échange de l'information est asynchrone et ne conduit à aucun blocage ni pour le processus qui écrit ni pour celui qui lit. Dans ce modèle la synchronisation doit être explicite, réalisée par des instructions spéciales.

Le parallélisme vient de la possibilité de créer plusieurs processus séquentiels qui opèrent simultanément sur la mémoire partagée. Chaque case mémoire voit donc une suite d'opérations (lecture ou écriture) réalisées par des processus différents ; cette suite est un entrelacement des suites d'opérations des différents processus. Comme les processus n'évoluent pas nécessairement à la même vitesse d'une exécution à une autre, le résultat de cet entrelacement n'est pas prévisible : on dit qu'il est non déterministe. Considérons par exemple le terme $x := 1 \parallel x := 2$. Les deux entrelacements possibles $x := 1; x := 2$ et $x := 2; x := 1$ n'ont pas le même effet ; x peut valoir 1 ou 2. Ces deux affectations sont en conflit car elles sont concurrentes et écrivent toutes deux sur une même zone de la mémoire.

Maintenant si l'on considère le cas de $x := x + 1 \parallel x := 2 * x$. Supposons que x vale 2 initialement. À première vue, les deux exécutions possibles $x := x + 1; x := 2 * x$ et $x := 2 * x; x := x + 1$ conduisent à une valeur finale pour x de 6 et 5 respectivement. Mais si chacune des instructions $x := x + 1$ et $x := 2 * x$ est réalisée par des opérations plus élémentaires de chargement et déchargement de la mémoire vers les registres internes du processeur et d'opérations arithmétiques opérant sur les valeurs contenues dans les registres, alors l'entrelacement des deux affectations entraîne l'entrelacement des opérations élémentaires réalisées par le processeur.

$$\begin{array}{lll} 1: & R \leftarrow x & \\ 2: & R \leftarrow R + 1 & \parallel \\ 3: & x \leftarrow R & \end{array} \qquad \begin{array}{lll} 4: & R' \leftarrow x & \\ 5: & R' \leftarrow R' * 2 & \\ 6: & x \leftarrow R' & \end{array}$$

L'ordre 1-2-4-5-6-3 conduit à $x = 3$, et l'ordre 4-1-2-3-5-6 conduit à $x = 4$, valeurs incohérentes et ne correspondant à aucune des deux interprétations séquentielles du $\parallel$. Pour éviter ce problème, il est nécessaire de pouvoir rendre une suite d'opérations *atomique* afin d'interdir tout entrelacement pendant la durée de son exécution.

Pour assurer l'atomicité, de nouvelles structures de contrôle sont disponibles. Dans le modèle X3H5, il est possible de placer une séquence d'instructions à l'intérieur d'une *section critique*. Un processus entrant dans une section critique se voit attribuer l'exclusivité de l'accès aux variables partagées référencées dans la séquence pour toute la durée de l'exécution de la séquence.

Ce problème de l'atomicité existe aussi dans les systèmes d'exploitation multitâches où le parallélisme est simulé ; un petit intervalle de temps de calcul est attribué à chacune des tâches actives qui s'exécutent ainsi de manière entrelacée.

Ce modèle de communication inter-processus s'accommode aussi bien d'une architecture à mémoire partagée réelle que d'une architecture à mémoire distribuée et passages de messages. Dans le cas d'une architecture distribuée, la mémoire partagée est simulée. Les accès mémoire non locaux engendrent des requêtes sur le réseau et déclenchent des actions de maintien de la cohérence. Différentes stratégies de maintien de la cohérence ont été proposées. La stratégie la plus simple, dite de *consistance séquentielle*, suppose que les opérations sur la mémoire surviennent les unes après les autres comme sur une machine uni-processeur ; des messages doivent alors être envoyés à tous les processeurs à chaque fois qu'un objet partagé est modifié. Les stratégies plus évoluées tentent de diminuer la quantité de données transférées en associant les actions de maintien de la cohérence avec les opérations de synchronisation et/ou en minimisant le nombre de processus concernés par la réception des messages de mise à jour des objets partagés.

Le modèle X3H5 n'exige pas, pour l'implémentation du modèle, la consistance séquentielle mais seulement que les actions nécessaires à la cohérence des objets partagés soient effectuées aux points de synchronisation explicites ou implicites qu'il définit. Il est de la responsabilité du programmeur d'assurer la consistance séquentielle, si son algorithme le demande, en employant les instructions de synchronisation appropriées.

2.2.2 Processus

En X3H5 plusieurs processus peuvent participer à l'exécution d'un programme. Un processus X3H5 est défini comme une unité séquentielle d'exécution proche du modèle de thread tel qu'il existe dans les systèmes d'exploitation. Le thread se démarque d'un processus Unix, en ce qu'il partage la plupart de ses ressources et notamment l'espace d'adressage avec ses semblables. Son contexte d'exécution étant plus réduit, les opérations de création, d'initialisation, de synchronisation et de changement de contexte sont plus rapides.

Un processus peut se trouver dans l'un des trois états suivants : *actif*, *bloqué* ou *en attente de travail*. Un processus actif se compose d'un *pointeur d'instruction* et d'un *environnement de données*.

2.2.3 Modèle conceptuel d'exécution

Le parallélisme est introduit au moyen des constructions structurées `PARALLEL DO` et `PARALLEL SECTIONS` qui peuvent être imbriquées.

L'exécution d'un programme commence avec un processus initial unique jusqu'à la rencontre d'une construction parallèle pour laquelle il devient le *processus de base*. Une construction parallèle spécifie un ensemble d'unités de travail exécutables en parallèle par plusieurs processus. Chaque section d'un `PARALLEL SECTIONS` ou chaque itération d'un `PARALLEL DO` représente

une *unité de travail* assignable à au plus un processus.

L'exécution d'une construction parallèle commence par la création d'une *équipe* de processus, dont le processus de base peut faire partie, et à chaque membre de laquelle une ou plusieurs unités de travail peuvent être affectées. Les processus de l'équipe qui ont terminé le travail assigné ou qui n'ont pas reçu d'unités de travail, attendent à la fin de la construction parallèle que l'ensemble des unités ait été exécuté. C'est l'ordre d'affectation des unités de travail à un processus qui détermine l'ordre de leur exécution.

Lorsque tous les membres de l'équipe ont terminé leur travail, l'équipe est dissoute et le processus de base reprend l'exécution séquentielle jusqu'à la rencontre d'une autre construction parallèle ou de la fin du programme. Dans le cas de constructions parallèles imbriquées, chaque processus devient le processus de base pour la construction parallèle rencontrée.

Nous décrivons informellement la sémantique de ces constructions.

2.2.4 Structures de contrôle

Parallel Do

```
PARALLEL DO I=1,N (ORDERED)
  ...
END PARALLEL DO
```

Un `PARALLEL DO` indique la possibilité de mettre en jeu plus d'un processus pour l'exécution du corps de la boucle : les différentes itérations de la boucle sont distribuées aux membres de l'équipe. Si l'option `ORDERED`[1] est utilisée, l'affectation des itérations doit se faire dans l'ordre défini par la variable de boucle ; sinon l'affectation peut se faire dans n'importe quel ordre (cf. exemples figure 2.1). Toute itération est affectée à exactement un processus ; il n'y a pas d'exécution de code redondante.

La variable de boucle d'un `PARALLEL DO` a le statut de variable privée.

	ordered	*non ordered*
P_1	1;4	1;4
P_2	2;3	3;2
P_1	1;2	2;1
P_2	3;4	3;4

FIG. 2.1 – *Deux exemples de distributions* ordered *et* non ordered *avec* `N` $= 4$ *et deux processus* P_1 *et* P_2.

1. L'utilisation de constructions parallèles «ordered» ne définit en rien l'ordre d'exécution des tâches (sections ou itérations) et ne remplace donc pas l'utilisation de synchronisations explicites.

Parallel Sections

```
PARALLEL SECTIONS (ORDERED)
  SECTION A
  ...
  SECTION B WAIT A
  ...
  SECTION C
  ...
END PARALLEL SECTIONS
```

Un `PARALLEL SECTIONS` indique la possibilité de mettre en jeu plus d'un processus pour l'exécution de blocs d'instructions disjoints délimités par le mot `SECTION`. Les sections d'un `PARALLEL SECTIONS` sont distribuées aux membres de l'équipe. Si l'option `ORDERED` est utilisée, l'affectation des sections aux processus de l'équipe doit se faire dans l'ordre lexical ; sinon l'affectation peut se faire dans n'importe quel ordre. Toute section est affectée à exactement un processus.

Un ordre partiel d'exécution des sections peut être précisé. C'est le cas dans l'exemple ci-dessus où la clause `WAIT A` interdit le commencement de la `SECTION B` avant la fin de la `SECTION A`. Afin de simplifier le traitement des `PARALLEL SECTIONS`, nous considérerons dans la suite que les `WAIT` de sections ont été remplacés par des couples `POST` / `WAIT` ordinaires sur des événements (Cf. 2.2.6). La transformation est immédiate, pour l'exemple précédent, on obtient :

```
EVENT SA
CLEAR(SA)
PARALLEL SECTIONS (ORDERED)
  SECTION A
  ...
  POST(SA)
  SECTION B
  WAIT(SA)
  ...
  SECTION C
  ...
END PARALLEL SECTIONS
```

On pourrait alors simplifier la syntaxe du langage :

```
section     →   sectionhead body
```

2.2.5 Environnements

L'environnement d'un processus est une fonction qui à un nom de scalaire ou de tableau associe l'adresse d'un objet, c'est à dire d'une zone mémoire.

L'environnement du processus initial est constitué de tous les objets déclarés en tête du programme.

Pour l'exécution d'une construction parallèle, l'environnement de données d'un processus rassemble les objets référencés à l'intérieur de la construction parallèle. Chaque objet de l'environnement possède un attribut dont la valeur est soit *privé* soit *partagé*. Il est possible qu'un objet ayant un attribut privé pour une construction parallèle ait un attribut partagé pour une construction parallèle imbriquée. La situation inverse est interdite : un objet partagé pour une construction parallèle ne peut être privé pour une construction imbriquée. Donc l'ensemble des processus pour lequel un objet a l'attribut partagé peut évoluer dynamiquement au cours de l'exécution du programme.

Si un objet a l'attribut privé alors chaque processus en possède une copie qui est inaccessible aux autres membres de l'équipe. Une lecture ou une écriture par un processus de l'équipe sur cet objet n'opère que sur sa propre copie. L'accès à la copie privée ne peut engendrer aucun conflit, la sémantique de ces opérations est donc identique à celle qui se trouve dans un langage séquentiel.

Si un objet a l'attribut partagé pour un ensemble E de processus alors tous les processus de E peuvent écrire ou lire sur cet objet. Cependant, comme aucune hypothèse d'atomicité sur ces opérations n'est faite, et comme ces opérations sont asynchrones, leur sémantique est différente du cas séquentiel. Ces opérations ne sont pas toujours définies car on ne peut pas supposer qu'un objet soit *accessible* à n'importe quel moment.

La lecture ou l'écriture par un processus sur un objet partagé est définie seulement si cet objet est accessible par ce processus.

La notion d'accessibilité est dérivée d'une notion plus globale de *consistance* : un objet partagé par un ensemble E de processus est accessible à un processus de E

- s'il a été modifié par ce processus
- ou bien s'il est consistant relativement à E

La consistance garantit à tous les processus qui opèrent en lecture sur un objet d'obtenir la même valeur.

Au commencement de chaque construction parallèle, tout objet partagé est consistant. Un objet partagé qui n'est jamais modifié reste toujours consistant. Un objet partagé qui est modifié cesse d'être consistant ; il ne peut le redevenir qu'après une opération de synchronisation. La différence entre consistance et accessibilité est qu'un objet modifié par un processus reste accessible par ce processus même s'il n'est pas consistant (ie. même si les autres processus de l'équipe n'ont pas été informés de cette modification).

À l'entrée d'une construction parallèle, la valeur d'un objet partagé est celle associée à cet objet dans l'environnement du processus (de base) qui a rencontré la construction parallèle. Pour un objet privé il y a deux cas :

- si le nom de cet objet existe dans l'environnement du processus de base alors sa valeur est celle associée à cet objet dans l'environnement du processus de base.

- sinon sa valeur n'est pas définie.

Une *synchronisation* joue un rôle de coordination entre les processus et permet d'assurer la consistance des objets partagés.

La figure 2.2 illustre la notion d'accessibilité à une variable partagée x par deux processus concurrents P_1 et P_2. Au départ, la valeur de x est supposée être consistante pour $\{P_1, P_2\}$. Le couple post / wait opérant sur le même événement, une synchronisation a lieu entre P_1 et P_2 ; elle permet au processus P_2 d'accéder à la valeur que P_1 a affectée à x.

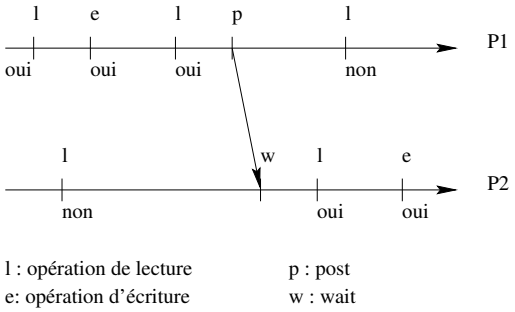

FIG. 2.2 – *Diagramme d'accessibilité d'une variable partagée par 2 processus concurrents*

2.2.6 Synchronisation

L'emploi d'une instruction de synchronisation dans une construction parallèle permet d'assurer la consistance des objets partagés par un groupe de processus.

Il y a deux types de synchronisation : *implicite* et *explicite*.

Une synchronisation concerne un groupe de processus. Pour une synchronisation implicite, le groupe concerné est l'équipe toute entière ou l'équipe plus le processus de base si celui-ci ne fait pas partie de l'équipe. Pour une synchronisation explicite, les processus concernés sont les processus partageant la variable de synchronisation.

Quand un processus P exécute une opération de synchronisation, les actions suivantes sont réalisées :

- les objets, résultat des modifications par P d'objets partagés, sont rendus consistants pour l'ensemble de processus concernés par l'opération de synchronisation.
- réciproquement, le résultat des modifications par les autres processus d'objets partagés est rendu accessible à P.

La seconde fonction des opérations de synchronisation est de signaler aux autres processus qu'ils peuvent accéder à l'objet modifié ou inversement de les en empêcher. Par exemple un couple post / wait établit un lien de causalité : le post précède le wait, une section critique garantit l'unicité de l'accès à un objet. Dans un programme correct, un processus qui n'a reçu aucun signal des autres processus est assuré qu'un objet n'a pas été modifié.

Synchronisation explicite

X3H5 propose plusieurs mécanismes de synchronisation explicite : les verrous, les sections critiques, les événements et les ordinaux. Nous avons écarté les verrous et les sections critiques qui peuvent être utilisés pour écrire des programmes non déterministes desquels il est difficile de dériver une version séquentielle au comportement équivalent.

Un événement est une variable de type `EVENT` dont les valeurs sont *clear* et *posted*. Il reçoit une valeur au moyen des instructions atomiques `CLEAR` et `POST`. L'instruction `WAIT` teste la valeur de son argument. Si elle vaut *posted*, l'exécution continue ; sinon le test est répété ultérieurement.

Un ordinal est un objet de type `ORDINAL` implémentant un sémaphore à compteur. Un ordinal prend des valeurs entières, est initialisé par l'instruction `SET`, incrémenté par `POST` et testé par `WAIT` :

- `SET`(ω, e_1, e_2) positionne la valeur initiale de l'ordinal ω à la valeur de l'expression e_1 et son incrément à la valeur de l'expression e_2.
- `POST`(ω, e) compare la valeur o de l'ordinal ω à la valeur n de l'expression e moins la valeur i de l'incrément :
  - si $o < n - i$ et $i > 0$, ou $o > n - i$ et $i < 0$, ce test est répété ultérieurement ;
  - sinon la valeur n est affectée à l'ordinal ω et le contrôle passe à l'instruction suivante.
- `WAIT`(ω, e) compare la valeur de l'ordinal ω à la valeur n de l'expression e :
  - si $o < n$ et $i > 0$, ou $o > n$ et $i < 0$, ce test est répété ultérieurement ;

– sinon, le contrôle passe à l'instruction suivante.

Les ordinaux servent à synchroniser les itérations d'un nid de boucles selon une progression arithmétique.

La valeur d'une variable d'événement ou d'ordinal est modifiée de façon atomique.

Un ordinal peut être remplacé par un tableau d'événements au prix d'une surconsommation de mémoire. L'inverse est faux à cause de l'attente implicite réalisée par le `POST` ordinal comme on peut le constater sur l'exemple ci-dessous. On a, à gauche, un fragment de programme qui utilise un tableau d'événements et à droite, sa transcription qui utilise un ordinal à la place.

```
EVENT E(MAXN)
initialisation de E
PARALLEL DO (ORDERED) I=4,N
  C(I) = FUNC(B(I))
  WAIT E(I-3)
  B(I) = B(I) + B(I-3)*C(I)
  POST E(I)
END PARALLEL DO
```

```
ORDINAL E
SET (E,3,1)
PARALLEL DO (ORDERED) I=4,N
  C(I) = FUNC(B(I))
  WAIT (E,I-3)
  B(I) = B(I) + B(I-3)*C(I)
  POST (E,I)
END PARALLEL DO
```

Ces deux fragments de programme ont le même effet sur la mémoire. Mais dans le programme de droite, l'instruction `POST`, pour chaque itération, doit attendre que l'instruction `POST` de l'itération précédente ait terminé alors que pour le programme de gauche jusqu'à trois itérations peuvent s'exécuter de manière totalement indépendante. Le remplacement d'un tableau d'événements par un ordinal induit une contrainte sur l'ordre d'exécution qui n'est pas toujours nécessaire. En revanche le programme de droite consomme moins de ressource mémoire.

Synchronisation implicite

Une synchronisation implicite est déclenchée au début et à la fin d'une construction parallèle :

– au début, afin que chaque processus de l'équipe nouvellement formée ait accès à la valeur courante de tout objet partagé.

– à la fin, pour que le processus de base ait accès à l'ensemble des modifications faites par les membres de l'équipe sur les objets partagés, comme s'il avait exécuté seul la construction parallèle.

2.3 Problèmes liés aux synchronisations

2.3.1 Instructions d'attente

Les instructions d'attente ont pour effet d'interrompre le flot de contrôle tant qu'une condition n'est pas remplie. Les instructions qui engendrent une attente explicite sont le `WAIT` sur les événements et les ordinaux, le `POST` sur les ordinaux et les `WAIT` de sections. Les instructions qui engendrent une attente implicite sont les constructions parallèles `PARALLEL DO` et `PARALLEL SECTIONS` à la fin desquelles le processus de base attend que les membres de l'équipe aient terminé l'exécution des unités qui leur ont été attribuées.

Soit p un programme, nous montrons que si p ne contient pas d'instructions d'attente, alors il termine.

Nous prouvons cette propriété par induction sur la structure syntaxique du programme :

- une expression s'évalue en un temps fini.
- une instruction simple termine (par hypothèse).
- une liste d'instruction est soit vide soit de la forme $s :: s^*$. Si elle est vide alors elle termine ; sinon elle termine si s termine et s^* termine.
- une instruction conditionnelle est de la forme `if` b s_1^* s_2^*. Elle termine si l'évaluation de b termine, s_1^* termine et s_2^* termine.
- une boucle `do` est de la forme `do` i lb ub s^*. Elle termine si l'évaluation de lb et ub termine, si la boucle engendre un nombre fini d'instances $s^*(i)$ et si toute instance $s^*(i)$ termine.

La finitude du nombre d'instances engendrées par une boucle est garantie par le fait que l'indice de boucle n'est pas modifiable à l'intérieur de la boucle.

En fait, nous aurions pu garder les constructions parallèles mais cela nous aurait obliger à considérer les processus qu'elles créent.

La non-terminaison d'un programme ne peut donc résulter que d'un blocage dû à une instruction d'attente.

À l'intérieur d'une construction parallèle, un événement, simple ou ordinal, attendu par une instruction d'attente (`WAIT` sur des événements ou des ordinaux, `POST` sur des ordinaux) mais jamais posté interdit au processus qui exécute l'instruction d'attente d'achever l'exécution des tâches qui lui ont été assignées empêchant le processus de base de terminer l'exécution de la construction parallèle. Ce raisonnement s'applique récursivement à la construction parallèle englobante jusqu'à atteindre la séquence principale du programme. L'attente indéfinie d'un événement provoque le blocage du programme.

2.3.2 Blocage et nombre de processus

Le standard X3H5 ne prévoit pas la possibilité de spécifier un nombre minimum de processus pour l'exécution d'une construction parallèle. Pour X3H5, l'exécution d'un programme correct doit terminer quel que soit le nombre de processus disponible, en particulier avec un seul. Nous avons retenu cette exigence dans le cadre de cette étude.

Dans l'exemple ci-dessous, si un seul processus est alloué pour l'exécution du `PARALLEL SECTIONS`, le programme bloquera indéfiniment sur le `WAIT`.

```
CLEAR E
PARALLEL SECTIONS (ORDERED)
  SECTION A
  WAIT E
  ...
  SECTION B
  POST E
  ...
END PARALLEL SECTIONS
```

Ici, il se produit un inter-blocage quelque soit le nombre de processus :

```
CLEAR E
CLEAR F
PARALLEL SECTIONS (ORDERED)
  SECTION A
  WAIT E
  ...
  POST F
  SECTION B
  WAIT F
  ...
  POST E
END PARALLEL SECTIONS
```

Bien qu'avec un seul processus disponible, le programme soit exécuté séquentiellement, il n'y a pas en général une exécution mono-processus mais plusieurs, à cause de l'ordre d'affectation des unités de travail aux processus, qui est susceptible de varier d'une exécution à une autre du programme.

Dans la suite, nous imposerons l'option `ORDERED` aux constructions parallèles pour qu'il y ait une (unique) exécution mono-processus au cours de laquelle les unités de travail sont affectées dans un ordre compatible avec l'ordre textuel des sections ou l'ordre d'évolution des indices de boucles.

Chapitre 3

Dépendances de données

Dans un langage séquentiel, une dépendance de données résulte de deux accès successifs à un même emplacement mémoire, dont au moins un en écriture, par deux instances d'instruction différentes lors de l'exécution d'un programme.

Dans ce chapitre nous formalisons cette définition et explicitons les restrictions nécessaires à sa formulation logique. Mais auparavant nous donnons une définition précise aux éléments syntaxiques permettant d'y aboutir.

3.1 Définitions syntaxiques

3.1.1 Nid de boucles et espace d'itération

Définition 1 *Pour une instruction a, on appelle* nid de boucles *de a la liste éventuellement vide des instructions* `DO` *ou* `PARALLEL DO` *ordonnées de l'extérieur vers l'intérieur, ancêtres de a dans l'arbre syntaxique.*

Le nid de boucles de a est noté $\mathfrak{N}(a)$, ses éléments sont représentés par des tuples à quatre éléments (C, i, l, u), où C est l'un des symboles `do` ou `pdo`, i un nom d'indice, et l et u les expressions des bornes inférieures et supérieures respectivement.

Le *vecteur d'itération* $\mathfrak{I}(a)$ est la liste des noms d'indices, supposés tous différents, apparaissant dans $\mathfrak{N}(a)$.

Si $\mathfrak{N}$ est un nid de boucles, sa longueur est notée $|\,\mathfrak{N}\,|$; la profondeur d'imbrication d'une instruction a est $|\,\mathfrak{N}(a)|$, le nombre de boucles (séquentielles ou parallèles) entourant a.

Dans cet exemple,

```
   DO I=1,N
     PARALLEL DO J=1,I
       DO K = I-J,I
a:       A(I,J,K) = 0
```

```
      END DO
    END PARALLEL DO
  END DO
```

la profondeur d'imbrication de l'instruction a est 3, son vecteur d'itération est $\mathfrak{I}(a) = [\mathtt{I},\mathtt{J},\mathtt{K}]$ et son nid de boucles, $\mathfrak{N}(a)$, est la liste [`(do,I,1,N), (pdo,J,1,I), (do,K,I-J,I)`].

Nous définissons également le nid de boucles d'une expression e. $\mathfrak{N}(e)$ est égal au nid de boucles de l'instruction parente de e la plus proche dans l'arbre de syntaxe abstraite du programme.

Nous devons considérer qu'une expression `iexp` dans un programme (voir le chapitre 2) est liée par son nid de boucles. Quand nous créerons une formule logique à partir d'une telle expression, nous substituerons des variables logiques aux noms d'indices. Cette substitution sera rendue explicite par la notation $[\boldsymbol{x}/\boldsymbol{i}]$ dans laquelle $\boldsymbol{x}$ représente un vecteur de variables et $\boldsymbol{i}$ un vecteur d'entiers ou encore la notation $[\boldsymbol{x}/\,\mathfrak{I}(a)]$ s'il s'agit d'une expression apparaissant dans une instruction a.

Définition 2 *Pour toute paire d'instruction a, b, on appelle* nid de boucles commun, *noté $\mathfrak{N}(a,b)$, le préfixe commun des nids de boucles des instructions a et b et,* vecteur d'itération commun, *noté $\mathfrak{I}(a,b)$, la liste des noms d'indices apparaissant dans $\mathfrak{N}(a,b)$.*

Définition 3 *On appelle* espace d'itération *d'une instruction a, l'ensemble dans lequel varie le vecteur d'itération d'une instruction a, pour un ensemble de paramètres $\boldsymbol{p}$ fixé.*

Plutôt qu'un ensemble, on préfère manipuler une formule logique $\mathrm{Iter}_a(\boldsymbol{p},\boldsymbol{x}) = is(\mathfrak{N}(a),\boldsymbol{x})$ qui dépend des paramètres $\boldsymbol{p}$ présents dans les bornes des boucles et d'une liste de variables $\boldsymbol{x}$ associée au vecteur d'itération $\mathfrak{I}(a)$. C'est possible grâce aux restrictions syntaxiques énoncées en 2.1.2.

Définition 4 *La formule décrivant l'espace d'itération d'une instruction est calculée par :*

$$\begin{aligned} is([\,],[\,]) &= \mathit{true} \\ is((\_,i,l,u)::\boldsymbol{n'},x::\boldsymbol{x'}) &= (l([\boldsymbol{x}/\boldsymbol{i}]) \leqslant x \leqslant u([\boldsymbol{x}/\boldsymbol{i}])) \wedge is(\boldsymbol{n'},\boldsymbol{x'}) \end{aligned}$$

Cette formule ne dépend pas du type de boucle (`do` ou `pdo`).

Quand les paramètres du programme reçoivent des valeurs entières, l'ensemble $\{\boldsymbol{n};\ \mathrm{Iter}_a(\boldsymbol{p},\boldsymbol{n})\}$ détermine l'espace d'itération de a, qui est un sous-ensemble de $\mathbb{Z}^d$, où d est la longueur de $\mathfrak{N}(a)$, pour une instance donnée du programme.

Définition 5 *Pour toute paire d'instruction a, b, l'*espace d'itération commun *est l'ensemble $\{\boldsymbol{n};\ \mathrm{CIter}_{a,b}(\boldsymbol{p},\boldsymbol{n})\}$ où $\mathrm{CIter}_{a,b}$ est défini par la formule :* $\mathrm{CIter}_{a,b}(\boldsymbol{p},\boldsymbol{x}) = is(\mathfrak{N}(a,b),\boldsymbol{x})$.

3.1.2 Instruction et instance d'instruction

Une instruction du langage appartient à la catégorie syntaxique `stmt`. Ce peut donc être une instruction simple (affectation, instruction de synchronisation) ou une instruction structurée (boucle séquentielle ou parallèle, conditionnelle, section parallèle).

```
stmt  →  assign | if | do | pdo | psections | epost | ewait | clear
       | opost | owait | set | continue
```

Nous emploierons le terme d'instruction pour l'expression booléenne d'un `if`, l'en-tête d'un `psections`, d'un `do` ou d'un `pdo` bien que ces éléments ne fasse pas partie de la catégorie `stmt`. Ceci revient à assimiler l'exécution d'un `if` à l'évaluation de sa condition booléenne, l'exécution d'un `do` ou d'un `pdo` à l'évaluation de ses bornes et l'exécution d'un `psections` à l'exécution d'une instruction sans effet situé à l'emplacement de son en-tête.

La présence des boucles `DO` et `PARALLEL DO` nous conduit à introduire la notion d'*instance d'instruction*. À toute instruction a nous associons un ensemble d'instances de sorte que chaque instance soit exécutée au plus une fois.

Si $\mathfrak{N}(a)$ est non vide, une instance d'instruction a est obtenue en donnant une valeur $\boldsymbol{i}_0$ au vecteur d'itération de a. Notons que si $\mathfrak{N}(a)$ est vide, a n'admet qu'une seule instance.

On notera une instance d'instruction a par un couple : $\langle a, \boldsymbol{i} \rangle$ où $\boldsymbol{i} \in \mathbb{Z}^d$ avec $d = |\,\mathfrak{N}(a)|$.

En conséquence une instruction contenue dans un nid de boucles engendre une infinité dénombrable d'instances dont au plus un nombre fini, déterminé par la formule d'itération et les conditions booléennes englobantes, sera effectivement exécuté lors d'une exécution d'une instance du programme.

3.1.3 Ordre séquentiel

Sur la version séquentielle (Cf. 4.4.1) du programme, l'ordre d'exécution de deux instances instructions $\langle a, \boldsymbol{i}_a \rangle$ et $\langle b, \boldsymbol{i}_b \rangle$ est exprimable statiquement par une formule qui dépend de la position relative des instructions dans le texte du programme (l'ordre lexical) et de l'ordre des itérations $\boldsymbol{i}_a$ et $\boldsymbol{i}_b$ sur le nid de boucles commun $\mathfrak{N}(a, b)$.

Définition 6 ***Ordre lexical***
On appelle ordre lexical*, la relation $a\,T\,b$ définie sur les instructions, qui dit que a est avant b dans le texte du programme.*

Précisons que $a\,T\,a = \mathit{false}$.

Définition 7 ***Ordre d'exécution séquentielle dans un nid de boucles***
Dans un nid de boucles $\mathfrak{N}$, la relation de précédence séquentielle entre deux

itérations est donnée par la formule $\ll$ définie ci-dessous.

$$\begin{aligned} [\,] \ll [\,] &= \mathit{false} \\ i :: \boldsymbol{i'} \ll j :: \boldsymbol{j'} &= (i < j) \vee ((i = j) \wedge (\boldsymbol{i'} \ll \boldsymbol{j'})) \end{aligned}$$

On reconnaît l'*ordre lexicographique* qui compare deux chaînes caractère par caractère en commençant par la gauche. Ici, les chaînes sont remplacées par les vecteurs d'indices et la comparaison commence par l'indice de la boucle la plus externe.

Définition 8 ***Ordre séquentiel de deux instances***
Soit $\boldsymbol{k}_a$ et $\boldsymbol{k}_b$ les préfixes de longueur $|\mathfrak{N}(a,b)|$ de $\boldsymbol{i}_a$ et $\boldsymbol{i}_b$ respectivement. L'ordre séquentiel, noté $\prec$, de deux instances $\langle a, \boldsymbol{i}_a\rangle$ et $\langle b, \boldsymbol{i}_b\rangle$ est définie par

$$\langle a, \boldsymbol{i}_a\rangle \prec \langle b, \boldsymbol{i}_b\rangle = ((\boldsymbol{k}_a = \boldsymbol{k}_b) \wedge (a\,T\,b)) \vee (\boldsymbol{k}_a \ll \boldsymbol{k}_b)$$

Définition 9 ***Formule de précédence séquentielle***
La formule de précédence séquentielle, notée $(a \prec b)(\boldsymbol{x}, \boldsymbol{y})$, entre deux instances $\langle a, \boldsymbol{i}_a\rangle$ et $\langle b, \boldsymbol{i}_b\rangle$ est obtenue à partir de la relation précédente en associant la liste de variables $\boldsymbol{x}$ à $\boldsymbol{i}_a$ et la liste $\boldsymbol{y}$ à $\boldsymbol{i}_b$:

$$(a \prec b)(\boldsymbol{x}, \boldsymbol{y}) = (\langle a, \boldsymbol{i}_a\rangle \prec \langle b, \boldsymbol{i}_b\rangle)[\boldsymbol{x}/\boldsymbol{i}_a, \boldsymbol{y}/\boldsymbol{i}_b]$$

3.1.4 Ensembles $\mathfrak{In}$ et $\mathfrak{Out}$ d'une instruction

Pour calculer la relation de dépendance entre deux instructions, il est utile de connaître quelles sont les références mémoire utilisées en entrée (ensemble $\mathfrak{In}$) et en sortie (ensemble $\mathfrak{Out}$) par les instructions. Pour cela, on définit l'ensemble $\mathfrak{Ref}$ des références scalaire ou tableau présentes dans une expression. Le mot « référence » a ici un sens purement syntaxique.

$$\begin{aligned} \mathfrak{Ref}(\mathit{constant}) &= \varnothing \\ \mathfrak{Ref}(E1\ \mathit{op}\ E2) &= \mathfrak{Ref}(E1) \cup \mathfrak{Ref}(E2) \\ \mathfrak{Ref}(\mathit{op}\ E1) &= \mathfrak{Ref}(E1) \\ \mathfrak{Ref}(\mathit{name}) &= \{\mathit{name}\} \\ \mathfrak{Ref}(\mathit{name}(E_1, \ldots E_n)) &= \{\mathit{name}(E_1, \ldots E_n)\} \cup \mathfrak{Ref}(E_1) \cup \ldots \cup \mathfrak{Ref}(E_n) \end{aligned}$$

Dans la suite du texte, on parlera de dimension d'une référence pour signifier la longueur de sa liste d'expressions.

Les définitions suivantes donnent, pour chaque type d'instruction simple, la définition des ensembles $\mathfrak{In}$ et $\mathfrak{Out}$. On y considère une référence scalaire comme une référence tableau avec une liste d'expressions vide.

– a est une affectation

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{Ref}(a.\texttt{rhs}) \cup \mathfrak{Ref}(a.\texttt{lhs.iexp}^*) \\ \mathfrak{Out}(a) &= \{a.\texttt{lhs}\} \end{aligned}$$

– a est un `POST`

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{Ref}(a.\texttt{lhs.iexp}^*) \\ \mathfrak{Out}(a) &= \{a.\texttt{lhs}\} \end{aligned}$$

– a est un `WAIT`

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{Ref}(a.\texttt{lhs}) \\ \mathfrak{Out}(a) &= \varnothing \end{aligned}$$

– a est un `CLEAR`

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{Ref}(a.\texttt{lhs.iexp}^*) \\ \mathfrak{Out}(a) &= \{a.\texttt{lhs}\} \end{aligned}$$

– a est un `POST` ordinal

$$\begin{aligned} \mathfrak{In}(a) &= \{a.\texttt{lhs}\} \cup \mathfrak{Ref}(a.\texttt{iexp}) \cup \mathfrak{Ref}(a.\texttt{lhs.iexp}^*) \\ \mathfrak{Out}(a) &= \{a.\texttt{lhs}\} \end{aligned}$$

– a est un `WAIT` ordinal

$$\begin{aligned} \mathfrak{In}(a) &= \{a.\texttt{lhs}\} \cup \mathfrak{Ref}(a.\texttt{iexp}) \cup \mathfrak{Ref}(a.\texttt{lhs.iexp}^*) \\ \mathfrak{Out}(a) &= \varnothing \end{aligned}$$

– a est un `SET`

$$\begin{aligned} \mathfrak{In}(a) &= \mathfrak{Ref}(a.\texttt{iexp}) \cup \mathfrak{Ref}(a.\texttt{lhs.iexp}^*) \\ \mathfrak{Out}(a) &= \{a.\texttt{lhs}\} \end{aligned}$$

3.1.5 Ensembles InNames et OutName d'une instruction

Des ensembles $\mathfrak{In}$ et $\mathfrak{Out}$ sont dérivés respectivement les ensembles d'identificateurs $\mathfrak{InNames}$ et $\mathfrak{OutName}$ de la façon suivante : une référence tableau est dépouillée de sa liste d'expressions tandis qu'une référence scalaire reste inchangée. Cette correspondance nommée "strip" est telle que :

$$\begin{aligned} \text{strip}(name) &= name \\ \text{strip}(name(E_1, \ldots E_n)) &= name \end{aligned}$$

Elle est appliquée à chaque élément de $\mathfrak{In}$ (resp. $\mathfrak{Out}$) pour donner $\mathfrak{InNames}$ (resp. $\mathfrak{OutName}$) :

$$\begin{aligned} \mathfrak{InNames}(a) &= \{n; \exists r \in \mathfrak{In}(a) \wedge \text{strip}(r) = n\} \\ \mathfrak{OutName}(a) &= \{n; \exists r \in \mathfrak{Out}(a) \wedge \text{strip}(r) = n\} \end{aligned}$$

3.2 Dépendances de données

3.2.1 Couple potentiellement conflictuel

Soit un couple $(a, b) \in (\texttt{assign}\times\texttt{assign})\cup(\texttt{assign}\times\texttt{if.bexp})\cup(\texttt{if.bexp}\times\texttt{assign})$. Un couple de références mémoire r_1,r_2 issue de ce couple a, b est potentiellement conflictuel si r_1 et r_2 peuvent désigner le même emplacement mémoire et qu'au moins l'un d'entre eux est modifié.

Trois cas peuvent se présenter :

- si $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(b) \neq \varnothing$, soit n un nom de variable tel que $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(b) = \mathfrak{OutName}(a) = \{n\}$, et r_o tel que $\mathfrak{Out}(a) = \{r_o\}$, et $\text{strip}(r_o) = n$; pour toute référence $r \in \mathfrak{In}(b)$ telle que $\text{strip}(r) = n$, on a le couple en conflit « écriture-lecture » (wr ou out-in) (r_o, r)

- si $\mathfrak{OutName}(b) \cap \mathfrak{InNames}(a) \neq \varnothing$, soit n un nom de variable tel que $\mathfrak{OutName}(b) \cap \mathfrak{InNames}(a) = \mathfrak{OutName}(b) = \{n\}$, et r_o tel que $\mathfrak{Out}(b) = \{r_o\}$, et $\text{strip}(r_o) = n$; pour toute référence $r \in \mathfrak{In}(a)$ telle que $\text{strip}(r) = n$, on a le couple en conflit « lecture-écriture » (rw ou in-out) (r, r_o)

- si $\mathfrak{OutName}(a) \cap \mathfrak{OutName}(b) \neq \varnothing$, soit n un nom de variable tel que $\mathfrak{OutName}(a) \cap \mathfrak{OutName}(b) = \mathfrak{OutName}(a) = \mathfrak{OutName}(b) = \{n\}$, et r_o, s_o les références telles que $\mathfrak{Out}(a) = \{r_o\}$, $\mathfrak{Out}(b) = \{s_o\}$, et $\text{strip}(r_0) = \text{strip}(s_0) = n$; on a le couple en conflit « écriture-écriture » (ww ou out-out) (r_o, s_o).

3.2.2 Formule de dépendance

A chaque couple de références potentiellement conflictuelles $u = (r_1, r_2)$ est associée une formule de dépendance notée $\text{Dep}_u : a \to b$ où a est appelée la *source* et b la *cible* de la dépendance, la flèche rappelant le sens de la dépendance. Soit d la dimension commune des références r_1 et r_2 et soient $r_1.\texttt{iexp}^* = [e_1, \ldots, e_d]$, et $r_2.\texttt{iexp}^* = [f_1, \ldots, f_d]$ ces listes.

$\text{Dep}_u : a \to b$ est la conjonction des équations et inéquations du système suivant :

$$\begin{cases} [\boldsymbol{x}/\mathfrak{I}(a)]e_i = [\boldsymbol{y}/\mathfrak{I}(b)]f_i \quad i = 1, \ldots, d \\ (a \prec b)(\boldsymbol{x}, \boldsymbol{y}) \\ \text{Iter}_a(\boldsymbol{p}, \boldsymbol{x}) \\ \text{Iter}_b(\boldsymbol{p}, \boldsymbol{y}) \end{cases}$$

où $\boldsymbol{x}$ (resp. $\boldsymbol{y}$) est un tuple de variables entières de longueur $|\mathfrak{N}(a)|$ (resp. $|\mathfrak{N}(b)|$),$\prec$ est l'ordre séquentiel défini en 3.1.3 et Iter_a (resp. Iter_b) la formule donnant l'espace d'itération de a (resp. b) défini en 3.1.1.

Nous disons qu'il y a une dépendance entre a et b, à cause du conflit u, si la formule de dépendance $\mathrm{Dep}_u : a \to b$ est satisfaisable, c'est à dire, possède une solution en $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{p}$. Selon que le conflit est *wr*, *rw* ou *ww*, nous avons respectivement une *dépendance de flot*, une *anti-dépendance* ou une *dépendance de sortie*.

Prenons l'exemple suivant :

```
      DO I = 1, 10
a:     A(I) = ...
b:     ... = A(3*I-5)
      END DO
```

Il y a deux couples de références potentiellement conflictuelles, un $wr : u = (A(I), A(3{*}I{-}5))$ et un $rw : v = (A(3{*}I{-}5), A(I))$. Dep_u est $x = 3y-5 \wedge x \leqslant y \wedge 1 \leqslant x \leqslant 10 \wedge 1 \leqslant y \leqslant 10$, Dep_v est $x = 3y - 5 \wedge y < x \wedge 1 \leqslant x \leqslant 10 \wedge 1 \leqslant y \leqslant 10$. On vérifie que $(1, 2)$ est solution de Dep_u, $(4, 3)$ est solution de Dep_v donc il y a effectivement une dépendance de flot de a vers b et une anti-dépendance de b vers a.

Remarquons que si le conflit porte sur des références scalaires, alors l'équation de dépendance disparaît, et l'ensemble des solutions est le graphe de la relation $\prec$ dans l'espace produit $\{\boldsymbol{n};\ \mathrm{Iter}_a(\boldsymbol{p}, \boldsymbol{n})\} \times \{\boldsymbol{n};\ \mathrm{Iter}_b(\boldsymbol{p}, \boldsymbol{n})\}$.

3.2.3 Caractéristiques d'une dépendance

Soit u un couple conflictuel impliquant les instructions a et b et soit $(\boldsymbol{x}, \boldsymbol{y})$ une solution vérifiant la formule de dépendance $\mathrm{Dep}_u : a \to b$.

La différence $\Delta = \boldsymbol{y}_{|\,\mathfrak{N}(a,b)} - \boldsymbol{x}_{|\,\mathfrak{N}(a,b)}$ est appelée *vecteur de dépendance*. Soit $\mathrm{Vec}_u(\boldsymbol{p})$ l'ensemble des vecteurs de dépendance pour une même valeur de $\boldsymbol{p}$.

La dépendance est *uniforme* si pour tout $\boldsymbol{p}$, $\mathrm{Vec}_u(\boldsymbol{p})$ est réduit à un seul élément.

Si pour tout $\boldsymbol{p}$, $\mathrm{Vec}_u(\boldsymbol{p}) = \{\mathbf{0}\}$, la dépendance est alors indépendante de l'itération.

Étant donnée une solution, si k est le rang du premier élément non nul de Δ alors cette solution est dite de rang k et la plus petite valeur de k quand Δ parcourt $\mathrm{Vec}_u(\boldsymbol{p})$ est appelée *profondeur* de la dépendance.

Pour une solution de rang k, on appelle *support* de la solution la boucle à la profondeur k : il est séquentiel si c'est une boucle `DO`, parallèle si c'est un `PARALLEL DO`.

Dans cet exemple,

```
      PARALLEL DO I = 1,N
        DO J = 1,M
  a:      A(I,J) = ...
  b:      ... = A(I-1,J-1)
```

```
      END DO
    END PARALLEL DO
```

la dépendance $a \rightarrow b$ possède une solution de rang 1 dont le support est parallèle (boucle `PARALLEL DO`) et une solution de rang 2 dont le support est séquentiel (boucle `DO`). La profondeur de la dépendance est 1.

Dans l'exemple ci-dessous,

```
    PARALLEL DO I = 1,N
      DO J = 1,M
a:      A(I,J) = ...
b:      ... = A(I,J-I)
      END DO
    END PARALLEL DO
```

contrairement à l'exemple précédent, la dépendance $a \rightarrow b$ ne possède plus de solution de rang 1 mais seulement des solutions de rang 2. Si $((x_1, x_2), (y_1, y_2))$ est une solution de rang 2 alors le vecteur de dépendance est

$$\Delta = \begin{pmatrix} 0 \\ y_1 \end{pmatrix}$$

Le vecteur de dépendance dépend de l'itération et n'est donc pas réduit à un seul élément, dans ce cas la dépendance est non uniforme.

Le support d'une dépendance indépendante de l'itération est l'en-tête de contrôle commun (Cf. chapitre 4) du couple d'instructions ; il est séquentiel si l'en-tête de contrôle commun est ';' et parallèle s'il est '||'.

3.2.4 Graphe de dépendance

Pour un programme donné, nous rassemblons l'ensemble des informations relatives aux dépendances, dans un graphe $\mathcal{DG} = (\mathcal{DS}, \mathcal{DA})$.

Définition 10 *L'ensemble des sommets $\mathcal{DS}$ contient les instructions d'affectation et les expressions booléennes des* `IF` *présentes à l'intérieur des constructions parallèles.* [1]

L'ensemble des arcs $\mathcal{DA}$ est l'ensemble des couples potentiellement conflictuels c'est-à-dire ceux pour lesquels une dépendance peut exister.

C'est seulement l'étude des systèmes Dep d'équations et d'inéquations en variables entières qui permettra de conclure si oui ou non les dépendances existent vraiment. Chaque arc est accompagné de la nature du conflit :

– *ww* pour un conflit de type écriture-écriture,

1. C'est, en effet, inutile de s'intéresser aux parties séquentielles du programme qui sont naturellement correctes du point de vue de la sémantique que nous avons choisie : celle du programme séquentiel équivalent que nous définissons au chapitre suivant.

– wr pour un conflit de type écriture-lecture,

– rw pour un conflit de type lecture-écriture.

Dans le programme ci-dessous, l'instruction a n'est pas incluse dans une construction parallèle donc

$$\begin{aligned}\mathcal{DS} &= \{b, c\} \text{ et}\\ \mathcal{DA} &= \{(b,b,ww),(b,b,wr),(b,b,rw),(b,c,ww),(b,c,wr),(b,c,rw),\\ &\quad (c,c,ww),(c,c,wr),(c,c,rw)\}\end{aligned}$$

Le graphe est représenté fig. 3.1.

```
        DO I=1,M
a:        A(I)=I
          POST(E(I))
        ENDDO
        PARALLEL SECTIONS (ORDERED)
          SECTION
            PARALLEL DO (ORDERED) I=M+1,3*M
              WAIT(E(I-M))
b:            A(I)=I*A(I-M)
              POST(E(I))
            END PARALLEL DO
          SECTION
            PARALLEL DO J=1,M+1
              WAIT(E(3*M+1-J))
            END PARALLEL DO
            PARALLEL DO I=3*M+1,3*M+N
c:            A(I)=I*A(I-M)
            END PARALLEL DO
        END PARALLEL SECTIONS
```

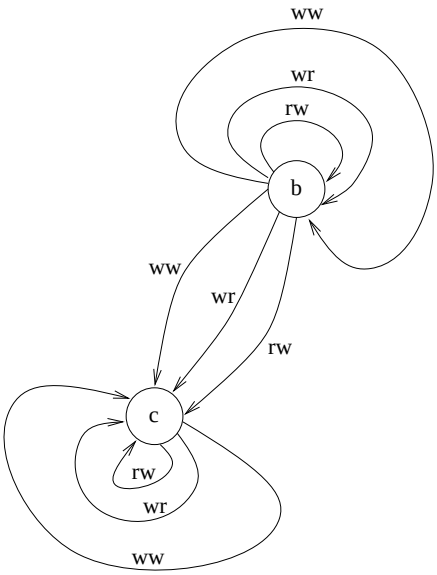

FIG. 3.1 – *Graphe de dépendance*

Chapitre 4

Précédences

4.1 Précédence de contrôle

La relation de précédence de contrôle exprime la précédence entre deux instructions a et b telle qu'elle résulte de la structure du contrôle du programme, sans tenir compte de l'effet des instructions de synchronisation explicite.

Les restrictions que nous avons faites sur la syntaxe nous permettent d'exprimer statiquement cette relation par une formule notée Pre^0. Elle ne dépend que de variables associées aux variables d'indice de boucle.

4.1.1 En-tête de contrôle commun à deux instructions

Pour toute paire d'instructions a, b, on définit l'en-tête de contrôle commun $\mathfrak{C}(a,b)$ qui peut valoir l'un des trois symboles ';', '||' ou '?'. Il exprime une relation de contrôle séquentielle, parallèle ou alternative entre a et b qui dépend de l'ancêtre commun le plus proche de a et de b dans l'arbre de syntaxe abstraite du programme.

$\mathfrak{C}(a,b)$ est défini par les clauses ci-dessous dans lesquelles c désigne le père de a si $a = b$, le plus petit ancêtre commun à a et à b sinon.

- Si $op(c) = \texttt{stmt}^*$ alors $\mathfrak{C}(a,b) = ;$
- Si $op(c) = \texttt{if}$ alors
  - si $a = c$ ou $b = c$ alors $\mathfrak{C}(a,b) = ;$
  - sinon $\mathfrak{C}(a,b) = ?$
- Si $op(c) = \texttt{section}^*$, alors a et b appartiennent à deux sections distinctes que l'on appelle s_a et s_b. Soit $h_a = s_a.\texttt{sectionhead}$ (respectivement $h_b = s_b.\texttt{sectionhead}$) l'en-tête de la section s_a (respectivement

s_b) alors

- si h_a.`name` $\in$ h_b.`sectionwait` ou h_b.`name` $\in$ h_a.`sectionwait` alors $\mathfrak{C}(a,b) = ;$
- sinon $\mathfrak{C}(a,b) = ||$

$\mathfrak{C}$ est commutatif par construction : $\mathfrak{C}(a,b) = \mathfrak{C}(b,a)$. Voir figure 4.1 pour des exemples de calcul de $\mathfrak{C}$.

4.1.2 Formule de précédence de contrôle

Pour chaque paire d'instruction a, b, on définit le prédicat $\mathrm{Pre}^0_{a,b}$ de précédence de contrôle déterminé par le nid de boucles et l'en-tête communs à a et à b.

Ordre d'exécution dans un nid de boucles

Soit $\boldsymbol{n}$ un nid de boucles. La relation de précédence entre deux itérations $\boldsymbol{i}$ et $\boldsymbol{j}$ de $\boldsymbol{n}$ dépend de la nature des boucles (parallèle ou séquentielle) du nid ; elle est donnée par la relation $\ll_{\boldsymbol{n}}$ définie ci-dessous.

$$
\begin{aligned}
[\,] \ll_{[\,]} [\,] &= \mathit{false} \\
i :: \boldsymbol{i'} \ll_{(\mathtt{pdo},\_,\_,\_)::\boldsymbol{n'}} j :: \boldsymbol{j'} &= (i = j) \wedge (\boldsymbol{i'} \ll_{\boldsymbol{n'}} \boldsymbol{j'}) \\
i :: \boldsymbol{i'} \ll_{(\mathtt{do},\_,\_,\_)::\boldsymbol{n'}} j :: \boldsymbol{j'} &= (i < j) \vee ((i = j) \wedge (\boldsymbol{i'} \ll_{\boldsymbol{n'}} \boldsymbol{j'}))
\end{aligned}
$$

Ordre d'exécution de deux instances

Soit $\boldsymbol{k}_a$ et $\boldsymbol{k}_b$ les préfixes de longueur $|\,\mathfrak{N}(a,b)|$ de $\boldsymbol{i}_a$ et $\boldsymbol{i}_b$ respectivement. L'ordre d'exécution noté $\prec_{\mathfrak{N}(a,b),\mathfrak{C}(a,b)}$ dépend du nid de boucles commun et de l'en-tête de contrôle commun à a et à b. Il est défini par

$$
\begin{aligned}
\langle a, \boldsymbol{i}_a\rangle \prec_{\mathfrak{N}(a,b),;} \langle b, \boldsymbol{i}_b\rangle &= ((\boldsymbol{k}_a = \boldsymbol{k}_b) \wedge (a\,T\,b)) \vee (\boldsymbol{k}_a \ll_{\mathfrak{N}(a,b)} \boldsymbol{k}_b) \\
\langle a, \boldsymbol{i}_a\rangle \prec_{\mathfrak{N}(a,b),||} \langle b, \boldsymbol{i}_b\rangle &= \boldsymbol{k}_a \ll_{\mathfrak{N}(a,b)} \boldsymbol{k}_b \\
\langle a, \boldsymbol{i}_a\rangle \prec_{\mathfrak{N}(a,b),?} \langle b, \boldsymbol{i}_b\rangle &= \boldsymbol{k}_a \ll_{\mathfrak{N}(a,b)} \boldsymbol{k}_b
\end{aligned}
$$

À partir de cette relation, nous définissons la formule de précédence de contrôle notée $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y})$ entre deux instances $\langle a, \boldsymbol{i}_a\rangle$ et $\langle b, \boldsymbol{i}_b\rangle$ en associant la liste de variables $\boldsymbol{x}$ à $\boldsymbol{i}_a$ et la liste $\boldsymbol{y}$ à $\boldsymbol{i}_b$:

$$\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\langle a, \boldsymbol{x}\rangle \prec_{\mathfrak{N}(a,b),\mathfrak{C}(a,b)} \langle b, \boldsymbol{y}\rangle)$$

Des exemples de calcul de Pre^0 sont donnés figure 4.1.

```
    PARALLEL DO I=1,10
a:    A(I) = F(...)
      DO J = 1,10
        PARALLEL SECTIONS
          SECTION
b:          B(J) = A(J) + B(J-1)
          SECTION
c:          C(J) = A(J) + C(J-1)
        END PARALLEL SECTIONS
      END DO
    END PARALLEL DO
```

Les instructions a et b satisfont :

- $\mathfrak{J}(a) = [\mathtt{I}]$; $\mathfrak{J}(b) = [\mathtt{I}, \mathtt{J}]$
- $\mathfrak{N}(a, b) = [(\mathtt{pdo}, \mathtt{I}, 1, 10)]$
- $\mathfrak{J}(a, b) = [\mathtt{I}]$
- $\mathfrak{C}(a, b) =$; car l'ancêtre commun à a et à b est $\mathtt{stmt}^*$
- $a\,T\,b = true$
- $\mathrm{Pre}^0_{a,b}([x_1], [y_1, y_2]) = ((x_1 = y_1) \wedge true) \vee false$
  $= (x_1 = y_1)$

Les instructions b et c satisfont :

- $\mathfrak{J}(b) = \mathfrak{J}(c) = [\mathtt{I}, \mathtt{J}]$
- $\mathfrak{N}(b, c) = [(\mathtt{pdo}, \mathtt{I}, 1, 10), (\mathtt{do}, \mathtt{J}, 1, 10)]$
- $\mathfrak{J}(b, c) = [\mathtt{I}, \mathtt{J}]$
- $\mathfrak{C}(b, c) = ||$ car l'ancêtre commun à b et à c est $\mathtt{section}^*$ et les sections contenant b et c ne sont pas ordonnées par un `WAIT` de section
- $b\,T\,c = true$
- $\mathrm{Pre}^0_{b,c}([x_1, x_2], [y_1, y_2]) = (x_1 = y_1) \wedge ((x_2 < y_2) \vee ((x_2 = y_2) \wedge false))$
  $= (x_1 = y_1) \wedge (x_2 < y_2)$

FIG. 4.1 – *En-tête de contrôle commun et précédence de contrôle*

4.2 Formule de synchronisation

4.2.1 Couple de synchronisation

Synchronisation par événements

Soit a une instruction `POST` et b une instruction `WAIT` opérant sur des événements de même nom. On a $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(b) = \{n\}, \mathfrak{Out}(a) = \{r_p\}, r_w \in \mathfrak{In}(b)$ et $\text{strip}(r_p) = \text{strip}(r_w) = n$. On appelle couple de synchronisation par événements, le couple (r_p, r_w).

Synchronisation par ordinaux

Soit a une instruction `POST` et b une instruction `WAIT` opérant sur des ordinaux de même nom. On a $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(b) = \{n\}, \mathfrak{Out}(a) = \{r_p\}, r_w \in \mathfrak{In}(b)$ et $\text{strip}(r_p) = \text{strip}(r_w) = n$. On appelle couple de synchronisation par ordinaux, le couple (r_p, r_w).

La valeur de l'expression a.`iexp` est appelée le rang de l'ordinal posté et la valeur de b.`iexp` le rang de l'ordinal attendu.

Soit a et b deux instructions `POST` opérant sur des ordinaux de même nom. On a $\mathfrak{Out}(a) = \{r_p\}$ et $\mathfrak{Out}(b) = \{r_{p'}\}$. Le couple $(r_p, r_{p'})$ est appelé couple de synchronisation ordinale implicite. Notons qu'on peut avoir le couple de synchronisation (r_p, r_p) si $a = b$.

4.2.2 Formule de synchronisation

A chaque couple de synchronisation $u = (r_p, r_w)$ est associée une formule de synchronisation $\text{Sync}_u : a \rightarrow b$ notée aussi $\text{Sync}_{p,w}$. r_p et r_w sont des références scalaires ou tableaux. Appelons d la dimension commune de leur liste d'expressions et soient $r_p.\texttt{iexp}^* = [e_1, \ldots, e_d]$, et $r_w.\texttt{iexp}^* = [f_1, \ldots, f_d]$ ces listes.

- Pour un couple de synchronisation par événements, $\text{Sync}_u : a \rightarrow b$ est la conjonction des équations du système suivant :

$$[\boldsymbol{x}/\,\mathfrak{I}(a)]e_i \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]f_i \quad i = 1, \ldots, d$$

- Pour un couple de synchronisation par ordinaux, la formule de synchronisation est la conjonction de :

$$\begin{cases} [\boldsymbol{x}/\,\mathfrak{I}(a)]e_i \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]f_i & i = 1, \ldots, d \\ [\boldsymbol{x}/\,\mathfrak{I}(a)]a.\texttt{iexp} \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]b.\texttt{iexp} \end{cases}$$

où $\boldsymbol{x}$ (resp. $\boldsymbol{y}$) est un tuple de variables entières de longueur $|\,\mathfrak{N}(a)|$ (resp. $|\,\mathfrak{N}(b)|$).

À un couple de synchronisation ordinale implicite, nous associons une formule de synchronisation qui est la disjonction du système :

$$\begin{cases} [\boldsymbol{x}/\,\mathfrak{I}(a)]e_i \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]f_i & i = 1,\ldots,d \\ [\boldsymbol{x}/\,\mathfrak{I}(a)]a.\texttt{iexp} + s \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]b.\texttt{iexp} \end{cases}$$

et du système :

$$\begin{cases} [\boldsymbol{x}/\,\mathfrak{I}(a)]e_i \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]f_i & i = 1,\ldots,d \\ [\boldsymbol{y}/\,\mathfrak{I}(b)]b.\texttt{iexp} + s \;=\; [\boldsymbol{x}/\,\mathfrak{I}(a)]a.\texttt{iexp} \end{cases}$$

où s est l'incrément de l'ordinal, $\boldsymbol{x}$ et $\boldsymbol{y}$ sont des tuples de variables entières de longueur $|\,\mathfrak{N}(a)|$ (resp. $|\,\mathfrak{N}(b)|$). Seul un des deux systèmes peut être satisfait car l'incrément s est strictement positif. Dans la plupart des cas, on aura $a = b$ et la formule de synchronisation sera la conjonction des équations et inéquations du système suivant :

$$\begin{cases} [\boldsymbol{x}/\,\mathfrak{I}(a)]e_i \;=\; [\boldsymbol{y}/\,\mathfrak{I}(a)]e_i & i = 1,\ldots,d \\ [\boldsymbol{x}/\,\mathfrak{I}(a)]a.\texttt{iexp} + s \;=\; [\boldsymbol{y}/\,\mathfrak{I}(b)]a.\texttt{iexp} \end{cases}$$

On notera que la forme de ces systèmes rappelle celle du système associé à la formule de dépendance définie au chapitre 3.

Nous allons étudier dans quelles conditions la formule Sync peut être interprétée comme une relation de précédence entre deux instructions à l'instar de Pre^0 et comment ces deux formules peuvent être combinées pour déterminer un prédicat de précédence générale sur un programme parallèle.

4.3 Précédence généralisée

Nous cherchons à définir un prédicat général de précédence que nous notons $\mathrm{Pre}_{a,b}(\boldsymbol{i}_a, \boldsymbol{i}_b)$ sur le programme parallèle qui exprime que si les instances $\langle a, \boldsymbol{i}_a\rangle$ et $\langle b, \boldsymbol{i}_b\rangle$ sont exécutées alors $\langle a, \boldsymbol{i}_a\rangle$ est exécutée avant $\langle b, \boldsymbol{i}_b\rangle$ pour toute exécution parallèle.

Autres notations :

- Si $\alpha = \langle a, \boldsymbol{i}_a\rangle$ et $\beta = \langle b, \boldsymbol{i}_b\rangle$ sont deux instances d'instruction, on note $\mathrm{Pre}(\alpha, \beta)$ (resp. $\mathrm{Pre}^0(\alpha, \beta)$) à la place de $\mathrm{Pre}_{a,b}(\boldsymbol{i}_a, \boldsymbol{i}_b)$ (resp. $\mathrm{Pre}^0_{a,b}(\boldsymbol{i}_a, \boldsymbol{i}_b)$).
- On confond une instance d'instruction avec l'instruction elle-même si celle-ci ne possède qu'une seule instance. Par exemple, a est utilisé à la place de $\langle a, [\,]\rangle$.

4.3.1 Graphe de flot

Nous généralisons ici, aux programmes parallèles, le graphe de flot de contrôle $\mathcal{FG}$ [1]. Ce graphe est introduit dans le but de définir le prédicat

de précédence sur les instances d'instruction du programme parallèle. C'est un graphe d'instances[1]. Donc si l'on prend un programme contenant des boucles, son graphe possédera un nombre infini de sommets. C'est pourquoi, un autre graphe de flot sera utilisé par la suite pour vérifier effectivement l'existence d'une précédence entre deux instructions soumises à une condition supplémentaire de dépendance.

Les sommets du graphe sont des instances d'éléments syntaxiques qui appartiennent à l'une des quatre catégories suivantes :

- instructions simples: `assign`, `epost`, `ewait`, `clear`, `opost`, `owait`, `set`;
- instructions structurées : `if`, `do`, `pdo`, `psections`;
- condition booléenne d'un `if`;
- en-tête de boucle `do`.

Les arcs sont soit des arcs de contrôle séquentiel soit des arcs de synchronisation. On a un arc de contrôle entre une instance $\langle a, \boldsymbol{i}_a \rangle$ et une instance $\langle b, \boldsymbol{i}_b \rangle$ dans les cas suivants :

1. Instruction simple (voir figure 4.2)
   (a) si a est une instruction simple et si $succ(a)$ (Cf. page 44) existe alors $b = succ(a)$ et $\boldsymbol{i}_a = \boldsymbol{i}_b$
   (b) si a est une instruction simple telle que $succ(a)$ n'existe pas, soit c l'instruction structurée ancêtre de a le plus proche dans l'arbre de syntaxe abstraite et ayant un successeur alors $b = succ(c)$ et $\boldsymbol{i}_a = \boldsymbol{i}_b @ \boldsymbol{i}$ pour tout $\boldsymbol{i} \in \mathbb{Z}^k$ avec $k = |\boldsymbol{i}_a| - |\boldsymbol{i}_b|$
   (c) (cas des `WAIT` de sections) si a est une instruction simple telle que $succ(a)$ n'existe pas, soit c la `psections` ancêtre de a la plus proche telle qu'il n'existe pas dans sa descendance d'instruction structurée ancêtre de a ayant un successeur, $s_1 \in c.\texttt{section}^*$ ancêtre de a, $s_2 \in c.\texttt{section}^*$ tels que $s_1 \neq s_2$ et $s_1.\texttt{sectionhead.name} \in s_2.\texttt{sectionhead.sectionwait.name}^*$, alors $b = first(s_2.\texttt{body.stmt}^*)$ et $\boldsymbol{i}_a = \boldsymbol{i}_b @ \boldsymbol{i}$ pour tout $\boldsymbol{i} \in \mathbb{Z}^k$ avec $k = |\boldsymbol{i}_a| - |\boldsymbol{i}_b|$
   (d) (cas des itérations successives d'un `DO`) si a est une instruction simple telle que $succ(a)$ n'existe pas, soit c l'instruction `do` ancêtre de a la plus proche telle qu'il n'existe pas dans sa descendance d'instruction structurée ancêtre de a ayant un successeur, alors $b = c.\texttt{head}$ et $\boldsymbol{i}_a = \boldsymbol{i}_c @ [i] @ \boldsymbol{i}$, $\boldsymbol{i}_b = \boldsymbol{i}_c @ [i+1]$ pour tout $i \in \mathbb{Z}$ et pour tout $\boldsymbol{i} \in \mathbb{Z}^k$ avec $k = |\boldsymbol{i}_a| - (|\boldsymbol{i}_b| + 1)$

1. Dans la terminologie de [1], un *graphe de flot* est un graphe dont les sommets sont pris dans l'ensemble des nœuds de l'arbre syntaxique.

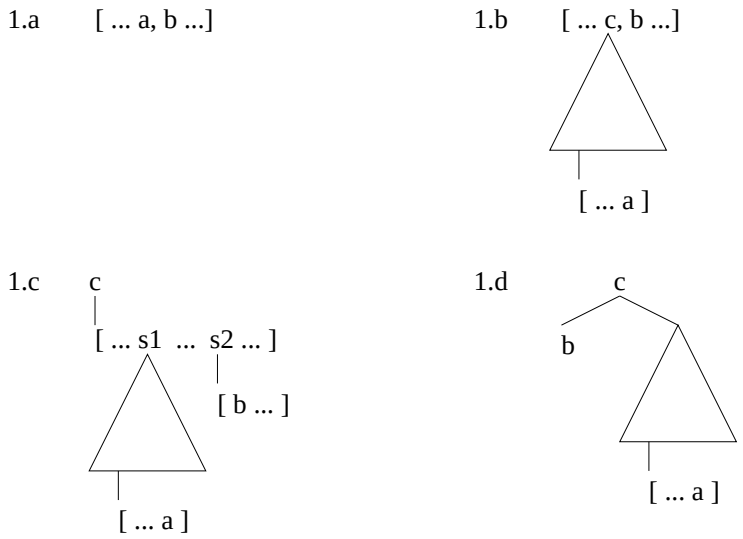

FIG. 4.2 – *Cas où un arc du graphe de flot a pour origine une instruction simple*

2. `if`
   - si a est un `if`, $b = a$.`bexp` et $\boldsymbol{i}_a = \boldsymbol{i}_b$
3. `if.bexp`
   - si $a = c$.`bexp` où c est un `if`, $b = first(c.\texttt{thenbody.stmt}^*)$ ou $first(c.\texttt{elsebody.stmt}^*)$ et $\boldsymbol{i}_a = \boldsymbol{i}_b$
4. `do`
   - si a est un `do`, $b = a$.`head` et $\boldsymbol{i}_b = \boldsymbol{i}_a @[i]$ pour tout $i \in \mathbb{Z}$
5. `do.head`
   - si $a = c$.`head` où c est un `do`, $b = first(a.\texttt{body.stmt}^*)$ et $\boldsymbol{i}_a = \boldsymbol{i}_b$
6. `pdo`
   - si a est un `pdo`, $b = first(a.\texttt{body.stmt}^*)$ et $\boldsymbol{i}_b = \boldsymbol{i}_a @[i]$ pour tout $i \in \mathbb{Z}$
7. `psections`
   - si a est une `psections`, $s \in a.\texttt{section}^*$ tel que $s.\texttt{sectionhead}.\texttt{sectionwait.name}^* = [\,]$, $b = first(s.\texttt{body.stmt}^*)$ et $\boldsymbol{i}_a = \boldsymbol{i}_b$

Les clauses ci-dessus utilisent les opérateurs *succ* et $first$ sur les listes d'instructions. *succ* représente le successeur d'une instruction s'il existe, $first$ le premier élément de la liste d'instructions. On a supposé pour simplifier qu'une liste d'instructions contenait au moins un élément. Bien que les cas des instructions `do` et `pdo` soient formellement identiques, nous les traitons différemment ; pour le `do`, nous faisons intervenir son en-tête (`do.head`) qui est la cible de l'arc en retour de la boucle (cas 1d).

Soit a et b deux instructions de synchronisation, on a un arc de synchronisation entre une instance $\langle a, \boldsymbol{i}_a \rangle$ et une instance $\langle b, \boldsymbol{i}_b \rangle$ dans les cas suivants :

– si a est un `epost` et b un `ewait` sur des événements de même nom et si

$$[\boldsymbol{i}_a / \mathfrak{I}(a)]e_k \;=\; [\boldsymbol{i}_b / \mathfrak{I}(b)]f_k \quad k = 1, \ldots, d$$

– si a est un `opost` et b un `owait` sur des ordinaux de même nom et si

$$\begin{cases} [\boldsymbol{i}_a / \mathfrak{I}(a)]e_k \;=\; [\boldsymbol{i}_b / \mathfrak{I}(b)]f_k & k = 1, \ldots, d \\ [\boldsymbol{i}_a / \mathfrak{I}(a)]a.\texttt{iexp} \;=\; [\boldsymbol{i}_b / \mathfrak{I}(b)]b.\texttt{iexp} \end{cases}$$

– si a et b sont deux `opost` sur des ordinaux de même nom dont l'incrément vaut s et si

$$\begin{cases} [\boldsymbol{i}_a / \mathfrak{I}(a)]e_k \;=\; [\boldsymbol{i}_b / \mathfrak{I}(b)]f_k & k = 1, \ldots, d \\ [\boldsymbol{i}_a / \mathfrak{I}(a)]a.\texttt{iexp} + s \;=\; [\boldsymbol{i}_b / \mathfrak{I}(b)]b.\texttt{iexp} \end{cases}$$

ou bien

$$\begin{cases} [\boldsymbol{i}_a / \mathfrak{I}(a)]e_k \;=\; [\boldsymbol{i}_b / \mathfrak{I}(b)]f_k & k = 1, \ldots, d \\ [\boldsymbol{i}_b / \mathfrak{I}(b)]b.\texttt{iexp} + s \;=\; [\boldsymbol{i}_a / \mathfrak{I}(a)]a.\texttt{iexp} \end{cases}$$

où $[e_1, \ldots, e_d] = a.\texttt{lhs.iexp}^*$ et $[f_1, \ldots, f_d] = b.\texttt{lhs.iexp}^*$.

On représente un arc de contrôle par une flèche courte $\rightarrow$, un arc de synchronisation par une flèche $\rightsquigarrow$ et un chemin par une flèche longue $\longrightarrow$.

Si α et β sont deux sommets du graphe alors on note $\alpha \longrightarrow \beta$ la relation « il existe un chemin de α à β » : $\longrightarrow = (\rightarrow \cup \rightsquigarrow)^*$.

4.3.2 Chemin et précédence

En général, l'existence d'un chemin entre deux instances α et β ne suffit pas pour dire que si α et β sont exécutées alors α est exécutée avant β.

Proposition 1 *i. S'il existe un chemin entre une instance α et une instance β ne contenant que des arcs de contrôle, alors* $\mathrm{Pre}^0(\alpha, \beta)$ *est vrai et par suite α est exécutée avant β.*

ii. Si tous les chemins de α à β contiennent au moins un arc de synchronisation et si α et β sont exécutées, alors on ne peut pas conclure que α précède β dans l'ordre d'exécution.

Nous allons démontrer i par récurrence sur la longueur du chemin.

Soit $\alpha = \langle a, \boldsymbol{i}_a \rangle$ et $\beta = \langle b, \boldsymbol{i}_b \rangle$ deux instances d'instruction, supposons que α et β sont reliés par un seul arc de contrôle. Nous devons examiner cas par cas chacune des situations où un arc de contrôle existe entre α et β (Cf. page 42) :

1. (a) nous avons nécessairement $a\,T\,b$, de plus $\boldsymbol{i}_a = \boldsymbol{i}_b$ donc $\mathrm{Pre}^0(\alpha, \beta)$.
   (b) nous avons $a\,T\,b$, de plus $\boldsymbol{i}_a =_{|\,\mathfrak{N}(a,b)} \boldsymbol{i}_b$ donc $\mathrm{Pre}^0(\alpha, \beta)$.
   (c) nous avons $a\,T\,b$, de plus $\boldsymbol{i}_a =_{|\,\mathfrak{N}(a,b)} \boldsymbol{i}_b$ donc $\mathrm{Pre}^0(\alpha, \beta)$.
   (d) nous avons $b\,T\,a$ mais $\boldsymbol{i}_a \ll_{|\,\mathfrak{N}(a,b)} \boldsymbol{i}_b$ donc $\mathrm{Pre}^0(\alpha, \beta)$.
2. nous avons $a\,T\,b$ et $\boldsymbol{i}_a = \boldsymbol{i}_b$ donc $\mathrm{Pre}^0(\alpha, \beta)$.
3. idem cas 2.
4. nous avons $a\,T\,b$ et $\boldsymbol{i}_a =_{|\,\mathfrak{N}(a,b)} \boldsymbol{i}_b$ donc $\mathrm{Pre}^0(\alpha, \beta)$.
5. idem cas 2.
6. idem cas 4.
7. idem cas 2.

Donc, si deux instances α et β sont reliés par un arc de contrôle du graphe de flot alors α est exécutée avant β.

Nous supposons par récurrence que cette propriété est vraie pour un chemin de longueur $n-1$. Montrons qu'elle est vraie pour un chemin de longueur n. Soit $\gamma = \langle c, \boldsymbol{i}_c \rangle$ une instance d'instruction telle qu'il existe un chemin de contrôle de longueur $n-1$ entre α et γ et un arc de contrôle simple entre γ et β, si nous montrons que $\mathrm{Pre}^0(\gamma, \beta)$ est vraie alors grâce à l'hypothèse de récurrence et à la transitivité de Pre^0, nous pourrons en déduire que $\mathrm{Pre}^0(\alpha, \beta)$ est vraie. La démonstration est la même que pour le cas $n = 1$: il suffit de remplacer a par c dans chacun des cas possibles. Ceci prouve le i de la proposition.

Pour le *ii*, nous allons fournir deux contre-exemples :

1. Dans l'exemple suivant, c'est l'exécution concurrente de deux `epost` de même événement qui empêche d'avoir la précédence.

```
      PARALLEL SECTIONS (ORDERED)
       SECTION
a:       ...
```

```
p1:     POST(E)
       SECTION
a':       ...
p2:     POST(E)
       SECTION
w:      WAIT(E)
b:       ...
     END PARALLEL SECTIONS
```

On a seulement : $a \longrightarrow b \wedge a' \longrightarrow b \Rightarrow \mathrm{Pre}(a,b) \vee \mathrm{Pre}(a',b)$.

2. De même l'exécution concurrente d'un `epost` et d'un `clear` de même événement empêche d'avoir la précédence. Dans l'exemple suivant, la précédence $\mathrm{Pre}(a,b)$ n'est pas assurée car l'instruction c peut être exécutée après p.

```
     PARALLEL SECTIONS (ORDERED)
      SECTION
a:     ...
p:     POST(E)
      SECTION
        ...
c:     CLEAR(E)
      SECTION
w:     WAIT(E)
b:     ...
     END PARALLEL SECTIONS
```

□

4.3.3 Restrictions

Ceci nous conduit à imposer les restrictions suivantes :

- **Hypothèse S1** Pour toute instance γ d'instruction `clear` référençant un événement e, si ξ est une instance d'instruction `epost` ou `ewait` référençant le même événement, alors γ et ξ sont exécutées dans des branches alternatives d'un `if` ou sont dans la relation de précédence de contrôle : $\mathrm{Pre}^0(\gamma,\xi) \vee \mathrm{Pre}^0(\xi,\gamma)$.

- **Hypothèse S2** Pour toute instance d'instruction `ewait` référençant un événement e, s'il existe plus d'une instance d'instruction `epost` référençant le même événement e alors, elles sont exécutées dans des branches alternatives d'un `if`.

L'hypothèse S2 est illustrée par l'exemple ci-dessous.

```
      PARALLEL SECTIONS (ORDERED)
       SECTION
         IF (B)
           THEN
a:           ...
p1:          POST(E)
           ELSE
a':          ...
p2:          POST(E)
       SECTION
w:       WAIT(E)
b:       ...
      END PARALLEL SECTIONS
```

Nous faisons des restrictions analogues pour les synchronisations par ordinaux :

- **Hypothèse S1'** Pour toute instance γ d'instruction `set` référençant un ordinal o, si ξ est une instance d'instruction `opost` ou `owait` référençant le même ordinal, alors γ et ξ sont exécutées dans des branches alternatives d'un `if` ou sont dans la relation de précédence de contrôle : $\mathrm{Pre}^0(\gamma, \xi) \vee \mathrm{Pre}^0(\xi, \gamma)$.

- **Hypothèse S2'** Pour toute instance d'instruction `owait` testant un ordinal o à la valeur n, s'il existe plus d'une instance d'instruction `opost` susceptible d'incrémenter l'ordinal o à une valeur supérieure ou égale à n, alors elles sont exécutées dans des branches alternatives d'un `if`.

- **Hypothèse S3'** Pour toute instance d'instruction `opost` testant un ordinal o à la valeur n, s'il existe plus d'une instance d'instruction `opost` susceptible d'incrémenter l'ordinal o à une valeur égale à n, alors elles sont exécutées dans des branches alternatives d'un `if`.

Ces hypothèses sont nécessaires pour assurer que tout couple (π, ω) de synchronisation établit bien une précédence entre π et ω. Remarquons qu'une même instance d'instruction `epost` (resp. `opost`) peut poster un événement (resp. un ordinal) lu par deux ou plusieurs instances d'instruction `ewait` (resp. `owait`) non exclusives.

4.3.4 Précédence généralisée

On peut penser naïvement que le prédicat Pre est obtenu à partir de Pre^0 et de Sync par simple composition le long d'un chemin du graphe de

flot. Mais celle-ci n'est pas transitive, $\mathrm{Pre}(\alpha,\beta)$ et $\mathrm{Pre}(\beta,\gamma)$ n'impliquent pas $\mathrm{Pre}(\alpha,\gamma)$, comme le montre la construction ci-dessous :

```
      PARALLEL SECTIONS (ORDERED)
      SECTION
p:     POST(E)
      SECTION
       IF (B)
       THEN
w:      WAIT(E)
       ELSE
        A=1
       ENDIF
a:      A=2
      END PARALLEL SECTIONS
```

On voit que p précède w, et w précède a, mais p ne précède pas a, parce que la branche `ELSE` peut être prise, et a exécutée concurremment avec p sans attendre `E`. Il y a précédence si on peut assurer que p et w sont exécutées.

Au lieu de la transitivité, on a seulement une transitivité modulo un prédicat d'exécution $\mathrm{Exe}^{\mathsf{s}}$ défini en 4.3.5 :

$$\mathrm{Pre}(\alpha,\beta) \wedge \mathrm{Exe}^{\mathsf{s}}(\beta) \wedge \mathrm{Pre}(\beta,\gamma) \Rightarrow \mathrm{Pre}(\alpha,\gamma)$$

Définition 11 *Soit P un chemin reliant α à β, il a la forme suivante :*

$$\alpha \stackrel{*}{\rightarrow} \pi_1 \rightsquigarrow \omega_1 \stackrel{*}{\rightarrow} \pi_2 \rightsquigarrow \omega_2 \stackrel{*}{\rightarrow} \ldots \stackrel{*}{\rightarrow} \pi_n \rightsquigarrow \omega_n \stackrel{*}{\rightarrow} \beta,$$

où $\stackrel{*}{\rightarrow}$ est true s'il s'agit du chemin vide et la relation Pre^0 sinon, $\rightsquigarrow$ un Sync *et π_i, ω_i un couple* `epost,ewait` *ou* `opost,owait` *ou encore* `opost,opost`*. Le prédicat associé à P est :*

$$\begin{aligned}\mathrm{Pre}(P) \quad = \quad & \mathrm{Pre}^0(\alpha,\pi_1) \wedge \mathrm{Exe}^{\mathsf{s}}(\pi_1) \wedge \mathrm{Sync}(\pi_1,\omega_1) \wedge \mathrm{Exe}^{\mathsf{s}}(\omega_1) \wedge \mathrm{Pre}^0(\omega_1,\pi_2) \\ & \wedge \ldots \wedge \mathrm{Exe}^{\mathsf{s}}(\omega_n) \wedge \mathrm{Pre}^0(\omega_n,\beta)\end{aligned}$$

La relation $\mathrm{Pre}(\alpha,\beta)$ est obtenue en considérant tous les chemins de précédence entre les instances α et β, ce qui se traduit par :

$$\text{pour tout } P : \alpha \longrightarrow \beta,\ \mathrm{Pre}(P) \Rightarrow \mathrm{Pre}(\alpha,\beta)$$

ce qui est équivalent à :

$$(\bigvee_{P:\alpha\longrightarrow\beta} \mathrm{Pre}(P)) \Rightarrow \mathrm{Pre}(\alpha,\beta)$$

où $\bigvee_{P:\alpha\longrightarrow\beta}$ est la disjonction des relations $\mathrm{Pre}(P)$.

Comme toute l'information de précédence est contenue dans le graphe on donne pour Pre la définition suivante :

Définition 12 *Pour toute instance α et β, si $P : \alpha \longrightarrow \beta$ est un chemin du graphe de flot reliant α à β et si* $\mathrm{Pre}(P)$ *est le prédicat associé à P défini ci-dessus, alors*

$$\mathrm{Pre}(\alpha, \beta) = \bigvee_{P:\alpha \longrightarrow \beta} \mathrm{Pre}(P) \tag{4.1}$$

Soulignons que Pre est un prédicat et non une formule ; il est défini par la disjonction d'un ensemble éventuellement infini de prédicats ce qui a un sens pour un prédicat mais qui n'en aurait pas pour une formule de logique du premier ordre.

4.3.5 Prédicat d'exécution

Nous voudrions définir un prédicat $\mathrm{Exe}(\alpha)$ pour chaque instance d'instruction α qui représente la condition pour que α soit exécutée dans toute exécution parallèle. Pour les instructions `ewait`, `owait` et `opost` cela signifie « passer à l'instruction suivante ».

Dans le cadre de la syntaxe que nous avons retenue, son expression dépend des bornes de boucles englobantes et des expressions booléennes des `if` englobants. Nous avons restreint les bornes de boucles à des expressions contenant des paramètres et des indices de boucles englobantes. Par contre les expressions booléennes des `if` peuvent contenir n'importe quelles variables du programme. Or, tant qu'un programme parallèle n'a pas été prouvé correct, la valeur d'une expression contenant des variables quelconques n'est pas forcément définie : elle peut être différente d'une exécution à une autre ou ne pas exister. On peut seulement affirmer que si le programme parallèle est correct (au sens de la sémantique séquentielle), la valeur de l'expression sera égale à celle calculée par sa version séquentielle.

On va donc s'intéresser à la condition d'exécution $\mathrm{Exe}^{\mathbf{s}}$ d'une instruction dans la version séquentielle.

Pour définir $\mathrm{Exe}^{\mathbf{s}}$, dont l'expression ne dépend pas seulement de la valeur du vecteur d'itération mais de l'état mémoire tout entier, nous devons introduire la fonction $\mathcal{B} : \texttt{exp} \to (\mathbf{U} \to (\mathbf{S} \to \mathbf{B}))$ qui à une expression booléenne associe sa dénotation. Cette dernière prend en arguments un environnement $\rho \in \mathbf{U}$ et un état mémoire $\sigma \in \mathbf{S}$, et renvoie un booléen. Précisons qu'un environnement $\rho \in \mathbf{U}$ associe à un nom muni ou non d'une liste d'entiers un emplacement mémoire et qu'un état mémoire $\sigma \in \mathbf{S}$ associe à un emplacement mémoire une *valeur* dont nous ne détaillerons pas le domaine.

Si a est une instruction, pour décrire la valeur d'une expression $a.e$ de type `iexp` comme une borne de boucle, qui ne dépend que d'indices de boucles englobantes et de constantes, nous noterons simplement $e_{\boldsymbol{i}_a}$ où $\boldsymbol{i}_a$ est un vecteur d'entiers de longueur $|\,\mathfrak{N}(a)|$.

Soit a une instruction, c un `if`, b un `do` ou un `pdo`, s une `psections` et p un programme, Exe^{s} est défini sur les instances d'instruction par

- si $a \in p.\texttt{stmt}^*$ alors $\boldsymbol{i}_a = []$ et $\text{Exe}^{\text{s}}(\langle a, [] \rangle) = \mathit{true}$
- si $a \in s.\texttt{section.body.stmt}^*$ alors $\boldsymbol{i}_a = \boldsymbol{i}_s$ et $\text{Exe}^{\text{s}}(\langle a, \boldsymbol{i}_a \rangle) = \text{Exe}^{\text{s}}(\langle s, \boldsymbol{i}_s \rangle)$
- si $a \in c.\texttt{thenbody.stmt}^*$ alors $\boldsymbol{i}_a = \boldsymbol{i}_c$ et $\text{Exe}^{\text{s}}(\langle a, \boldsymbol{i}_a \rangle) = \text{Exe}^{\text{s}}(\langle c, \boldsymbol{i}_c \rangle) \wedge \mathcal{B}[\![c.\texttt{bexp}]\!]\rho_{\boldsymbol{i}_a}\sigma_{\boldsymbol{i}_a}$
- si $a \in c.\texttt{elsebody.stmt}^*$ alors $\boldsymbol{i}_a = \boldsymbol{i}_c$ et $\text{Exe}^{\text{s}}(\langle a, \boldsymbol{i}_a \rangle) = \text{Exe}^{\text{s}}(\langle c, \boldsymbol{i}_c \rangle) \wedge \neg(\mathcal{B}[\![c.\texttt{bexp}]\!]\rho_{\boldsymbol{i}_a}\sigma_{\boldsymbol{i}_a})$
- si $a \in b.\texttt{body.stmt}^*$ alors $\boldsymbol{i}_a = \boldsymbol{i}_b@[i]$ et $\text{Exe}^{\text{s}}(\langle a, \boldsymbol{i}_a \rangle) = \text{Exe}^{\text{s}}(\langle b, \boldsymbol{i}_b \rangle) \wedge ((b.\texttt{head.lb})_{\boldsymbol{i}_b} \leqslant i \leqslant (b.\texttt{head.ub})_{\boldsymbol{i}_b})$

$\sigma_{\boldsymbol{i}}$ (resp. $\rho_{\boldsymbol{i}}$) est l'état mémoire (resp. l'environnement) dans lequel s'exécute l'instance $\langle a, \boldsymbol{i} \rangle$.

4.4 Correction d'un programme

4.4.1 Sémantique séquentielle

Notre but est de prouver la correction d'un programme parallèle. Notre critère de correction est l'équivalence sémantique du programme parallèle avec sa version séquentielle.

Définition 13 *La version séquentielle d'un programme parallèle ne comportant pas d'objets privés est obtenue en transformant les* `PARALLEL DO` *en* `DO`*, les* `PARALLEL SECTIONS` *en des blocs de code consécutifs et les* `POST`*,* `WAIT`*,* `CLEAR` *et* `SET` *en* `CONTINUE`*.*

La figure 4.3 montre un exemple de transformation.

```
A(0) = ...                              A(0) = ...
POST(E(0))                              CONTINUE
PARALLEL DO (ORDERED) I=1,N             DO I=1,N
  WAIT(E(I-1))                            CONTINUE
  ...  = A(I-1)                           ...  = A(I-1)
  A(I) = ...                              A(I) = ...
  POST(E(I))                              CONTINUE
END PARALLEL DO                         END DO
...                                     ...
```

FIG. 4.3 – *Un programme et sa version séquentielle*

Un programme parallèle correct peut être vu comme une parallélisation d'un programme séquentiel dans le but d'accroître la vitesse d'exécution si plusieurs processeurs sont disponibles.

La présence de variables privées explicites complique la situation. Nous discutons une possible transformation à la section 4.4.2.

Si nous sommes assurés que tous les programmes corrects écrits dans notre langage ont une version séquentielle au comportement équivalent il n'en est pas de même dans le cadre d'un langage parallèle plus étendu. Si, par exemple, nous avions ajouté les sections critiques, la transformation simple d'un programme par élimination des constructions parallèles n'aurait pas toujours conduit à un programme équivalent.

L'exemple ci-dessous montre un programme non déterministe conforme au standard X3H5. Le verrou `GMAXA` contrôle l'accès à la variable `MAXA`. Le tableau `B` est normalisé par `MAXA` d'une façon non déterministe qui dépend du nombre de processus disponibles, de l'ordre d'affectation des sections aux processus et de leur vitesse relative. En particulier, la valeur de `MAXA` peut aussi bien être celle à l'entrée de la procédure que le plus grand élément du tableau `A` calculée par la première section et qui est la valeur à la sortie de la procédure. Ce programme ne peut recevoir une sémantique séquentielle équivalente.

```
      SUBROUTINE ND (A,B,MAXA,GMAXA,N)
      REAL A(N), B(N), MAXA
      TYPE(LATCH) GMAXA
      PARALLEL SECTIONS
      SECTION
        REAL AM = A(1)
        DO 10 I=2,N
           IF(AM.LT.A(I))AM=A(I)
10      CONTINUE
        CRITICAL SECTION (GMAXA) GUARDS(MAXA)
           IF(MAXA.LT.AM) MAXA=AM
        END CRITICAL SECTION (GMAXA)
      SECTION
        REAL AM
        CRITICAL SECTION (GMAXA)
           AM=MAXA
        END CRITICAL SECTION (GMAXA) GUARDS(MAXA)
        DO 20 I=1,N
           B(I)=B(I)/AM
20      CONTINUE
      END PARALLEL SECTIONS
      END
```

4.4.2 Gestion des objets privés

Nous nous posons ici la question de savoir comment considérer les variables privées vis à vis des notions de dépendance et de version séquentielle.

Rappelons que des variables privées peuvent apparaître sous deux formes : déclarées explicitement en début d'une construction parallèle ou implicites pour les indices des `PARALLEL DO`.

Pour les indices de boucles, nous avons supposé qu'ils n'étaient pas modifiables. Il peuvent cependant être impliqués dans une dépendance comme le montre le fragment de programme suivant :

```
  PARALLEL DO I = ...
    ...
    X = I
    ...
  END PARALLEL DO
```

dans lequel `X` est une variable partagée. La dépendance sur `I` est trivialement préservée par le flot de contrôle puisque la variable `I` est privée pour chaque processus engagé dans l'exécution du `PARALLEL DO`. La situation est plus compliquée pour les variables privées explicites.

Pour l'exécution du corps d'une construction parallèle C, chaque processus reçoit, dans son environnement, une copie de chaque variable privée déclarée dans la construction parallèle. Donc si P_1 et P_2 sont deux processus appartenant à une même équipe E et si x est une variable privée alors on ne peut avoir de dépendance entre le x de P_1 et le x de P_2. Au contraire, si u_1 et u_2 sont deux unités de travail définies par C, alors on peut avoir une dépendance entre une occurrence de x dans u_1 et une occurrence de x dans u_2 si u_1 et u_2 sont exécutées par le même processus. L'existence de dépendances de données sur une variable privée dépend de la distribution des unités aux processus de l'équipe ce qui peut poser des problèmes pour établir une version séquentielle équivalente. Considérons l'exemple suivant dans lequel la variable `T` est privée :

```
  PARALLEL SECTIONS
    INTEGER T
    SECTION A
a:    T = 0
    SECTION B WAIT A
b:    X = T
  END PARALLEL SECTIONS
```

Dans la version séquentielle de ce programme telle que nous l'avons définie jusqu'à présent, il y a une dépendance sur `T` entre les instructions a et b qui conduit à affecter la valeur 0 à la variable `X` car `T` est traitée comme une variable ordinaire. Les exécutions parallèles où les sections `A` et `B` sont

affectées à un même processus se comportent comme la version séquentielle en raison du `WAIT` de section. Cependant, dans toute exécution parallèle où la section `A` est affectée à un processus P_1 et la section `B` à un processus différent P_2, la dépendance est supprimée puisque les processus P_1 et P_2 ne référencent pas le même emplacement mémoire pour `T`, laissant ainsi la variable `X` à une valeur indéfinie. Bien que le critère de préservation des dépendances soit respecté, toutes les exécutions parallèles ne se comportent pas comme la version séquentielle. La définition de la version séquentielle (Cf. définition 13 page 50) se révèle donc insuffisante en présence de variables privées.

Nous voulons conserver à la version séquentielle son caractère de référence et préserver l'usage des variables privées. L'exemple précédent a montré que la présence d'une variable privée pouvait rendre un programme non déterministe. Nous sommes donc obligés de restreindre l'utilisation d'une variable privée à la création d'une variable temporaire dont la portée est restreinte à une unité de travail.

Pour ce faire, nous proposons l'hypothèse suivante :

> **Hypothèse P1** Toute unité de travail u, dans laquelle existe une référence r en lecture à une variable T ayant l'attribut privé pour la construction parallèle qui définit u, doit initialiser r auparavant.

Cette hypothèse **P1** est vérifiable syntaxiquement. Elle pallie le problème d'initialisation des variables privées. Le fragment de programme précédent ne vérifie pas cette hypothèse car dans la `section B`, la variable `T` est lue mais n'a pas été initialisée dans la section ; à la place, on doit écrire par exemple :

```
  PARALLEL SECTIONS
    INTEGER T
    SECTION A
a:    T = 0
    SECTION B WAIT A
      T = 0
b:    X = T
  END PARALLEL SECTIONS
```

L'hypothèse **P1** nous permet de relier la portée d'une variable privée à une unité de travail, notion syntaxique indépendante de l'exécution, plutôt qu'à un processus, notion dynamique, et ce, sans perdre de dépendances donc sans changer le sens du programme parallèle en ce qui concerne les variables privées. C'est ce que nous formalisons dans l'hypothèse ci-dessous :

> **Hypothèse P2** Soit C une construction parallèle, u_1 et u_2 deux unités de travail de C et T une variable privée pour C, si une occurrence de T dans u_1 fait référence à une adresse α_1 et si une

occurrence de T dans u_2 fait référence à une adresse α_2 alors $\alpha_1 \neq \alpha_2$.

Avec cette hypothèse **P2**, nous modifions la sémantique du langage, pour qu'une copie de chaque variable privée soit allouée à chaque unité de travail quelque soit la distribution des unités aux processus de l'équipe, ce qui reflète le cas d'un parallélisme maximal où un processus est attribué à chaque unité de travail.

Munis de ces deux hypothèses, nous pouvons dériver une version séquentielle de référence de tout programme parallèle, qui tient compte du statut particulier des variables privées.

Définition 14 *La version séquentielle d'un programme parallèle comportant des objets privés est obtenue en transformant les* `PARALLEL DO` *en* `DO`*, les* `PARALLEL SECTIONS` *en des blocs de code consécutifs et les* `POST`*,* `WAIT`*,* `CLEAR` *et* `SET` *en* `CONTINUE`*, ceci comme précédemment. Mais, de plus, les variables privées introduites dans un* `PARALLEL SECTIONS` *sont renommées et les variables privées introduites dans un* `PARALLEL DO` *sont transformées en vecteur s'il s'agit de scalaires ou en tableau de dimension* $d+1$ *s'il s'agit de tableaux de dimension* d*.*

Nous donnons, ci-dessous, quelques exemples de transformation.

Par renommage :

```
PARALLEL SECTIONS                       INTEGER TPS1,TPS2
  INTEGER T
  SECTION
    T=0                                 TPS1=0
  SECTION
    T=0                                 TPS2=0
    X=T                                 X=TPS2
END PARALLEL SECTIONS
```

La variable `T` apparaît dans deux sections distinctes, référençant deux cases mémoire distinctes ; elle n'induit pas de dépendances donc chaque occurrence est transformée en une nouvelle variable.

Par promotion en tableau :

```
                                        REAL TDO
PARALLEL DO I=3,100                     DIMENSION TDO(3:100)
  REAL T                                DO I=3,100
  T=A(I)**2                               TDO(I)=A(I)**2
  B(I)=T*(T-1)                            B(I)=TDO(I)*(TDO(I)-1)
END PARALLEL DO                         END DO
```

La dépendance sur `T` est transformée en une dépendance sur `TDO(I)` pour tout `I`.

L'exemple suivant illustre le cas de constructions parallèles imbriquées : la variable scalaire `T` est privée pour le `PARALLEL DO` et partagée pour le `PARALLEL SECTIONS`.

```
                                      REAL TDO
                                      DIMENSION TDO(1:100)
  PARALLEL DO I=1,100                 DO I=1,100
    REAL T
    PARALLEL SECTIONS (ORDERED)
      SECTION
        T=A(I)                          TDO(I)=A(I)
        POST(E(I))
      SECTION
        WAIT(E(I))
        B(I)=T                          B(I)=TDO(I)
    END PARALLEL SECTIONS
  END PARALLEL DO                     END DO
```

La valeur de `T` utilisée par la $i$ème itération du `PARALLEL DO` a été calculée par cette même itération mais dans une section différente du `PARALLEL SECTIONS`.

4.5 Équivalence sémantique

Nous étudions la correction séquentielle d'un programme parallèle c'est-à-dire l'équivalence sémantique du programme parallèle avec sa version séquentielle telle que nous l'avons définie plus haut.

4.5.1 Préservation des dépendances

Sur la version séquentielle du programme qui constitue la sémantique de référence, tout emplacement mémoire induit une dépendance de donnée entre deux instances d'instruction y accédant, l'un au moins des accès étant en écriture. Cette dépendance de donnée induit un ordre partiel sur les instances d'instructions qui doit être préservé par toute exécution parallèle pour qu'aucun conflit mémoire n'apparaisse. C'est ce que nous traduisons par la phrase :

> Si deux instances d'instructions a et b de vecteur d'itération $\boldsymbol{i}$ et $\boldsymbol{j}$ respectivement sont en dépendance $\mathrm{Dep}(\langle a, \boldsymbol{i}\rangle, \langle b, \boldsymbol{j}\rangle)$, ce qui implique que $\langle a, \boldsymbol{i}\rangle$ est exécutée avant $\langle b, \boldsymbol{j}\rangle$ dans la version séquentielle, et si ces deux instances sont exécutées, alors $\langle a, \boldsymbol{i}\rangle$ doit être exécutée avant $\langle b, \boldsymbol{j}\rangle$ dans le programme parallèle.

Plus formellement, en utilisant les prédicats que nous avons introduits, pour toutes instances d'instructions α et β et pour toute exécution parallèle, on doit avoir

$$\mathrm{Exe}^{\mathsf{s}}(\alpha) \wedge \mathrm{Exe}^{\mathsf{s}}(\beta) \wedge \mathrm{Dep}(\alpha, \beta) \Rightarrow \mathrm{Pre}(\alpha, \beta)$$

4.5.2 Approximations des prédicats

Dans la pratique, il est souvent impossible d'obtenir statiquement, l'expression exacte des prédicats Dep, $\mathrm{Exe}^{\mathsf{s}}$ et Pre. C'est pourquoi nous nous intéressons à des approximations conservatives de ces prédicats. Elles sont conservatives dans le sens où elles ne conduisent pas à une réponse positive quand la préservation des dépendances n'est pas vérifiée. Mais elles conduisent à la réponse « ne sait pas » ou « faux » même s'il y a préservation.

Nous considérons des approximations $\mathrm{Dep}^{\star}$ et $\mathrm{Pre}_{\star}$ telles que $\mathrm{Dep} \Rightarrow \mathrm{Dep}^{\star}$ et $\mathrm{Pre}_{\star} \Rightarrow \mathrm{Pre}$ qui signifient respectivement : « une dépendance peut exister » et « une précédence doit exister ».

Le prédicat Pre fait intervenir les prédicats Pre^0, Sync et $\mathrm{Exe}^{\mathsf{s}}$. Pre^0 est assez facilement exprimable ; Sync aussi, compte tenu des restrictions faites sur les expressions d'indices de vecteurs. Pour approximer Pre, nous pouvons prendre un prédicat $\mathrm{Exe}^{\mathsf{s}}_{\star}$ tel que $\mathrm{Exe}^{\mathsf{s}}_{\star} \Rightarrow \mathrm{Exe}^{\mathsf{s}}$ ou, considérer seulement certains chemins.

Pour accélérer le calcul de la relation de dépendance, nous avons utilisé une définition conservative de la dépendance (donc un $\mathrm{Dep}^{\star}$) qui omet la clause suivante :

> Si (a, b) est un couple d'instructions en conflit sur une référence r alors il n'existe pas d'instruction c modifiant r et exécutée entre a et b.

La prise en compte de cette clause dans la détermination des dépendances « exactes » est un souci des concepteurs d'outils de parallélisation et fait l'objet de nombreuses recherches afin d'en réduire le surcoût.

4.5.3 Théorème d'équivalence sémantique

L'équivalence sémantique entre le programme parallèle et sa version séquentielle est démontrée dans [7]. Nous donnons ci-dessous l'énoncé du théorème :

Théorème 1 *Si*

i. les hypothèses S1, S2, S1', S2', S3', sont vérifiées ;

ii. toute exécution mono-processus termine ;

iii. pour toutes instances d'instruction α *et* β, $\mathrm{Exe}^{\mathsf{s}}(\alpha) \wedge \mathrm{Exe}^{\mathsf{s}}(\beta) \wedge \mathrm{Dep}(\alpha, \beta) \Rightarrow \mathrm{Pre}(\alpha, \beta)$;

alors le prédicat d'exécution est bien défini et est précisément $\mathrm{Exe}^{\mathsf{s}}$*, et le programme parallèle est sémantiquement équivalent à sa version séquentielle.*

Chapitre 5

Un algorithme de vérification

Le but de cet algorithme est de vérifier la condition *iii* du théorème 1. Il suppose que les prédicats intervenant dans la condition sont formulables statiquement, ce qui n'est en général pas possible pour $\text{Exe}^{\mathbf{s}}$.

On s'intéresse aux couples (a, b) potentiellement conflictuels, c'est-à-dire ceux pour lesquels $\text{Dep}_{a,b}$ est susceptible d'être satisfaisable.

5.1 Principe de l'algorithme

Pour chaque couple conflictuel (r, s) provenant d'un couple (a, b) d'instructions, les étapes suivantes sont réalisées :

- Si $\mathfrak{N}(a, b)$ ne contient aucun `pdo` et si $\mathfrak{C}(a, b) \neq ||$ alors a et b ne peuvent pas être exécutées concurremment donc la dépendance, si elle existe, ne peut être qu'une dépendance préservée (Cf. 4.5.1) et n'a pas à être considérée.

- Si $\mathfrak{C}(a, b) = ||$ et si $\mathfrak{N}(a, b) = [\,]$ alors nécessairement $\text{Pre}^0_{a,b} = \mathit{false}$. Si la formule de dépendance qui n'est alors constituée que d'un seul terme conjonctif, est satisfaisable, il faut analyser les synchronisations.

- Sinon, la formule de dépendance est une disjonction $\text{Dep} = \bigvee_k \text{Dep}_k$ à cause de la formule $(a \prec b)(\boldsymbol{x}, \boldsymbol{y})$, la satisfaisabilité de chaque Dep_k doit être testée pour k allant de 1 à $|\,\mathfrak{N}(a, b)|$:

  - si Dep_k est satisfaisable, on regarde le support (défini en 3.2.3) de la boucle à cette profondeur k : s'il est séquentiel (boucle `do`) alors la dépendance est préservée, sinon il est parallèle (boucle `pdo`) et la dépendance peut ne pas être préservée ; le calcul de la précédence $\text{Pre}_{a,b}$ est nécessaire : si la formule de précédence de contrôle $\text{Pre}^0_{a,b}$ vaut *true*, la dépendance est préservée au rang k ; si $\text{Pre}^0_{a,b}$ vaut *false*, les synchronisations doivent être analysées. Pour cela, les

chemins de précédence et les formules associées faisant intervenir les synchronisations doivent être construits.

- sinon c'est que Dep n'a pas de solution de rang k.

Dans la suite, nous montrons comment calculer effectivement une approximation conservative de la précédence généralisée, définie au chapitre précédent, en développant des chemins finis dans des graphes que nous définissons.

5.2 Calcul des précédences

5.2.1 Graphe de précédence

Le graphe de flot que nous avons utilisé pour définir le prédicat de précédence Pre était un graphe d'instances. Tout programme comportant une boucle `DO` ou `PARALLEL DO` engendre un graphe de flot de taille infinie. Le graphe que nous construisons pour le calcul effectif des précédences a pour sommets des éléments pris dans la syntaxe abstraite du langage ; comme le graphe de dépendance, c'est un graphe fini étiqueté par des formules logiques. On note $\mathcal{PG}$ ce graphe de précédence.

Nous construisons $\mathcal{PG}$ par projection, à partir du graphe de flot $\mathcal{FG}$ défini au chapitre 4. Soit p la projection qui à un sommet α du graphe de flot associe l'élément syntaxique (instruction, condition booléenne, en-tête de boucle `do`) $p(\alpha)$ dont α est une instance, autrement dit : $p(\langle a, \boldsymbol{i} \rangle) = a$.

Définition 15 *L'ensemble des sommets de $\mathcal{PG}$ est l'ensemble formé par $p(\alpha)$ quand α parcourt l'ensemble des sommets du graphe de flot. L'ensemble des arcs de $\mathcal{PG}$ comprend des arcs de contrôle et des arcs de synchronisation :*

- *$u : a \to b$ est un arc de contrôle de $\mathcal{PG}$ s'il existe un arc de contrôle dans le graphe de flot entre une instance de a et une instance de b.*

- *$u : a \to b$ est un arc de synchronisation de $\mathcal{PG}$ s'il existe un arc de synchronisation dans le graphe de flot entre une instance de a et une instance de b.*

Remarque Deux sommets de $\mathcal{PG}$ peuvent être reliés à la fois par un arc de contrôle et un arc de synchronisation.

En outre, nous associons à chaque arc u de $\mathcal{PG}$ une formule logique P_u dont l'expression dépend de la nature de l'arc : arc de contrôle ou arc de synchronisation.

Il existe deux types d'arcs de contrôle, les arcs « avant » qui vont d'un sommet a vers un sommet b situé après a dans le texte du programme et les

arcs «arrière» qui vont d'un sommet a à un en-tête de boucle `do` situé avant lui dans le texte du programme.

Définition 16 *Soit u un arc allant d'un sommet a à un sommet b de $\mathcal{PG}$, $\boldsymbol{x}$ et $\boldsymbol{y}$ des n-uplets de variables de longueur $|\mathfrak{N}(a)|$ et $|\mathfrak{N}(b)|$ respectivement, la formule P_u est définie selon les cas par :*

- *si u est un arc de contrôle avant alors $P_u(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{|\mathfrak{N}(a,b)} = \boldsymbol{y}_{|\mathfrak{N}(a,b)})$.*
- *si u est un arc de contrôle arrière alors $P_u(\boldsymbol{x},\boldsymbol{y}) = ((\boldsymbol{x'} = \boldsymbol{y'}) \wedge (x+1 = y))$ où $\boldsymbol{x'}@[x] = \boldsymbol{x}_{|\mathfrak{N}(a,b)}$ et $\boldsymbol{y'}@[y] = \boldsymbol{y}_{|\mathfrak{N}(a,b)}$.*
- *si u est un arc de synchronisation, soit Sync_u la formule de synchronisation qui lui est associée (Cf. paragraphe 4.2.2), alors $P_u(\boldsymbol{x},\boldsymbol{y}) = \mathrm{Sync}_u(\boldsymbol{x},\boldsymbol{y})$.*

On remarquera que $\mathcal{PG}$, tout comme $\mathcal{FG}$ d'ailleurs, ne porte pas de conditions d'exécutabilité.

Le graphe de flot $\mathcal{FG}$ nous avait permis de définir la relation de précédence entre deux instances. Malheureusement ce graphe est potentiellement infini comme on l'a vu. On voudrait donc pouvoir prouver l'existence de relations de précédence entre instances d'instruction à l'aide du graphe $\mathcal{PG}$.

L'équivalence des deux graphes est énoncée par les deux propositions ci-dessous :

Proposition 2 *Pour tout arc $u : \langle a,\boldsymbol{i}\rangle \to \langle b,\boldsymbol{j}\rangle$ dans $\mathcal{FG}$, il existe un arc $v : a \to b$ dans $\mathcal{PG}$ tel que $P_v(\boldsymbol{i},\boldsymbol{j})$ est vrai.*

Proposition 3 *Pour tout arc $v : a \to b$ dans $\mathcal{PG}$ et pour tous n-uplets $\boldsymbol{i},\boldsymbol{j}$ tels que $P_v(\boldsymbol{i},\boldsymbol{j})$, il existe un arc $u : \langle a,\boldsymbol{i}\rangle \to \langle b,\boldsymbol{j}\rangle$ dans $\mathcal{FG}$.*

Démonstration de la proposition 2 Considérons un arc $u : \langle a,\boldsymbol{i}\rangle \to \langle b,\boldsymbol{j}\rangle$ de $\mathcal{FG}$, l'existence d'un arc $v : a \to b$ dans $\mathcal{PG}$ découle trivialement de la définition du graphe $\mathcal{PG}$. Nous devons vérifier que $P_v(\boldsymbol{i},\boldsymbol{j})$ est vrai. On distingue les trois cas suivants :

- Si u est un arc de synchronisation, $P_v(\boldsymbol{i},\boldsymbol{j}) = \mathrm{Sync}(\boldsymbol{i},\boldsymbol{j})$ est vrai par définition de Sync.
- Si u est un arc de contrôle entre a une instruction simple et b un `do.head`, on a $u : \langle a,\boldsymbol{i'}@[i]@\boldsymbol{k}\rangle \to \langle b,\boldsymbol{i'}@[i+1]\rangle$. Comme $P_v(\boldsymbol{x},\boldsymbol{y}) = ((\boldsymbol{x'} = \boldsymbol{y'}) \wedge (x+1 = y))$ où $\boldsymbol{x'}@[x] = \boldsymbol{x}_{|\mathfrak{N}(a,b)}$ et $\boldsymbol{y'}@[y] = \boldsymbol{y}_{|\mathfrak{N}(a,b)}$, on a donc $P_v(\boldsymbol{i'}@[i]@\boldsymbol{k},\boldsymbol{i'}@[i+1])$ vrai car $\mathfrak{N}(b) = \mathfrak{N}(a,b)$.
- Autrement, u est un arc de contrôle ordinaire et $P_v(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{|\mathfrak{N}(a,b)} = \boldsymbol{y}_{|\mathfrak{N}(a,b)})$. On vérifie que $P_v(\boldsymbol{i},\boldsymbol{j})$ est vrai quelque soit l'allure de la relation entre $\boldsymbol{i}$ et $\boldsymbol{j}$:
  - si $\boldsymbol{i} = \boldsymbol{j}$ alors $P_v(\boldsymbol{i},\boldsymbol{i})$ est vrai.

- si $\boldsymbol{i} = \boldsymbol{j}@\boldsymbol{k}$ pour tout $\boldsymbol{k}$ alors $P_v(\boldsymbol{j}@\boldsymbol{k}, \boldsymbol{j})$ est vrai car $\mathfrak{N}(b) = \mathfrak{N}(a,b)$.
- si $\boldsymbol{j} = \boldsymbol{i}@[k]$ pour tout k alors $P_v(\boldsymbol{i}, \boldsymbol{i}@[k])$ est vrai car $\mathfrak{N}(a) = \mathfrak{N}(a,b)$. □

Démonstration de la proposition 3 Soit $u : a \to b$ un arc de $\mathcal{PG}$; on cherche à montrer qu'étant donnés deux n-uplets $\boldsymbol{i}, \boldsymbol{j}$ qui vérifient $P_u(\boldsymbol{i}, \boldsymbol{j})$, il existe un arc $v : \langle a, \boldsymbol{i}\rangle \to \langle b, \boldsymbol{j}\rangle$ dans $\mathcal{FG}$. On distingue trois cas selon la nature de u :

- u un arc de synchronisation, soit $\boldsymbol{i}, \boldsymbol{j}$ deux n-uplets qui vérifient $\mathrm{Sync}(\boldsymbol{i}, \boldsymbol{j})$. Par conséquent il existe un arc de synchronisation de $\langle a, \boldsymbol{i}\rangle$ vers $\langle b, \boldsymbol{j}\rangle$ dans $\mathcal{FG}$.
- u est un arc de contrôle arrière de $\mathcal{PG}$, soit $\boldsymbol{i}, \boldsymbol{j}$ deux n-uplets qui vérifient $(\boldsymbol{i'} = \boldsymbol{j'}) \wedge (i + 1 = j)$ où $\boldsymbol{i'}@[i] = \boldsymbol{i}_{|\,\mathfrak{N}(a,b)}$ et $\boldsymbol{j'}@[j] = \boldsymbol{j}_{|\,\mathfrak{N}(a,b)}$. a est une instruction simple, $b = c.\texttt{head}$ où c est un `do` ancêtre de a, $\mathfrak{N}(b) = \mathfrak{N}(a,b)$, donc $\boldsymbol{j} = \boldsymbol{i'}@[i+1]$ et par suite il existe un arc de $\langle a, \boldsymbol{i}\rangle$ vers $\langle b, \boldsymbol{j}\rangle$ dans $\mathcal{FG}$.
- u est un arc de contrôle avant de $\mathcal{PG}$, soit $\boldsymbol{i}, \boldsymbol{j}$ deux n-uplets qui vérifient $\boldsymbol{i}_{|\,\mathfrak{N}(a,b)} = \boldsymbol{j}_{|\,\mathfrak{N}(a,b)}$, on a plusieurs cas suivant la nature syntaxique de a :
  - si a est une instruction simple, on a trois cas :
    - a possède un successeur, $\mathfrak{N}(a) = \mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{i} = \boldsymbol{j}$.
    - a n'a pas de successeur mais est le fils d'une instruction structurée qui en a un, $\mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{j} = \boldsymbol{i}_{|\,\mathfrak{N}(a,b)}$.
    - a n'a pas de successeur mais est le fils d'une `section` attendue par une autre et telle qu'il n'existe pas d'instruction structurée ayant un successeur et située entre elle et a dans l'arbre de syntaxe abstraite, $\mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{j} = \boldsymbol{i}_{|\,\mathfrak{N}(a,b)}$.
  - a est un `if` : $\mathfrak{N}(a) = \mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{i} = \boldsymbol{j}$.
  - a est un `if.bexp` : $\mathfrak{N}(a) = \mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{i} = \boldsymbol{j}$.
  - a est un `do` : $\mathfrak{N}(b) = \mathfrak{N}(a)@[\texttt{do}, \_, \_, \_]$ donc $\boldsymbol{i} = \boldsymbol{j}_{|\,\mathfrak{N}(a,b)}$.
  - a est un `do.head` : $\mathfrak{N}(a) = \mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{i} = \boldsymbol{j}$.
  - a est un `pdo` : $\mathfrak{N}(b) = \mathfrak{N}(a)@[\texttt{pdo}, \_, \_, \_]$ donc $\boldsymbol{i} = \boldsymbol{j}_{|\,\mathfrak{N}(a,b)}$.
  - a est un `psections` : $\mathfrak{N}(a) = \mathfrak{N}(b) = \mathfrak{N}(a,b)$ donc $\boldsymbol{i} = \boldsymbol{j}$. □

Ces deux propositions établissent l'équivalence des deux graphes.

Pour montrer qu'une précédence existe entre deux instances d'instruction et sous réserve que ces instances soient exécutées, nous sommes donc ramenés

à la recherche d'un chemin dans un graphe fini $\mathcal{PG}$ et à la preuve de la formule associée à ce chemin.

Soit $P : a \longrightarrow b$ un chemin dans le graphe $\mathcal{PG}$ allant de a à b et passant par les sommets intermédiaires $s_1, s_2, \ldots, s_n$; on va associer à P une formule $F_P(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ où $\boldsymbol{x}$ (resp. $\boldsymbol{y}$) est un n-uplet de variables de longueur $|\mathfrak{N}(a)|$ (resp. $|\mathfrak{N}(b)|$) associées à a (resp. b) et les $\boldsymbol{z}_i$ des n-uplets de variables de longueur $|\mathfrak{N}(s_i)|$ associés aux s_i. Si on note $u_i : s_i \to s_{i+1}$ pour $1 \leqslant i \leqslant n-1$, $u_0 : a \to s_1$ et $u_n : s_n \to b$ alors P peut s'écrire comme la concaténation des $n+1$ arcs u_i : $P = u_0 \cdot u_1 \cdot \ldots \cdot u_n$.

Hypothèse de staticité Au chapitre 4, nous avions défini $\mathrm{Exe}^{\mathsf{s}}$ sur les instances d'instructions ; son expression faisait intervenir l'environnement et l'état mémoire. Nous supposons maintenant que $\mathrm{Exe}^{\mathsf{s}}$ ne dépend pas de variables autres que les indices de boucles appartenant à $\mathrm{Iter}(a)$ et pour a et $\boldsymbol{x}$ quelconques nous notons cette formule $\mathrm{Exe}^{\mathsf{s}}_a(\boldsymbol{x})$. Elle est définie par

$$\mathrm{Exe}^{\mathsf{s}}_a(\boldsymbol{x}) = \mathrm{Exe}^{\mathsf{s}}(\langle a, \boldsymbol{x}\rangle)$$

Définition 17 *F_P est la conjonction des formules*

- *P_{u_i} si u_i est un arc de contrôle*
- *$\mathrm{Exe}^{\mathsf{s}}_{s_i} \wedge P_{u_i} \wedge \mathrm{Exe}^{\mathsf{s}}_{s_{i+1}}$ si u_i est un arc de synchronisation*

Proposition 4 *Pour un chemin P, on a l'implication :*

$$\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n (F_P(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n) \Rightarrow \mathrm{Pre}_{a,b}(\boldsymbol{x}, \boldsymbol{y}))$$

Remarquons que cette formule peut encore s'écrire :

$$\forall \boldsymbol{x}, \boldsymbol{y} ((\exists \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n F_P(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)) \Rightarrow \mathrm{Pre}_{a,b}(\boldsymbol{x}, \boldsymbol{y}))$$

puisque Pre ne dépend pas de $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$.

Pour démontrer cette proposition, nous avons besoin de démontrer un résultat préliminaire qui permet de relier Pre^0 à la formule F_P associée à un chemin de contrôle.

Lemme 1 *Si $P : a \longrightarrow b$ est un chemin dans $\mathcal{PG}$ composé uniquement d'arcs de contrôle alors*

$$F_P \Rightarrow \mathrm{Pre}^0_{a,b}$$

Démonstration Nous démontrons cette propriété par récurrence sur la longueur de P. Pour chaque cas, nous faisons appel à un lemme intermédiaire énoncé et démontré plus loin.

Si P n'est composé que d'un seul arc : $P = u$, alors on applique le lemme 2 à u ce qui prouve la propriété pour $|P| = 1$.

Supposons que la proposition soit vraie pour un chemin $P : a \longrightarrow b$ de longueur n, on a donc $F_{P:a\longrightarrow b} \Rightarrow \mathrm{Pre}^0_{a,b}$, soit $u : b \to c$ un arc de contrôle, par le lemme 3 on a $\mathrm{Pre}^0_{a,b} \wedge P_u \Rightarrow \mathrm{Pre}^0_{a,c}$ et par conséquent $F_{P:a\longrightarrow b} \wedge P_u \Rightarrow \mathrm{Pre}^0_{a,c}$. Conclusion : pour tout chemin de contrôle $P : a \longrightarrow c$, on a : $F_{P:a\longrightarrow c} \Rightarrow \mathrm{Pre}^0_{a,c}$. □

Remarquons que nous n'avons pas l'équivalence des deux formules. Considérons l'exemple suivant :

```
DO I = 1,N
  b
  a
END DO
```

et soit u l'arc en retour allant de a vers b, x (resp. y) une variable d'instance associée à a (resp. b), on a :

$$\begin{aligned} P_u(x,y) &= (x+1=y) \\ \mathrm{Pre}^0_{a,b}(x,y) &= x<y \end{aligned}$$

Il est clair sur cet exemple que $\mathrm{Pre}^0_{a,b} \not\Rightarrow P_u$ donc $\mathrm{Pre}^0_{a,b} \Rightarrow F_{P:a\longrightarrow b}$ est faux en général.

Lemme 2 *Pour tout arc de contrôle $u : a \to b$ de $\mathcal{PG}$, on a : $P_u \Rightarrow \mathrm{Pre}^0_{a,b}$*

Démonstration $u : a \to b$ appartient à $\mathcal{PG}$ donc $\mathfrak{C}(a,b) =$ ';' et par conséquent $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = ((\boldsymbol{x}_{\mathfrak{N}(a,b)} = \boldsymbol{y}_{\mathfrak{N}(a,b)}) \wedge (a\,T\,b)) \vee (\boldsymbol{x}_{\mathfrak{N}(a,b)} \ll_{\mathfrak{N}(a,b)} \boldsymbol{y}_{\mathfrak{N}(a,b)})$.

Si u est un arc de contrôle avant alors $P_u(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{|\,\mathfrak{N}(a,b)} = \boldsymbol{y}_{|\,\mathfrak{N}(a,b)})$ et la première partie de la disjonction dans $\mathrm{Pre}^0_{a,b}$ est vérifiée.

Si u est un arc de contrôle arrière alors $P_u(\boldsymbol{x},\boldsymbol{y}) = ((\boldsymbol{x'} = \boldsymbol{y'}) \wedge (x+1=y))$ où $\boldsymbol{x'}@[x] = \boldsymbol{x}_{|\,\mathfrak{N}(a,b)}$ et $\boldsymbol{y'}@[y] = \boldsymbol{y}_{|\,\mathfrak{N}(a,b)}$ et c'est la seconde partie de la disjonction dans $\mathrm{Pre}^0_{a,b}$ qui est vérifiée. □

Lemme 3 *Pour tout couple d'instructions a, b sommets de $\mathcal{PG}$ et pour tout arc de contrôle $u : b \to c$ de $\mathcal{PG}$, on a :*

$$\mathrm{Pre}^0_{a,b} \wedge P_u \Rightarrow \mathrm{Pre}^0_{a,c}$$

Démonstration On a a priori trois cas possibles selon la position du nid de boucles de c par rapport au nid de boucles commun à a et à b (voir figure 5.1). Pour chacun de ces cas, on doit à nouveau distinguer quatre cas car Pre^0 et P_u peuvent prendre chacun deux formes : P_u peut être associé soit à un arc avant soit à un arc arrière et $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x} = \boldsymbol{y}) \wedge (a\,T\,b)$

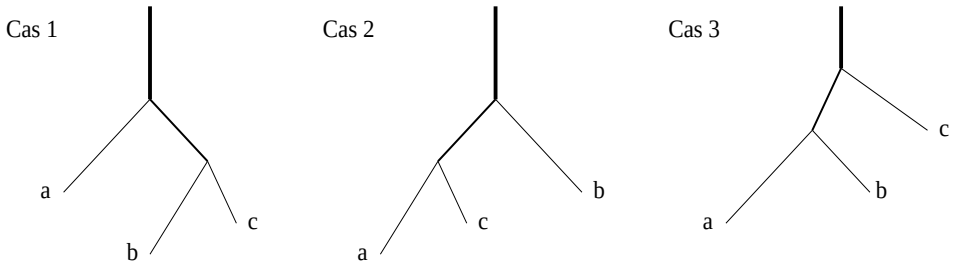

FIG. 5.1 – *Positions relatives de $\mathfrak{N}(c)$ par rapport à $\mathfrak{N}(a,b)$ représentées sur l'arbre des* `do`,`pdo` *(chacun des nœuds binaires représente un nid de boucles commun)*

ou $\mathrm{Pre}^0_{a,b} = \boldsymbol{x} \ll_{\mathfrak{N}(a,b)} \boldsymbol{y}$. Remarquons que si $u : a \to b$ est un arc arrière $P_u(\boldsymbol{x},\boldsymbol{y}) \Rightarrow \boldsymbol{x} \ll_{\mathfrak{N}(a,b)} \boldsymbol{y}$.

1. $\mathfrak{N}(a,c) \subseteq \mathfrak{N}(b,c)$ d'où $\mathfrak{N}(a,c) = \mathfrak{N}(a,b)$. Pour faciliter l'exposé de la démonstration, on décompose les variables d'itérations de la façon suivante :

$$\begin{cases} \boldsymbol{x} = \boldsymbol{x}_{a,c}@\boldsymbol{x'} \\ \boldsymbol{y} = \boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta@\boldsymbol{y'} \\ \boldsymbol{z} = \boldsymbol{z}_{a,c}@\boldsymbol{z}_\Delta@\boldsymbol{z'} \end{cases}$$

   en notant $\boldsymbol{t}_{a,c} = \boldsymbol{t}_{|\,\mathfrak{N}(a,c)}$ et $\boldsymbol{t}_\Delta = \boldsymbol{t}_{|\,\mathfrak{N}(b,c)-\mathfrak{N}(a,c)}$. On remarque que $\boldsymbol{t}_{a,c} = \boldsymbol{t}_{a,b}$ et que $\boldsymbol{t}_{b,c} = \boldsymbol{t}_{a,b}@\boldsymbol{t}_\Delta$.

   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c} = \boldsymbol{y}_{a,c}) \wedge (a\,T\,b)$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta = \boldsymbol{z}_{a,c}@\boldsymbol{z}_\Delta)$ (arc avant) on en déduit par transitivité que $\boldsymbol{x}_{a,c} = \boldsymbol{z}_{a,c}$. De $a\,T\,b$ et $b\,T\,c$, car u est un arc avant, on déduit $a\,T\,c$ ce qui permet de conclure que $\mathrm{Pre}^0_{a,c}$ est vrai.
   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c} = \boldsymbol{y}_{a,c}) \wedge (a\,T\,b)$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta \ll \boldsymbol{z}_{a,c}@\boldsymbol{z}_\Delta)$ (arc arrière). On a deux cas : si $\boldsymbol{y}_{a,c} \ll \boldsymbol{z}_{a,c}$ alors $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$. Si $\boldsymbol{y}_{a,c} = \boldsymbol{z}_{a,c}$ alors $\boldsymbol{x}_{a,c} = \boldsymbol{z}_{a,c}$, de plus $\boldsymbol{y}_\Delta \neq []$ donc $\mathfrak{N}(a,c) \subset \mathfrak{N}(b,c)$ (inclusion stricte) et par suite $a\,T\,c$. Dans les deux cas, on en déduit que $\mathrm{Pre}^0_{a,c}$ est vrai.
   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c} \ll \boldsymbol{y}_{a,c})$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta = \boldsymbol{z}_{a,c}@\boldsymbol{z}_\Delta)$ (arc avant) on en déduit que $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$ donc $\mathrm{Pre}^0_{a,c}$.
   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c} \ll \boldsymbol{y}_{a,c})$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta \ll \boldsymbol{z}_{a,c}@\boldsymbol{z}_\Delta)$ (arc arrière) on a deux cas : si $\boldsymbol{y}_{a,c} \ll \boldsymbol{z}_{a,c}$, on en déduit par transitivité de $\ll$ que $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$. Si $\boldsymbol{y}_{a,c} = \boldsymbol{z}_{a,c}$, on en déduit également $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$. Donc on a $\mathrm{Pre}^0_{a,c}$ dans les deux cas.

2. $\mathfrak{N}(a,b) \subset \mathfrak{N}(a,c)$ d'où $\mathfrak{N}(b,c) = \mathfrak{N}(a,b)$. Ce cas viole l'hypothèse selon laquelle $u : b \to c$ est un arc de contrôle de $\mathcal{PG}$ car

   - si u est un arc avant, on a nécessairement $b\,T\,c$ ce qui est faux puisque l'hypothèse implique le contraire.
   - si u est un arc arrière alors il existe une instruction `do` d telle que $c = d$.`head` ancêtre de b ce qui entraîne que $\mathfrak{N}(a,c) \subseteq \mathfrak{N}(b,c)$ en contradiction avec l'hypothèse.

3. $\mathfrak{N}(a,c) \subset \mathfrak{N}(a,b)$ d'où $\mathfrak{N}(a,c) = \mathfrak{N}(b,c)$. On peut décomposer les variables d'itérations comme suit :

$$\begin{cases} \boldsymbol{x} = \boldsymbol{x}_{a,c}@\boldsymbol{x}_\Delta@\boldsymbol{x'} \\ \boldsymbol{y} = \boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta@\boldsymbol{y'} \\ \boldsymbol{z} = \boldsymbol{z}_{a,c}@\boldsymbol{z'} \end{cases}$$

   en notant $\boldsymbol{t}_{a,c} = \boldsymbol{t}_{|\,\mathfrak{N}(a,c)}$ et $\boldsymbol{t}_\Delta = \boldsymbol{t}_{|\,\mathfrak{N}(a,b)-\mathfrak{N}(a,c)}$. On remarque que $\boldsymbol{t}_{a,c} = \boldsymbol{t}_{b,c}$ et que $\boldsymbol{t}_{a,b} = \boldsymbol{t}_{a,c}@\boldsymbol{t}_\Delta$.

   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c}@\boldsymbol{x}_\Delta = \boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta) \wedge (a\,T\,b)$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c} = \boldsymbol{z}_{a,c})$ (arc avant) alors par transitivité on a $\boldsymbol{x}_{a,c} = \boldsymbol{z}_{a,c}$. u étant un arc avant, il en découle que $b\,T\,c$ est vrai et comme $a\,T\,b$ est vrai par hypothèse, on a aussi $a\,T\,c$ donc $\mathrm{Pre}^0_{a,c}$.
   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c}@\boldsymbol{x}_\Delta = \boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta) \wedge (a\,T\,b)$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c} \ll \boldsymbol{z}_{a,c})$ (arc arrière) alors par transitivité on a $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$ et donc $\mathrm{Pre}^0_{a,c}$.
   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c}@\boldsymbol{x}_\Delta \ll \boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta) \wedge (a\,T\,b)$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c} = \boldsymbol{z}_{a,c})$ (arc avant) on a deux cas. Si $\boldsymbol{x}_{a,c} = \boldsymbol{y}_{a,c}$ alors $\boldsymbol{x}_{a,c} = \boldsymbol{z}_{a,c}$. Pour prouver $a\,T\,c$, considérons d l'instruction `do` plus proche ancêtre ancêtre commun à a et à b mais non à c, si on avait $c\,T\,a$ alors on aurait aussi $c\,T\,d$ et par suite $c\,T\,b$ ce qui contredit le fait que u est un arc avant. Si $\boldsymbol{x}_{a,c} \ll \boldsymbol{y}_{a,c}$ alors $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$. Donc on a $\mathrm{Pre}^0_{a,c}$ dans les deux cas.
   - Si $\mathrm{Pre}^0_{a,b}(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}_{a,c}@\boldsymbol{x}_\Delta \ll \boldsymbol{y}_{a,c}@\boldsymbol{y}_\Delta) \wedge (a\,T\,b)$ et $P_u(\boldsymbol{y},\boldsymbol{z}) = (\boldsymbol{y}_{a,c} \ll \boldsymbol{z}_{a,c})$ (arc arrière) alors on a deux cas. Si $\boldsymbol{y}_{a,c} \ll \boldsymbol{z}_{a,c}$, on en déduit par transitivité de $\ll$ que $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$. Si $\boldsymbol{y}_{a,c} = \boldsymbol{z}_{a,c}$, on en déduit également $\boldsymbol{x}_{a,c} \ll \boldsymbol{z}_{a,c}$. Donc on a $\mathrm{Pre}^0_{a,c}$ dans les deux cas. □

Démonstration de la proposition 4 On considère a, b deux sommets de $\mathcal{PG}$, $\boldsymbol{i}, \boldsymbol{j}$ deux n-uplets d'entiers de longueur $|\,\mathfrak{N}(a)|$ et $|\,\mathfrak{N}(b)|$ respectivement et $P : a \longrightarrow b$ un chemin passant par n sommets intermédiaires s_i. Supposons qu'il existe des n-uplets d'entiers $\boldsymbol{k}_i$ tels que $F_P(\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}_1,\ldots,\boldsymbol{k}_i,\ldots,\boldsymbol{k}_n)$

soit vrai. Le chemin P est de la forme : $a = s_0 \to s_1 \to \ldots s_i \to s_{i+1} \to \ldots s_{n+1} = b$; par la proposition 3, on peut dire qu'il existe un arc $v_i : \langle s_i, k_i\rangle \to \langle s_{i+1}, k_{i+1}\rangle$ dans $\mathcal{FG}$ pour tout $i \in [0, n]$ (en notant $\boldsymbol{k}_0 = \boldsymbol{i}$ et $\boldsymbol{k}_{n+1} = \boldsymbol{j}$) et par suite, il existe un chemin Q, composition des v_i, allant de $\langle a, \boldsymbol{i}\rangle$ à $\langle b, \boldsymbol{j}\rangle$ dans $\mathcal{FG}$. Il faut montrer que $\mathrm{Pre}(Q)$ est vrai.

Si $v_i : \langle s_i, k_i\rangle \to \langle s_{i+1}, k_{i+1}\rangle$ est un arc de synchronisation, $\mathrm{Exe}^{\mathsf{s}}(\langle s_i, k_i\rangle) \wedge \mathrm{Sync}(\langle s_i, k_i\rangle, \langle s_{i+1}, k_{i+1}\rangle) \wedge \mathrm{Exe}^{\mathsf{s}}(\langle s_{i+1}, k_{i+1}\rangle)$ est vrai puisqu'une formule identique, $\mathrm{Exe}^{\mathsf{s}}_{s_i}(k_i) \wedge \mathrm{Sync}_{s_i,s_{i+1}}(k_i, k_{i+1}) \wedge \mathrm{Exe}^{\mathsf{s}}_{s_{i+1}}(k_{i+1})$ associée à l'arc $s_i \to s_{i+1}$, est un terme de la conjonction F_P qui est vraie.

Si $v_i \cdots v_{i+l} : \langle s_i, k_i\rangle \longrightarrow \langle s_{i+l+1}, k_{i+l+1}\rangle$ pour $l \geqslant 0$ est un chemin composé d'arcs de contrôle uniquement, le lemme 1 énoncé ci-dessus prouve que $\mathrm{Pre}^0(\langle s_i, k_i\rangle, \langle s_{i+l+1}, k_{i+l+1}\rangle)$ est vrai puisque la formule $F_{P:s_i \longrightarrow s_{i+l+1}}$ (associée au chemin de contrôle $s_i \longrightarrow s_{i+l+1}$ de $\mathcal{PG}$) est vraie en tant qu'élément de la conjonction F_P qui est vraie.

Donc $\mathrm{Pre}(Q)$ est vrai, et par suite $\mathrm{Pre}_{a,b}(\boldsymbol{i}, \boldsymbol{j})$ aussi car il est défini comme une disjonction dont $\mathrm{Pre}(Q)$ est un terme (Cf. définition 12 page 49). □

5.2.2 Non localité des synchronisations

Les effets des synchronisations peuvent « traverser » les constructions parallèles comme le montre l'exemple suivant :

```
PARALLEL SECTIONS
 SECTION
  POST(E(1))
  PARALLEL DO I=2,N
   a
   POST(E(I))
   WAIT(F(I-1))
   b
  END PARALLEL DO
 SECTION
  DO I = 1,N
   WAIT(E(I))
   POST(F(I))
  END DO
END PARALLEL SECTIONS
```

La précédence entre $\langle a, i\rangle$ et $\langle b, i{+}1\rangle$, pour $i \geqslant 2$ provient des synchronisations, non du flot de contrôle à l'intérieur du `PARALLEL DO` ; la chaîne de précédence est $\langle a, i\rangle \to \mathtt{POST(E}(i)\mathtt{)} \rightsquigarrow \mathtt{WAIT(E}(i)\mathtt{)} \to \mathtt{POST(F}(i)\mathtt{)} \rightsquigarrow \mathtt{WAIT(F}(i)\mathtt{)} \to \langle b, i+1\rangle$.

En conséquence, pour calculer les précédences à l'intérieur d'une construction parallèle, les synchronisations nous obligent à considérer le programme

dans son entier. Pour diminuer cet inconvénient, nous allons isoler des blocs d'instructions de taille maximale à l'intérieur desquels la précédence est calculable localement.

5.2.3 Le graphe de blocs

Pour faciliter le calcul des précédences, un graphe de blocs $\mathcal{BG}$ est construit. C'est un graphe dont les sommets sont des séquences d'instructions quelconques, simples ou structurées, qui généralisent les blocs de base décrits dans [1] qui eux n'admettent que des instructions simples. L'ensemble des arcs représente la relation de précédence généralisée (contrôle et synchronisation) définies sur les instructions et étendue aux blocs. On trouve une structure semblable dans [6].

Définition 18 *Chaque sommet du graphe $\mathcal{BG}$ appartient à l'un des six types suivants : bloc conditionnel, blocs de synchronisation, bloc ordinaire, bloc « en-tête de* `DO` *», bloc « en-tête de* `PARALLEL SECTIONS` *» et bloc vide.*

- *un bloc conditionnel contient uniquement l'expression booléenne d'une instruction* `IF`*.*
- *un bloc de synchronisation est*
  - *soit une séquence d'instructions commençant par un* `WAIT` *ou finissant par un* `POST`*, mais on interdit d'avoir les deux à la fois ; les autres instructions ne contiennent pas d'instructions de synchronisation,*
  - *soit une instruction* `POST` *seule.*
- *un bloc ordinaire est une séquence d'instructions ne contenant pas d'instructions de synchronisation.*
- *un bloc « en-tête de* `DO` *» marque le début du corps d'un* `DO`*. Il est le successeur du dernier bloc du corps du* `DO`*.*
- *un bloc « en-tête de* `PARALLEL SECTIONS` *» marque le début d'une* `PARALLEL SECTIONS`*.*
- *un bloc vide est un bloc de jonction utilisé pour rassembler plusieurs arcs incidents.*

Les blocs vides ne servent que pendant la construction du graphe et peuvent être éliminés une fois celle-ci terminée.

Une séquence dans un bloc peut contenir non seulement des instructions simples (affectations, synchronisations, conditions booléennes de tests) mais aussi des instructions structurées, parallèles ou séquentielles, dans lesquelles

ne figure pas d'instruction de synchronisation à quelque niveau d'imbrication que ce soit.

Définition 19 *Les arcs du graphe sont de deux types :*

- *contrôle :*
  - *d'un bloc vers le suivant dans une séquence*
  - *d'un bloc « en-tête de* `PARALLEL SECTIONS` *» vers le premier bloc des sections qui ne comporte pas de clause* `WAIT`
  - *du dernier bloc d'une section s vers le premier bloc des sections qui ont s dans leur clause* `WAIT`
  - *du dernier bloc de chaque branche d'un* `IF` *vers un bloc de jonction*
  - *du dernier bloc de chaque section dont le nom ne figure dans aucune clause* `WAIT` *vers un bloc de jonction*
  - *du dernier bloc d'un* `DO` *vers le bloc « en-tête de* `DO` *» correspondant.*
  - *d'un bloc conditionnel vers le premier bloc de chacune de ses deux branches*
- *synchronisation d'un bloc «* `POST` *» vers un bloc «* `WAIT` *» pour des événements ou des ordinaux de même nom, et d'un bloc «* `POST` *ordinal» vers lui-même.*

Chaque arc du graphe de blocs est étiqueté par une formule logique analogue à celle donnée à chaque arc du graphe de précédence.

Définition 20 *Soit u un arc allant d'un bloc K à un bloc K' de $\mathcal{BG}$, $\boldsymbol{x}$ et $\boldsymbol{y}$ des n-uplets de variables de longueur $|\,\mathfrak{N}(K)|$*[1] *et $|\,\mathfrak{N}(K')|$ respectivement, la formule B_u associée à cet arc u est :*

- *si u est un arc de contrôle simple :*

$$B_u = \boldsymbol{x}_{|\,\mathfrak{N}(K,K')} = \boldsymbol{y}_{|\,\mathfrak{N}(K,K')}$$

- *si u est un arc de retour de boucle* `do` *:*

$$B_u = (\boldsymbol{x'} = \boldsymbol{y'}) \wedge (x + 1 = y)$$

  si $\boldsymbol{x'}@[x] = \boldsymbol{x}_{|\,\mathfrak{N}(K,K')}$ et $\boldsymbol{y'}@[y] = \boldsymbol{y}_{|\,\mathfrak{N}(K,K')}$.

- *si u est un arc de synchronisation, soit p l'instruction* `epost` *ou* `opost` *que contient K et soit w l'instruction* `ewait` *ou* `opost` *que contient K' alors :*

$$B_u = \mathrm{Sync}_v(\boldsymbol{x}, \boldsymbol{y})$$

  si v est l'arc $p \to w$ de $\mathcal{PG}$.

1. Le nid de boucles d'un bloc est défini page 71

Propriétés

- Les blocs sont disjoints : une instruction ne peut pas être dans deux blocs distincts.

- La décomposition en blocs est telle que :

  1. les blocs de synchronisation soient maximaux
  2. les blocs ordinaires soient maximaux

  Par exemple, un bloc conditionnel est créé seulement si au moins une branche contient une synchronisation ; deux blocs ordinaires ne sont jamais consécutifs.

- Toutes les instructions (de premier niveau) dans un bloc ont la même formule d'exécution et le même vecteur d'itération.

- Les précédences à l'intérieur d'un bloc sont calculées avec Pre^0.

Programme agrégé

On peut voir un bloc qui rassemble plus d'une instruction d'une séquence comme une macroinstruction. Il ne peut s'agir que de blocs ordinaires, de blocs « `WAIT` » et de blocs « `POST` ». Nous définissons une syntaxe étendue de programmes « à blocs» par adjonction de trois nouvelles catégories d'instruction à la grammaire du langage décrite en 2.1.1 table 2.1 :

`stmt` → ... | `block` | `block-wait` | `block-post`

Le découpage d'un programme p en blocs produit un programme $\tilde{p}$ écrit dans la syntaxe étendue. Nous donnons ci-dessous un exemple (fig. 5.2).

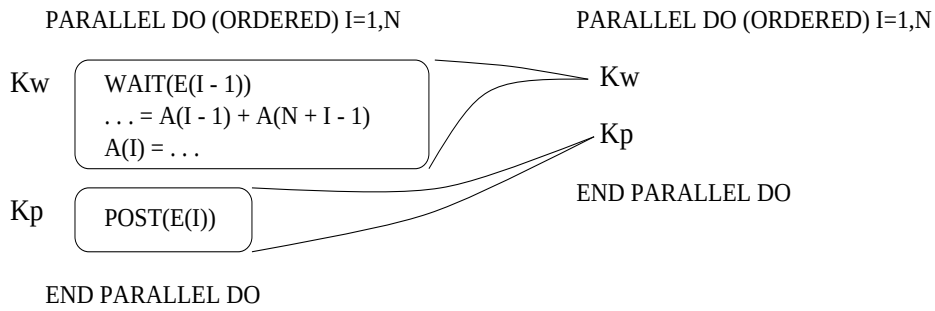

FIG. 5.2 – *Un programme découpé en blocs : à gauche avant agrégation, à droite après agrégation*

Les caractéristiques syntaxiques définies jusqu'à maintenant sur les instructions : nid de boucles, vecteur d'itération, en-tête de contrôle et précédence textuelle s'étendent naturellement aux macroinstructions. Nous pouvons définir un prédicat d'exécution sur les macro-instructions puisqu'elles ont une situation précise dans l'arbre de syntaxe abstraite. Nous pouvons donc parler d'instance d'une macroinstruction puisqu'elle possède un vecteur d'itération et une condition d'exécutabilité.

La relation de précédence de contrôle, dont la formule ne dépend que de la position relative des vecteurs d'itération sur le nid de boucles commun et de la précédence lexicale reste valable sur la syntaxe étendue.

La relation de précédence due aux synchronisations qui peut exister entre un `block-post` et un `block-wait` est héritée de celle qui peut exister entre les instructions `epost` (resp. `opost`) et `ewait` (resp. `owait`) contenues respectivement dans ces blocs.

Nous conserverons, pour toutes les notions qui précèdent, les mêmes notations que celles introduites précédemment.

Précédence

Le graphe de bloc a été introduit pour réduire le coût lié au calcul de la précédence. Étant données deux instructions a et b, soit K_a et K_b les blocs contenant a et b respectivement et soit P un chemin de $\mathcal{BG}$ allant de K_a à K_b. Si P est non vide, il peut être décomposé en une suite $u_0 \cdot u_1 \cdot \ldots \cdot u_n$ telle que u_i est soit un arc de synchronisation soit un chemin de contrôle (la composition d'arcs de contrôle) de manière à ce que deux chemins de contrôle ne soient pas consécutifs. Soit $K_1, K_2, \ldots, K_n$ les n blocs intermédiaires tels que $u_0 : K_a \to K_1$, $u_n : K_n \to K_b$ et $u_i : K_i \to K_{i+1}$ pour $1 \leqslant i \leqslant n-1$.

Nous donnons, ci-dessous, la définition de la formule B_P associée au chemin P de $\mathcal{BG}$ pour le couple a, b.

Définition 21 *Si $P = \Lambda$ (chemin vide) ou $P = u_0$ et u_0 est un chemin de contrôle alors $B_P = \mathrm{Pre}^0_{a,b}$. Autrement, B_P est la conjonction des formules suivantes :*

- $\mathrm{Pre}^0_{K_i,K_{i+1}}$ *si $u_i : K_i \to K_{i+1}$ est un chemin de contrôle.*
- $\mathrm{Exe}^{\mathsf{s}}_{K_i} \wedge \mathrm{Sync}_{K_i,K_{i+1}} \wedge \mathrm{Exe}^{\mathsf{s}}_{K_{i+1}}$ *si $u_i : K_i \to K_{i+1}$ est un arc de synchronisation et $i \neq 0 \wedge i \neq n$.*
- $\mathrm{Pre}^0_{a,p} \wedge \mathrm{Exe}^{\mathsf{s}}_{K_a} \wedge \mathrm{Sync}_{K_a,K_1} \wedge \mathrm{Exe}^{\mathsf{s}}_{K_1}$ *si u_0 est un arc de synchronisation et si p est l'instruction* `epost` *ou* `opost` *de K_a.*
- $\mathrm{Exe}^{\mathsf{s}}_{K_n} \wedge \mathrm{Sync}_{K_n,K_b} \wedge \mathrm{Exe}^{\mathsf{s}}_{K_b} \wedge \mathrm{Pre}^0_{w,b}$ *si u_n est un arc de synchronisation et si w est l'instruction d'attente de K_b.*

Remarque Supposons que P comporte au moins un arc de synchronisation. Si u_0 est un chemin de contrôle, $u_1 : K_1 \to K_2$ est un arc de synchronisation ; on choisit alors d'associer à u_0 la formule $\mathrm{Pre}^0_{a,p}$ où p est l'instruction `epost` ou `opost` de K_1. De même si u_n est un chemin de contrôle, on lui associe la formule $\mathrm{Pre}^0_{w,b}$ où w est l'instruction d'attente de K_n. Ainsi, dans le cas d'un chemin P non réduit à un chemin de contrôle, B_P commence toujours par $\mathrm{Pre}^0_{a,p}$ et se termine toujours par $\mathrm{Pre}^0_{w,b}$ qu'un chemin de contrôle existe ou pas entre K_a et le premier bloc de synchronisation K_p et entre le dernier bloc de synchronisation K_w et K_b.

Nous proposons, ci-après, deux lemmes utiles pour la démonstration de la proposition 5 qui suit.

Lemme 4 *Soit K un bloc non vide et soit a une instruction de K telle qu'il n'existe pas d'instruction interne à K et parente de a dans l'arbre de syntaxe, on a* $\mathrm{Exe}^{\mathtt{s}}_K = \mathrm{Exe}^{\mathtt{s}}_a$.

La démonstration de ce lemme est triviale. En effet l'instruction a n'étant pas imbriquée dans une instruction structurée interne à K, elle a le même nid de boucles et la même suite d'instructions `if` parentes que le bloc K lui-même et par conséquent la même condition d'exécution.

Lemme 5 *Soit a et b deux instructions telles que a soit interne au bloc K_1 et b interne au bloc K_2 distinct de K_1, on a :* $\mathrm{Pre}^0_{K_1,K_2} = \mathrm{Pre}^0_{a,b}$

Démonstration La formule Pre^0 fait intervenir le nid de boucles commun et la relation de précédence lexicale T. Si l'on prouve que $\mathfrak{N}(K_1, K_2) = \mathfrak{N}(a, b)$ et que $K_1 \, T \, K_2 \Leftrightarrow a \, T \, b$ alors on aura prouvé l'équivalence des formules Pre^0.

Si $K_1 \, T \, K_2$ alors toutes les instructions présentes dans K_1 précédent lexicalement toutes les instructions présentes dans K_2 puisque deux blocs distincts ne peuvent se recouvrir, on a donc en particulier $a \, T \, b$. Réciproquement si $\neg(K_1 \, T \, K_2)$ alors $K_2 \, T \, K_1$ et par suite $b \, T \, a$ donc $\neg(a \, T \, b)$. Ceci prouve que $K_1 \, T \, K_2 \Leftrightarrow a \, T \, b$.

On a $\mathfrak{N}(K_1) \subseteq \mathfrak{N}(a)$ et $\mathfrak{N}(K_2) \subseteq \mathfrak{N}(b)$ ce qui entraîne que $\mathfrak{N}(K_1, K_2) \subseteq \mathfrak{N}(a, b)$. Supposons que $\mathfrak{N}(K_1, K_2) \subset \mathfrak{N}(a, b)$, alors il existe une instruction c, `do` ou `pdo`, appartenant à $\mathfrak{N}(a, b)$ mais n'appartenant pas à $\mathfrak{N}(K_1, K_2)$. On a alors trois cas :

1. $c \notin \mathfrak{N}(K_1)$, mais $c \in \mathfrak{N}(a)$ donc c est interne à K_1. Par ailleurs $c \in \mathfrak{N}(b)$ donc b est descendante de c dans l'arbre de syntaxe et par suite b est interne à K_1 ce qui contredit $K_1 \neq K_2$, les blocs étant disjoints par construction.

2. $c \notin \mathfrak{N}(K_2)$, le raisonnement est identique à celui de 1.

3. $c \notin \mathfrak{N}(K_1) \wedge c \notin \mathfrak{N}(K_2)$ donc c est interne à K_1 et à K_2 ce qui contredit à nouveau $K_1 \neq K_2$.

Conclusion : $\mathfrak{N}(K_1, K_2) = \mathfrak{N}(a, b)$. □

Proposition 5 *Pour un chemin $P : K_a \longrightarrow K_b$, on a l'implication :*

$$B_P \Rightarrow \mathrm{Pre}_{a,b}$$

Démonstration Soit $\boldsymbol{i}, \boldsymbol{j}$ deux n-uplets d'entiers de longueur $|\mathfrak{N}(a)|$ et $|\mathfrak{N}(b)|$ respectivement. Le chemin P est de la forme : $K_a = K_0 \rightarrow K_1 \rightarrow \ldots K_i \rightarrow K_{i+1} \rightarrow \ldots K_{n+1} = K_b$. On sait, par hypothèse, qu'il existe des n-uplets d'entiers $\boldsymbol{k}_i$ de longueur $|\mathfrak{N}(K_i)|$ tels que $B_P(\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}_1, \ldots, \boldsymbol{k}_i, \ldots, \boldsymbol{k}_n)$ soit vrai.

La démonstration consiste à mettre en évidence un chemin $P' : \langle a, \boldsymbol{i}\rangle \longrightarrow \langle b, \boldsymbol{j}\rangle$ du graphe de flot $\mathcal{FG}$ et à montrer que la formule $\mathrm{Pre}(P')$ est vraie.

On construit P' à partir de P. Tous les sommets de P sont soit des blocs «`POST`» soit des blocs «`WAIT`» à l'exception peut-être des blocs K_a et K_b. Si K_i est un bloc «`POST`», il contient par définition une et une seule instruction de postage, appelons la p_i. Si K_i est un bloc «`WAIT`», il contient par définition une et une seule instruction d'attente, appelons la w_i. On prend pour P' le chemin ci-dessous :

$$\langle a, \boldsymbol{i}\rangle \longrightarrow \langle p_1, \boldsymbol{k}_1\rangle \rightarrow \langle w_2, \boldsymbol{k}_2\rangle \longrightarrow \ldots \langle p_{n-1}, \boldsymbol{k}_{n-1}\rangle \rightarrow \langle w_n, \boldsymbol{k}_n\rangle \longrightarrow \langle b, \boldsymbol{j}\rangle$$

ce qui a un sens puisque $\mathfrak{N}(s) = \mathfrak{N}(K_i)$ pour $s = p_i$ ou $s = w_i$. La formule associée à P' est :

$$\begin{aligned}
&\mathrm{Pre}(P') = \\
&\quad \mathrm{Pre}^0(\langle a, \boldsymbol{i}\rangle, \langle p_1, \boldsymbol{k}_1\rangle) \wedge \\
&\quad \mathrm{Exe}^{\mathsf{s}}(\langle p_1, \boldsymbol{k}_1\rangle) \wedge \mathrm{Sync}(\langle p_1, \boldsymbol{k}_1\rangle, \langle w_2, \boldsymbol{k}_2\rangle) \wedge \mathrm{Exe}^{\mathsf{s}}(\langle w_2, \boldsymbol{k}_2\rangle) \wedge \\
&\quad \ldots \mathrm{Exe}^{\mathsf{s}}(\langle p_{n-1}, \boldsymbol{k}_{n-1}\rangle) \wedge \mathrm{Sync}(\langle p_{n-1}, \boldsymbol{k}_{n-1}\rangle, \langle w_n, \boldsymbol{k}_n\rangle) \wedge \mathrm{Exe}^{\mathsf{s}}(\langle w_n, \boldsymbol{k}_n\rangle) \wedge \\
&\quad \mathrm{Pre}^0(\langle w_n, \boldsymbol{k}_n\rangle, \langle b, \boldsymbol{j}\rangle)
\end{aligned}$$

Or, par définition $\mathrm{Sync}_{K_i,K_{i+1}}(k_i, k_{i+1}) = \mathrm{Sync}_{p_i,w_{i+1}}(\boldsymbol{k}_i, \boldsymbol{k}_{i+1})$ pour $1 \leqslant i \leqslant n-1$; d'après le lemme 4, $\mathrm{Exe}^{\mathsf{s}}_{K_i}(k_i) = \mathrm{Exe}^{\mathsf{s}}_{s}(k_i)$ pour $s = p_i$ ou $s = w_i$ et $1 \leqslant i \leqslant n$; d'après le lemme 5, $\mathrm{Pre}^0_{K_i,K_{i+1}}(k_i, k_{i+1}) = \mathrm{Pre}^0_{w_i,p_{i+1}}(k_i, k_{i+1})$ pour $1 \leqslant i \leqslant n-1$. Donc chaque élément de la conjonction $\mathrm{Pre}(P')$ est égal à un élément de la conjonction B_P qui est vraie et par suite la formule est vraie. □

La conséquence de cette proposition est que pour prouver la condition principale de préservation d'une dépendance entre une instruction a et une instruction b, il suffira de prouver qu'il existe un chemin $P : K_a \longrightarrow K_b$ dans $\mathcal{BG}$ tel que $\mathrm{Dep}_{a,b} \Rightarrow B_P$.

5.2.4 Calcul des chemins

On considère deux instructions a et b telles que $\mathrm{Dep}_{a,b}$. Nous nous plaçons ici dans le cas général où il est nécessaire de prendre en compte les synchronisations pour assurer la précédence $\mathrm{Pre}_{a,b}$. Soit K_a le bloc contenant a et K_b le bloc contenant b. On s'intéresse aux chemins allant de K_a à K_b. L'ensemble des chemins entre deux nœuds d'un graphe constitue un langage rationnel sur l'alphabet des arcs du graphe ; il est donc possible, par un algorithme ad hoc, de représenter ce langage par une expression rationnelle. Dans notre cas, on pourrait associer à un couple de blocs, ou d'instructions, une expression rationnelle représentant tous les chemins de contrôle et de synchronisation entre eux. Il est cependant difficile d'exploiter les expressions obtenues dès qu'elles comportent plusieurs niveaux d'étoile, car elles conduisent à des équations non-linéaires. Nous avons donc renoncé à cette représentation, pour adopter une énumération en largeur des chemins, avec un critère de classement des successeurs d'un nœud. Nous reviendrons plus en détail sur la représentation par expression rationnelle au chapitre 8.

Cette méthode énumérative est évidemment incomplète car nous ne développons que des chemins finis de longueur fixée. Si nous trouvons un chemin P tel que $\mathrm{Exe}_a^{\mathsf{s}} \wedge \mathrm{Exe}_b^{\mathsf{s}} \wedge \mathrm{Dep}_{a,b} \Rightarrow B_P$ est satisfaisable alors la dépendance est préservée. Mais en cas d'insuccès, nous ne prouvons pas la non préservation.

Pour accélérer la découverte d'un chemin assurant la précédence, le critère de classement doit être choisi de manière à éviter d'itérer plusieurs fois de suite une boucle `do` avant de prendre un arc de synchronisation, sachant que la relation Pre^0 contient l'information de précédence associée à N itérations, N étant quelconque.

5.3 Preuve de la formule de correction

Il s'agit de prouver, pour un chemin P dans $\mathcal{BG}$, que

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \exists \vec{\boldsymbol{z}} \, (\mathrm{Exe}_a^{\mathsf{s}} \wedge \mathrm{Exe}_b^{\mathsf{s}} \wedge \mathrm{Dep}_{a,b} \Rightarrow \mathrm{Pre}_P) \tag{5.1}$$

est valide. Les variables quantifiées de cette formule sont :

- les variables d'itération $\boldsymbol{x}$ de a et $\boldsymbol{y}$ de b,
- le vecteur des variables d'itération $\vec{\boldsymbol{z}}$ des blocs sur le chemin P, figurant seulement à droite de l'implication.

Cette formule contient également les paramètres $\boldsymbol{p}$ du programme, non quantifiés. Notons que Pre_P est une approximation B_P de Pre calculée le long d'un chemin P.

$\mathrm{Dep}_{a,b}$ est une disjonction $Dep_1 \vee \ldots \vee Dep_m$, à cause de sa composante $(a \prec b)$; chaque Dep_i est un système d'équations et d'inéquations. Seuls les termes Dep_i qui n'ont pas été prouvés insatisfaisables par le test Omega

(Cf. 5.4) sont considérés ici. Rappelons (Cf. 4.5.2 page 56) que la formule de dépendance considérée ici est une approximation conservative $\mathrm{Dep}^\star$.

$\mathrm{Exe}^{\mathtt{s}}$ est une conjonction de conditions des `if` et d'inéquations exprimant les bornes des boucles, contenant uniquement des paramètres et des variables d'indice, selon l'hypothèse de staticité (Cf. page 63).

Pre_P est une conjonction (le long du chemin P) de formules $\mathrm{Exe}^{\mathtt{s}}$, Sync et Pre^0. Sync est elle-même une conjonction d'équations et Pre^0 est une disjonction d'équations et d'inéquations, de façon analogue à $\prec$.

Si nous exprimons les deux membres de l'implication sous forme normale disjonctive, i.e. comme une disjonction de systèmes, la formule 5.1 devient :

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \exists \vec{\boldsymbol{z}} \, (\vee_i D_i \Rightarrow \vee_j P_j)$$

Elle est encore équivalente à

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \, \wedge_i \vee_j \, (D_i \Rightarrow \exists \vec{\boldsymbol{z}} \, P_j)$$

Nous devons donc prouver la formule

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \forall i \exists j \, (D_i \Rightarrow \exists \vec{\boldsymbol{z}} \, P_j)$$

Il suffit de montrer que pour tout i, il existe un système P_j, tel que

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \, D_i(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \exists \vec{\boldsymbol{z}} \, P_j(\boldsymbol{x}, \boldsymbol{y}, \vec{\boldsymbol{z}})$$

soit valide. C'est seulement une condition suffisante à cause de l'inversion du $\exists j$ avec le $\forall \boldsymbol{x} \forall \boldsymbol{y}$, qui a l'avantage de pouvoir être soumise au test Omega. Cette simplification revient à limiter la préservation d'une dépendance à une précédence uniforme, i.e., qui ne dépende pas des variables d'itération puisque j ne dépend plus de $\boldsymbol{x}, \boldsymbol{y}$.

5.4 Le test Omega

Les programmes de tests utilisés dans les systèmes de parallélisation automatique apportent une réponse binaire (oui/non) au problème d'existence de solutions à un système linéaire d'équations et d'inéquations. Ils ont l'intérêt d'être rapides pour la détermination des dépendances de données entre éléments de tableaux mais sont insuffisants pour traiter un problème comme le nôtre.

L'algorithme que nous utilisons est l'Omega-test [30], basé sur une extension de l'algorithme de Fourier-Motzkin d'élimination de variable. Il a été développé par W. Pugh à l'Université du Maryland et utilisé pour résoudre des problèmes d'analyse des dépendances de données de façon exacte [32].

Ce test s'applique seulement aux systèmes *linéaires* d'équations et d'inéquations, c'est pourquoi nous ne pouvons nous occuper que des programmes dans lesquels les expressions d'indice et les bornes de boucles sont linéaires.

L'Omega-test n'est pas un simple système de décision ; il est capable de réaliser des projections symboliques d'un problème sur un sous-ensemble des variables, éliminant ainsi des variables, ce qui donne une forme réduite du système de contraintes. Nous utilisons cette capacité combiné au calcul d'une formule dite du « gist » pour prouver notre implication.

5.4.1 Projection et calcul du gist

L'opération de base du test Omega est la projection. Étant donné un système d'équations et d'inéquations sur un ensemble V de variables, la projection du système de contraintes sur un ensemble de variables $\widehat{V} \subset V$ produit un système de contraintes sur $\widehat{V}$ qui a les mêmes solutions entières pour $\widehat{V}$ que le système initial. Cette opération de projection est itérée jusqu'à l'obtention d'un système à une seule variable pour lequel il est facile de vérifier si des solutions entières existent.

L'opération de projection est réalisée par un algorithme d'élimination de variable adapté de l'algorithme de Fourier-Motzkin.

L'algorithme d'élimination de Fourier-Motzkin consiste à prendre deux contraintes sur une variable, disons z, l'une représentant une borne inférieure $\beta \leqslant bz$ et l'autre une borne supérieure $az \leqslant \alpha$, α, β ne contenant pas z, et à les combiner pour obtenir $a\beta \leqslant abz \leqslant b\alpha$. La projection ou « l'ombre réelle » de ce couple de contraintes est $a\beta \leqslant b\alpha$. L'ombre d'un ensemble de contraintes est calculée en combinant les contraintes ne contenant pas la variable à éliminer avec le résultat de chaque combinaison d'une borne inférieure et supérieure sur la variable à éliminer.

Une version étendue de cet l'algorithme est utilisé par l'Omega-test car il s'intéresse à l'existence de solutions entières. Même si $a\beta \leqslant b\alpha$, il peut ne pas exister de solution entière à z telle que $a\beta \leqslant abz \leqslant b\alpha$. Toutefois, si $a\beta + (a-1)(b-1) \leqslant b\alpha$, alors une solution entière à z doit exister. C'est «l'ombre noire» de ce couple de contraintes. L'ombre noire est une approximation par «en dessous» de l'ombre entière d'un ensemble de contraintes. Quelquefois, la projection d'un système de contraintes S «le long» d'une variable z (notée $\pi_{\neg z}$) peut produire un ensemble de systèmes $S_0, S_1, \ldots, S_p$. L'ombre entière $\pi_{\neg z}(S)$ est alors égale à $\cup_{i=0}^{p} S_i$. Il y a «éclatement» du problème et si S_0 est l'ombre noire et si l'on appelle T l'ombre réelle alors on a $S_0 \subseteq \pi_{\neg z}(S) \subseteq T$.

Le test Omega permet toutefois de traiter des problèmes plus généraux que l'existence de solutions à des systèmes d'équations et d'inéquations, notamment des implications (Cf. [33]).

Supposons que p et q soient des conjonctions d'équations et d'inéquations linéaires. Prouver que $\forall x\,(p \Rightarrow \exists y\, q)$ est vrai équivaut à montrer que $p \Rightarrow \pi_{\neg y}(q)$ est une tautologie où $\pi_{\neg y}(q)$ représente la projection du problème q sur toutes les variables sauf y.

Pour prouver l'implication, nous utilisons une formule gist définie de sorte que $(gist\, p\, given\, q) \wedge q$ soit égale à $p \wedge q$. Il en résulte que $gist\, p\, given\, q =$

$true \Leftrightarrow q = (p \wedge q) \Leftrightarrow (q \Rightarrow p)$. Intuitivement, $gist\, p\, given\, q$ représente l'information nouvelle apportée par p étant donnée celle déjà contenue dans q. Par exemple :

$$\begin{array}{lcl} gist\{1 \leqslant i \leqslant 10\}\, given\, \{i \leqslant 5\} & = & \{1 \leqslant i\} \\ gist\{1 \leqslant i \leqslant 10\}\, given\, \{i \leqslant 12\} & = & \{1 \leqslant i \leqslant 10\} \end{array}$$

mais

$$gist\{i \leqslant 1\}\, given\, \{5 \leqslant i \leqslant 10\} \quad = \quad false$$

Donc pour prouver nos implications, qui sont de la forme $\forall \boldsymbol{xy}\, (D \Rightarrow \exists \vec{\boldsymbol{z}} P)$, nous calculons la formule $gist\, \pi_{\boldsymbol{x},\boldsymbol{y}}(P)\, given\, D$, et vérifions qu'elle est vraie.

Pratiquement, nous calculons le gist selon l'algorithme suivant :

$$\begin{array}{lcll} gist\, [\,]\, q & = & [\,] & \\ gist\, (e :: p)\, q & = & e :: (gist\, p\, (e :: q)) & \text{si } \neg e \wedge p \wedge q \text{ est satisfaisable} \\ gist\, (e :: p)\, q & = & gist\, p\, q & \text{sinon} \end{array}$$

Dans cet algorithme, p est représenté par une liste de contraintes dont les égalités ont été préalablement transformées en un couple d'inégalités logiquement équivalent.

Chapitre 6

Implémentation

Dans ce chapitre, nous donnons quelques éléments sur le contexte de l'implémentation du système de vérification. Nous évoquons l'environnement Centaur [5] et l'utilisation de la bibliothèque Omega.

L'algorithme de construction du graphe de blocs est décrit précisément.

6.1 L'environnement de développement Centaur

Le système Centaur est un générateur d'environnements qui fournit les formalismes de spécification syntaxique (Metal [21]) et sémantique (Typol [21]) dont des outils spécifiques sont dérivés et intégrés sous une même interface.

Le travail d'implémentation s'est fait dans le prolongement d'un outil de parallélisation de programmes Fortran 77 réalisé à l'INRIA, et construit à l'aide de Centaur : Piaf [15, 28, 16]. Cet outil de parallélisation a été modifié en étendant la spécification de la syntaxe et du pretty-printer de Fortran 77 à Fortran X3H5. Nous avons bénéficié du vérificateur de types de Fortran 77, que nous avons étendu à X3H5, et de l'interface graphique de Piaf, que nous avons complétée.

Mais l'essentiel du travail d'implémentation a porté sur la réalisation du vérificateur de correction des constructions parallèles. Le langage d'implémentation est principalement Le-Lisp [8] pour la construction de graphes et de systèmes d'équations à partir des informations contenues dans l'arbre de syntaxe abstraite du programme. La manipulation de ce dernier se fait grâce aux fonctions Le-Lisp du « Virtual Tree Processor » (VTP) de Centaur. La preuve des formules arithmétiques est confiée à un système externe écrit en C dont les fonctions ont été interfacées avec Le-Lisp pour pouvoir être appelées directement à partir de celui-ci.

Le générateur Centaur a également été utilisé pour construire un paralléliseur interactif pour un sous-ensemble de Fortran, ParaGraph [10]. La sémantique dynamique et le calcul des dépendances ont été spécifiés dans le

formalisme Typol.

6.2 L'interface avec l'Omega-test

L'intérêt de l'Omega-test, outre sa rapidité et la précision des résultats qu'il fournit, est de se présenter sous forme d'une bibliothèque de fonctions C décrite dans [31], indépendantes de tout système applicatif destiné, par exemple, à la parallélisation ou à l'optimisation.

Dans l'Omega-test un problème de programmation linéaire en nombres entiers (un système d'équations et d'inéquations linéaires à coefficients entiers) est représenté par une structure `_problem` qui comporte notamment les champs `_EQs` et `_GEQs`, respectivement tableau des équations et des inéquations. Une équation ou inéquation est représentée par un tableau de ses coefficients.

La principale fonction de la bibliothèque est la fonction C :

```
int _simplifyProblem(Problem *)
```

qui prend en argument un pointeur sur un problème. Elle retourne 1 si une solution existe et 0 sinon, en outre le problème est simplifié.

Nous avons utilisé les possibilités d'interfaçage externe du langage Le-Lisp pour relier les fonctions de la bibliothèque Omega à l'environnement Centaur. Une quarantaine de fonctions C ont ainsi été définies pour réaliser l'interface entre le système de vérification et les principales fonctionnalités du système Omega.

6.3 La construction du graphe de blocs

Rappelons que la décomposition en blocs doit respecter ces deux contraintes :

1. les blocs de synchronisation soient maximaux
2. les blocs ordinaires soient maximaux

Par exemple, un bloc conditionnel est créé seulement si au moins une branche contient une synchronisation ; deux blocs ordinaires ne sont jamais consécutifs.

6.3.1 Structure de données

Les blocs sont implémentés par des structures chaînées représentées à la Figure 6.1. Le chaînage est assuré par trois champs : `succ, back` et `sync`. Le champ `succ` pour les arcs de contrôle « avant », le champ `back` pour l'arc de contrôle « arrière » dans un nid de boucles `DO`, et `sync` pour les arcs de synchronisation. Ces champs peuvent être vides.

Pour construire un bloc ordinaire, nous utilisons la structure Le-Lisp ci-dessous qui définit en fait une classe d'objets :

```
(defstruct block
  lstat    ; liste des instructions
  succ     ; liste des blocs successeurs (contrôle avant)
  back     ; éventuellement bloc de début d'un do (contrôle arrière)
  nest     ; liste des boucles do et pdo englobantes
  exe      ; liste des if englobants
  is-eq    ; liste des eq. et ineq. représentant l'espace d'itération
  exe-eq   ; liste des eq. et ineq. représentant la condition d'exécution
  )
```

Les autres types de blocs sont représentés par des sous-classes de la structure `block` définie ci-dessus, i.e. elles en héritent les champs ; par exemple pour un bloc « post » :

```
(defstruct #:block:block-post
  event    ; nom de la variable événement
  lexp     ; liste d'expressions
  synch    ; liste d'instructions wait de même événement
  synch-eq ; liste d'eq.
  )
```

Les blocs vides n'héritent pas de la structure `block` car ils n'ont pas besoin de tous ses champs : seuls les champs `succ` et `back` sont nécessaires. Ils sont donc tous représentés par une structure analogue à celle-ci, au nom près :

```
(defstruct psections-block
  succ
  back
  )
```

6.3.2 Algorithme

L'algorithme ci-dessous décrit le processus de décomposition en blocs et de construction du graphe avec ses liens de contrôle. Il est écrit dans le langage Lisp.

```
(defun block-graph (p)
  (let ((bstart (make-block 'start ())))
    (split-seq (program-stats p) nil nil bstart)
    bstart))
```

La fonction principale `block-graph` prend en argument un arbre de syntaxe abstraite et retourne le bloc de départ du graphe. Elle appelle la fonction `split-seq` qui prend en argument la liste d'instructions restant à traiter, la liste d'instructions du bloc en cours de construction, le type du bloc en cours de construction et enfin le bloc qui précède.

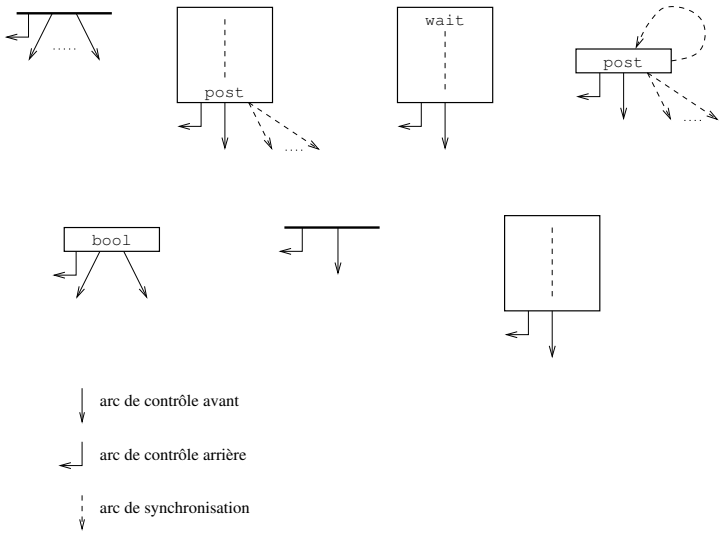

FIG. 6.1 – *Les blocs*

La fonction `split-seq` (Cf. figure 6.2 page 83) est chargée de traiter une séquence d'instructions `stats` placées dans une liste. Si la liste est vide alors le bloc courant est fermé, sinon on regarde la première instruction de la liste. Si elle ne contient pas de synchronisation alors elle est ajoutée au bloc courant et la fonction est rappelée récursivement sur le reste de la liste. Si elle contient des synchronisations alors il y a plusieurs cas possibles suivant la nature de l'instruction :

1. `stat` est un `ewait` ou `owait`.

   ```
   (defun wait-case (stat stats bstats bkind bpre)
     (let ((newbpre (close-block bstats bkind bpre)))
       (split-seq (cdr stats) (cons stat ()) 'wait newbpre)))
   ```

   Une instruction `WAIT` sur les événements ou les ordinaux commence un bloc. Le bloc courant est donc fermé et un nouveau bloc de type `wait` est commencé.

2. `stat` est un `epost`.

   ```
   (defun epost-case (stat stats bstats bkind bpre)
     (if (eq bkind 'wait)
         (let ((bw (make-block bkind bstats))
               (bp (make-block 'post (cons stat ()))))
           (link bpre bw)
           (link bw bp)
           (split-seq (cdr stats) () nil bp))
       (let ((bp (make-block 'post (cons stat bstats))))
   ```

```
(defun split-seq (stats bstats bkind bpre)
  ; stats : liste d'instructions restant à placer
  ; bstats : liste d'instructions appartenant au bloc en cours de construction
  ; bkind : type du bloc en cours de construction
  ; bpre : bloc précédent
  (if (null stats)
      (let ((newbpre (close-block bstats bkind bpre)))
        newbpre)
      (let ((stat (car stats)))
        (if (synch-free? stat)
            ; vrai si stat est différent de ewait,owait,epost,opost
            ; ou bien récursivement si chaque instruction du corps de stat
            ; est synch-free? si stat est une instruction structurée
            (split-seq (cdr stats) (cons stat bstats) bkind bpre)
            (selectq (kind-of-stat stat)
              ((ewait owait)
               (wait-case (stat stats bstats bkind bpre)))
              (epost
               (epost-case (stat stats bstats bkind bpre)))
              (opost
               (opost-case (stat stats bstats bkind bpre)))
              (if
               (if-case (stat stats bstats bkind bpre)))
              (pdo
               (pdo-case (stat stats bstats bkind bpre)))
              (do
               (do-case (stat stats bstats bkind bpre)))
              (psections
               (psections-case (stat stats bstats bkind bpre)))
              )))))
```

FIG. 6.2 – *Fonction de découpage d'une séquence d'instructions en blocs*

```
    (link bpre bp)
    (split-seq (cdr stats) () nil bp))))
```

Une instruction `POST` sur les événements termine un bloc sauf s'il s'agit d'un bloc de type `wait` qui est alors fermé et un nouveau bloc de type `epost` est commencé.

3. `stat` est un `opost`.

```
(defun opost-case (stat stats bstats bkind bpre)
  (let ((newbpre (close-block bstats bkind bpre))
        (bop (make-block 'opost (cons stat ()))))
    (link newbpre bop)
    (split-seq (cdr stats) () nil bop)))
```

Une instruction `POST` sur les ordinaux est seule dans un bloc donc le bloc courant est fermé et un bloc de type `opost` est créé.

4. `stat` est un `if`.

```
(defun if-case (stat stats bstats bkind bpre)
  (let ((newbpre (close-block bstats bkind bpre))
        (bif (make-block 'if (cons stat ()))))
    (link newbpre bif)
    (let ((bthen (split-seq (then-stats stat) () nil bif))
          (belse (split-seq (else-stats stat) () nil bif)))
      (let ((bj (make-block 'join ())))
        (link bthen bj)
        (link belse bj)
        (split-seq (cdr stats) () nil bj))))))
```

Un `if` représentant la condition booléenne est seule dans un bloc donc le bloc courant est fermé et un bloc de type `if` est créé. Les branches du `if` sont découpées en blocs par des appels récursifs et un bloc vide de jonction des branches est créé.

5. `stat` est un `pdo`.

```
(defun pdo-case (stat stats bstats bkind bpre)
  (let ((newbpre (close-block bstats bkind bpre)))
    (let ((bpdo (split-seq (pdo-stats stat) () nil newbpre)))
      (split-seq (cdr stats) () nil bpdo))))
```

Le bloc courant est fermé et le corps du `pdo` est découpé en blocs par un appel récursif.

6. `stat` est un `do`.

```
(defun do-case (stat stats bstats bkind bpre)
  (let ((newbpre (close-block bstats bkind bpre))
```

```
    (begin-do (make-block '1do ())))
  (link newbpre begin-do)
  (let ((bdo (split-seq (do-stats stat) () nil begin-do)))
    (link-back bdo begin-do)
    (split-seq (cdr stats) () nil bdo))))
```

Le bloc courant est fermé, un bloc vide est créé pour repérer le début du `do` et le corps du `do` est découpé en blocs par un appel récursif.

7. `stat` est un `psections`.

```
(defun psections-case (stat stats bstats bkind bpre)
  (let ((newbpre (close-block bstats bkind bpre)))
    (let ((bs (mapcar
               (lambda (s) (split-seq
                            (section-stats s) () nil newbpre))
               (section-list stat))))
      (let ((bj (make-block 'join ())))
        (mapc (lambda (b) (link b bj)) bs)
        (split-seq (cdr stats) () nil bj))))))
```

Le bloc courant est fermé et le corps de chaque `section` est découpé en blocs par un appel récursif.

L'algorithme fait appel à des fonctions de navigation dans l'arbre de syntaxe abstraite et aux fonctions auxiliaires suivantes :

- `synch-free?` : prédicat qui est vrai si `stat` est différent d'un `ewait, owait, epost` ou `opost` ou bien récursivement si chaque instruction du corps de `stat` est `synch-free?` au cas où `stat` est une instruction structurée.

- `close-block` : crée un bloc si la liste d'instructions n'est pas vide

```
(defun close-block (bstats bkind bpre)
  (if bstats
      (let ((b (make-block bkind bstats))) (link bpre b) b)
      bpre))
```

- `link` : rajoute un élément à la liste des successeurs (champ `succ`)

- `link-back` : affecte le champ `back`.

Une fois le graphe construit, les blocs vides sont supprimés, le champ `sync` est affecté, et les clauses `WAIT` des sections prises en compte.

Un exemple de décomposition d'un programme en blocs est donné Figure 6.3.

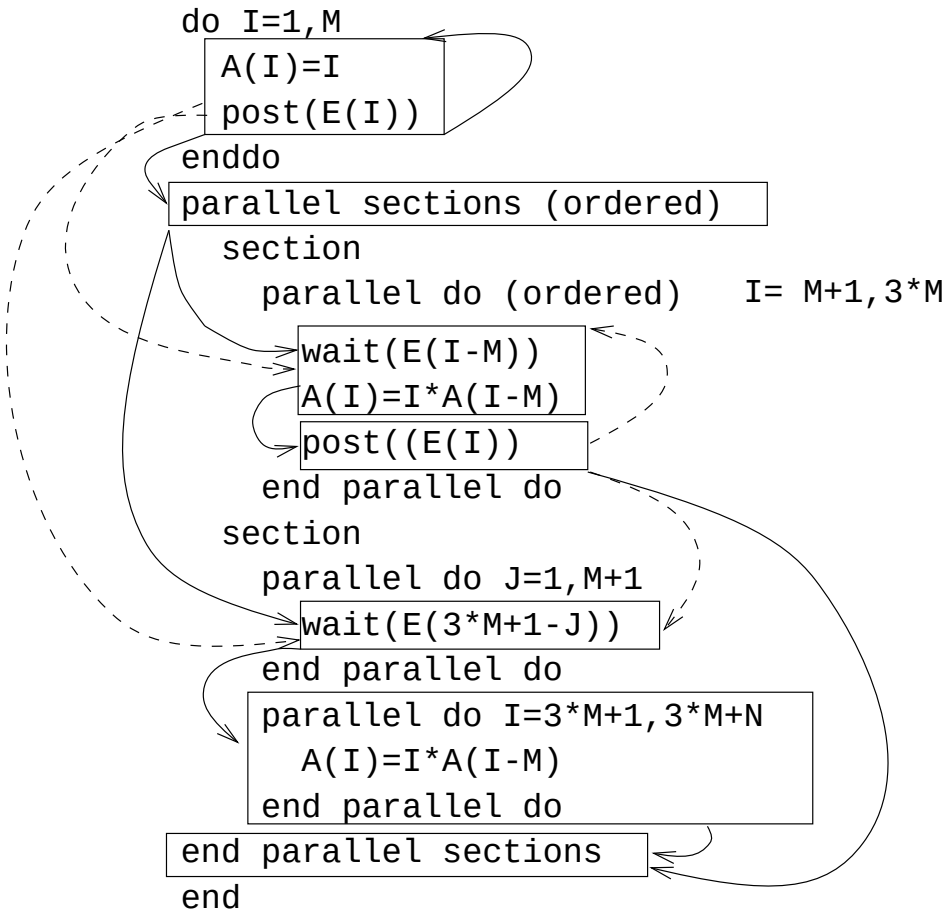

FIG. 6.3 – *Exemple de décomposition d'un programme en blocs. Les arcs de contrôle sont en traits pleins, les arcs de synchronisations sont en traits pointillés.*

Chapitre 7

Exemples

Ce chapitre montre cinq exemples de programmes. Le premier exemple présente le cas d'une dépendance uniforme dont la préservation est assurée par une séquentialisation complète de le boucle. Le deuxième exemple montre le cas d'une dépendance non uniforme. Dans le troisième exemple, nous montrons un cas de contrôle complexe par des synchronisations. Le quatrième exemple illustre un cas où les diverses expressions dépendent de paramètres du programme. Le dernier exemple est une variante du premier exemple qui fait échouer notre algorithme de vérification ; bien que la boucle soit complètement séquentialisée, le développement de chemins de longueur bornée ne permet pas de prouver la préservation de toutes les dépendances.

L'analyse de ces programmes est développée en partie « à la main» pour plus de clarté. Les résultats fournies par le système sont présentées sous forme d'une trace. On y trouve les informations suivantes :

- le conflit sur lequel porte l'analyse ;
- le nombre et la longueur des chemins testés (ligne `allpaths= ...`) dans laquelle chaque `&` est un arc ;
- les systèmes de dépendance et de précédence tels que l'Omega-test les représente ;
- le système résultat de la projection du système de précédence sur les variables universelles ;
- les inéquations composant le « gist». Si la liste vide « `()`» apparaît cela signifie *true* puisque le « gist» est une conjonction.

7.1 Exemple 1 : un cas simple

```
A(0) = ...
POST(E(0))
```

```
      PARALLEL DO (ORDERED) I=1,N
w:     WAIT(E(I-1))
b:     ... = A(I-1)
a:     A(I) = ...
p:     POST(E(I))
      END PARALLEL DO
```

Certaines instances des instructions a et b peuvent être exécutées concurremment, nous devons examiner si elles peuvent être en conflit. Les instructions a et b vérifient :

- $\mathfrak{I}(a) = [\mathtt{I}]$; $\mathfrak{I}(b) = [\mathtt{I}]$
- $\mathfrak{N}(a, b) = [(\mathtt{pdo}, I, 1, N)]$
- $\mathfrak{I}(a, b) = [\mathtt{I}]$
- $\mathfrak{C}(a, b) = ;$
- $a\,T\,b = \mathit{false}$
- $\mathfrak{In}(b) = \{A(I-1)\}$
- $\mathfrak{Out}(a) = \{A(I)\}$
- $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(b) = \{A\}$

La formule de dépendance doit donc être construite :

$$\mathrm{Dep}_{a,b}(x, y) = (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x = y - 1)$$

Cette formule étant satisfaisable, il faut étudier les précédences. La précédence de contrôle est :

$$\begin{aligned} \mathrm{Pre}^0_{a,b}(x, y) &= (x = y) \wedge \mathit{false} \\ &= \mathit{false} \end{aligned}$$

Comme $\mathrm{Pre}^0_{a,b}(x, y)$ n'est pas conséquence de $\mathrm{Dep}_{a,b}(x, y)$, les synchronisations doivent être considérées, donc le graphe de blocs doit être construit. Il est représenté sur la figure 7.1.

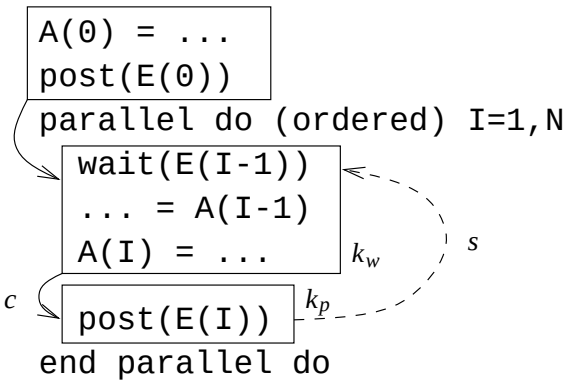

FIG. 7.1 – *Graphe de blocs : exemple 1*

On a $\mathrm{Exe}_a^{\mathsf{s}}(x) = (1 \leqslant x \leqslant N)$ et $\mathrm{Exe}_b^{\mathsf{s}}(y) = (1 \leqslant y \leqslant N)$. Les instructions a et b sont contenues dans un même bloc k_w, et il y a un arc de synchronisation s de k_p à k_w avec la formule $\mathrm{Sync}_{p,w}(x, y) = (x = y - 1)$; les chemins de k_w à lui-même sont de la forme $cs(cs)^*$.

Nous commençons par le chemin le plus court, cs. La formule $\mathrm{Pre}_{cs}(x, y)$ est $\mathrm{Pre}^0_{a,p}(x, z_1) \wedge \mathrm{Exe}_p(z_1) \wedge \mathrm{Sync}_{p,w}(z_1, z_2) \wedge \mathrm{Exe}_w(z_2) \wedge \mathrm{Pre}^0_{w,b}(z_2, y)$.

Nous devons prouver :

$$\forall x \forall y \exists z_1 \exists z_2 \, (\mathrm{Dep}_{a,b}(x, y) \wedge \mathrm{Exe}_a^{\mathsf{s}}(x) \wedge \mathrm{Exe}_b^{\mathsf{s}}(y) \Rightarrow \mathrm{Pre}_s(x, y, z_1, z_2))$$

i.e.,

$$\begin{array}{l} \forall x \forall y \exists z_1 \exists z_2 \\ ((1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x = y - 1) \\ \Rightarrow \\ (x = z_1) \wedge (1 \leqslant z_1 \leqslant N) \wedge (z_1 = z_2 - 1) \wedge (1 \leqslant z_2 \leqslant N) \wedge (z_2 = y)) \end{array}$$

Ce problème est transmis au test Omega, qui le projette sur les variables x, y et N et calcule son *gist* étant donné le système de dépendance. La trace suivante peut être obtenue :

```
--------- projection of precedence system
-----------------------------------------
reduced problem:
variables = (x, n, y)
y = 1+x
1+x <= n
1 <= x
-----------------------------------------
dependence and projected precedence system:
dependence system:
0) y = 1+x
0) 1 <= x
1) x <= n
2) 1 <= y
3) y <= n
4) 1+x <= y
precedence system:
5) y <= 1+x
6) 1+x <= y
7) 1+x <= n
8) 1 <= x
gist: ()
```

Le *gist* est identiquement vrai (système vide), ce qui suffit à prouver que le programme est correct.

Si on remplace `E(I-1)` par `E(I+1)` dans l'instruction `WAIT`, la trace suivante donne pour le *gist* la formule $1 + y \leqslant x$, qui n'est pas valide, les variables x et y étant quantifiées universellement :

```
--------- projection of precedence system
-----------------------------------------
reduced problem:
variables = (x, n, y)
1+y = x
x <= n
2 <= x
-----------------------------------------
dependence and projected precedence system:
dependence system:
0) y = 1+x
0) 1 <= x
1) x <= n
2) 1 <= y
3) y <= n
4) 1+x <= y
precedence system:
5) 1+y <= x
6) x <= 1+y
7) x <= n
8) 2 <= x
gist: (5)
1+y <= x
```

Comme il s'agit de l'unique chemin de précédence possible, c'est une indication sérieuse pour que le programme soit incorrect.

7.2 Exemple 2 : une dépendance non uniforme

```
      PARALLEL SECTIONS (ORDERED)
       SECTION
        DO I=1,L
a:       A(I) = ...
p:       POST(E(I))
        END DO
       SECTION
        DO J=1,M
         DO K=1,N
```

```
w:        WAIT(E(J+K))
b:        ... = A(J+K)
         END DO
        END DO
      END PARALLEL SECTIONS
```

Les instructions a et b vérifient :

- $\mathfrak{I}(a) = [\mathtt{I}]$; $\mathfrak{I}(b) = [\mathtt{J},\mathtt{K}]$
- $\mathfrak{N}(a,b) = [\,]$
- $\mathfrak{C}(a,b) = ||$
- $a\,T\,b = true$
- $\mathfrak{In}(b) = \{A(J+K)\}$
- $\mathfrak{Out}(a) = \{A(I)\}$
- $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(b) = \{A\}$.

La formule de dépendance est :

$$\mathrm{Dep}_{a,b}(x,y_1,y_2) = (1 \leqslant x \leqslant L) \wedge (1 \leqslant y_1 \leqslant M) \wedge (1 \leqslant y_2 \leqslant N) \wedge (x = y_1 + y_2)$$

La précédence de contrôle est :

$$\mathrm{Pre}^0_{a,b}(x,y_1,y_2) = false$$

Nous devons donc considérer les synchronisations. Le graphe de blocs est représenté sur la figure 7.2.

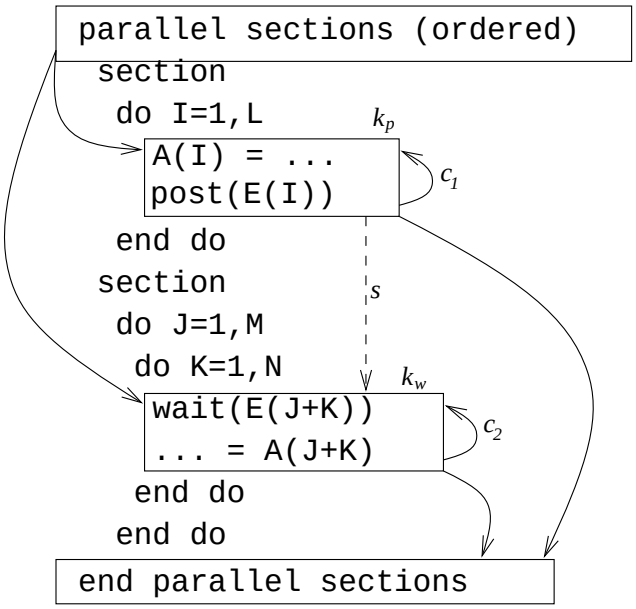

FIG. 7.2 – *Graphe de blocs : exemple 2*

On a $\mathrm{Exe}^{\mathbf{s}}_a(x) = (1 \leqslant x \leqslant L)$ et $\mathrm{Exe}^{\mathbf{s}}_b(y_1, y_2) = (1 \leqslant y_1 \leqslant M) \wedge (1 \leqslant y_2 \leqslant N)$. Les instructions a et b sont contenues respectivement dans les blocs k_p et k_w, et il y un arc de synchronisation s de k_p à k_w avec la formule $\mathrm{Sync}_{p,w}(x, y_1, y_2) = (x = y_1 + y_2)$; le plus court chemin allant de k_p à k_w est simplement constitué de l'arc s. La formule de précédence $\mathrm{Pre}_s(x, y_1, y_2)$ est $\mathrm{Pre}^0_{a,p}(x, z_1) \wedge \mathrm{Exe}_p(z_1) \wedge \mathrm{Sync}_{p,w}(z_1, z_2, z_3) \wedge \mathrm{Exe}_w(z_2, z_3) \wedge \mathrm{Pre}^0_{w,b}(z_2, z_3, y_1, y_2)$ avec

$$\begin{aligned}
\mathrm{Pre}^0_{a,p}(x, z_1) &= (x < z_1) \vee (x = z_1) \\
\mathrm{Pre}^0_{w,b}(z_2, z_3, y_1, y_2) &= (z_2 < y_1) \vee (z_2 = y_1 \wedge z_3 < y_2) \vee (z_2 = y_1 \wedge z_3 = y_2) \\
\mathrm{Sync}_{p,w}(z_1, z_2, z_3) &= (z_1 = z_2 + z_3)
\end{aligned}$$

Mise sous forme disjonctive, la formule Pre_s comporte six alternatives. Celle pour laquelle $x = z_1 \wedge z_2 = y_1 \wedge z_3 = y_2$ donne une formule de précédence identique à $\mathrm{Dep}_{a,b}(x, y_1, y_2)$. La vérification de « Dep $\Rightarrow$ Pre » est donc immédiate comme le montre la trace suivante :

```
parameter list = (n m l)

------- conflict:
- conflict on name = a
- statement A =
      a(i173) = x(i173)
- statement B =
      y(j172+k171) = a(j172+k171)
- type = wr
- head operator = par
- A T B = true
- common nest =
- dependence equation =
i173 = j172+k171
- iteration space for statement A =
1 <= i173
i173 <= l
- iteration space for statement B =
1 <= j172
j172 <= m
1 <= k171
k171 <= n
allpaths=((& &) (& & &))
testing path of length 2
```

```
path=(() 0)

alternative sys:
path problem:
- Eq:
0) y1 = z2
1) y0 = z1
2) z0 = x0
3) z1+z2 = z0
- GEq:
0) 1 <= z0
1) z0 <= l
2) 1 <= z1
3) z1 <= m
4) 1 <= z2
5) z2 <= n
dependence system:
0) y0+y1 = x0
0) y1 <= n
1) 1 <= y1
2) y0 <= m
3) 1 <= y0
4) x0 <= l
5) 1 <= x0
projected precedence system:
6) y0+y1 <= x0
7) x0 <= y0+y1
8) x0 <= n+y0
9) x0 <= l
10) 1 <= y0
11) y0 <= m
12) 1+y0 <= x0
gist: ()
```

C'est un cas de dépendance non uniforme, où les méthodes de parallélisation usuelles, qui commencent pas déterminer les dépendances, ne peuvent calculer de distance de dépendance ; ici le traitement formel des équations permet d'achever la preuve de correction.

7.3 Exemple 3 : synchronisation à travers des boucles

```
      PARALLEL SECTIONS (ORDERED)
       SECTION
```

```
        DO I=1,M
a:       A(2*I) = ...
p:       POST(EA(2*I-1))
w:       WAIT(EB(I))
b:       ... = B(I)
        END DO
       SECTION
        DO J=1,N
b':      B(3*J) = ...
p':      POST(EB(3*J))
w':      WAIT(EA(J+1))
a':      ... = A(J)
        END DO
      END PARALLEL SECTIONS
```

Intéressons nous aux instructions a et a' ; elles vérifient :

- $\mathfrak{I}(a) = [\mathtt{I}]$; $\mathfrak{I}(a') = [\mathtt{J}]$
- $\mathfrak{N}(a, a') = [\,]$
- $\mathfrak{C}(a, a') = ||$
- $a\,T\,a' = true$
- $\mathfrak{In}(a') = \{A(J)\}$
- $\mathfrak{Out}(a) = \{A(2 * I)\}$
- $\mathfrak{OutName}(a) \cap \mathfrak{InNames}(a') = \{A\}$.

La formule de dépendance entre a et a' est :

$$\mathrm{Dep}_{a,a'}(x, y) = (1 \leqslant x \leqslant M) \wedge (1 \leqslant y \leqslant N) \wedge (2x = y)$$

La précédence de contrôle est :

$$\mathrm{Pre}^0_{a,b}(x, y) = false$$

Nous devons donc considérer les synchronisations. Le graphe de blocs est représenté sur la figure 7.3.

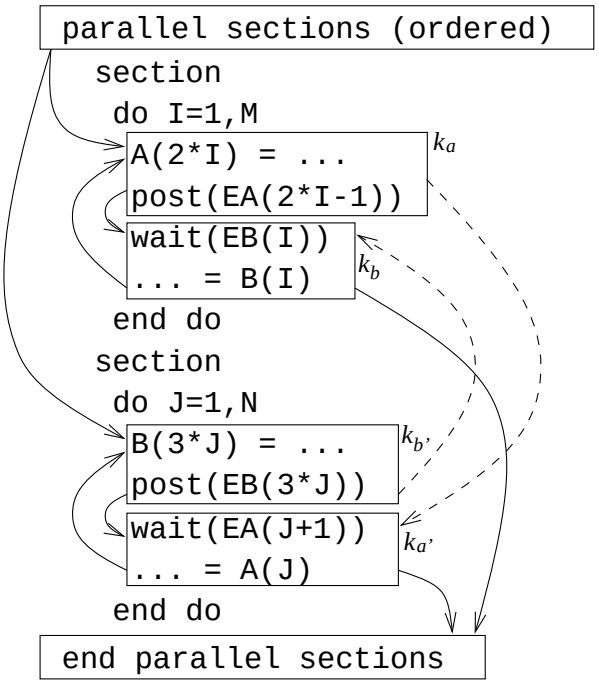

FIG. 7.3 – *Graphe de blocs : exemple 3*

À l'arc de synchronisation s entre le bloc k_a contenant a et le bloc $k_{a'}$ contenant a' correspond la formule de synchronisation

$$\mathrm{Sync}_{p,w}(x,y) = (2x - 1 = y + 1)$$

à ce chemin correspond la formule de précédence

$$\mathrm{Pre}_s(x,y,z_1,z_2) = \\ \mathrm{Pre}^0_{a,p}(x,z_1) \wedge \mathrm{Exe}^{\mathsf{s}}_p(z_1) \wedge \mathrm{Sync}_{p,w'}(z_1,z_2) \wedge \mathrm{Exe}^{\mathsf{s}}_{w'}(z_2) \wedge \mathrm{Pre}^0_{w',a'}(z_2,y)$$

i.e., le système

$$\begin{cases} 1 \leqslant x \leqslant M \\ 1 \leqslant z_1 \leqslant M \\ x \leqslant z_1 \\ 1 \leqslant z_2 \leqslant N \\ 2z_1 - 1 = z_2 + 1 \\ 1 \leqslant y \leqslant N \\ z_2 \leqslant y \end{cases}$$

La projection de ce système sur les variables x, y, M, N est calculée, puis son *gist* étant donné le système de dépendance :

```
--------- projection of precedence system
```

```
----------------------------------------
problem reduced:
variables = (x, y, m, n)
2 <= m
2 <= y
y <= n
1 <= x
x <= m
2x <= 2+y
----------------------------------------
dependence and projected precedence system:
dependence system:
0) y = 2x
0) 1 <= x
1) x <= m
2) 1 <= y
3) y <= n
precedence system:
4) 2 <= m
5) 2 <= y
6) y <= n
7) 1 <= x
8) x <= m
9) 2x <= 2+y
gist: (4)
2 <= m
```

Le résultat $2 \leqslant m$ est une condition sur le paramètre m pour que le programme soit correct du point de vue de la préservation des dépendances. Sans cette contrainte, l'emplacement mémoire $A(2)$ peut être lu par l'instance $\langle a', 1\rangle$ avant d'être écrit par l'instance $\langle a, 2\rangle$.

7.4 Exemple 4

```
      DO I=1,M
       A(I)=I
       POST(E(I))
      ENDDO
      PARALLEL SECTIONS (ORDERED)
       SECTION
        PARALLEL DO (ORDERED) I=M+1,3*M
w:       WAIT(E(I-M))
a:       A(I)=I*A(I-M)
p:       POST(E(I))
```

```
        END PARALLEL DO
       SECTION
        PARALLEL DO J=1,M+1
w':      WAIT(E(3*M+1-J))
        END PARALLEL DO
        PARALLEL DO I=3*M+1,3*M+N
a':      A(I)=I*A(I-M)
        END PARALLEL DO
      END PARALLEL SECTIONS
```

En ne s'intéressant qu'aux instructions qui peuvent être exécutées concurremment, nous avons quatre dépendances représentées sur la figure 7.4 avec le graphe de blocs. Les dépendances δ_1 (dépendance de flot) et δ_2 (anti-

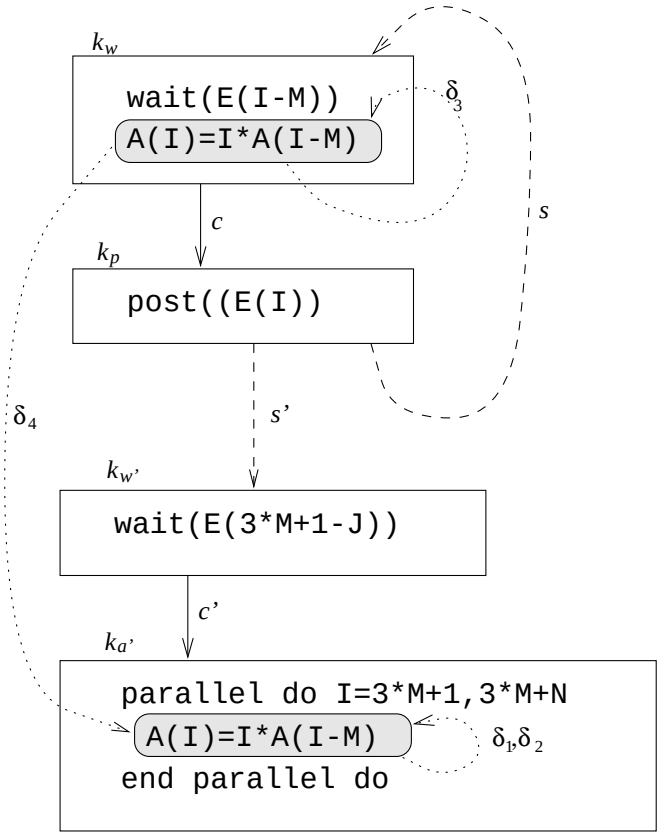

FIG. 7.4 – *Graphe de blocs et de dépendances : exemple 4*

dépendance) entre l'instruction a' et elle-même sont trivialement non préservées parce qu'il n'existe pas de chemin de précédence entre $k_{a'}$ et lui-même.

C'est ce qu'indique les traces suivantes :

```
------- conflict:
- conflict on name = a
- statement A =
      a(i162) = i162*a(i162-m)
- statement B =
      a(i162) = i162*a(i162-m)
- type = wr
- head operator = seq
- A T B = false
- common nest =
paralleldo i162
- dependence equation =
i162 = -m+i162
- iteration space for statement A =
1+ 3m <= i162
i162 <= 3m+n
- iteration space for statement B =
1+ 3m <= i162
i162 <= 3m+n

allpaths=()

------- conflict:
- conflict on name = a
- statement A =
      a(i162) = i162*a(i162-m)
- statement B =
      a(i162) = i162*a(i162-m)
- type = rw
- head operator = seq
- A T B = false
- common nest =
paralleldo i162
- dependence equation =
-m+i162 = i162
- iteration space for statement A =
1+ 3m <= i162
i162 <= 3m+n
- iteration space for statement B =
1+ 3m <= i162
i162 <= 3m+n
```

```
allpaths=()
```

Pour la dépendance $\delta_3 : a \rightarrow a$, nous devons étudier les chemins utilisant les synchronisations car la précédence de contrôle est $\mathrm{Pre}^0_{a,a}(x, y) = \mathit{false}$. La trace suivante montre que la précédence associée au chemin le plus court, cs, assure la préservation de la dépendance :

```
------- conflict:
- conflict on name = a
- statement A =
      a(i160) = i160*a(i160-m)
- statement B =
      a(i160) = i160*a(i160-m)
- type = wr
- head operator = seq
- A T B = false
- common nest =
paralleldo i160
- dependence equation =
i160 = -m+i160
- iteration space for statement A =
1+m <= i160
i160 <= 3m
- iteration space for statement B =
1+m <= i160
i160 <= 3m

allpaths=((& & &) (& & & & &) (& & & & & & &) (& & & & & & & & &))
testing path of length 3
path=(() pre0 0)

alternative sys:
path problem:
- Eq:
0) y0 = z1
1) z0 = x0
2) z1 = m+z0
- GEq:
0) 1+m <= z0
1) z0 <= 3m
2) 1+m <= z1
3) z1 <= 3m
dependence system:
0) y0 = m+x0
```

```
0) 1+x0 <= y0
1) y0 <= 3m
2) 1+m <= y0
3) x0 <= 3m
4) 1+m <= x0
projected precedence system:
5) y0 <= m+x0
6) m+x0 <= y0
7) 1+m <= x0
8) x0 <= 2m
gist: ()
```

La préservation de $\delta_4 : a \rightarrow a'$ n'est pas assurée par le contrôle car $\mathrm{Pre}^0_{a,a'}(x,y) = \mathit{false}$ mais il existe des chemins passant par les synchronisations. La trace suivante montre que la précédence associée au chemin le plus court, $cs'c'$, garantit la préservation de la dépendance :

```
------- conflict:
- conflict on name = a
- statement A =
      a(i144) = i144*a(i144-m)
- statement B =
      a(i146) = i146*a(i146-m)
- type = wr
- head operator = par
- A T B = true
- common nest =
- dependence equation =
i144 = -m+i146
- iteration space for statement A =
1+m <= i144
i144 <= 3m
- iteration space for statement B =
1+ 3m <= i146
i146 <= 3m+n
- status = ()
- preserved = ()
allpaths=((& & & &) (& & & & & &) (& & & & & & & &))
%%%% testing path of length 4
path=((() . &) (pre0 . &) (1 . &) (pre0 . &))

alternative sys:
path problem:
- Eq:
```

```
0) z0 = x0
1) 1+3m = z0+z1
- GEq:
0) 1+m <= z0
1) z0 <= 3m
2) 1 <= z1
3) z1 <= m
dependence system:
0) y0 = m+x0
0) y0 <= n+3m
1) 1+3m <= y0
2) x0 <= 3m
3) 1+m <= x0
projected precedence system:
4) 1+2m <= x0
5) x0 <= 3m
gist: ()
```

7.5 Exemple 5

```
      PARALLEL DO (ORDERED) I = 1, N
w:     WAIT(E(I-1))
b:     ...  = A(I-1) + A(N+1-I)
a:     A(I) = ...
p:     POST(E(I))
      END PARALLEL DO
```

L'analyse des dépendances montre qu'il existe une anti-dépendance δ_1 de b vers a dont la formule est :

$$\mathrm{Dep}_{\delta_1}(x,y) = (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x \leqslant y) \wedge (N+1-x = y)$$

et deux dépendances de flot, δ_2 et δ_3 de a vers b dont les formules sont :

$$\begin{aligned} \mathrm{Dep}_{\delta_2}(x,y) &= (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x = N+1-y) \\ \mathrm{Dep}_{\delta_3}(x,y) &= (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x < y) \wedge (x = y-1) \end{aligned}$$

La preuve de la préservation de δ_3 ne pose pas de problème particulier au système ; un cas analogue est décrit en détail à l'exemple 1. Nous n'y reviendrons pas. Nous allons nous intéresser au cas de la préservation de δ_2. La précédence de contrôle est :

$$\mathrm{Pre}^0_{a,b}(x,y) = \mathit{false}$$

Nous devons donc considérer les synchronisations. Le graphe de blocs est représenté sur la figure 7.5.

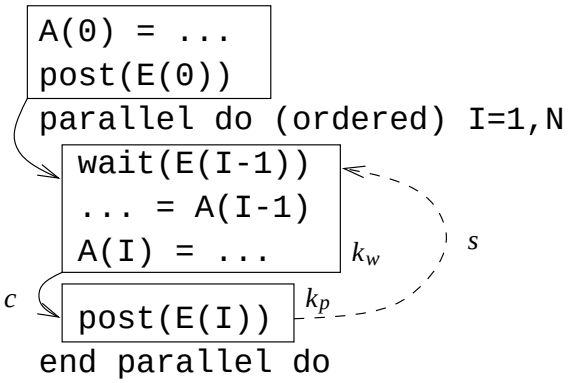

FIG. 7.5 – *Graphe de blocs : exemple 1*

Il est identique à celui de l'exemple 1. Les chemins du graphe allant de a à b sont de la forme $cs(cs)^*$. Nous allons calculer la formule Pre_π, sa projection sur les variables universelles et les paramètres et le *gist* du système projeté étant donné le système de contraintes fourni par la formule de dépendance, successivement pour les chemins $\pi = cs, cscs, cscscs$ et $cscscscs$.

Pour $\pi = cs$, nous obtenons la trace suivante :

```
dependence system:
0) 1+n = x0+y0
0) 1+x0 <= y0
1) y0 <= n
2) 1 <= y0
3) x0 <= n
4) 1 <= x0
projected precedence system:
5) y0 <= 1+x0
6) 1+x0 <= y0
7) 1 <= x0
8) 1+x0 <= n
gist: (5)
y0 <= 1+x0
```

Pour $\pi = cscs$, nous obtenons la trace suivante :

```
dependence system:
0) 1+n = x0+y0
0) 1+x0 <= y0
1) y0 <= n
2) 1 <= y0
3) x0 <= n
4) 1 <= x0
```

```
projected precedence system:
5) y0 <= 2+x0
6) 2+x0 <= y0
7) 1 <= x0
8) 2+x0 <= n
gist: (6 5)
2+x0 <= y0
y0 <= 2+x0
```

Pour $\pi = cscscs$, nous obtenons la trace suivante :

```
dependence system:
0) 1+n = x0+y0
0) 1+x0 <= y0
1) y0 <= n
2) 1 <= y0
3) x0 <= n
4) 1 <= x0
precedence system:
5) y0 <= 3+x0
6) 3+x0 <= y0
7) 1 <= x0
8) 3+x0 <= n
gist: (6 5)
3+x0 <= y0
y0 <= 3+x0
```

Pour $\pi = cscscscs$, nous obtenons la trace suivante :

```
dependence system:
0) 1+n = x0+y0
0) 1+x0 <= y0
1) y0 <= n
2) 1 <= y0
3) x0 <= n
4) 1 <= x0
precedence system:
5) y0 <= 4+x0
6) 4+x0 <= y0
7) 1 <= x0
8) 4+x0 <= n
gist: (6 5)
4+x0 <= y0
y0 <= 4+x0
```

Nous pourrions continuer comme cela en prenant des chemins de plus en plus longs mais ça ne servirait à rien. Pourtant la dépendance est bien

préservée puisque les synchronisations rendent la boucle parallèle complètement séquentielle. Nous voyons là une limite de notre méthode qui consiste à énumérer successivement des chemins de longueurs finies. Si nous pouvions considérer la formule associée à la clôture transitive $cs(cs)^*$, nous pourrions prouver la formule de préservation Dep $\Rightarrow$ Pre. Nous verrons au chapitre 8 comment on peut exprimer une formule de précédence à partir d'une expression régulière de chemin et ce que l'on peut en faire dans certains cas favorables.

Chapitre 8

Une amélioration

Dans ce chapitre, nous montrons comment dans certains cas il est possible d'obtenir une réponse alors que par la méthode exposée précédemment nous ne pouvions en obtenir. La principale différence tient au fait que nous allons considérer l'ensemble des chemins entre l'instruction source de la dépendance et l'instruction cible là où nous ne prenions que les chemins de longueur bornée. Dans les cas où il est possible d'exprimer la formule logique correspondant à ces chemins sous la forme d'une formule de Presburger, nous pouvons conclure sur la préservation ou non de la dépendance étudiée. Nous le montrons sur un exemple simple mais significatif en utilisant les possibilités du calculateur Omega pour prouver la satisfaisabilité de l'implication Dep $\Rightarrow$ Pre.

8.1 Problème

Reprenons l'exemple 7.5 de la page 101 :

```
      PARALLEL DO (ORDERED) I = 1, N
       WAIT(E(I-1))
a:     ...  = A(I-1) + A(N+1-I)
b:     A(I) = ...
       POST(E(I))
      END PARALLEL DO
```

le couple `POST`/`WAIT` assure trivialement la séquentialisation complète de la boucle. Il est cependant impossible de prouver que les dépendances entre les instances $\langle a, i\rangle$ et $\langle b, N+1-i\rangle$ et entre les instances $\langle b, i\rangle$ et $\langle a, N+1-i\rangle$ sont préservées par les synchronisations.

Prenons, par exemple, le cas de la dépendance entre a et b, dont la formule est :

$$\mathrm{Dep}_{a,b}(x, y) = (1 \leqslant x \leqslant N) \wedge (1 \leqslant y \leqslant N) \wedge (x \leqslant y) \wedge (N + 1 - x = y)$$

La boucle est parallèle, donc nous devons considérer des chemins passant par les synchronisations. Le graphe de blocs de cet exemple est dessiné figure 8.1. L'ensemble des chemins du bloc K_w à lui-même peut être représenté par l'expression régulière $(1 \cdot 2)^*$ (voir section 8.2). La méthode par énumération de chemins de longueur croissante nous amène à étudier les chemins $1 \cdot 2, 1 \cdot 2 \cdot 1 \cdot 2, 1 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 2, \ldots$ jusqu'à une longueur limite l donnée. Mais aucun de ces chemins ne permet de satisfaire la condition de préservation. Dans ce cas, une approximation de la précédence par un chemin fini ne suffit pas, il faudrait pouvoir associer une formule logique à l'expression $(1 \cdot 2)^*$.

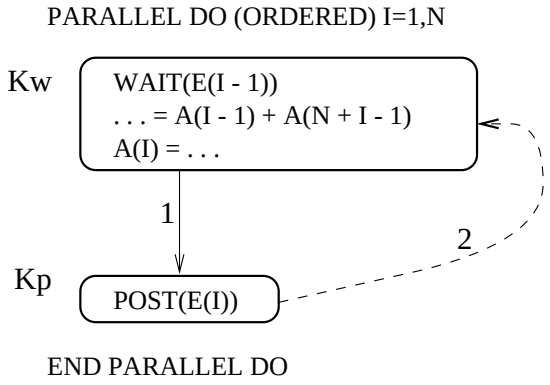

FIG. 8.1 – *Graphe de blocs*

8.2 Calcul des chemins

L'ensemble des chemins entre une instruction a et une instruction b peut être synthétisé par une expression régulière [38] dont la syntaxe est présentée ci dessous :

$$R := \Lambda \mid \varnothing \mid a \mid R^* \mid R \cdot R \mid R + R$$

où a représente un arc, Λ le chemin vide, $\varnothing$ l'absence de chemin et où les opérations $\cdot, ^*, +$ symbolisent respectivement la concaténation, la clôture rélexive et transitive et la réunion. La présence de l'opérateur * dans une expression régulière produit une infinité de chemins.

L'utilisation d'expressions régulières pour représenter des chemins dans un graphe ainsi que l'algorithme de construction des expressions de chemins proviennent de [36, 35].

Pour le fragment de programme ci-dessous, l'ensemble des chemins entre le sommet a et le sommet d est

$$P = ((5 \cdot (4 \cdot 3)^* \cdot 4 \cdot 6 + 1) \cdot 2)^* \cdot 5 \cdot (4 \cdot 3)^*$$

```
      PARALLEL SECTIONS (ORDERED)
       SECTION
        DO I=1,M
a:       A(2*I) = ...
p:       POST(EA(2*I-1))
w:       WAIT(EB(I))
b:       ... = B(I)
        END DO
       SECTION
        DO J=1,N
c:      B(3*J) = ...
p':      POST(EB(3*J))
w':      WAIT(EA(J+1))
d:      ... = A(J)
        END DO
       END PARALLEL SECTIONS
```

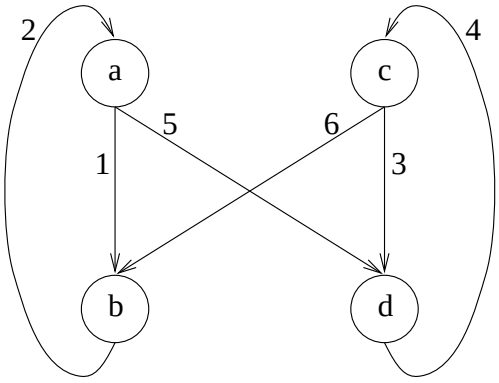

FIG. 8.2 – *Graphe de blocs simplifié*

Dans les sections qui suivent, nous allons présenter le calcul de la formule logique associée à une expression régulière. Pour cela, nous emploierons un formalisme empruntée à la Sémantique Naturelle que nous décrivons brièvement ci-dessous.

8.3 Sémantique naturelle

En Sémantique Naturelle [23], une proposition est associée à des hypothèses dans un *séquent*:

$$hypothèses \vdash conséquent$$

où *hypothèses* est une liste d'expressions et *conséquent* une proposition. Par exemple, pour exprimer qu'une expression arithmétique e comportant des variables a pour valeur v dans l'environnement ρ, nous écrirons :

$$\rho \vdash e : v$$

Le type d'un séquent est déclaré par un *jugement*. Par exemple, le jugement :

$$ENV \vdash EXPR : INT$$

indique dans quels ensembles sont définis les composants du séquent précédent.

Une règle d'inférence a la forme suivante :

$$\text{(sujet de la règle)} : \frac{prémisses}{conclusion}$$

le numérateur est une liste de séquents, le dénominateur un seul séquent et quand toutes les *prémisses* sont vraies, la *conclusion* l'est aussi. Un axiome est une règle qui ne comporte pas de prémisses.

8.4 Formule de précédence associée à un arc

Nous reformulons ici l'expression de la précédence associée à chaque arc du graphe de blocs en utilisant le formalisme de la sémantique naturelle présenté ci-dessus.

Jugement :

$$C \vdash F : A$$

$c \in C$ est un ensemble de tuples de variables : $c = \boldsymbol{x}_1 : v_1, \ldots, \boldsymbol{x}_n : v_n$. Chaque tuple $\boldsymbol{x}_i$ est associé à un sommet v_i du graphe $\mathcal{BG}$ et est distinct de tous les autres ; il a pour longueur $|\,\mathfrak{N}(v_i)|$.

$e \in F$ est une formule de la logique du premier ordre sans quantificateur sur l'arithmétique de Presburger.

$u^{a \to b} \in A$ est un arc du graphe de blocs qui relie le sommet a au sommet b.

Axiomes :

$$(\textbf{seq}) : \frac{}{\boldsymbol{x} : a, \boldsymbol{y} : b \vdash \boldsymbol{x}_{|\,\mathfrak{N}(a,b)} = \boldsymbol{y}_{|\,\mathfrak{N}(a,b)} : u^{a \to b}}$$

si u est un arc de contrôle simple.

$$(\textbf{backloop}) : \frac{}{\boldsymbol{x} : a, \boldsymbol{y} : b \vdash (\boldsymbol{x'} = \boldsymbol{y'}) \wedge (x + 1 = y) : u^{a \to b}}$$

$$\text{où } \boldsymbol{x'}@[x] = \boldsymbol{x}_{|\,\mathfrak{N}(a,b)} \text{ et } \boldsymbol{y'}@[y] = \boldsymbol{y}_{|\,\mathfrak{N}(a,b)}$$

si u est un arc de retour de boucle `DO`.

$$(\textbf{synchro}) : \frac{}{\boldsymbol{x} : a, \boldsymbol{y} : b \vdash \mathrm{Exe}^{\mathsf{s}}_u(\boldsymbol{x}) \wedge \mathrm{Sync}_{u,v}(\boldsymbol{x}, \boldsymbol{y}) \wedge \mathrm{Exe}^{\mathsf{s}}_v(\boldsymbol{y}) : u^{a \to b}}$$

si u est un arc de synchronisation.

8.5 Formule associée à une expression régulière

Les règles d'inférences présentées ci-dessous, complétées par les axiomes précédents, relient les expressions régulières aux formules logiques de précédence.

Jugement :

$$C \vdash F : R$$

$c \in C$ est un ensemble de tuples de variables comme précédemment.

$e \in F$ est une formule de la logique du premier ordre sur l'arithmétique de Presburger.

$r \in R$ est une expression régulière représentant un chemin entre deux sommets du graphe de blocs.

Règles :

$$(\varnothing) : \frac{}{\vdash \mathit{false} : \varnothing}$$

$$(\Lambda) : \frac{}{\boldsymbol{x}, \boldsymbol{y} : a \vdash \boldsymbol{x} = \boldsymbol{y} : \Lambda^a}$$

$$(\cdot) : \frac{\boldsymbol{x} : a, \boldsymbol{z}_1 : c \vdash e_1 : r_1^{a \to c} \quad \boldsymbol{z}_2 : c, \boldsymbol{y} : b \vdash e_2 : r_2^{c \to b}}{\boldsymbol{x} : a, \boldsymbol{y} : b \vdash \exists \boldsymbol{t}(e_1[\boldsymbol{t}/\boldsymbol{z}_1] \wedge e_2[\boldsymbol{t}/\boldsymbol{z}_2]) : (r_1 \cdot r_2)^{a \to b}}$$

si $\boldsymbol{t}$ n'apparaît ni dans e_1 ni dans e_2

$$(+) : \frac{\boldsymbol{x} : a, \boldsymbol{y} : b \vdash e_1 : r_1^{a \to b} \quad \boldsymbol{z} : a, \boldsymbol{t} : b \vdash e_2 : r_2^{a \to b}}{\boldsymbol{x} : a, \boldsymbol{y} : b \vdash e_1 \vee e_2[\boldsymbol{x}/\boldsymbol{z}, \boldsymbol{y}/\boldsymbol{t}] : (r_1 + r_2)^{a \to b}}$$

$$(*) : \frac{\boldsymbol{x}, \boldsymbol{y} : a \vdash e : \Lambda^a + r^{a \to a} \cdot (r^*)^{a \to a}}{\boldsymbol{x}, \boldsymbol{y} : a \vdash e : (r^*)^{a \to a}}$$

Donc, si nous pouvons prouver $c \vdash e : r$, nous posons :

$$\mathrm{Pre}_r = e$$

L'emploi de la règle $(*)$ induit une branche infinie dans l'arbre de preuve. Une proposition dans laquelle l'expression de chemin contient l'opérateur de clôture transitive réflexive n'est donc pas prouvable.

Nous donnons, figure 8.3, un exemple d'application des règles pour le chemin $P = (u^*)^{a\to a}$ où u est un arc de retour de boucle `DO`. La figure ne montre que deux itérations de la règle $(*)$.

$x, y : a \vdash (x = y) \vee \exists t_2((t_2 = x + 1) \wedge ((t_2 = y) \vee \exists t((t = t_2 + 1) \wedge e[t/z_3, t_2/z_2]))) :$
$(u^*)^{a\to a}$
par la règle $(*)$
$x, y : a \vdash (x = y) \vee \exists t_2((t_2 = x + 1) \wedge ((t_2 = y) \vee \exists t((t = t_2 + 1) \wedge e[t/z_3, t_2/z_2]))) :$
$\Lambda^a + u^{a\to a} \cdot (u^*)^{a\to a}$
par la règle $(+)$
$x, y : a \vdash \exists t_2(t_2 = x + 1 \wedge ((t_2 = y) \vee \exists t((t = t_2 + 1) \wedge e[t/z_3, t_2/z_2]))) :$
$u^{a\to a} \cdot (u^*)^{a\to a}$
par la règle $(\cdot)$
$z_2, y : a \vdash (z_2 = y) \vee \exists t((t = z_2 + 1) \wedge e[t/z_3]) : (u^*)^{a\to a}$
par la règle $(*)$
$z_2, y : a \vdash (z_2 = y) \vee \exists t((t = z_2 + 1) \wedge e[t/z_3]) : \Lambda^a + u^{a\to a} \cdot (u^*)^{a\to a}$
par la règle $(+)$
$z_2, y : a \vdash \exists t((t = z_2 + 1) \wedge e[t/z_3]) : u^{a\to a} \cdot (u^*)^{a\to a}$
par la règle $(\cdot)$
$z_3, y : a \vdash e : u^{a\to a}$
par la règle $(*)$
$z_2, z_1 : a \vdash z_1 = z_2 + 1 : (u^*)^{a\to a}$
par la règle (`DO`)
$z, t : a \vdash z = t : \Lambda^a$
par la règle (Λ)
$x, z_1 : a \vdash z_1 = x + 1 : u^{a\to a}$
par la règle (`DO`)
$z, t : a \vdash z = t : \Lambda^a$
par la règle (Λ)

FIG. 8.3 – *Dérivation de* $(u^*)^{a\to a}$ *où* u *est un arc de retour de boucle* `DO`

La présence de l'opérateur de clôture réflexive transitive dans une expression de chemin ne permet pas, dans le cas général, de déduire la formule de précédence associée à l'expression. Toutefois, dans certains cas, il est possible d'exprimer la clôture transitive par une formule finie. Si celle-ci est linéaire, le problème de préservation de la dépendance sera solvable avec les outils classiques de la programmation linéaire en nombres entiers. C'est ce que nous allons illustrer en traitant complètement l'exemple présenté à la section 8.1. Le problème de la satisfaisabilité de l'implication sera soumis au calculateur Omega qui est le successeur de l'Omega-test présenté au chapitre 6.

8.6 Le calculateur et la bibliothèque Omega

Le calculateur Omega [24] est une interface de type « texte » à la bibliothèque Omega. La bibliothèque Omega rassemble des fonctions C++ qui permettent de manipuler des relations entre n-uplets d'entiers ou des ensembles de n-uplets d'entiers. Ces relations et ensembles sont décrits par des formules de Presburger, un sous-ensemble des formules de logique du premier ordre construit à partir de contraintes affines sur des variables entières, des connecteurs logiques $\neg$, $\wedge$ et $\vee$, et des quantificateurs $\forall$ et $\exists$. Les formules peuvent aussi contenir des variables libres, ce qui permet d'utiliser des relations paramétrées.

Le langage permet de définir de nouvelles relations à partir de relations existantes grâce à des opérateurs relationnels. Pour des relations simples, le système est capable d'exprimer exactement sa clôture transitive. Nous allons nous servir de cette caractéristique pour résoudre notre problème de préservation sur un exemple que nous ne pouvions pas traiter par la méthode présentée au chapitre 5.

8.7 Vérification de la formule de correction

Soit Dep la formule de dépendance et soit Pre la formule de précédence obtenue par application des règles d'inférence à l'expression régulière de chemin entre a et b.

Pour verifier si Dep $\Rightarrow$ Pre, nous pouvons tester si Dep est un sous-ensemble de Pre en soumettant Dep `subset` Pre au calculateur Omega. La réponse obtenue est binaire (vrai/faux) ; si elle est «vrai» alors la dépendance est préservée. Pour avoir davantage d'informations, nous pouvons calculer la différence Dep $-$ Pre ; si elle est vide, ce qui s'exprime dans le langage du calculateur Omega par la relation $\{\boldsymbol{x} \rightarrow \boldsymbol{y} : \mathit{false}\}$, alors la dépendance est préservée sinon la relation résiduelle retournée indique l'ensemble des couples $(\boldsymbol{x}, \boldsymbol{y})$ de l'espace d'itération du couple d'intructions (a, b) pour lesquels la dépendance n'est pas préservée. Enfin, il peut être intéressant de calculer `gist` Pre `given` Dep comme auparavant si l'on espère une relation intéressante portant sur les paramètres du programme.

Nous allons appliquer cette méthode au cas de l'exemple présenté en 8.1. La formule de dépendance entre l'instruction a et l'instruction b se traduit dans le langage du calculateur Omega par la relation suivante :

$$\mathrm{Dep} = \{[x] \rightarrow [y] : (1 \leqslant x, y \leqslant N) \wedge (x \leqslant y) \wedge (N + 1 - x = y)\}$$

où x (resp. y) est une variable entière associée au vecteur d'itération de l'instruction a (resp. b) et N le paramètre du programme.

Les chemins de précédence entre a et b, exprimés en utilisant le graphe

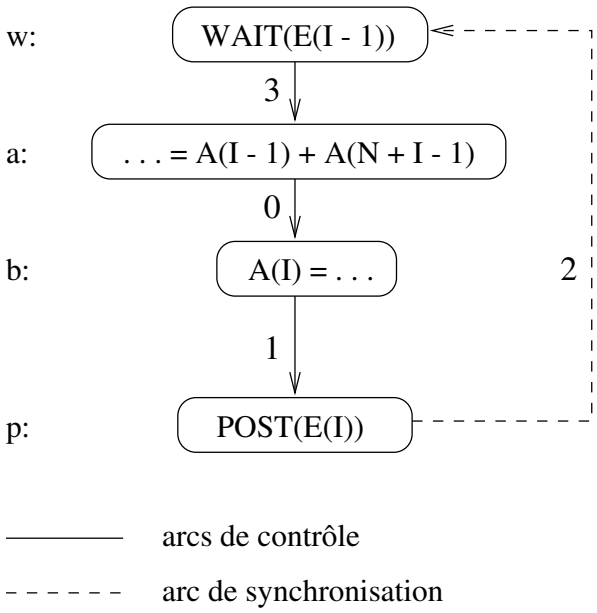

FIG. 8.4 – *Graphe de précédence*

de précédence représenté figure 8.4, sont de la forme :

$$a \xrightarrow{0} b$$
$$a \xrightarrow{0} b \xrightarrow{1} p \xrightarrow{2} w \xrightarrow{3} a \xrightarrow{0} b$$
$$a \xrightarrow{0} b \xrightarrow{1} p \xrightarrow{2} w \xrightarrow{3} a \xrightarrow{0} b \xrightarrow{1} p \xrightarrow{2} w \xrightarrow{3} a \xrightarrow{0} b$$
$$\ldots$$

Ils sont décrits par l'expression régulière suivante :

$$(0 \cdot 1 \cdot 2 \cdot 3)^* \cdot 0$$

Les formules de Presburger associées aux arcs, exprimées sous forme de relations, sont les suivantes :

$$\begin{array}{rl} 0, 1, 3 : & \{[x] \to [y] : x = y\} \\ 2 : & \{[x] \to [y] : (1 \leqslant x, y \leqslant N) \wedge (x = y - 1)\} \end{array}$$

En remarquant que pour les arcs $0, 1, 3$ la relation est l'identité, nous en déduisons que la relation de précédence est :

$$\mathrm{Pre} = \{[x] \to [x]\} \cup \{[x] \to [y] : (1 \leqslant x, y \leqslant N) \wedge (x = y - 1)\}+$$

où le signe $+$ indique la clôture transitive de la relation.

Si nous soumettons ce problème au calculateur Omega, nous obtenons la trace suivante :

```
# Omega Calculator [v1.00, Mar 96]:
# #Exemple 1
# #Chemin exprimé par une expression régulière
#
# symbolic N;
#
#
# dep := [x] -> [y] : 1<= x,y <= N and x <= y and N+1-x = y;
#
# pre := [x] -> [x] union [x] -> [y] : 1<= x,y <= N and x = y-1+;
#
#
# dep;

[x] -> [N-x+1] : 1 <= x && 2x <= 1+N

#
# pre;

[x] -> [Out_1] : 1 <= x < Out_1 <= N union
 [x] -> [x]

#
# dep subset pre;

True
#
# dep - pre;

[x] -> [y]  : FALSE

#
#
```

8.8 Limite de la méthode et conclusion

Cette méthode est limitée par le fait qu'on ne sait pas toujours exprimer la clôture transitive d'une relation et dans les cas où on le sait, la clôture transitive d'une relation contrainte par une formule de Presburger fournit en général une relation contrainte par une formule arithmétique non linéaire.

Par exemple, soit la relation

$$L = \{[x] \to [y] : y = ax + b\}$$

définie par une formule arithmétique linéaire dans laquelle a et b sont des constantes symboliques, sa clôture transitive peut s'exprimer par la relation

$$L^+ = \{[x] \to [y] : \exists n > 0\,(y = a^n x + nb)\}$$

L'introduction d'une variable en exposant dans la formule nous place en dehors du cadre de l'arithmétique de Presburger. Les méthodes de résolution de problèmes de programmation linéaire en nombres entiers ne sont plus applicables.

Reprenons la relation L précédente pour $a = 1$, appelons-la L_1 :

$$L_1 = \{[x] \to [y] : y = x + b\}$$

sa clôture transitive est

$$L_1^+ = \{[x] \to [y] : \exists p > 0\,(y = x + pb)\}$$

elle est encore définie par une formule de Presburger, mais la clôture transitive de L_1^+ est

$$L_1^{++} = \{[x] \to [y] : \exists q > 0\,(\exists p > 0\,(y = x + qpb))\}$$

sa formule n'appartient plus à l'arithmétique de Presburger à cause du produit qp.

L'exemple du « papillon » (Cf. 8.2) montre un cas où un double niveau d'étoiles apparaît dans l'expression régulière représentant l'ensemble des chemins allant de l'instruction source de la dépendance à l'instruction cible. Nous ne pouvons donc pas associer une formule de Presburger à cette expression régulière.

La contrainte de linéarité des expressions d'indice de tableau et de bornes de boucle ne suffit pas pour assurer la linéarité des formules logiques résultant du calcul de la précédence généralisée. Par conséquent, la solvabilité du problème de préservation d'une dépendance n'est pas toujours possible en utilisant les outils informatiques conçus pour résoudre des problèmes de programmation linéaire en nombres entiers.

Chapitre 9

Conclusion

Notre but était d'étudier le problème de la vérification de programmes parallèles développées pour le calcul scientifique. Le domaine d'application nous a conduits à choisir les extensions X3H5 de Fortran 77 en cours de normalisation à l'ANSI. Ce projet de norme décrit un modèle de parallélisme asynchrone et à mémoire partagée implémentable sur un grand nombre d'architectures parallèles.

Nous nous intéressons à une propriété de correction sémantique encore peu abordée. La notion de correction étudiée, qui est l'équivalence entre le programme parallèle et sa version séquentielle, bien qu'artificielle dans le cadre général de la programmation parallèle, semble appropriée dans le cas des programmes conçus pour le calcul scientifique. Un sous-ensemble significatif des constructions parallèles proposées par X3H5 a été retenu : sections et boucles parallèles et synchronisations par événements. En revanche, les sections critiques ont été écartées car il n'est pas toujours possible de déduire une version séquentielle d'un programme qui les utilise.

Les techniques utilisées sont souvent celles bien connues en parallélisation, notamment le calcul des dépendances. Cependant, la perspective est différente, puisqu'il faut prouver que les dépendances qui existent dans la version séquentielle sont préservées par les constructions parallèles et les synchronisations, ceci afin d'éviter des situations de course qui conduiraient à des données incohérentes. L'analyse est rendue délicate par le fait que, tant que l'on n'a pas prouvé que toutes les dépendances sont préservées, on ne peut même pas supposer qu'une expression a une valeur bien déterminée. Nous avons donc imposé un certain nombre de restrictions syntaxiques, qui correspondent pour la plupart à un style de programmation recommandable. Les restrictions les plus fortes proviennent toutefois de la partie séquentielle, puisque les boucles `WHILE` et les `GOTO` ne sont pas traités. Sur le fragment ainsi constitué, on parvient à réaliser une analyse formelle du programme, au sens où l'on construit des formules arithmétiques qui doivent être prouvées. Ces formules peuvent être vues comme des « contraintes» qu'il s'agit de véri-

fier ou de simplifier ; il suffit de vérifier l'une de ces formules pour assurer que ces contraintes sont respectées par le contrôle et les synchronisations. Pour ça, nous cherchons à prouver la satisfaisabilité d'une formule de la forme Dep $\Rightarrow$ Pre faisant intervenir le prédicat de dépendance et un prédicat de précédence que nous définissons.

Nous avons explicité les conditions d'emploi des synchronisations pour qu'elles puissent établir une précédence. Moyennant ces hypothèses, nous avons défini une précédence générale sur les programmes parallèles, qui combine la précédence de contrôle et la précédence due aux synchronisations. Cette dernière dépend du prédicat d'exécution attaché à chaque instruction impliquée dans une relation de synchronisation.

Plutôt que de déterminer d'abord les dépendances et de montrer qu'elles sont préservées, il nous a paru intéressant et plus puissant de construire globalement la formule exprimant cette préservation dans tous les cas où la syntaxe rend possible une dépendance, et de montrer que cette formule est valide. Ceci exige une part importante de manipulations formelles, mais dans des cas usuels, la formule sera trivialement vérifiable. Ces cas usuels incluent des dépendances non uniformes où l'autre approche serait infructueuse. La simplification des formules peut donner des informations utiles au programmeur, en particulier sur les relations entre les paramètres du programme qui peuvent garantir la correction.

Du point de vue algorithmique, la vérification d'une unité de programme conduit à construire un graphe, à énumérer des chemins, à construire des systèmes d'équations et d'inéquations linéaires assez importants, et à calculer une formule *gist* pour prouver l'implication. Le calcul de la précédence est optimisé par l'introduction d'un nouveau type de graphe dont les sommets sont des blocs d'instructions. La réduction et la vérification des systèmes d'équations et d'inéquations font appel au test Omega. Ce solveur est capable de fournir des solutions exactes à des problèmes de programmation linéaire en nombres entiers.

L'implémentation de ces différents algorithmes est intégré dans un environnement logiciel qui comprend un éditeur syntaxique, un « pretty printer » et un vérificateur de type sur l'ensemble de la syntaxe du Fortran 77 étendu à l'ensemble des constructions parallèles proposées par le comité X3H5 pour ce langage.

Un problème avec notre algorithme est qu'il utilise une approximation de de la formule de précédence qui, dans certains cas, ne permet pas de conclure de manière définitive quant à la préservation ou non d'une dépendance. Nous avons présenté une amélioration qui exploite la possibilité de représenter un ensemble de chemins par une expression régulière. Cependant, la formule associée à cette expression est rarement traitable car elle est généralement non linéaire.

Travaux en rapport avec cette thèse

Un travail plus théorique [7] fournit une preuve détaillée du théorème d'équivalence sémantique. Certaines restrictions syntaxiques sont levées : l'instruction `WHILE` est admise, les conditions booléennes des `IF` et des `WHILE`, les expressions des bornes de boucles et d'indice de tableau peuvent contenir des variables quelconques. Cependant les expressions d'indice de tableau contenues dans une instruction `WAIT` doivent seulement contenir des indices de boucles et des paramètres.

Un travail de thèse actuellement en cours a pour objectif d'une part la formalisation de la sémantique des constructions parallèles dans un calcul de processus (π-calcul) [18] et sa représentation dans un meta-système logique (Coq) [19]. D'autre part, cette représentation en Coq doit permettre de construire des preuves formelles de propriétés sémantiques sur le langage lui-même, dont le théorème d'équivalence sémantique.

Annexe A

Notations

Nid de boucle	$\mathfrak{N}$
Vecteur d'itération	$\mathfrak{I}$
Formule d'itération	Iter
Vecteurs d'entiers	$\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, \ldots$
Paramètres du programme	$\boldsymbol{p}$
Instructions	$a, b, c, \ldots$
Instances d'instructions	$\alpha, \beta, \gamma, \ldots$ $\langle a, \boldsymbol{i} \rangle$
Ordre lexical	T
Ordre lexicographique	$\ll$
Ordre séquentiel	$\prec$
Formule de dépendance	Dep
Graphe de dépendance	$\mathcal{DG}$
Graphe de flot de contrôle	$\mathcal{FG}$
Graphe de précédence	$\mathcal{PG}$
Graphe de blocs	$\mathcal{BG}$
En-tête de contrôle	$\mathfrak{C}$
Ordre lexicographique sur programme parallèle	$\ll_{\boldsymbol{n}}$
Ordre d'exécution sur programme parallèle	$\prec_{\boldsymbol{n},c}$
Formule de précédence de contrôle	Pre^0
Formule de synchronisation	Sync
Prédicat de précédence générale	Pre
Prédicat d'exécution	Exe
Prédicat d'exécution sur version séquentielle	$\mathrm{Exe}^{\mathsf{s}}$

Bibliographie

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] R. Allen. *Dependence analysis for subscripted variables and its application to program transformations*. PhD thesis, Rice University, Houston, Texas, April 1983.

[3] C. Amza et al. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[4] U. Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2:133–149, 1988.

[5] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the SIGSOFT'88, Third Annual Symposium on Software Development Environments*, Boston, 1988.

[6] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, March 1990. ACM Press.

[7] G. Caplain. Correctness properties in a control-parallel extension of Fortran. Technical Report 94-29, CERMICS, 1994.

[8] J. Chailloux. *Le-Lisp Version 15.22, le manuel de référence*. I.N.R.I.A., janvier 1989.

[9] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] B. Dion, L. Angeli, and A. Bravo Lastra. Paragraph : an interactive environment for parallelizing fortran programs. Technical Report RR-1920, INRIA, mai 1993.

[11] Paul Feautrier. Parallélisation et vectorisation automatique, état de l'art et recherches récentes. In Paul Feautrier and Gérard Noguez, editors, *Actes des Journées Firtech Systèmes et Télématique*, pages 1–20, Paris, novembre 1988.

[12] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

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

[14] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[15] M. C. Giboulot and F. Thomasset. Automatic parallelization of structured if statements without if conversion. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II (SHPC)*, pages 127–144, Amsterdam, October 1991. Elsevier Science Publishers B.V., North-Holland. Version étendue : RR–1408, INRIA, juin 1991.

[16] M.C. Giboulot, M. Loyer, G. Popovitch, and F. Thomasset. An interactive parallelizer under the Centaur environment. Technical report, ESPRIT-II, 1990.

[17] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168. ACM Press, May 1993.

[18] D. Hirschkoff. Bisimulation proofs for the pi-calculus in the calculus of constructions. Technical Report 96-62, CERMICS, 1996.

[19] D. Hirschkoff. Up-to context proofs for the pi-calculus in the Coq system. Technical Report 96-82, CERMICS, 1996.

[20] C. A. R. Hoare. Communicating sequential processes. *Communications of the Association for Computing Machinery*, 21(8):666–677, August 1978.

[21] INRIA, Sophia-Antipolis. *The Centaur Documentation – Version 1.0.*

[22] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. Crl: High-performance all-software distributed shared memory. In *ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 213–228. ACM Press, December 1995.

[23] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, number 247 in LNCS, Passau,

Germany, February 1987. Springer-Verlag. The paper is also available as INRIA Report 601, February, 1987.

[24] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library*, April 1996.

[25] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30, pages 196–204, June 1995.

[26] Bruce Leasure. *Parallel Processing Model for High Level Programming Languages*. ANSI Technical Committee X3H5, March 1993. (Proposed Standard).

[27] Bruce Leasure. *X3H5 Parallel Extensions for Fortran*. ANSI Technical Committee X3H5, February 1994.

[28] A. Lichnewsky and F. Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *International Conference on Supercomputing*, Saint Malo, France, June 1988. ACM.

[29] C. Pancake. *Parallel Processing Model for High Level Programming Languages*. ANSI, March 1992. (Proposed Standard).

[30] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[31] W. Pugh et al. *The Omega Test Users Manual*.

[32] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. Technical Report UMIACS-TR-92-114, Dept. of Computer Science, Univ. of Maryland, College Park, MD, December 1992.

[33] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):204–211, February 1995.

[34] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, practice and experience*, 2(4):315–339, December 1990.

[35] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[36] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[37] R. Triolet, P. Feautrier, and F. Irigoin. Automatic parallelization of fortran programs in the presence of procedure calls. In *ESOP 86 European Symposium on Programming*, number 213 in LNCS, Saarbrücken, Federal Republic of Germany, March 1986. Springer-Verlag.

[38] Mark Wegman. Summarizing graphs by regular expressions. In *Proc. of the 15th ACP POPL*, pages 203–216, January 1983.

[39] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, MA, 1989.

[40] X3H5. *FORTRAN 77 Binding of X3H5 Model for Parallel Programming Constructs.* ANSI, September 1992. (draft version).

[41] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.