Remerciements

Je tiens tout d'abord à remercier Pierre Boullier qui a accepté de diriger ma thèse, avec toutes les signatures que cela pouvait comporter, et également de présider ce jury.

Je remercie Martin Jourdan qui m'a encadré pendant ces trois ans et qui m'a fait connaître et presque apprécier les grammaires attribuées.

Je remercie également Didier Parigot qui a su me guider dans mon implantation et me montrer les méandres de FNC-2, mais surtout d'avoir supporté avec patience les nombreuses critiques que j'ai pu lui en faire avec les éclats de voix qui les accompagnaient.

Je remercie Paul Franchi-Zannettacci et Catherine Roucairol d'avoir accepté d'être mes rapporteurs malgré le peu de temps dont ils disposent. Je tiens à remercier également François Le berre et Matthijs Kuiper qui ont bien voulu participer à mon jury.

Je remercie également tous les membres du projet Chloé pour leur accueil chaleureux et qui, par les problèmes qu'ils m'ont si souvent soumis, m'ont permis de progresser dans la voie du système, sans oublier Philippe Deschamp qui m'a appris une grande partie de ce que je sais dans ce domaine. Je remercie aussi les nombreux correcteurs orthographiques qui ont bien voulu lire cette thèse et qui l'ont expurgée d'un grand nombre de coquilles...

Je remercie tous les différents stagiaires et thésards qui sont ou ont été dans le projet avec moi, et qui ont su égayer certains moments de dur labeur !

Mais surtout je dédie cette thèse à Nathalie, Antoine et Thomas qui ont su me soutenir pendant ces trois ans.

Enfin cette thèse n'est dédiée ni aux bugs systèmes ni aux nombreux bugs que j'ai moi même, par trop de bonté, introduit dans le code.

Table des matières

Table des figures

Introduction

De plus en plus de constructeurs proposent dans leur gamme des machines parallèles, pour un coût qui, s'il reste élevé, devient de plus en plus abordable. Malheureusement, si les machines parallèles ont fait de gros progrès, il reste aux logiciels encore beaucoup de chemin à parcourir avant de pouvoir utiliser le potentiel de ces multiprocesseurs. Le point crucial concerne la génération automatique de programmes parallèles à partir de spécifications où n'apparaît pas la notion de parallélisme. En effet, écrire directement un programme parallèle demande beaucoup d'efforts et de précautions, notre mode de pensée n'étant pas adapté au parallélisme. La mise au point est délicate et rendue difficile par le côté aléatoire et non reproductible de l'exécution. La correction des algorithmes déjà difficile à assurer en séquentiel, devient critique en parallèle. Autre point d'importance, la parallélisation automatique permet de profiter de l'existence de nombreux programmes et bibliothèques déjà écrits en séquentiel. Même si ces algorithmes ne sont pas toujours au mieux adaptés à la parallélisation, il est tout de même plus intéressant de les paralléliser automatiquement plutôt que de les réécrire en totalité. Ainsi, un grand nombre de travaux concerne la transformation de programmes séquentiels en code parallèle.

Nos travaux se situent dans le cadre des grammaires attribuées. Leur but est de montrer qu'il est possible, à l'aide des grammaires attribuées, de produire des programmes parallèles efficaces, sans avoir recours à la spécification explicite du parallélisme. Pour cela, les grammaires attribuées sont particulièrement bien adaptées puisqu'elles permettent de spécifier les calculs à entreprendre pour la résolution d'un problème, sans préciser l'ordre dans lequel ces calculs doivent être effectués. L'ordre est calculé par le système à partir d'un ensemble de graphes de flot de données qui sont connus statiquement. La détection du parallélisme revient à l'analyse de ces graphes.

En plus des avantages généraux des grammaires attribuées, la classe des grammaires attribuées l-ordonnées permet de calculer un très grand nombre d'informations de façon statique. La détention d'informations statiques est très importante car elle permet de minimiser les informations parallèles à calculer dynamiquement, ce qui diminue le surcoût de la mise en œuvre parallèle et par conséquent augmente d'autant l'efficacité du code produit. Pour profiter de tous ces avantages, nous nous sommes donc intéressés à la parallélisation de l'évaluation pour cette classe de grammaires.

Un certain nombre de travaux théoriques ayant déjà été effectués, nous avons pensé qu'il était important d'implanter dans un système performant la parallélisation de l'évaluation des attributs afin de montrer qu'il est possible, grâce au formalisme des grammaires attribuées, de produire du code réellement efficace. Cela a été réalisé au sein de l'axe compilation du projet `Chloé` à l'`Inria`, où nous avons inséré la parallélisation de l'évaluation à l'intérieur du système FNC-2 qui y est développé. La performance et la complexité du système FNC-2, qui a dépassé depuis longtemps le stade du prototype, donne à notre implantation une valeur significative au niveau des mesures effectuées.

La première partie de cette thèse rappellera les différents concepts propres aux grammaires attribuées et décrira les principales notations que nous utiliserons dans la suite.

Un état de l'art en ce qui concerne les travaux sur la parallélisation de l'évaluation des attributs est passé en revue dans la deuxième partie, y compris pour les modèles différents de celui que nous avons choisi, notamment des études sur architectures distribuées. Nous présentons les différentes méthodes d'évaluation, séquentielles et parallèles, qui ont été étudiées par le passé. Le nombre croissant de travaux dans le domaine de la parallélisation est significatif de l'importance de l'enjeu. En effet, la parallélisation des grammaires attribuées va complètement dans le sens de la diminution du prix de revient des compilateurs sur machine parallèle puisqu'elle permet de construire une version séquentielle et une version parallèle, immédiatement à partir des mêmes spécifications du compilateur.

La méthode de parallélisation que nous avons choisie est décrite dans le troisième chapitre. Nous y exposons la transformation du graphe de dépendance en un graphe de "parallélisation" qui respecte le modèle synchrone. Quelques méthodes sont également données pour améliorer les résultats de cette parallélisation, notamment pour éviter la création trop importante de tâches de durée trop courte.

L'implantation effectuée est décrite dans la quatrième partie. On y trouvera des comparaisons entre les différentes implantations du modèle : une implantation procédurale des séquences de visite et une implantation par automate. Nous comparons une même implantation sur des architectures différentes, toutes basées sur le concept de mémoire partagée. La comparaison sur différentes machines parallèles nous permet d'abstraire les spécificités internes des différentes architectures et ainsi de voir les points forts et les points faibles de notre modèle. Les résultats prouvant la validité de notre approche sont explicités dans cette même partie.

Enfin, une véritable efficacité ne saurait être atteinte sans une optimisation mémoire de l'évaluation des attributs. La cinquième partie présente succintement la théorie de l'optimisation mémoire dans les évaluateurs séquentiels d'attributs qui permet le stockage hors de l'arbre de tous les attributs d'une grammaire. Avec une telle méthode, les informations sémantiques ne sont plus dans l'arbre syntaxique mais dans des structures externes.

L'optimisation mémoire dans l'évaluation est absolument nécessaire si nous voulons concurrencer l'écriture manuelle de programmes parallèles. Elle permet en effet une économie d'espace mémoire très importante, et l'élimination de calculs propres aux méthodes des grammaires attribuées : les règles de copies. Dans le dernier chapitre, nous proposons une théorie qui permet d'étendre la méthode d'optimisation séquentielle pour le cas de l'évaluation parallèle et, de cette façon, retirer toute la performance qui y est attachée.

Chapitre 1

La théorie des grammaires attribuées

Ce chapitre introduit les différents concepts et notations utilisés dans les grammaires attribuées et auxquels nous ferons référence tout au long de cette thèse.

1.1 Concepts de base

Une grammaire attribuée est une extension du concept de grammaire indépendante du contexte. À chaque non-terminal d'une production, on associe un ensemble d'attributs qui décrivent la sémantique attachée à ce non-terminal. Le calcul de la valeur de ces attributs est exprimé par des fonctions sémantiques attachées aux productions. Ces règles sémantiques induisent ainsi des dépendances entre les attributs, et on peut le plus souvent trouver un ordre d'évaluation pour le calcul des attributs. Nous donnerons d'abord quelques définitions formelles de ces différents objets avant d'en exhiber un exemple.

Définition 1.1.1 (Grammaire indépendante du contexte) *Une* grammaire indépendante du contexte *est un quadruplet* $G = (N, T, Z, P)$ *où :*

- N *est un ensemble fini non vide de symboles non-terminaux ;*
- T *est un ensemble fini de symboles terminaux,* $N \cap T = \emptyset$*;*
- Z *est l'axiome de la grammaire,* $Z \in N$*;*
- P *est un ensemble fini de productions,*
  $p : X_0 \rightarrow X_1 \ldots X_{n_p}$ *avec* $X_0 \in N$ *et* $X_i \in (T \cup N)$.

Définition 1.1.2 (Relation dérive) *On note* $\longrightarrow_G$ *la relation binaire issue des productions de la grammaire* G. $X \rightarrow_G Y \Leftrightarrow \exists p \in P$ *telle que* $p : X \rightarrow X_1 \ldots Y \ldots X_{n_p}$.

On note $\stackrel{*}{\longrightarrow}_G$ *la fermeture transitive de* $\longrightarrow_G$.

Définition 1.1.3 (Sous-grammaire) *Soit $G = (N, T, Z, P)$ une grammaire indépendante du contexte.*

Soit $Z' \in N$ un non-terminal de G.

On appelle sous-grammaire de G de racine Z' la grammaire $G' = (N', T', Z', P')$ telle que :

- $X \in N' \Leftrightarrow (Z' \xrightarrow{*}_G X)$
- $x \in T' \Leftrightarrow (Z' \xrightarrow{*}_G x)$
- $p : X_0 \to X_1 \ldots X_{n_p} \in P' \Leftrightarrow X_0 \in N'$

Notations pour une grammaire indépendante du contexte G :

$L(G)$ est le langage engendré par G ;

t est un arbre syntaxique de G ;

u est un nœud de t ;

$label(u)$ représente le non-terminal associé à u ;

$prod(u)$ représente la production au nœud u ;

$pos(u, i)$ représente le fils du nœud u en position i ;

$arite(u)$ est égal au nombre de fils de u.

Définition 1.1.4 (Grammaire attribuée) *On appelle* Grammaire attribuée *un triplet $AG = (G, A, F)$ où :*

- *$G = (N, T, Z, P)$ est une grammaire indépendante du contexte ;*
- *A est l'ensemble des attributs de la grammaire. À chaque non-terminal X est associé un sous-ensemble de A noté $A(X)$ dont les éléments sont notés $X.a$. L'ensemble $A(X)$ représente les valeurs sémantiques du non-terminal X. On note $A = \bigcup_{X \in N} A(X)$*
- *$F = \bigcup F(p)$ est un ensemble de règles sémantiques où $F(p) = \{r_{p,a,X,i}\}$, $r_{p,a,X,i}$ désignant la règle sémantique qui définit l'attribut a du non-terminal X en position i dans la production p.*

Définition 1.1.5 (Règle de copie) *Une règle de copie est une règle sémantique de la forme $X.a = Y.b$ où $=$ est la fonction d'identité.*

Les règles de copie représentent une grande partie des règles sémantiques, et sont à l'origine d'une certaine inefficacité des évaluateurs. Les chapitres 5 et 6 montrent comment nous pouvons éliminer certaines de ces règles.

Définition 1.1.6 (Arbre d'Entrée) *Un arbre d'entrée t pour une grammaire attribuée $AG = (G, A, F)$ est un arbre syntaxique associé à une phrase du langage $L(G)$.*

Dans une production, plusieurs nœuds peuvent être étiquetés par un même non-terminal X, nous parlerons alors d'occurrence du non-terminal X.

Dans un arbre d'entrée, chaque nœud étiqueté X est appellé instance de X.

La notion (statique) d'occurrence et la notion (dynamique) d'instance sont aussi valides pour les attributs.

L'ensemble d'attributs $A(X)$ est l'union de l'ensemble $HER(X)$ des attributs hérités, et de l'ensemble $SYN(X)$ des attributs synthétisés :

- $A(X) = HER(X) \cup SYN(X)$
- $\forall X \in N, HER(X) \cap SYN(X) = \emptyset$

Pour une production $p : X_0 \rightarrow X_1 \ X_2 \ \ldots \ X_{n_p} \qquad p \in P$, on définit deux ensembles d'occurrences d'attributs :

- $$ENT(p) = \{X_i.a/(i = 0 \wedge X_0.a \in HER(X_0)) \vee (i \geq 1 \wedge X_i.a \in SYN(X_i))\}$$
- $$SOR(p) = \{X_i.a/(i = 0 \wedge X_0.a \in SYN(X_0)) \vee (i \geq 1 \wedge X_i.a \in HER(X_i))\}$$

Définition 1.1.7 (Règle sémantique) *Soit p une production de P, pour tout $X.a \in SOR(p)$, il existe dans $F(p)$ une et une seule règle sémantique de la forme :*
$r_{p,a,X,i} : X_i.a = f_{p,a,X,i}(X_{k_1}.b_1, \ldots, X_{k_n}.b_n)$

La signification de ces deux ensembles est simple : les attributs d'entrée ($ENT(X)$) sont les attributs calculés en dehors de la production, tandis que les attributs de sortie ($SOR(X)$) sont calculés à l'intérieur de la production.

1.1.1 Graphe de dépendances local

Les règles sémantiques associées à une production induisent des dépendances sur les occurrences d'attributs des non-terminaux de cette production. Pour formaliser ces dépendances, nous utilisons la notion de graphe de dépendances.

Pour une production p et une occurrence i de l'attribut a de p, on définit D_{p,a_i} comme étant l'ensemble des occurrences d'attributs de la production p utile au calcul de l'attribut a. $D_{p,a} = \{X_i.b/\exists r_{p,a,X_k,k} \in F$ avec $r_{p,a,X_k,k} : X_k.a = f(\ldots, X_i.b, \ldots)\}$

Définition 1.1.8 (Graphes de dépendances locaux) *Pour chaque production $p \in P$, le graphe de dépendances local $DP(p)$ est le graphe orienté $DP(p) = (V, E)$ avec :*

- $V = A(X_0) \cup \ldots \cup A(X_n)$

- $E = \{(X_i.a, X_j.b) / X_j.b = f(\dots, X_i.a, \dots)$ *est une règle sémantique de* $F\}$

Notation : on note $DP^+(p)$ la fermeture transitive de $DP(p)$.

Par extension, on définit le graphe de dépendances global sur un arbre t comme l'union des instanciations des graphes de dépendances locaux sur l'ensemble des instances de productions de t. On note $\xrightarrow{t}$ sa relation associée.

Définition 1.1.9 (Forme normale) *Une règle sémantique est en forme normale si et seulement si les paramètres de la fonction sémantique associée sont tous des attributs d'entrée de la production.*

Définition 1.1.10 (Grammaire en forme normale) *Une grammaire attribuée G est en forme normale si et seulement si toutes ses règles sémantiques sont en forme normale.*

Remarque : le résultat d'une règle sémantique est toujours un attribut de sortie de la production.

Il est bon de rappeler que toute grammaire attribuée peut se mettre en forme normale par une transformation des règles sémantiques, si pour tout $p \in P$, le graphe de dépendances locales (définition 1.1.8) de p est acyclique. Considérer seulement des règles sémantiques en forme normale ne restreint donc pas le pouvoir d'expression. Par conséquent, nous supposerons dans la suite que les règles sémantiques sont en forme normale.

- Pour une occurrence d'attribut a sur une production p, on note *suivant(a)* l'ensemble des occurrences d'attributs b tel que b dépend de a.

- Pour une occurrence d'attribut a sur une production p, on note *précédent(a)* l'ensemble des occurrences d'attributs b tel que a dépend de b.

- Pour une production $p \in P$ et a une occurrence d'attribut attachée à la production, on note $pos(a, p)$ la position dans p du non-terminal portant a.

1.1.2 Notations sur les graphes

Définition 1.1.11 (Graphe) *Un graphe G est la donnée d'un couple (V, E) où V est un ensemble fini et E définit une relation binaire entre les éléments de V.*

Définition 1.1.12 (Cycle) *Un cycle est un n-uplet $(e_1, e_2 \dots, e_n)$ tel que :*

- $\forall i\,; e_i \in V$

- $\forall i \in [1, n-1]\,; (e_i, e_{i+1}) \in E$

- $e_1 = e_n$

Définition 1.1.13 (Graphe acyclique) *Un graphe est dit acyclique ou non-circulaire si et seulement si il ne contient pas de cycle.*

Définition 1.1.14 (Maximum et minimum d'un graphe acyclique) *Soit G=(V,E) un graphe et soit $V' \subset V$. On note*
$Max(G) = \{a \in V / \textit{ précédent}(a) = \emptyset\}$
$Min(G) = \{a \in V / \textit{ suivant}(a) = \emptyset\}$
$Max_G(V') = \{a \in V' / \textit{ précédent}(a) \cap V' = \emptyset\}$
$Min_G(V') = \{a \in V' / \textit{ suivant}(a) \cap V' = \emptyset\}$
La notion de maximum caractérise les sommets d'entrée d'un graphe, tandis que la notion de minimum caractérise les sommets de sortie.

Définition 1.1.15 (Profondeur dans un graphe acyclique) *Soit $G = (V, E)$ un graphe acyclique qui vérifie : $\exists$unique $U \in V \ / \ \forall u \in V$ on a $U \xrightarrow{*}_G u$*
La profondeur d'un nœud u dans G est la longueur du plus court chemin de U à u.

1.2 Évaluation de grammaires attribuées

Étant donnée une grammaire attribuée, l'évaluation des attributs sur un arbre d'entrée t, consiste à évaluer l'ensemble des attributs associés à chaque nœud de l'arbre d'entrée conformément aux règles sémantiques de la grammaire attribuée.

- À chaque nœud u de l'arbre d'entrée t on associe un ensemble d'**instances d'attributs** :

$$IA(u) = \{u.a/a \in A(label(u))\} ;$$

- À chaque nœud u et à chaque règle sémantique de $F_{label(u)}$ de la forme :

$$r_{label(u),a,X,k} : X_k.a = f_{label(u),a,X,k}(X_{f(1)}.a_1, \cdots, X_{f(q)}.a_q)$$

  on associe l'équation :

$$u_k.a = f_{label(u),a,X,k}(u_{f(1)}.a_1, \cdots, u_{f(q)}.a_q)$$

  où u_i représente le nœud en position i.

Nous définissons pour un arbre t l'ensemble de ses instances d'attributs $IA(t)$ comme l'ensemble contenant toutes les instances d'attributs associées à tous les nœud de t.

Nous obtenons finalement un système d'équations $K(t)$ dont les inconnues sont les instances d'attributs de l'arbre d'entrée t.

Évaluer les attributs sur t, c'est résoudre le système $K(t)$.

Nous nous restreignons en fait au calcul du résultat de la grammaire attribuée pour un arbre t, aussi appelé *valeur sémantique*, qui est la valeur des instances d'attributs synthétisés de la racine. Le problème revient donc à calculer uniquement les valeurs des instances d'attributs qui sont nécessaires au calcul des attributs de Z.

Nous appelons impasse statique toute occurrence d'attribut qui, quel que soit l'arbre d'entrée, ne sert jamais au calcul du résultat de la grammaire attribuée.

Nous appelons impasse dynamique toute instance d'un attribut qui n'est pas une impasse statique et telle qu'il existe un arbre dans lequel elle ne sert pas au calcul du résultat.

Le système $K(t)$ a une unique solution s'il n'est pas circulaire. On dit qu'une grammaire attribuée est non-circulaire ou bien formée si, quel que soit l'arbre d'entrée t, le système $K(t)$ n'est pas circulaire.

Dans ce cas, il est possible, moyennant un réordonnancement des équations, de construire un système "triangulaire supérieur". Il existe alors un ordre simple d'évaluation des instances d'attributs.

Cette méthode de résolution semble la plus simple et la plus générale, malheureusement elle n'est pas très viable étant donné la place et le temps qu'elle nécessite. C'est pourquoi beaucoup d'auteurs se sont penchés sur différentes méthodes d'évaluation des attributs.

1.3 Les méthodes d'évaluation

Une première famille d'évaluateurs est basée sur la méthode dite méthode dynamique, très proche de la résolution du système. Elle consiste à trouver dynamiquement, pour un arbre d'entrée t, l'ordre sur les instances d'attributs induit par $\stackrel{t}{\rightarrow}$ et à calculer les instances d'attributs en allant des plus petites aux plus grandes au sens $\stackrel{t}{\rightarrow}^{+}$. Cette méthode très simple a été utilisée dans de nombreux systèmes [Fan72, Lor74, Lor77, CoH79, KeR79, Mad80, Jal83, Jou84a]. La classe des grammaires attribuées acceptée par cette méthode est la classe des grammaires attribuées non-circulaires. Il est possible de tester statiquement la non-circularité d'une grammaire attribuée. Malheureusement, les évaluateurs basés sur cette méthode sont **très peu efficaces** en temps car, pour chaque arbre d'entrée, ils doivent reconstruire l'ordre. D'autre part, le test statique de non-circularité d'une grammaire attribuée est exponentiel en la taille de la grammaire [JOR75, DJL84].

Par opposition à cette méthode, et pour pallier l'inefficacité des évaluateurs dynamiques, de nombreux auteurs ont cherché à déterminer de façon "statique" l'ordre d'évaluation.

Une première approche, utilisée dans la famille des évaluateurs dits "par passes", consiste à fixer d'une façon arbitraire la stratégie de parcours de l'arbre et en déduire un ordre de calcul des instances d'attributs. Les classes les plus importantes sont les classes L et *Sweep*. La classe L permet de calculer tous les attributs en parcourant l'arbre en plusieurs passes de gauche à droite. La classe *Sweep* revient à la classe L moyennant une permutation de la partie droite de certaines productions. Quelques systèmes sont basés sur cette approche et sont résumés dans [DJL88]. Les évaluateurs de cette famille sont très efficaces, mais la classe des grammaires attribuées acceptées est très restreinte.

Une troisième famille d'évaluateurs, basée sur des méthodes dites statiques, essaie, à partir des graphes de dépendances locales, de déduire statiquement un ordre d'évaluation des occurrences d'attributs d'une production, valable quel que soit l'arbre d'entrée. Les différentes méthodes se différencient par le type de l'ordre qu'elles construisent. Par opposition à la famille précédente où le parcours de l'arbre déterminait l'ordre, ici, c'est la grammaire attribuée, par sa définition, qui induit l'ordre et par conséquent le type de parcours de l'arbre. La classe des grammaires attribuées acceptées est donc plus large.

Nous allons nous intéresser plus particulièrement dans cette thèse à cette dernière famille d'évaluateurs. Ce choix se justifie car le compromis entre l'efficacité des évaluateurs et la classe des grammaires attribuées qu'ils acceptent, semble être le meilleur pour cette famille. D'autre part, ce travail se place au sein du système FNC-2 [JoP89] qui produit des évaluateurs basés sur ce type de méthode statique.

Nous décrirons maintenant de façon précise les classes de grammaires attribuées associées aux différentes méthodes statiques. Nous donnerons également une idée des évaluateurs attachés à chacune d'elles. Toutefois, il est préférable de parler en premier de la classe FNC, même si la plupart des travaux pour cette classe sont encore dynamiques.

1.3.1 La classe des grammaires attribuées FNC

La classe des grammaires attribuées fortement non-circulaires semble être, d'après les travaux de Jourdan [Jou82, Jou84c, Jou84b], la classe la plus large permettant l'utilisation d'une méthode statique. Cette méthode statique est dérivée de la construction théorique de Courcelle et Franchi-Zannettacci [CoF82, Fra82]. Elle est basée sur le résultat suivant :

> Soit une grammaire attribuée et un arbre d'entrée pour celle-ci, la valeur d'une instance d'attribut synthétisé attachée à un nœud donné de l'arbre ne dépend que :
>
> – des valeurs des instances d'attributs hérités de ce nœud ;
>
> – du sous-arbre issu de ce nœud.

La validité de ce résultat est une conséquence de la notion de calcul local à une instance de production. Il est naturel d'associer à chaque attribut synthétisé $a \in A(X)$, une fonction d'évaluation prenant comme paramètres un sous-arbre syntaxique t ayant pour racine u avec $label(u) = X$ et un ensemble de valeurs d'attributs hérités de X. Courcelle [CoF80] a appelé cet ensemble le sélecteur d'argument. Cette fonction a pour résultat la valeur de l'instance d'attribut $X.a$ attachée à la racine de t, étant données les valeurs de certaines instances d'attributs hérités associés à r.

L'ensemble des attributs hérités nécessaires au calcul d'un attribut synthétisé d'un même non-terminal dépend du sous-arbre issu de ce non-terminal. C'est une notion dynamique. Le *sélecteur d'arguments*, qui est une approximation des relations réelles entre les attributs hérités et synthétisés d'un non-terminal, permet de déterminer statiquement si on peut construire pour chaque attribut synthétisé un ensemble d'attributs hérités suffisant pour calculer la valeur de cet attribut synthétisé et n'induisant pas de circularité dans les calculs.

La classe des grammaires attribuées sur lesquelles il est possible de construire cet ensemble à partir du sélecteur d'arguments, est appelée classe des grammaires attribuées FNC.

Définition 1.3.1 (Fortement non-circulaire) *Une grammaire attribuée est FNC si et seulement si il existe une famille d'ordres partiels sur les attributs des non-terminaux* $\{R_{FNC}(X)/X \in N\}$ *telle que :*
$\forall p \in P, p : X_0 \longrightarrow X_1 \ldots X_n$ *on a*

1. *non-circularité :* $R_{FNC}(X_1) \cup \ldots \cup R_{FNC}(X_n) \cup DP(p)$ *est sans circuit ;*

2. *clôture : la restriction de* $R_{FNC}(X_1) \cup \ldots \cup R_{FNC}(X_n) \cup DP(p)$ *aux attributs de* X_0 *est incluse dans* $R_{FNC}(X_0)$.

À chaque attribut synthétisé a sur un non-terminal X on associe :

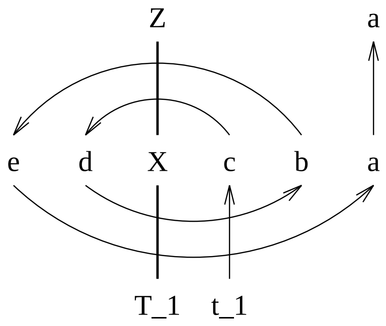
$$R_{FNC}(X, a) = \{b \in HER(X)/(b, a) \in R_{FNC}(X)\}.$$

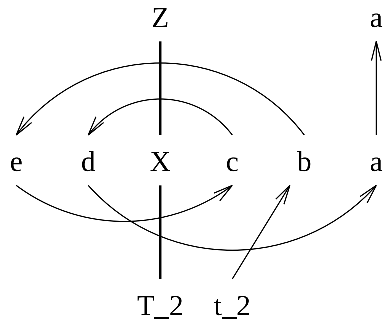
La propriété de clôture signifie que, quel que soit l'arbre d'entrée t, quel que soit l'attribut synthétisé a, $R_{FNC}(X, a)$ contient les attributs h tels que a dépend de h dans t. En d'autres termes, les dépendances induites du sélecteur d'arguments doivent contenir toutes les dépendances réelles. La propriété de non-circularité permet de vérifier qu'il n'existe pas de dépendance induite qui crée une circularité dans le calcul. En conséquence de quoi, il existe certaines grammaires attribuées (cf. figure 1.1) pour lesquelles il est impossible de trouver un sélecteur d'arguments qui vérifie les propriétés FNC, bien que celles-ci soient non-circulaires. Cette perte en pouvoir d'expression est très faible puisque toutes les grammaires attribuées "pratiques" que nous connaissons par expérience sont FNC.

Il existe un algorithme polynomial qui calcule pour une grammaire attribuée FNC une famille de relations d'ordre partiel $R_{FNC}(X)$ minimales. Cet algorithme est à la fois un test d'appartenance à la classe FNC et un algorithme de construction des relations d'ordre partiel.

L'ordre d'évaluation trouvé statiquement est a priori un ordre partiel sur les non-terminaux, et par conséquent on a un ordre partiel sur les productions. L'ordre effectif d'évaluation est déterminé dynamiquement sur un arbre d'entrée donné par la structure même des règles sémantiques. Du fait du sélecteur d'arguments, qui n'est qu'une approximation des relations réelles, la méthode FNC force le calcul sur un arbre donné de certaines instances d'attributs qui ne participent pas au calcul du résultat.

Cette GA est non-circulaire puisque les deux seuls arbres sont :





Pourtant elle n'est pas FNC puisque nécessairement :
$e \in R_{FNC}(X, c)$ et $d \in R_{FNC}(X, b) \Rightarrow$ circularité de $R_{FNC}(p_0)$.

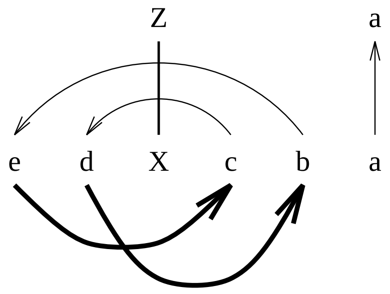


FIG. 1.1 - *Grammaire attribuée NC non FNC*

Il y a alors deux possibilités pour l'évaluation. Soit les instances d'attributs ne sont pas stockées dans l'arbre (elles sont vues comme les valeurs de retour de fonctions et donc certaines instances d'attributs sont recalculées lorsqu'elles sont utilisées plus d'une fois), soit elles sont stockées sur les nœuds de l'arbre et il est nécessaire de tester, avant d'utiliser leur valeur, si elles ont déjà été calculées. Le problème de la première solution est que chaque utilisation d'une instance d'attribut nécessite le recalcul complet de sa valeur, ce qui fait perdre en efficacité (complexité exponentielle). La deuxième évite cet inconvénient mais nécessite une grande consommation mémoire.

1.3.2 La classe des grammaires attribuées DNC

La méthode d'évaluation sur la classe DNC [Fil86] est comparable dans les grandes lignes à la méthode d'évaluation sur la classe FNC. Ces deux méthodes diffèrent principalement par le fait que l'évaluation DNC peut débuter sur n'importe quel nœud de l'arbre d'entrée. Ceci implique une évaluation ascendante de certaines instances d'attributs hérités et donc des fonctions d'évaluation qui remontent dans l'arbre d'entrée.

Cette méthode est basée sur le résultat associé à la classe des grammaires attribuées FNC et au résultat suivant :

> Soient une grammaire attribuée et un arbre d'entrée pour celle-ci. La valeur d'une instance d'attribut hérité attachée à un nœud donné de l'arbre ne dépend que :
>
> – des valeurs des instances d'attributs synthétisés de ce nœud ;
>
> – de l'arbre englobant le nœud.

L'arbre englobant d'un nœud u est l'arbre complet auquel a été enlevé le sous-arbre issu de u.

Comme pour les attributs synthétisés dans la méthode FNC, à chaque attribut hérité de X est associée une fonction qui ne dépend que de l'arbre englobant et d'un ensemble de valeurs d'attributs synthétisés de X. D'autre part, la notion de sélecteur d'arguments est étendue aux attributs hérités. Il faut trouver une famille de relations d'ordre partiel qui englobe les relations réelles pour tout arbre d'entrée.

Afin que le sélecteur d'arguments n'entraîne pas de circularité dans les calculs il faut que la famille d'ordres vérifie certaines propriétés.

Définition 1.3.2 (Doublement non circulaire) *Une grammaire attribuée est DNC si et seulement si il existe une famille d'ordres partiels $\{R_{DNC}(X)/X \in N\}$ telle que :*
$\forall p \in P, p : X_0 \longrightarrow X_1 \dots X_n$ *on a*

1. *non-circularité : $R_{DNC}(X_0) \cup \dots \cup R_{DNC}(X_n) \cup DP(p)$ est sans circuit ;*
2. *double clôture : la restriction de $R_{DNC}(X_0) \cup \dots \cup R_{DNC}(X_i - 1) \cup R_{DNC}(X_i + 1) \cup \dots \cup R_{DNC}(X_n) \cup DP(p)$ aux attributs de X_i est incluse dans $R_{DNC}(X_i)$.*

Il existe un algorithme polynomial [Bar82, Bar84] qui permet de caractériser la classe DNC et de calculer la famille des $R_{DNC}(X)$. On remarque que pour calculer cette famille, la grammaire attribuée doit déjà être FNC et les $R_{DNC}(X)$ doivent contenir les sélecteurs d'arguments FNC.

Cette classe semble être la classe la plus large pour laquelle il existe un ordre partiel connu statiquement permettant une évaluation ascendante. Cette classe est, d'après les remarques précédentes, une sous-classe de la classe FNC. Il existe des grammaires attribuées qui sont FNC sans être DNC (cf. figure 1.2).

1.3.3 La classe des grammaires attribuées *l*-ordonnées

Contrairement aux méthodes statiques présentées précédemment, la méthode d'évaluation *l*-ordonnée est basée sur le fait que l'on connaît statiquement l'**ordre total** d'évaluation des attributs d'un non-terminal. Cette connaissance est fondamentale quant aux actions que l'on peut entreprendre. En effet, le rôle d'un bon évaluateur est de calculer toutes les valeurs d'instances d'attributs nécessaires, sans calcul redondant (comme c'est le cas dans un évaluateur dynamique qui ne stocke pas les attributs). Pour être efficace, un évaluateur doit faire ce qu'on lui demande, sans plus. Or la seule information vraiment nécessaire pour évaluer correctement un arbre, c'est la connaissance de cet arbre. C'est là qu'est l'avantage de la classe *l*-ordonnée ; le seul travail qu'effectue l'évaluateur, en plus de l'évaluation, est de se poser la question de savoir à quoi ressemble l'arbre d'entrée. Ainsi, à chaque fois qu'il arrive sur un nouveau nœud, l'évaluateur se pose la question : sur quelle production suis-je ? Avec un tel évaluateur, il n'y a quasiment pas de surcoût à l'évaluation. Alors que pour un évaluateur FNC, l'évaluateur doit se demander : "ai-je déjà calculé cet attribut ?".

D'autre part, la classe *l*-ordonnée est la plus grande classe sur laquelle nous pouvons effectuer une optimisation mémoire, car l'ordre total nous permet de calculer les durées de vie de chaque instance d'attribut de l'arbre, ce qui n'est pas possible dans le cas de l'évaluation FNC ou DNC.

Cette classe de grammaires attribuées a été introduite par J. Engelfriet [EnF82] sous le nom de *simples multi-visites*.

Définition 1.3.3 (*l*-ordonnée) *Une grammaire attribuée est l-ordonnée si et seulement si il existe une famille d'ordres* **totaux** $\{R(X)/X \in N\}$ *telle que :*
$\forall p \in P, p : X_0 \longrightarrow X_1 \ldots X_n$ *on a*
non-circularité : $R(X_0) \cup \ldots \cup R(X_n) \cup DP(p)$ *est sans circuit.*

Si une grammaire attribuée est de classe *l*-ordonnée, alors on peut trouver statiquement un ordre total d'évaluation sur toute production. Il est possible de coder l'évaluateur sous forme de séquences de calculs, appelées séquences de visites dans [Kas80, Eng84] respectant l'ordre $DP(p)$. Celles-ci nous permettent de construire des évaluateurs **totalement déterministes** pour les grammaires attribuées *l*-ordonnées. Contrairement aux méthodes précédentes, il n'est pas nécessaire de tester si une instance d'attribut a déjà une valeur avant de la calculer.

La classe des grammaires attribuées *l*-ordonnées est incluse dans la classe des grammaires attribuées DNC. En effet l'ordre total induit la condition de double-clôture [Eng84, Der84] de la définition DNC.

Le test d'appartenance à la classe *l*-ordonnée ainsi que la construction des ordres totaux sont malheureusement des problèmes NP-complets [EnF82]. Il n'est donc guère envisageable de créer un système de génération d'évaluateurs pour cette classe.

Une solution pour utiliser les propriétés intéressantes de cette classe a été de trouver une sous-classe des grammaires attribuées *l*-ordonnées caractérisable en un temps polynomial : la classe OAG [Kas80].

Une autre approche, qui a été développée dans le système FNC-2, est de transformer les grammaires attribuées FNC en des grammaires attribuées *l*-ordonnées. En effet, il a été démontré [Fil83] que toute grammaire attribuée NC pouvait être transformée en une grammaire *l*-ordonnée équivalente, au risque d'augmenter la taille de la grammaire attribuée de façon exponentielle.

Une transformation optimisée [Par87] permet, dans le cas de la transformation FNC vers *l*-ordonnée, de faire disparaître en pratique le facteur exponentiel.

Comme nous nous plaçons dans le cadre d'évaluation de grammaires *l*-ordonnées, pour des raisons exposées plus tard, grâce à cette transformation, nous allons hériter de la possibilité de paralléliser les grammaires attribuées FNC.

Il existe bien sûr d'autres classes de grammaire, mais celles-ci ne sont pas utiles à notre discours. La figure 1.3 rappelle les classes précédemment citées ainsi que les liens d'inclusion entre elles.

1.4 Évaluation des attributs

Comme nous l'avons vu dans la section 1.3, les évaluateurs dynamiques ne sont pas efficaces. Quant aux évaluateurs FNC ou DNC, ils sont rendus peu efficaces par l'impossibilité de savoir statiquement si tel attribut est calculé à un instant précis.

Notre optique étant de faire le plus de choses possibles statiquement, nous avons été conduits à choisir la classe des grammaires attribuées *l*-ordonnées. L'évaluateur produit n'a jamais à se demander si un attribut est calculé ou non : il est purement déterministe. La seule question que se pose l'évaluateur est : en arrivant sur un nœud u, quelle production y est appliquée ? Ensuite, le parcours du graphe de dépendance se fait indépendamment de l'arbre d'entrée.

L'évaluateur *l*-ordonné peut se voir comme un automate avec trois actions principales :

– VISIT(i) : action de parcours d'arbre, l'évaluateur se déplace sur le i^{ème} fils du nœud courant et détermine la production associée à ce nœud pour sélectionner la séquence de visite correspondante.

– EVAL(X.a) : l'évaluateur doit calculer la valeur de l'attribut X.a.

– LEAVE : action de parcours d'arbre, l'évaluateur se déplace sur le père du nœud courant et continue la séquence de visite après le `VISIT` ayant provoqué la sous-visite.

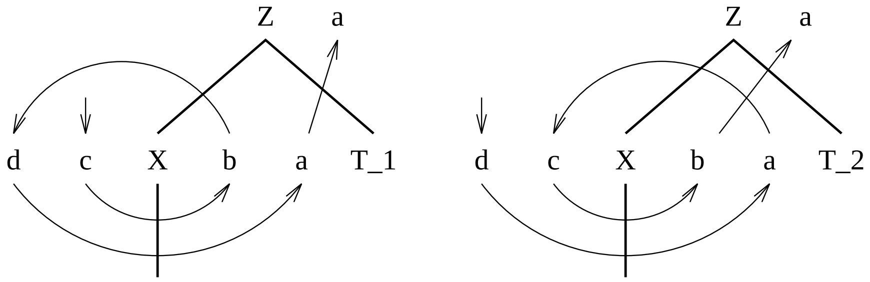
L'ensemble des instructions de l'automate, qui débute de l'entrée pour la i^{ème} fois sur un nœud X et termine par l'action LEAVE correspondante, est appelé i^{ème} séquence de visite sur X.

1.5 Exemple

La figure 1.4 donne un exemple de grammaire attribuée. La grammaire est décrite dans le langage Olga [JoP89] utilisé pour la description de grammaires attribuées dans le système Fnc-2. Cet exemple ne veut pas représenter une sémantique particulière, mais seulement introduire des graphes de dépendances significatifs par rapport aux problèmes rencontrés en parallélisme.

La figure 1.5 montre le graphe de dépendance de chacune des productions et la figure 1.6 explicite les graphes de dépendances augmentés par les relations FNC et DNC.

Cette GA est FNC puisque les deux seuls arbres sont :



Pourtant elle n'est pas DNC puisque : $a \in R_{DNC}(X,c)$ et $b \in R_{DNC}(X,d) \Rightarrow$ circularité de $R_{DNC}(p_2)$

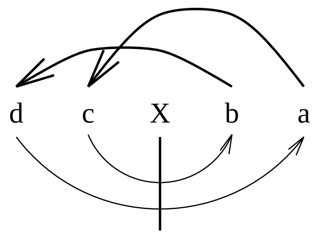


FIG. 1.2 - *Grammaire attribuée FNC non DNC*

Grammaires Attribuées Non-Circulaires

$\cup$

Grammaires Attribuées Fortement Non-Circulaires

$\cup$

Grammaires Attribuées Doublement Non-Circulaires

$\cup$

Grammaires Attribuées *l*-ordonnées

FIG. 1.3 - *Classification des grammaires attribuées.*

```
% -*-olga-*-
% { Document name: /home/minos3/chloe/marmol/these-rapport/grammaire.olga.tex }
% { Creator: Bruno & [marmol@phedre.inria.fr] }
% { Creation Date: Tue Jun 14 18:24:20 1994 }

attribute grammar exemple(arbre) : () is

    from translator_C import all ;
     attribute
     inherited   $h ( Z, X, Y, U, V ) : int;
     inherited  $a (T) :int;
     inherited   $b (T) :int;

     synthesized $s ( X, Y, U, V ) : int;
     synthesized $c (T) :int;
     synthesized $d (T) :int;

where Zop -> X use
    $s:= $s(X);
    $h(X):=1;
end where

where Xop0 -> use
    $s:=$h;
end where

where Xop1 -> Y1 Y2 Y3 use
    $s:= $s(Y1) + $s(Y2) + $s(Y3);
end where

where Xop2 -> X1 T use
    $s:= $s(X1) + $c(T) + $d(T);
    $h(X1):=$h;
    $a(T):=$h(X0);
end where

where Top -> T1 U V1 V2 use
    $a(T1):= $a;
    $b(T1):= $b;
    $h(U):= $a;
    $h(V1) := $s(U);
    $h(V2) := $s(U);
    $c  := $c(T1);
    $d := $s(U) + $s(V1) + $s(V2);

where Top0 -> use
    $d:=$a;
    $c:=$b;
end where

where Vop0 -> use
    $s:=$h;
end where

where Uop0 -> use
    $s:=$h;
end where

where Yop0 -> use
    $s:=$h;
end where
```

FIG. 1.4 - *Exemple en Olga d'une grammaire attribuée G*

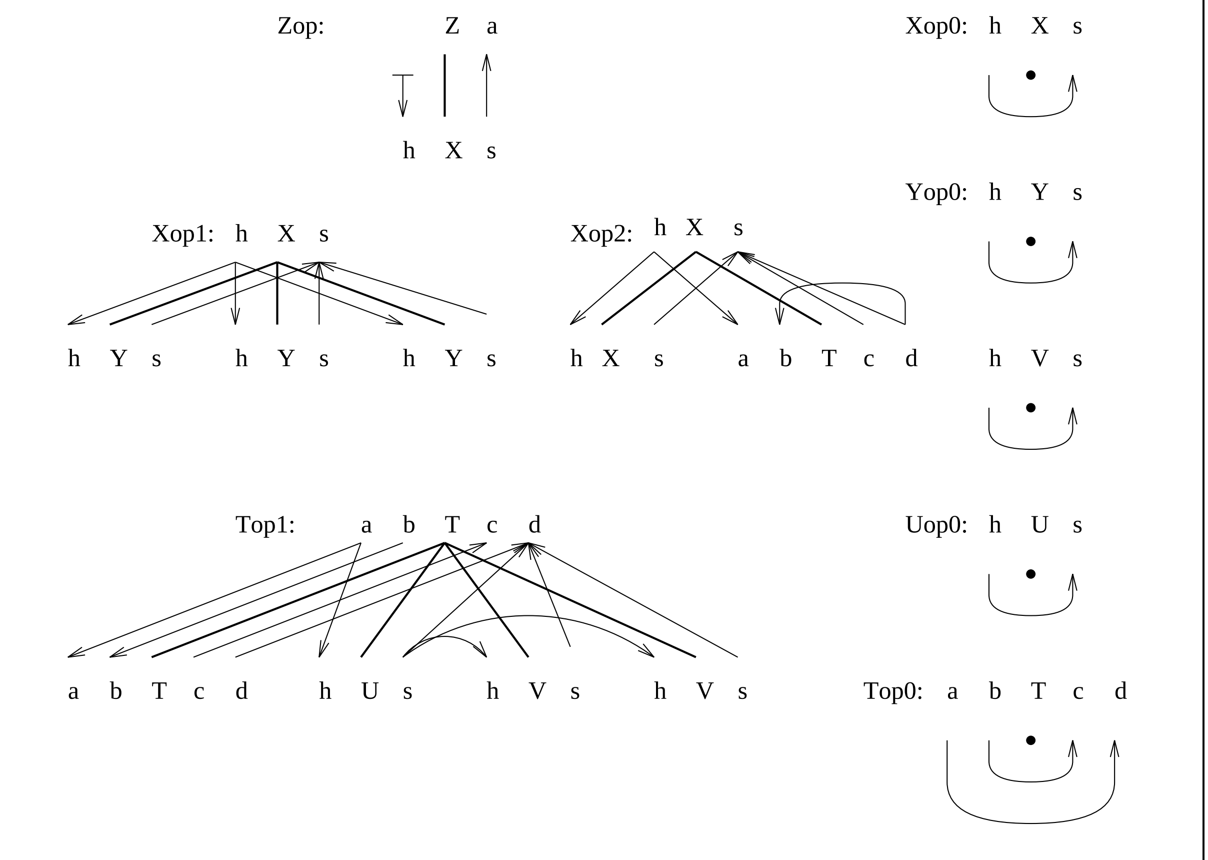


FIG. 1.5 - *Graphes de dépendance de G*

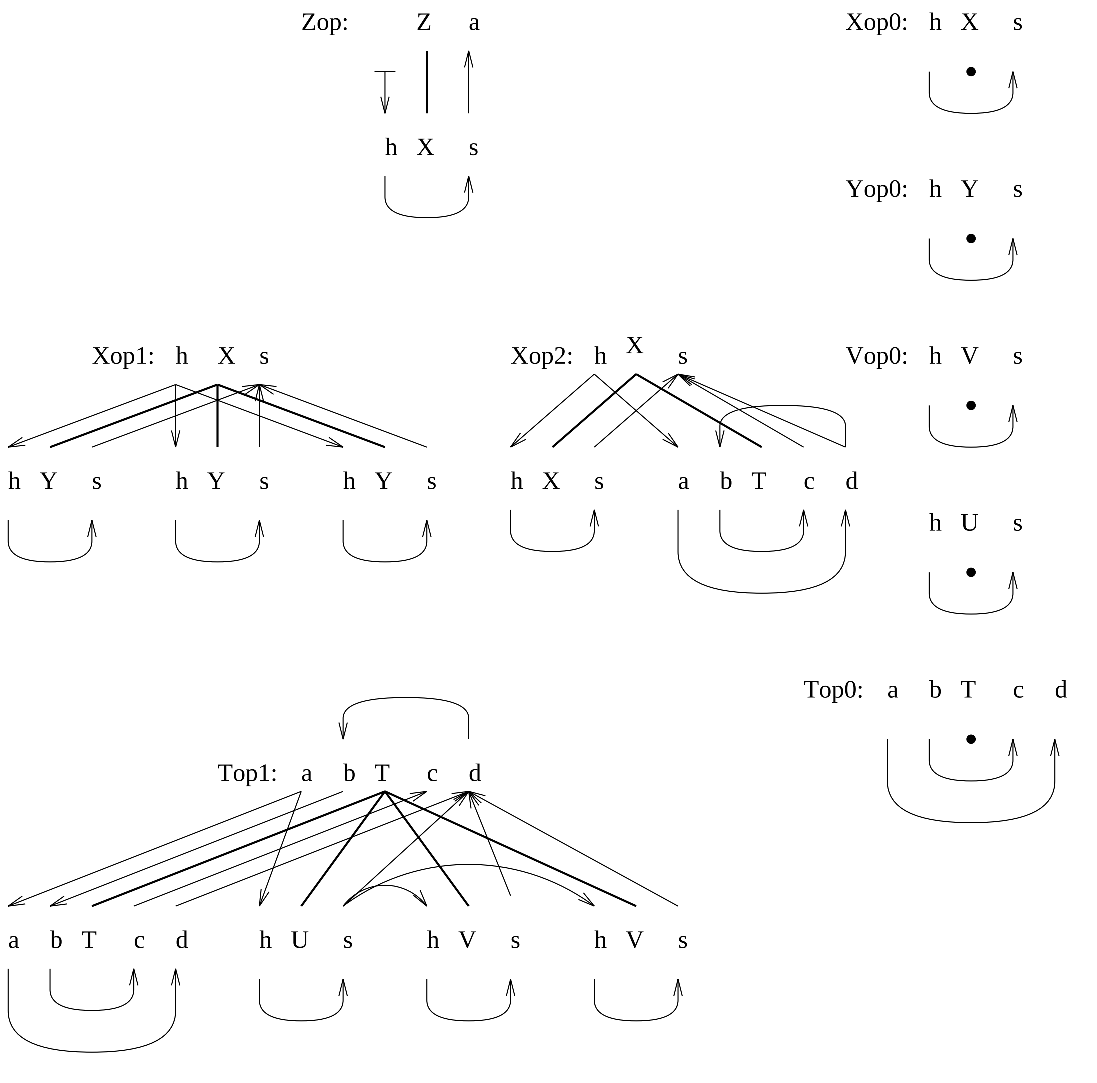


FIG. 1.6 - *Graphes de dépendance de G augmentés par les relations FNC/DNC*

Chapitre 2

Présentation générale de l'évaluation parallèle d'attributs

Avant de paralléliser l'évaluation, il est bon de s'interroger sur les différentes possibilités qui s'offrent à nous en ce qui concerne, d'une part les types d'architectures et les styles de parallélisation, et d'autre part, les différents types d'évaluations de grammaire attribuée. Ce chapitre se veut un bref tour d'horizon de quelques types d'architectures et des méthodes de parallélisation, ainsi qu'un état de l'art en ce qui concerne le parallélisme dans les grammaires attribuées.

2.1 Modèles d'architectures de machines parallèles

Il est à remarquer que les machines *data-parallèles* ne sont absolument pas indiquées pour l'évaluation de grammaires attribuées [Jou91]. Les structures manipulées par les grammaires attribuées n'étant pas uniformes, il est très difficile, voire impossible, de les plaquer à un modèle *data-parallèle*. En effet, de telles architectures fonctionnent très bien pour des structures de types tableaux, vecteurs... En grammaire attribuée, nous manipulons presque exclusivement des arbres, dont les nœuds ne sont pas tous de même type (pas le même nombre de fils, non-équilibrés...). Aucun travail à ce sujet n'a d'ailleurs été fait.

Nous nous intéressons donc plutôt à des machines de type MIMD. Parmi celles-ci, nous nous sommes consacrés aux machines à mémoire partagée. Nous pouvons ici distinguer deux classes de telles machines :

– à mémoire répartie : chaque processeur a sa propre mémoire. Cette dernière peut être vue par les autres processeurs, mais seulement au prix de communications entre les processeurs en cause.

– à mémoire partagée : les processeurs accèdent à la mémoire de façon indifférenciée.

Cette classification est loin d'être formelle. Elle veut pourtant caractériser les machines pour lesquelles il est important, pour des questions d'efficacité, de faire très attention au moment du stockage des données. Si des précautions ne sont pas prises, les performances du programme en seront gravement diminuées.

Au niveau utilisateur, la différence se fait sentir suivant la granularité du parallélisme. Plus le grain est fin, plus le surcoût engendré par les communications inter-processus doit être petit. Quand le grain est gros, il devient aisé de rentabiliser le coût des communications sophistiquées. Ainsi, suivant l'architecture que l'on considère, il faudra adapter sa parallélisation afin d'obtenir un grain très gros si on se place dans un cadre distribué où les communications sont coûteuses, et on pourra avoir un grain plus fin pour les machines à mémoire partagée, puisque les communications ont, dans ce cas, un coût moindre.

Mode de parallélisation

Un aspect plus logiciel du parallélisme concerne la communication entre les différentes tâches d'un modèle. La communication complète est donnée par le modèle asynchrone ; les tâches ne sont pas synchronisées entre elles, sauf au moment de leur initialisation. C'est à l'utilisateur de synchroniser les tâches quand cela s'avère nécessaire. L'utilisateur peut synchroniser les tâches entre elles à tout quel moment. Ce modèle permet à deux tâches de se synchroniser, et donc de communiquer, sur certains points (appelés points de synchronisation). En grammaire attribuée, les points de synchronisation sont les instances d'attributs. Si une tâche $T1$ a besoin d'une instance d'attribut a pour un calcul d'une instance d'attribut b et si a est calculé par $T2$, alors $T1$ et $T2$ vont devoir se synchroniser : $T2$ doit faire savoir à $T1$ que a est calculé. Ce modèle s'adapte très bien au graphe de dépendance $D(p)$.

Le modèle synchrone, quant à lui, synchronise les tâches père/fils au moment de la création et de la terminaison du fils. À part ces deux points de communication, les deux tâches ne peuvent pas se synchroniser. Deux tâches sœurs ne peuvent donc pas communiquer : elles sont totalement indépendantes.

2.2 Les différents modes d'évaluation

En ce qui concerne l'évaluation proprement dite des attributs, on peut également trouver différents critères qui changeront plus ou moins radicalement l'implantation et également le choix du parallélisme. Cette section contient un rappel des différents types d'évaluateurs. Cette classification des évaluateurs est plus ou moins indépendante de celle proposée dans le chapitre 1.

Calcul exhaustif

L'évaluation exhaustive des attributs consiste, pour un arbre d'entrée donné, à effectuer tous les calculs nécessaires à l'évaluation des attributs synthétisés de la racine. Si on a la possibilité de modifier l'arbre d'entrée, un calcul complet est réeffectué, même si la modification apportée ne porte que sur un petit nombre d'instances d'attributs. Dans le cas de la compilation, où on demande l'analyse du source une seule fois, cette solution est la plus répandue.

Calcul incrémental

Contrairement au calcul exhaustif, en cas de modification de l'arbre d'entrée, on veut effectuer un calcul partiel des attributs. Ce calcul ne touche que les attributs dont la valeur est susceptible d'être changée par la modification de l'arbre. Ce genre d'évaluation est indispensable dans un cadre interactif, où le nombre d'attributs à recalculer est souvent peu important et où le temps de réponse doit être minimal. Reps [Rep84] fût le premier à s'intéresser à l'évaluation incrémentale et à proposer un algorithme de recalcul incrémental des attributs.

Calcul dynamique de l'ordre d'évaluation

Nous avons vu que les règles sémantiques induisent un ordre des occurences d'attributs sur chaque production. Pour un arbre d'entrée donné, il est possible, pour toute grammaire non-circulaire, de trouver un ordre total d'évaluation sur cet arbre, qui est compatible avec les ordres de chacune des productions donnés par $D(p)$. Le problème de l'évaluateur est de connaître cet ordre. La solution dynamique consiste à le recalculer à chaque exécution de l'évaluateur. Ces méthodes permettent d'évaluer les classes de grammaires les plus grandes, mais se révèlent complètement inefficaces quant à l'implantation.

Calcul statique de l'ordre d'évaluation

L'ordre d'évaluation est calculé une fois pour toutes au moment de la création de l'évaluateur. L'ordre déterminé est général en ce sens qu'il doit être valide quel que soit l'arbre d'entrée. Les grammaires attribuées acceptées sont moins nombreuses, mais les évaluateurs produits sont beaucoup plus efficaces car l'ordre n'a pas à être recalculé à chaque fois.

2.3 Évaluation concurrente des grammaires attribuées

Les compilateurs sont des programmes largement utilisés, d'où l'intérêt de les optimiser, tant au niveau de leur écriture qu'au niveau de leur rapidité d'exécution. Les grammaires attribuées ont très vite été utilisées pour la spécification de compilateurs, répondant ainsi à l'une des optimisations. De nombreuses méthodes ont été utilisées, jouant sur les différentes classes de grammaire.

En ce qui concerne la parallélisation de compilateurs, la construction à la main est bien sûr possible [Lip79, Fra83, Van88, SWJ88, GZZ89]. Cette construction manuelle n'est toutefois pas souhaitable, d'une part parce que source d'erreur, et d'autre part car c'est un travail long et difficile. Enfin, les méthodes employées pour la parallélisation d'un compilateur pour un langage ne s'appliquent pas immédiatement à un compilateur d'un autre langage. Là encore, les grammaires attribuées [Knu68, DJL88, Alb91b] font leurs preuves. Car non seulement elles sont un excellent outil de spécification pour les compilateurs [Kas91], mais de plus, la spécification sous forme de graphe *data flow* s'adapte à merveille pour l'étude du parallélisme. Un compilateur spécifié par grammaires attribuées peut être parallélisé **automatiquement**.

La possibilité d'exploiter le parallélisme dans les grammaires attribuées a été reconnue très tôt, puisqu'elle était à la base de la première implantation faite par Fang dans le système FOLDS [Fan72]. La méthode utilisée était totalement dynamique, elle consistait à allouer un processus pour chaque instance d'attribut. Le processus, une fois lancé, attend que tous les attributs dont il dépend soient calculés. Cette méthode, bien que correcte théoriquement, ne peut être implantée efficacement.

En effet, le problème inhérent aux méthodes dynamiques est le surcoût nécessaire pour garantir l'ordre d'évaluation. Aussi s'est-on intéressé très tôt à des méthodes statiques.

Pour différencier les types d'évaluateurs certains auteurs ont essayé d'en établir une classification. Kuiper [Kui89, KuS90] a défini le concept de *distributeur*. Un *distributeur* est une fonction qui, à chaque instance d'attribut d'un arbre, associe un processus qui sera chargé de l'évaluer. Les *distributeurs* doivent répondre à deux critères :

- des attributs indépendants doivent, autant que possible, être associés à des processus différents ;

- la communication entre processus doit être minimisée.

Kuiper définit deux types de *distributeurs*. Le premier est formé par les distributeurs basés sur la structure de l'arbre. Dans cette catégorie, les attributs d'un même nœud sont calculés par le même processeur. Un distributeur partitionne l'arbre et associe un processus à chaque partition. Le partitionnement le plus utilisable est celui qui partage l'arbre en régions connexes. Cela rend le *distributeur* plus simple, car une région est définie par l'ancêtre commun. D'autre part, cela diminue le nombre de communications inter-processus. Ce modèle permet d'évaluer dans un contexte distribué puisqu'il fixe statiquement un processus par rapport à une région d'un arbre. En effet, l'arbre étant découpé un régions connexes, l'évaluateur chargé de l'évaluation d'une région trouve toutes les données localement. La communication entre processus n'a lieu qu'aux nœuds frontières de la région.

La deuxième forme des *distributeurs* de Kuiper est basée sur les attributs. Toutes les instances d'un même attribut sont calculées par un même processus. Ce genre de parallélisation est très utile pour calculer deux sémantiques totalement indépendantes. Il permet d'avoir plusieurs processus en calcul sur un même nœud à un instant donné.

Kuiper introduit ensuite une forme combinée des deux types précédents. Le *distributeur* partitionne l'arbre, puis alloue un processus par attribut d'une même région.

Les travaux de Kuiper sont intéressants car ils jettent une base pour la classification d'évaluateurs parallèles. Pourtant, il y manque un type particulier, celui basé sur le graphe de dépendance. Le partitionnement ne se fait ni sur l'arbre ni sur les attributs, mais partitionne le graphe de dépendance sur l'arbre en régions connexes. Pourtant, certaines méthodes de parallélisation, dont la nôtre, sont basées sur de tels *distributeurs*.

Un autre aspect des travaux de Kuiper est un algorithme qui permet de calculer statiquement l'indépendance de deux instances d'attributs d'une grammaire attribuée donnée. Cet algorithme est exponentiel mais peut être optimisé par des méthodes d'analyse de flot de grammaires [MöW91].

2.3.1 Méthodes statiques

Notre approche, fondée sur les séquences de visites, a déjà intéressé un certain nombre d'auteurs [Sch79, Kle91, Zar90, Mar90, Alb91a]. Tous ces travaux sont fondés sur l'incontournable concept synchrone **fork/join** et travaillent avec des grammaires l-ordonnées, cette classe étant la plus grande famille permettant de déterminer, de façon statique, un ordre total sur les attributs d'un non-terminal. L'ordre sur les attributs des non-terminaux, donné par le concept de grammaires l-ordonnée, est conservé, mais l'ordre à l'intérieur d'une production n'est pas séquentialisé. Si cette classe a particulièrement intéressé les auteurs, c'est qu'elle permet de connaître, de façon statique, un grand nombre d'informations pour le parallélisme, comme nous le verrons dans le chapitre 3.

Il existe toutefois plusieurs variations autour de cette méthode. Schell [Sch79] a été le premier à publier le modèle de la manière dont il est présenté ici. Il présente des modifications aux algorithmes de construction d'évaluateur de Kennedy et Warren [KeW76], Kastens [Kas80] et Bochmann [Boc76], tous fondés sur l'idée d'effectuer des visites aux sous-arbres en parallèle à chaque fois que cela est possible. Cependant, il est resté à un haut niveau d'abstraction et ne décrit pas les opérations parallèles autrement qu'avec *fork/join*. Autrement dit, il a montré qu'il était possible de paralléliser des grammaires attribuées l-ordonnées, mais sans souci de la moindre méthode de parallélisation qui, comme nous le verrons, est indissociable de la notion d'efficacité.

Zaring [Zar90], quant à lui, s'est intéressé à plusieurs types d'évaluation : il présente des algorithmes pour les modèles synchrones et asynchrones, dans le cas d'évaluation exhaustive et incrémentale.

2.3.2 Évaluation incrémentale

Kaplan et Kaiser [KaK86] ont proposé une modification de l'algorithme de propagation dynamique de Reps, utilisable pour l'évaluation incrémentale et qui s'adapte bien aux machines à mémoire partagée. Le graphe de dépendance (le *modèle* suivant la terminologie de Reps) est partagé par tous les processus.

Une procédure initiale, dont les paramètres sont l'arbre d'entrée t et un nœud r avec des attributs inconsistants, renvoie tous les attributs de t sans précédent dans un ensemble s. Pour chaque élément a de s, on lance un processus qui recalcule a et, si celui-ci a été modifié, il ajoute au graphe de dépendance l'ensemble des attributs qui dépendent de a puis retire a du graphe de dépendance.

Les procédures d'expansion et de réduction du graphe sont des régions critiques puisqu'elles modifient le modèle. Il est toutefois possible de limiter cette zone critique, car des modifications sur des régions disjointes peuvent s'effectuer simultanément, comme le soulignent [KaK86]. La synchronisation est directement intégrée à l'algorithme qui calcule dynamiquement si tous les précédents d'une instance d'attribut ont été calculés avant de lancer un processus asynchrone pour son évaluation.

Kaplan et Kaiser ont également modifié cet algorithme pour qu'il fonctionne en cas de modification de plusieurs nœuds. À notre connaissance, aucune implantation de cet algorithme n'a été faite.

Par la suite, une adaptation a été effectuée pour porter l'algorithme sur des machines distribuées, en utilisant une distribution basée sur un découpage de l'arbre. Le modèle est alors partagé suivant le découpage de l'arbre, et il est maintenu localement par chaque processus.

Zaring [Zar90] utilise, pour son évaluation incrémentale parallèle, la méthode séquentielle basée sur les séquences de visites décrite par Engelfriet [Eng84] ainsi que celle donnée par Reps [Rep84].

2.3.3 Méthodes pour architectures distribuées

Dans le cas d'architectures distribuées, il est nécessaire de rendre les données aussi locales que possible au processus qui les utilise, de façon à minimiser les communications entre processus. Böhm et Zwanepoel [BöZ87] ont donné une méthode de parallélisation des grammaires attribuées dans un système distribué. L'arbre d'analyse est partitionné au moment de sa construction par le *parser* et chaque sous-arbre est donné à un processus. Les nœuds, où le découpage de l'arbre est effectué, sont donnés par l'utilisateur par ajout d'information dans la grammaire. La méthode d'évaluation est une combinaison de méthode dynamique et de méthode statique. L'évaluation dynamique est utilisée pour tout nœud qui se trouve sur un chemin entre la racine de l'arbre local et une feuille d'un sous-arbre distant. L'appartenance d'un nœud n du sous-arbre à un tel chemin est calculé lors de la reconstruction du sous-arbre à partir de la forme linéarisée en provenance du réseau. Si ce n'est pas le cas, les attributs de n seront évalués par un évaluateur statique à base de séquences de visites, et aucune information de dépendance n'est calculée. Sinon, les attributs de n sont ajoutés au graphe de dépendance du processus pour une utilisation par l'évaluateur dynamique. Quand tous les prédécesseurs d'un attribut évalué statiquement sont calculés, l'évaluateur dynamique appelle l'évaluateur statique sur cet attribut.

L'efficacité d'un tel modèle repose avant tout sur le choix des nœuds de découpage de l'arbre. Ce choix doit minimiser la communication à travers le réseau. Böhm et Zwanepoel ont implanté leur méthode sur des stations de travail Sun-2 connectées par un réseau Ethernet à 10Mbit. Les machines utilisaient un système expérimental à base de messages. L'exemple choisi était une grammaire attribuée décrivant un compilateur d'un sous-ensemble de PASCAL vers l'assembleur de VAX. Les mesures sur des programmes d'environ 1000 lignes donnaient un *speedup* maximum de 3 pour 5 machines utilisées. Une analyse du comportement des évaluateurs a montré que seulement 0,1% des attributs sont évalués dynamiquement.

2.3.4 Autres approches

Bien que les évaluateurs à base de séquences de visites soient très utilisés, ce ne sont pas les seuls. Klein [Kle91, Kle92] a introduit la notion de segments. Le graphe de dépendance de chaque production est partitionné en un ensemble de segments, qui vérifient la condition suivante : 2 attributs d'un non-terminal X sont soit toujours dans le même segment, soit toujours dans 2 segments différents. Ce partitionnement est effectué statiquement au moment de la construction de l'évaluateur. À l'exécution, les segments locaux aux productions sont réunis (*vertically melted*), dès qu'ils ont des instances d'attributs en commun. Cela conduit finalement à un partitionnement global de l'arbre d'entrée en un certain nombre de segments. La notion de segment est une notion très générale, on retrouve donc la classification de Kuiper [Kui89] dans les différents types de segments possibles. Pourtant, la notion de segment autorise le découpage des attributs d'un même non-terminal, ce qui n'était pas possible avec la notion de distributeur. Toutefois, la construction dynamique des segments de l'arbre peut devenir coûteuse, car elle peut nécessiter plus d'une passe pour son calcul. Ensuite, les attributs d'un même segment sont ordonnés alors que ceux de segments distincts peuvent se calculer en parallèle. Les dépendances entre deux segments sont résolues par des points de synchronisation. La synchronisation est formalisée sous forme d'évènements, indépendamment de l'implantation sous-jacente. Klein met en évidence une nouvelle classe de grammaires, la classe POAG (pour Parallel Ordered Attributes Grammars). Cette classe, plus large que la classe OAG, permet de faire statiquement un grand nombre de calculs, afin de diminuer la partie dynamique due au type de segment utilisé. Klein étudie l'influence de la segmentation sur le grain du parallélisme [Kle92]. Un *scheduler* simulant une architecture multi-processeur à communications asynchrones a été utilisé pour faire des tests. Ceux-ci ont conduit a un *speedup* maximum de 4,67 pour 6 processeurs simulés (le maximum est atteint pour un programme relativement petit de 1000 lignes d'un Pascal simplifié). Pour 12 processeurs simulés, le maximum de *speedup* est de 9.79 atteint sur un texte d'entrée de 5000 lignes ! Ce qui représente 162 processus au total.

Une autre approche a été abordée par Klaiber et Gokhale [KlG92]. Leur méthode, basée sur la statégie d'évaluation de Katayama [Kat84], consiste à calculer le parallélisme statiquement, mais ils décident au moment de l'exécution lequel du code séquentiel ou du code parallèle doit être utilisé. Ce choix se fait à partir de poids calculés qui permettent une estimation du temps d'exécution d'une tâche. Si ce temps est inférieur au surcoût nécessaire au lancement d'une tâche, le code séquentiel est utilisé. Pour une meilleure approximation, l'écrivain de la grammaire peut spécifier la durée des règles sémantiques. Les poids sont des attributs synthétisés, placés sur chaque non-terminal, et les dépendances sont faites pour que ces attributs soient évalués par une première passe de l'évaluateur. La nature purement synthétisée de ces attributs fait qu'ils peuvent être calculés par l'analyseur syntaxique *LR*. Les inconvénients de cette méthode sont d'une part le fait d'effectuer une passe supplémentaire, d'autre part l'utilisateur doit spécifier des durées. La méthode n'a pas été complètement implantée, mais des simulations ont donné des résultats peu encourageants ; pour 10 processeurs, l'utilisation des processeurs est inférieur à 35%. Après ces mauvais résultats, Klaiber et Gokhale se sont intéressés à la transformation de la grammaire d'entrée pour ajouter du parallélisme ; la transformation consiste à remplacer les productions $X \to YX$ par des productions de la forme $X \to Y+$. D'après eux, les résultats de leur simulation étaient dus au fait que les arbres ne sont pas équilibrés. Ainsi, certaines parallélisations, pourtant intéressantes, étaient écartées.

Enfin, nous pouvons noter une implantation effectuée par Saraiva et Henriques (du département d'informatique de l'université de Minho Braga Portugal), non publiée à notre connaissance. Elle a été effectuée sur un *Solbourne Computer-Serie 700-* à 4 processeurs Sparc. Cette architecture est à mémoire partagée. Leur système comporte trois types de processus, le Principal, le Gestionnaire d'Évènements et les processus qui s'occupent de l'évaluation proprement dite des attributs. La communication entre deux types de processus est basée sur le concept d'évènement. Le processus principal analyse la grammaire et construit l'arbre qui est stocké en mémoire partagée. Un *splitter* parcourt alors l'arbre suivant une stratégie prédéfinie, et découpe l'arbre en sous-arbres. Chaque sous-arbre est envoyé au gestionnaire d'évènements qui l'enverra lui-même à un processus d'évaluation. Quand tous les sous-arbres sont envoyés, l'évaluation commence, le processus principal attend une notification de la part du gestionnaire d'évènements qui lui indiquera que l'évaluation est terminée. Un procédé similaire est utilisé pour la communication entre le gestionnaire d'évènements et les processus d'évaluation.

Le découpage de l'arbre est fait semi-automatiquement, c'est-à-dire que l'utilisateur doit lui-même indiquer les nœuds de découpage de l'arbre.

Les résultats de l'implantation sont assez encourageants, puisqu'ils atteignent un *speedup* supérieur à trois pour quatre processeurs.

2.4 Conclusion

De nombreux travaux ont été effectués dans la parallélisation des grammaires attribuées. De l'évaluation incrémentale à l'évaluation exhaustive, de l'évaluation statique à l'évaluation dynamique, les auteurs ont toujours voulu montrer que les grammaires attribuées étaient parallélisables. Malheureusement, nous pouvons déplorer le petit nombre d'implantations réelles, sur de véritables machines parallèles (et pas seulement un réseau de station monoprocesseur). La plupart des auteurs se sont arrêtés soit à un stade théorique, soit à une simulation sur machine séquentielle. Notre étude, dans la ligne de certains travaux déjà effectués, apporte l'expérience de la parallélisation des grammaires attribuées sur des machines parallèles.

Chapitre 3

La parallélisation

Les grammaires attribuées contiennent intrinsèquement un parallélisme très important. Les graphes de dépendances $DP(p)$ induisant un ordre partiel sur les attributs d'une production, il est assez aisé de détecter ce parallélisme au niveau de chaque production. Il s'agit de trouver des attributs indépendants dans le graphe $DP(p)$. Par conséquent, la parallélisation de l'évaluation des grammaires attribuées ne devrait poser aucun problème, du moins dans son côté formalisation. En effet, c'est de la parallélisation sur des graphes, et le graphe de dépendance nous assure que le parallélisme ainsi trouvé, sera sémantiquement correct. Le paradoxe vient du fait que les grammaires attribuées contiennent beaucoup trop d'opportunités de parallélisation. Le grain est très fin, ce qui correspond à un graphe contenant une proportion d'arcs importante par rapport au nombre de nœuds. Le nombre de tâches (même pour un arbre d'entrée de petite dimension) est grand et leur durée, ou temps d'exécution, est courte. Une parallélisation trop naïve entraînerait un surcoût en temps beaucoup trop important. Nous nous apercevons vite qu'il faut choisir très soigneusement les tâches si nous voulons obtenir une bonne efficacité, c'est-à-dire un rapport temps d'exécution parallèle/temps d'exécution séquentielle correspondant au nombre de processeurs mis en œuvre (*speedup*).

Les stratégies de parallélisation dépendent beaucoup de l'implantation que l'on veut faire et de l'architecture sur laquelle s'effectue cette implantation.

Paralléliser les grammaires attribuées sur un réseau de machines séquentielles ou sur une machine à mémoire répartie pose le problème de la répartition de l'arbre sur les différents processeurs afin de limiter au maximum les communications. En effet, plus un processus nécessite d'obtenir des valeurs d'attributs qui sont sur un autre processeur, plus le coût du parallélisme par rapport à l'exécution séquentielle sera important. Notre approche étant basée sur les machines à mémoire dite globale, le problème de la localisation des données est inexistant, du moins nous le considérons comme tel. Nous supposons donc que l'arbre est stocké dans la mémoire partagée, sans aucune précaution particulière. Toutefois, les algorithmes décrits par [Kui89] pour la détection de dépendances distantes peuvent être utilisés lors de l'optimisation mémoire.

Notre propos se rapporte particulièrement aux machines MIMD à mémoire partagée, comme par exemple le B8000 de Sequent, le Multimax de Encore, et la KSR1 de Kendall...

3.1 Type de parallélisme : notre choix

L'avantage de la classe des grammaires l-ordonnées est qu'un maximum d'informations concernant la manière d'évaluer les attributs peut être calculé de façon statique, au moment de la génération de l'évaluateur. Cette assertion s'applique à certaines informations parallèles, mais pas à toutes (ex : on ne peut pas connaître statiquement la durée des tâches, ni même le temps d'exécution des fonctions sémantiques). Notre optique étant de profiter des déterminations statiques, nous allons nous placer, dans notre sélection du parallélisme, dans le cadre de la classe des grammaires l-ordonnées et nous limiter à la détection qui peut être faite au moment de la génération de l'évaluateur. Pour rappeler l'exemple précédent, nous n'essaierons pas de calculer dynamiquement des informations pour le parallélisme, comme la profondeur de l'arbre, ce qui permettrait d'approcher la durée d'une visite (une visite sur un sous-arbre de grande profondeur a de fortes chances de prendre plus de temps qu'une visite sur un nœud feuille).

D'autre part, il est bon de rappeler que nous utilisons la transformation de Parigot [Par87] qui nous permet d'accepter des grammaires de classe FNC. Nous pouvons donc ainsi paralléliser des grammaires de la classe FNC. Enfin, pour des raisons de clarté, nous supposons que chaque partie du sélecteur d'argument est réduite à un seul attribut. Ce n'est pas une limitation pratique, puisque nous avons effectué une implantation réelle. En fait, cela revient seulement à considérer les attributs d'une même partie comme étant un seul attribut. Les dépendances sur cet attribut sont issues de tous les attributs qui appartiennent à la partie.

3.1.1 Le modèle

Le modèle choisi pour notre parallélisation est simple ; il est basé sur le concept synchrone *fork/join*. Aucune communication entre deux processus n'est nécessaire en dehors de l'initialisation d'une tâche, et de sa terminaison. La raison du choix d'un modèle synchrone s'explique par le fait que nous souhaitons que l'arbre d'entrée ne contienne aucune information concernant l'évaluation. Notre méthode de parallélisation vise donc à ne pas stocker d'information relative au parallélisme sur les nœuds de l'arbre. Le chapitre 6, quant à lui, montrera comment il est possible de sortir les attributs de l'arbre, comme c'est déjà le cas en séquentiel. Dans cette optique de libérer l'arbre d'entrée des informations d'évaluation, nous nous sommes aperçus que le modèle asynchrone n'était pas adapté. En asynchrone, il n'y a aucune raison de synchroniser les tâches à l'entrée d'une production. De cette façon, nous obtenons, au niveau d'une production donnée, des tâches qui peuvent venir (qui ont été créées) de n'importe où dans l'arbre. Par conséquent, deux tâches différentes qui veulent se synchroniser au niveau d'un attribut de l'arbre, peuvent ne pas se connaître du tout. Dans ce cas, la seule façon de les synchroniser est d'ajouter une information sur le nœud de l'arbre portant l'attribut, qui est la seule donnée que les deux tâches ont en commun.

Une tâche parallèle communique donc avec son père seulement lors de son lancement et de sa terminaison. La communication entre père et fils se restreint à la donnée des attributs d'entrée du fils et la communication fils/père à la donnée des attributs de sortie du fils.

Au niveau des conflits d'accès entre tâches, conflits induits par le parallélisme, nous pouvons envisager que deux tâches arrivent sur un nœud de l'arbre en même temps. Si elles nécessitent toutes deux la valeur d'un attribut non encore calculé, il faut empêcher que toutes deux calculent cet attribut. Là encore, les grammaires l-ordonnées sont avantageuses à paralléliser, puisque le fait de ne paralléliser que des visites fait qu'une seule tâche peut accéder à un nœud de l'arbre donné à un instant précis. Ainsi, aucune précaution n'a à être prise pour empêcher deux tâches de calculer le même attribut puique ce cas ne peut **jamais** se produire.

Toutes les informations concernant les communications sont connues statiquement, comme le nombre de tâches lancées, car tout se passe strictement à l'intérieur d'une seule et même production.

Le modèle *fork/join* choisi implique un graphe de dépendance très particulier, c'est-à-dire avec des *fork/join* bien parenthèsés. Or, le graphe de dépendances calculé par l'analyse statique de la grammaire a pour particularité d'être acyclique et d'avoir des arêtes d'articulation qui partagent le graphe suivant les différentes visites sur la production. Chaque composante n'a par conséquent qu'un seul point d'entrée et un seul point de sortie. À part ces restrictions, le graphe n'a pas de caractère particulier. La définition ci-dessous donne le type des graphes traités par notre méthode. L'algorithme de parallélisation a pour but de passer d'un graphe de dépendance G issu de l'analyse statique à un graphe G' qui respecte les bonnes propriétés. En fait, l'algorithme est chargé d'ajouter les sommets `COBEGIN/COEND` qui assurent la synchronisation. Ces sommets devront obligatoirement être bien parenthèsés de façon à respecter le modèle *fork/join*.

Pour que le procédé soit valide, il faut bien entendu que toute dépendance de G induise une dépendance sur G'. Sans cette condition, deux attributs a et b tels que $a \to_{G'} b$ pourraient être calculés en parallèle. Ceci serait totalement aberrant.

Soit $GD = (V, E)$ le graphe de dépendance d'une production $p \in P$ augmenté par les dépendances des relations l-ordonnées (et par inclusion des relations R_{FNC} et R_{DNC}).

À partir de GD, on construit G en enlevant les relations de transitivité.

L'algorithme de détection du parallélisme transforme alors ce graphe G en un autre graphe $G' = (V', E')$ appelé graphe de parallélisation, qui vérifie les propriétés suivantes :

- $V' = V \cup V''$ avec $V'' = \{Cobegin(n), Coend(n)\}_{n \geq 0}$
- E' vérifie : soit $a \in V'$, $S_a = \{s \in V' \,|(a, s) \in E'\}$, $P_a = \{p \in V' \,|(p, a) \in E'\}$
  i) si $(a, b) \in E$ alors $\exists (x_1, \ldots, x_k) \in (V'')^k | (a, x_1) \in E' \wedge \forall i \in [1, k-1] (x_i, x_{i+1}) \in E' \wedge (x_k, b) \in E'$

ii) si $a =$Cobegin(n) alors $card(S_a) > 1$

iii) si $a \in \{$Coend(n)$\} \cup V$ alors $card(S_a) = 1$

iv) si $a \in \{$Cobegin(n)$\} \cup V$ alors $card(P_a) = 1$

v) si $a =$Coend(n) alors $card(P_a) > 1$

vi) si $a =$Cobegin(n) et $a \rightarrow^+ b$ alors soit $b \rightarrow^+$ Coend(n) soit Coend(n) $\rightarrow^+ b$ où $\rightarrow^+ = E^+$.

vii) si $a =$Coend(n) et $b \rightarrow^+ a$ alors soit Cobegin(n) $\rightarrow^+ b$ soit $b \rightarrow^+$ Cobegin(n) où $\rightarrow^+ = E^+$.

La propriétés i) signifie que si une dépendance existait entre deux attributs a et b, alors la dépendance entre les sommets correspondants du nouveau graphe existe encore, au pire par le biais d'une transitivité avec des nœuds `COBEGIN` ou `COEND`. C'est-à-dire qu'entre deux attributs a et b dépendants, nous n'ajoutons que des sommets `COBEGIN` ou `COEND`. Les propriétés ii) et iii) indiquent que seuls les sommets `COBEGIN` peuvent et doivent avoir plusieurs successeurs, c'est-à-dire que partout ailleurs, l'ordre d'évaluation est obligatoirement fixé. iv) et v) indiquent que seuls les sommets `COEND` peuvent et doivent avoir plusieurs prédécesseurs, c'est-à-dire que ces sommets représentent les seuls points de synchronisation. Les propriétés vi) et vii), quant à elles, forcent le bon parenthèsage de `COBEGIN/COEND`, comme le démontre la preuve ci-dessous.

Preuve : On veut démontrer Cobegin(1) $\rightarrow^+$ Cobegin(2) $\Leftrightarrow$ Coend(2) $\rightarrow^+$ Coend(1)

- ($\rightarrow^+$) Supposons que Coend(1) $\rightarrow^+$ Coend(2)

  Soit a tel que $a \rightarrow Coend(2)$ et a et $Coend(1)$ incomparables.

  vii) $\Rightarrow Cobegin(2) \rightarrow^+ a$ donc $Cobegin(1) \rightarrow^+ Cobegin(2) \rightarrow^+ a$

  Contradiction : puisque $Cobegin(1) \rightarrow^+ a$ d'après vi) on devrait avoir $Coend(1)$ et a comparables.

- Le sens $\Leftarrow$ se démontre de la même manière puisque les propriétés sont symétriques.

□

Le graphe G', transformé du graphe initial G, met en évidence le parallélisme mis en œuvre sur chaque production. Nous remarquons que les restrictions imposées font que tout le parallélisme d'une grammaire n'est pas utilisé. La figure 3.1 montre un exemple de la limitation de cette parallélisation. Dans le graphe de parallélisation, le calcul de f est conditionné par le calcul préalable de e, alors que dans le graphe de départ, cette condition n'existe pas. Notre parallélisation induit donc de nouvelles dépendances. Il est intéressant de comprendre d'où viennent ces dépendances supplémentaires.

Dans la figure 3.1, nous aurions envie d'avoir les deux tâches parallèles A et B. Mais la tâche B nécessite le résultat intermédiaire d de la tâche A. Après avoir "prévenu" B,

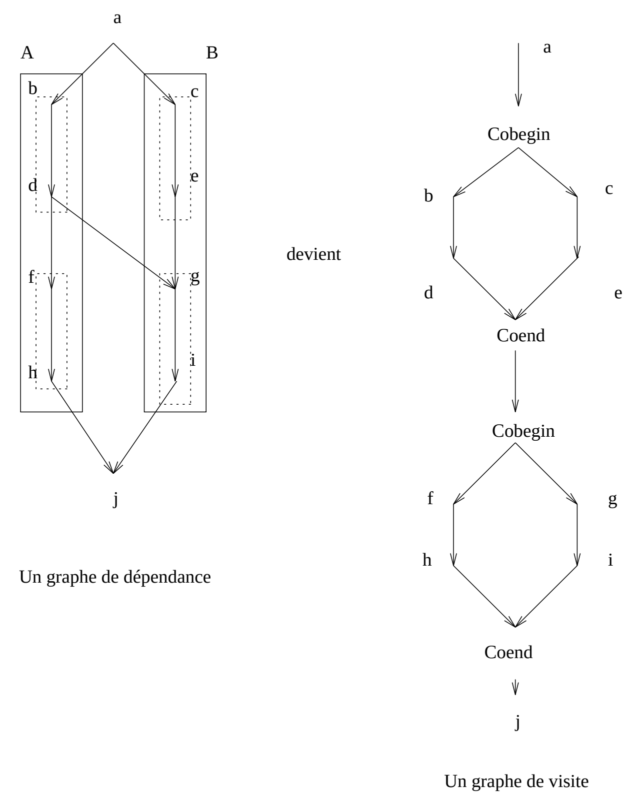


FIG. 3.1 - *Rendez-vous entre deux tâches parallèles.*

A pourrait très bien continuer son travail. Malheureusement, le modèle synchrone oblige deux tâches à se synchroniser par un *join* dès que celles-ci veulent communiquer, la tâche B ne pouvant pas, par définition, se poser la question “l’attribut est-il calculé ?”. Ces points de communications unilatéraux entre deux composantes parallèles nous obligent à ajouter des dépendances. Nous pouvons remarquer qu’en mode asynchrone, ils se traduisent par des rendez-vous entre deux processus et non en un *join*, ce qui permet de conserver les dépendances “naturelles”.

Inconvénient de la méthode. Une grande “quantité” de parallélisme nous échappe. Deux instances d’attributs indépendants au sens du graphe de dépendance sur l’arbre t, ne sont pas forcément calculés en parallèle. Par exemple, deux attributs d’une même visite sur un même non-terminal sont automatiquement évalués en séquentiel. La figure 3.2 montre un exemple où une parallélisation plus fine serait beaucoup plus avantageuse.

Avantage de la méthode. Le parallélisme à l’intérieur des grammaires attribuées est important et la granularité est extrêmement fine. Il n’est pas inutile de “bien” sélectionner les tâches à lancer pour augmenter cette granularité et ainsi diminuer le surcoût.

La détection du parallélisme sous forme de visite est intéressante, puisqu’elle nous donne

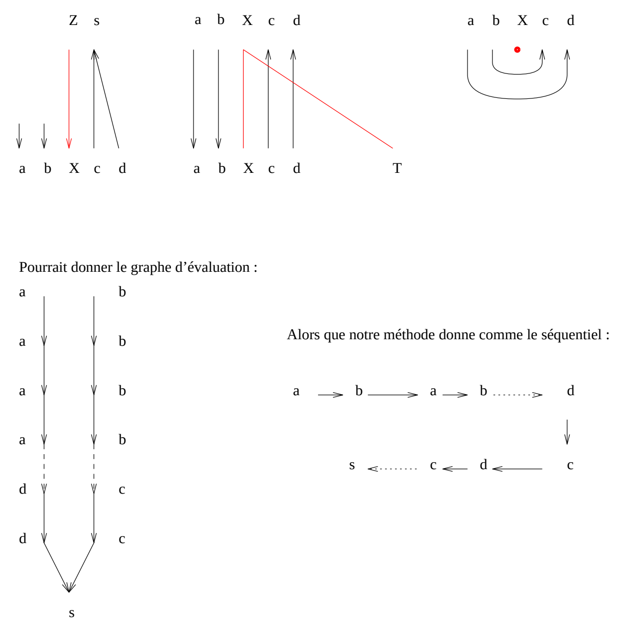


FIG. 3.2 - *Attributs parallèles d'une même visite*

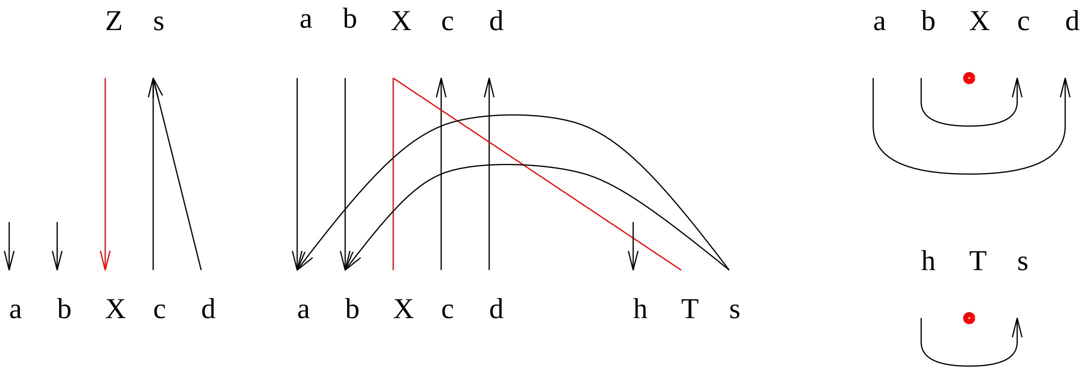
des tâches parallèles ayant le moins possible à communiquer les unes avec les autres. De cette façon, le surcoût induit par la parallélisation est minime et l'analyse est purement statique.

– Nous connaissons statiquement l'imbrication des processus. Un processus lance ses fils et récupère ensuite leurs résultats. La gestion des processus est alors très simple, ce qui diminue le surcoût.

– L'inconvénient signalé précédemment n'est pas aussi déterminant qu'il y paraît. La figure 3.3 montre que, sur un exemple similaire à celui de la figure 3.2, le comportement des processus parallèles est ici totalement différent. Dans un cas, les deux branches sont "très" parallèles (figure 3.2), c'est-à-dire que les deux tâches n'ont pas besoin de communiquer, sauf une fois à la fin, et dans l'autre cas (figure 3.3), une synchronisation est nécessaire à chaque étape avec la tâche qui calcule $T.h$ et $T.s$. Pourtant, dans les deux cas, les attributs des deux tâches sont indépendants.

– Par construction, les visites d'un même `COBEGIN/COEND` sont toutes sur des nœuds différents puisque deux visites sur un même nœud sont, dans l'évaluation l-ordonnée, obligatoirement dépendantes.

– Enfin, la garantie de n'avoir simultanément qu'une seule tâche par nœud évite un grand nombre de synchronisations. La seule tâche qui arrivera sur un nœud connaîtra les attributs déjà calculés et les attributs restant à calculer. Dans le cas de plusieurs tâches sur un même nœud, chaque tâche ignore si les attributs en charge des autres



Peut donner un graphe d'évaluation :

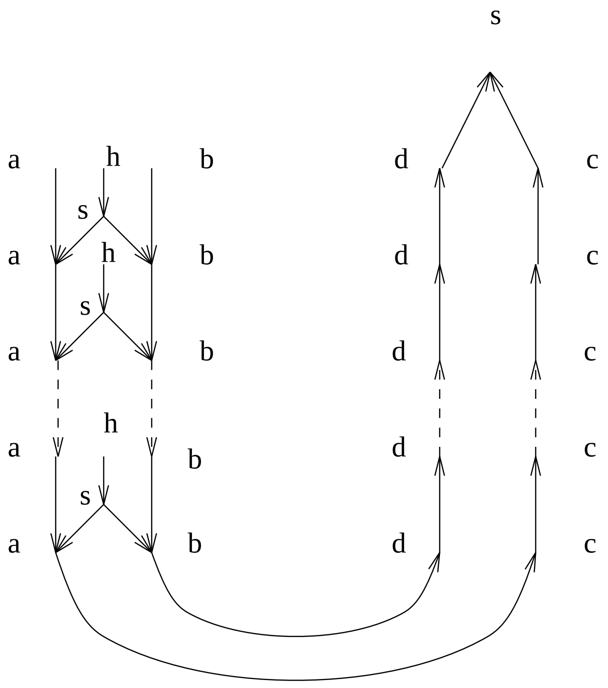


FIG. 3.3 - *Attributs parallèles d'une même visite avec points de synchronisation*

tâches sont ou non déjà calculés. Ceci nécessite une ou plusieurs synchronisations, suivant le nombre de tâches qui se partagent les calculs.

3.2 Algorithme de détection du parallélisme

3.2.1 Les intervalles doubles

Motivations

Une tâche parallèle, dans un modèle synchrone comme celui que nous avons choisi, ne communique qu'avec son père et ses fils. Au niveau du graphe de dépendance, nous remarquons qu'une tâche va correspondre à un sous-graphe dont la connection avec le reste du graphe respecte certaines propriétés. Pour présenter les choses de façon informelle, il est nécessaire qu'il n'y ait pas d'arc entrant à l'intérieur de la tâche. Autrement dit, tous les éléments de la tâche ne dépendent que des éléments de cette même tâche (sauf pour les premiers). Ceci correspond très bien à la notion d'intervalles dans un graphe. Cette propriété signifie qu'une tâche ne doit pas, en cours d'exécution, se poser la question de savoir si un attribut est, ou n'est pas calculé.

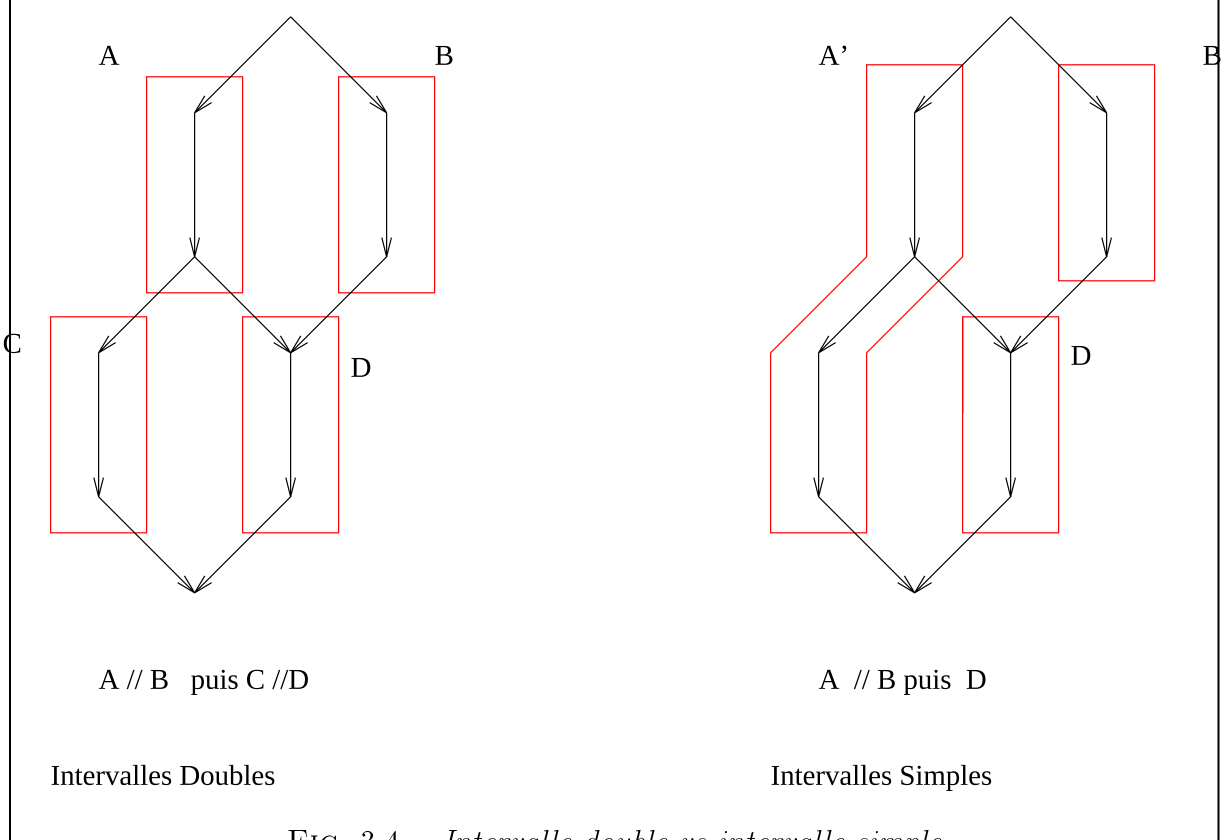


FIG. 3.4 - *Intervalle double vs intervalle simple*

Pourtant, sur certains graphes, nous pouvons trouver deux parallélisations différentes (voir figure 3.4). Dans l'exemple proposé, le cas $A//B$ puis $C//D$ est avantageux dans le cas de tâches de durées équivalentes. En effet, si toutes les tâches (A, B, C, D) durent 1 unité de temps, (seuls les sommets des tâches ont un coût) le temps d'exécution global est 2. Dans le deuxième cas, le temps d'exécution est de 3 (A' dure 2 et D dure 1). Pourtant, avec d'autres durées, la deuxième solution est parfois plus intéressante. Par exemple, si A dure 1, B dure 4, C dure 4 et D dure 1 : la première solution s'exécute en $4 + 4 = 8$ unités de temps, alors que la deuxième ne prend que $5 + 1 = 6$ unités de temps.

Nous choisissons la solution des intervalles doubles (section 3.2.1) qui présente l'avantage d'être plus efficace lors de tâches de durées équivalentes. D'autre part, nous ne voulons pas retarder le lancement d'une tâche sous prétexte que nous pouvons encore calculer certains attributs. Il est bien évident que ce n'est qu'une heuristique puisque nous ne sommes pas en mesure de déterminer statiquement la durée des différentes tâches. En effet, la durée d'un `EVAL` n'est pas connue, et la durée d'un `VISIT` dépend de la structure de l'arbre d'entrée qui est une donnée dynamique. La notion d'intervalle double représente le fait qu'une tâche n'a pas besoin de communiquer avec d'autres tâches tant que tous ses nœuds ne sont pas évalués, c'est-à-dire qu'elle n'a pas à prévenir d'autres tâches de l'état d'avancement de son travail, mais seulement de son terme. Ceci signifie qu'il ne doit pas y avoir d'arc sortant au milieu de la tâche. Seuls les nœuds minimaux (définition 1.1.14) ont le droit de communiquer. La définition suivante donne une formalisation de la notion d'intervalle double.

La figure 3.5 nous montre quelques exemples d'intervalles doubles en comparaison avec la notion d'intervalle.

Définition 3.2.1 (Intervalle double) *Soient $G = (V, E)$ un graphe et $D \subseteq V$ un ensemble de sommets. On appelle intervalle double issu de D pour G, un ensemble $I_D(D)$ tel que :*

- $D \subset I_D(D)$
- $\forall x \in I_D(D), \exists y \in D \mid x \to^+ y$ *ou* $y \to^+ x$
- $suivant_G(I_D(D) - Min_G(I_D(D))) \subseteq I_D(D)$
- $precedent_G(I_D(D) - Max_G(I_D(D))) \subseteq I_D(D)$

Définition 3.2.2 (Intervalle double maximal) *Un intervalle double $I_D(D)$ est dit maximal si et seulement si il est maximal au sens de l'inclusion.*

La maximalité d'un intervalle double représente l'ensemble des nœuds qui peuvent faire partie de la même tâche. Sans cette notion, nous arrivons à un parallélisme grossier où une communication est nécessaire après chaque sommet, puisqu'un sommet est à lui seul un intervalle double !

Remarque : Nous ne parlerons que d'intervalles doubles maxima. En conséquence, nous ferons souvent référence à des intervalles doubles maxima sans préciser qu'ils sont maxima.

Un intervalle double I est un ensemble d'attributs à évaluer (par évaluation directe ou par une visite). Dans notre modèle, à chaque intervalle double sera associé une tâche T qui aura en charge d'évaluer les attributs de I. Par abus de langage, nous parlerons indifféremment de tâche et d'intervalle double, suivant le contexte, ces deux notions étant identiques dans notre modèle de parallélisation.

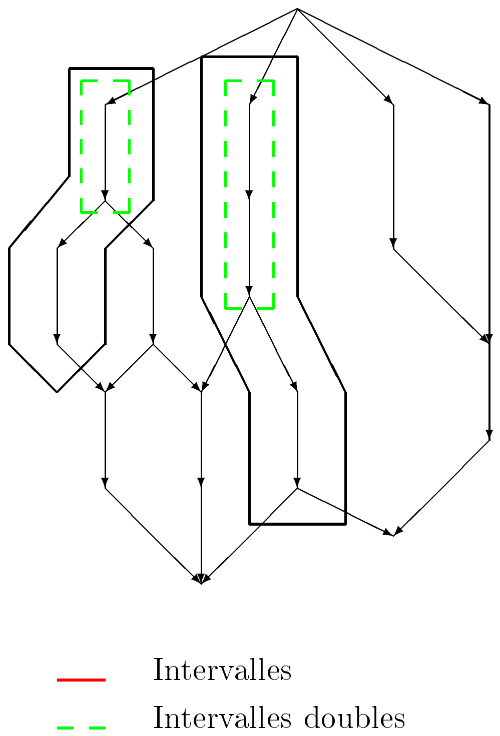


FIG. 3.5 - *Exemples d'intervalles doubles*

L'algorithme de calcul des intervalles doubles

Soit D l'ensemble des points à partir desquels nous calculons l'intervalle double. Dans notre algorithme, nous supposerons que D vérifie : $Max(I_D(D)) \subset D$. C'est une condition initiale forte qui signifie que les maxima de l'intervalle double se trouvent tous déjà dans l'ensemble des points de départ. De plus, il est nécessaire d'avoir des sommets sans lien de parenté. Avec la condition $Max(I_D(D)) \subset D$ et la propriété $D \in I_D(D)$, des liens de parenté pourraient conduire à l'inexistance d'un tel intervalle double. Nous donnons cette limitation car cela simplifie l'algorithme et, de plus, nous sommes toujours dans ce cas lors de son utilisation dans notre algorithme de détection du parallélisme dans les grammaires attribuées. L'algorithme de calcul des intervalles doubles est donné par la figure 3.6

Preuve : L'ensemble construit par l'algorithme vérifie la propriété de la définition 3.2.1.
Soit D l'ensemble construit par l'algorithme.
D vérifie les propriétés de la définition 3.2.2.

- $\textit{précédent}(D - Max_G(D)) \subseteq D$

  Supposons $\exists a \in D - Max_G(D)$ tel que $\exists b \in V$ et $b \notin D$ et $b \longrightarrow a$

  Le premier point de l'algorithme nous indique que si $b \notin D$ alors $a \notin D$ ce qui est contraire à l'hypothèse. Donc soit b a été mis dans D par le premier point (et oté plus tard), soit a n'a pas été mis dans D par le premier point.

  Comme la deuxième partie de l'algorithme n'ajoute pas de point dans D, la deuxième hypothèse n'est pas la bonne.

  Donc la deuxième partie de l'algorithme a retiré b. Or, si b a été retiré, $\{a$ tel que $b \xrightarrow{*} a\}$ a aussi été retiré.

  L'hypothèse de départ n'est donc pas valide.

- $\textit{suivant}(D - Min_G(D)) \subseteq D$

  Cette propriété est assurée par construction.

- Maximalité : Soit C_0 l'ensemble construit par la première phase de l'algorithme. On a $S \subset C_0$

  Soit C_i l'ensemble construit par l'algorithme à l'itération i. On a trivialement $C_i \subset C_j$ pour $i > j$, puisqu'on enlève des élements lors de chaque itération, la suite d'ensembles est décroissante.

  Le nombre d'éléments étant fini, l'algorithme est convergent. Par conséquent, le nombre d'itérations est fini. On note C_N l'ensemble résultat de l'algorithme.

  Hypothèse : Supposons qu'il existe un intervalle double S tel que $C_N \subset S$ et $C_N \neq S$.

  Remarque : $Max_G(S) = Max_G(C_N)$ par hypothèse de l'algorithme : les maxima de tous les intervalles doubles sont déjà dans D, l'ensemble de départ.

  D'après l'hypothèse, $Comp = S - C_N \neq \emptyset$. Soit $s_1 \in Max_G(Comp)$, on remarque que $s_1 \notin Max_G(S)$. La propriété de stabilité par les précédents nous donne que $\textit{précédent}(s_1) \subset S$. Mais $s_1 \in Max_G(Comp) \Rightarrow \textit{précédent}\ (s_1) \subset C_N$.

  Si $\textit{précédent}(s_1) \subset C_N$ alors s_1 a été ajouté à C lors de la première phase de l'algorithme. Comme il n'y est plus, c'est que la deuxième phase l'a retiré lors d'une itération i_1.

  D'après l'algorithme, si s_1 a été retiré, c'est qu'il existe deux sommets p_1 et s_2 tels que $p_1 \in C_{i_1}$ avec $p_1 \rightarrow s_1$ et $s_2 \notin C_{i_1}$ avec $p_1 \rightarrow s_2$.

```
/* calcul d'un intervalle "simple" */
    faire
        stab :=true
        pour a ∈ D faire
            S := suivant(a)
            pour b ∈ S
                si précédent(b) ⊆ (D) alors
                    D := D ∪ {b}
                    stab:= false
                fin si
            fin pour b
        fin pour a
    tant que stab=false

/* restriction par stabilité par suivant */

    faire
        stab := true
        pour a ∈ D faire
            S := suivant(a)
            si S ⊄ D et S ∩ D ≠ ∅ alors
                F := {c ∈ D tel que a --*--> c}
                D := D − F
                stab := false
            fin si
        fin pour
    tant que non(stab)
```

FIG. 3.6 - *Algorithme de calcul des intervalles doubles.*

s_2 n'est pas dans C_{i_1}, mais par stabilité par suivant de S, il est dans S. Donc il existe une étape i_2 pendant laquelle s_2 a été retiré. Ceci signifie, qu'il existe deux sommets p_2 et s_3 tels que $p_2 \in C_{i_2}$ avec $p_2 \to s_2$ et $s_3 \notin C_{i_2}$ avec $p_2 \to s_3$.

On peut donc construire une suite infinie $(s_1, s_2, \dots s_N, \dots)$. Or le graphe de départ étant fini, on arrive à une contradiction. L'hypothèse étant fondée sur le fait que $Comp \neq \emptyset$, on en déduit que l'intervalle construit est maximum.

□

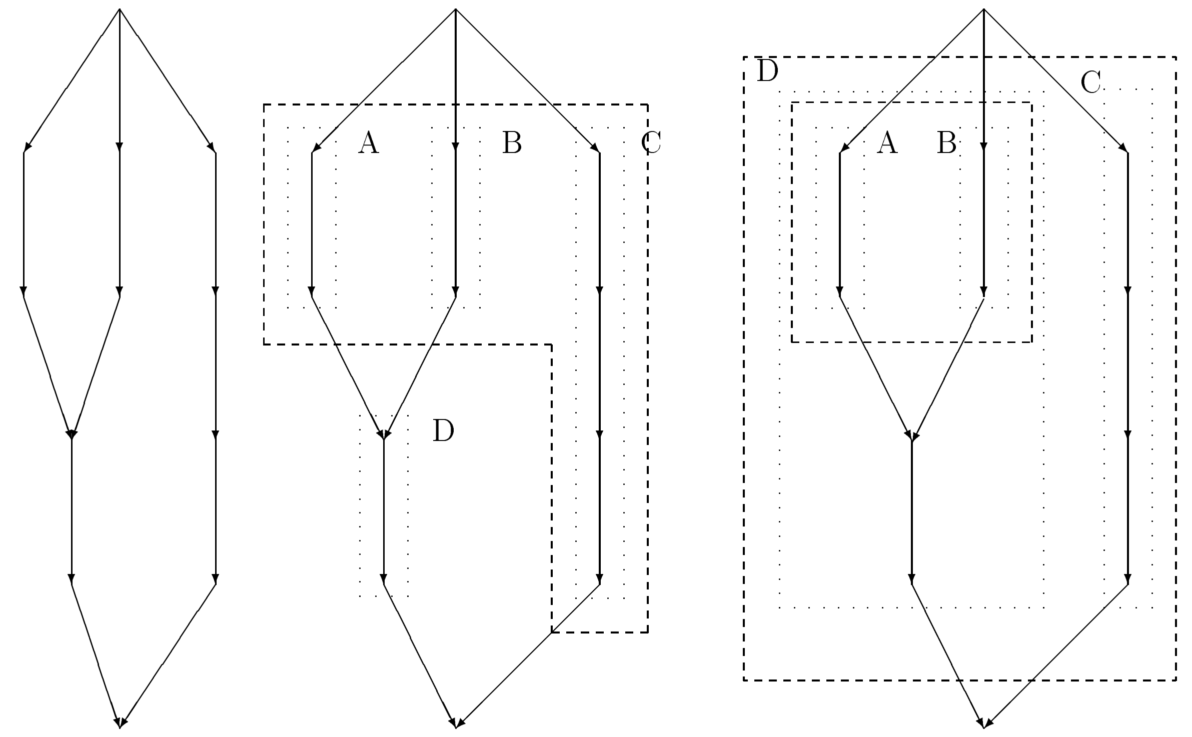
3.2.2 Positionnement des points de synchronisation

Motivations

Nous connaissons maintenant les différentes tâches à lancer en parallèle. Il faut maintenant déterminer les points de communication entre ces tâches, ou plutôt leur lien de parenté. Pour n tâches parallèles, faut-il lancer n tâches sœurs, ou bien lancer, par exemple, deux tâches sœurs qui lanceront chacune $n/2$ tâches filles ? Ce choix se heurte à plusieurs problèmes illustrés par les figures 3.7 et 3.8. Le premier est de savoir s'il est nécessaire d'imbriquer certaines tâches. Le deuxième est de connaître les tâches qui vont se synchroniser en un point précis. Dans la figure 3.7, la première parallélisation propose de calculer en parallèle les tâches A, B et C puis d'effectuer la tâche D et le reste du graphe en séquentiel. Cette parallélisation n'est pas bonne puisqu'elle introduit une dépendance inutile entre C et D . La deuxième parallélisation, quant à elle, imbrique l'exécution parallèle de A et B à l'intérieur d'une tâche. Les tâches A, B et D sont exécutées en parallèle avec C, dans cet exemple, aucune fausse dépendance n'est introduite.

La figure 3.8 nous montre un autre cas, ou nous ne trouvons pas de "bonne" parallélisation. De par le modèle synchrone, nous ajoutons toujours des dépendances qui n'existent pas au départ. Mais il faut tout de même sélectionner les tâches qui vont appartenir au même `COBEGIN/COEND`.

L'idée est d'étudier quels sont les différents sommets sur lesquels nous pourrions effectuer un *join*. Ces attributs font bien évidemment partie des suivants des différentes tâches en cause. Pourtant, tous les suivants ne sont pas à prendre en compte. En effet, notre but est d'effectuer les *joins* le plus tôt possible. En conséquence, il ne nous faut pas considérer les suivants dans lesquels dérivent d'autres suivants. L'ensemble d'attributs sur lequel nous effectuons les *joins* est donc l'ensemble constitué des maxima des suivants. La définition suivante nous donne les propriétés souhaitées pour que des tâches d'un ensemble $\mathcal{T}$ s'exécutent dans le même `COBEGIN/COEND`. Elle signifie que l'on peut passer d'une tâche à une autre par un chemin non-orienté dont deux sommets consécutifs ne sont jamais tous les deux hors des tâches de $\mathcal{T}$. C'est-à-dire que pour passer d'une tâche à une autre, on passe par un suivant commun à deux tâches.



Mauvaise Parallélisation
l(A)=1, l(B)=1, l(C)=4,l(D)=4
longueur totale: 8

Bonne parallélisation
l(A)=1, l(B)=1, l(C)=4,l(D)=4
longueur totale: 5

FIG. 3.7 - *Détections des tâches imbriquées(1).*

Définition 3.2.3 (Tâches sœurs) *Soient* $\mathcal{T} = \{T_i\}_{i\in\{1...N\}}$ *et* $S = Max_G(\{s \notin T_j$ *tel que* $\exists a \in T_i\,; a \rightarrow^{G_{//}} s\})$

Deux tâches T *et* T' *de* $\mathcal{T}$ *sont dites sœurs par rapport à* S *s'il existe une suite finie* $((a^1, s_1), (a_1^2, s_1), (a_2^2, s_2), (a_1^3, s_2) \ldots, (a_2^{n-1}, s_{n-1}), (a^n, s_{n-1}))$ *tel que :*

i) $a_i^j \in G_{//}$ *et* $s_i \in G_{//}$

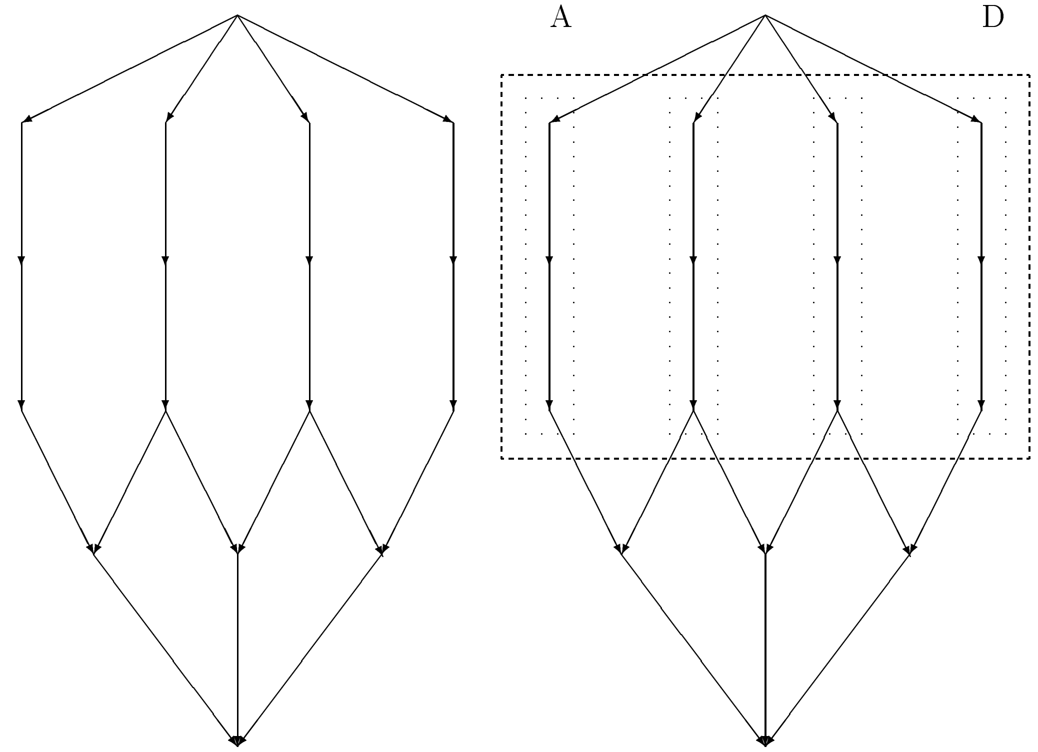
ii) a^1 *et* a^n *appartiennent respectivement à* T *et* T'

iii) a_1^i *et* a_2^i *appartiennent à* T_i

iv) $s_i \in S$ *et* $a_1^i \rightarrow^{G_{//}} s_{i-1}$ *et* $a_2^i \rightarrow^{G_{//}} s_i$

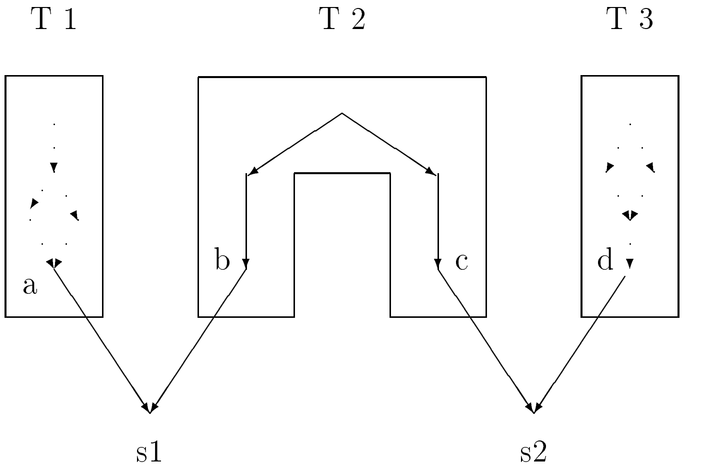
La figure 3.9 donne un exemple de tâches sœurs.

Lemme 3.2.1 *La relation "être sœur de" est une relation d'équivalence.*



A et D n'ont pas de suivant direct en commun
pourtant, ils seront synchronisés

FIG. 3.8 - *Détections des tâches imbriquées(2).*



On peut passer de T_1 à T_3 par $(a, s_1)(b, s_1)(c, s_2)(d, s_2)$

FIG. 3.9 - *Exemple de tâches sœurs.*

Preuve :

- Réflexive : Il suffit de considérer l'ensemble vide comme suite s_i.
- Symétrique : On considère la suite inverse.
- Transitive : On a pour T et T' $((a^1, s_1), (a_1^2, s_1), \ldots, (a_2^{n-1}, s_{n-1}), (a^n, s_{n-1}))$ et pour T' et T'' $((a^{1'}, s'_1), (a_1^{2'}, s'_1), \ldots, (a_2^{p-1'}, s'_{p-1}), (a^{p'}, s'_{p-1}))$

  la suite $((a^1, s_1), (a^2, s_1), \ldots, (a^n, s_{n-1}), (a^{1'}, s'_1), (a_1^{2'}, s'_1), \ldots, (a^{p'}, s'_{p-1}))$ prouve que T et T'' sont sœurs.

$\square$

Résultat : Notre parallélisation consiste à grouper, dans un seul `COBEGIN/COEND`, les tâches appartenant à la même classe d'équivalence.

3.2.3 Algorithme de partitionnement

Soit $\mathcal{T} = \{T_1, \ldots T_N\}$ un ensemble de tâches indépendantes.

Préliminaires :
Le but de l'algorithme suivant est de trouver les tâches sœurs. Pour cela, on considère les ensembles S_i qui représentent les suivants des tâches T_i. Chaque élément des S_i est un *join* pour la tâche T_i ou pour un de ses parents. Ensuite, le calcul de S, consiste justement à faire la part des choses et à ne laisser dans les S_i, que les *joins* immédiats des T_i. Ainsi, une tâche qui doit se synchroniser avec un groupe de tâches aura un S_i final vide. En effet, les S_i identifient les tâches sœurs. Une tâche qui doit se synchroniser avec un groupe de tâches n'a donc pas de sœurs. L'algorithme est donné par la figure 3.10

Preuve : l'ensemble construit par l'algorithme vérifie la définition. L'algorithme est trivialement convergent.

À chaque ajout d'une tâche T dans P, nous pouvons exhiber au moins un triplet (a, s, b) avec

- $s \in S$ (car $S_j \cap S_p \neq \emptyset$)
- $b \in T \mid b \to s$ (car $s \in suivant(T)$)
- $\exists T_i \in P$ et $a \in T_i \mid a \to s$

Algorithme :

$S_i := \{a \in V - T_i$ tel que $\exists b \in T_i$ tel que $b \longrightarrow a\}$
$S := \cup S_i$
si $\exists (a,b) \in S \times S$ tel que $a \overset{*}{\longrightarrow} b$ **alors** on retire b de S.

$S_i := S_i \cap S$ /* On redéfinit S_i pour tenir compte des attributs ôtés :*/
$P := \{T_1\}$
$S_P := S_1$
$R := \mathcal{T} - P$
faire
    stab := True
    Pour $T_j \in R$ **faire**
        si $S_j \cap S_P \neq \emptyset$ **alors**
            $P := P \cup \{T_j\}$
            $S_P := S_P \cup S_j$
            $R := R - \{T_j\}$
            stab := False
        fin si
    fin pour
tant que non(stab)

Résultat : les tâches de P sont sœurs et comme $T_1 \in P$, ce sont les sœurs de T_1.
En itérant l'algorithme, nous obtenons un ensemble d'ensembles de tâches sœurs. Nous verrons dans l'algorithme global comment sont organisés les groupes de tâches entre eux.

FIG. 3.10 - *Algorithme de partitionnement en tâches sœurs.*

Ceci implique que T_i et T sont sœurs. Par récurrence, toutes les tâches de P sont bien sœurs.

Une tâche sœur de T_1 appartient obligatoirement à P.

Au pire, tous les couples de la définition seront ajoutés les uns après les autres.

□

3.2.4 L'algorithme de parallélisation

L'algorithme de la figure 3.12 présente dans son ensemble l'algorithme de transformation du graphe de dépendance en un graphe de parallélisation. De façon informelle, l'algorithme agit de la façon suivante.

Un graphe G lui est donné en entrée. Nous calculons tout d'abord les maxima de ce graphe et les intervalles doubles qui leur correspondent. Récursivement, chacun de ces intervalles doubles sera transformé. Nous appliquons ensuite l'algorithme de partitionnement en tâches sœurs afin de pouvoir placer quelques `COBEGIN/COEND`. Il y aura un `COBEGIN/COEND` pour chaque élément du partitionnement (c'est-à-dire pour chaque classe d'équivalence de la relation "être soeur"). Ce procédé est itéré jusqu'à ce qu'il n'y ait plus qu'une seule et unique tâche.

L'étape suivante consiste à recommencer avec le reste du graphe. Nous connecterons la parallélisation obtenue sur le reste du graphe avec la tâche de la première étape.

Lorsqu'un graphe n'a qu'un seul maximum, nous avons besoin d'avancer dans ce graphe pour arriver à un nœud où un choix est possible quant à la sélection du suivant. Si le graphe est en fait un chemin, l'algorithme retournera l'identité.

L'algorithme se poursuit jusqu'à ce que tout le graphe G de départ soit traité.

Pour simplifier la présentation détaillée de cet algorithme, nous définissons les fonctions présentées par la figure 3.11

Exemple

L'algorithme utilisé est récursif. La complexité n'a pas été étudiée car la taille de nos graphes est relativement petite. De plus, notre but est d'optimiser le code de l'évaluateur lui-même, pas celui du générateur. L'exemple qui suit va nous permettre d'expliciter simplement la façon dont fonctionne notre algorithme. Pour cela, considérons le graphe de la figure 3.13.

Nous cherchons les maxima de ce graphe. Leur nombre étant supérieur à un, nous calculons pour chacun d'eux l'intervalle double qui en est issu. Nous trouvons les intervalles doubles A, B, C, D, E. Nous parallélisons chacun de ces sous-graphes en appelant récursivement l'algorithme. Sur A, B, D et E, l'algorithme retourne l'identité puisque ces graphes sont des chemins.

La figure 3.14 montre la parallélisation sur le sous-graphe C. Nous calculons tout d'abord l'ensemble des maxima de ce sous-graphe. Son cardinal est 1, donc nous prenons

$Card()$	Argument : un ensemble E. Valeur de retour : le cardinal de cet ensemble.
$Connect()$	Arguments : deux graphes de parallélisation $(V_1, E_1), (V_2, E_2)$. Valeur de retour : un graphe de parallélisation (V, E) tel que $V = V_1 \cup V_2$ et $E = E_1 \cup E_2 \cup \{(min(V_1), max(V_2))\}$.
$IntervalleDouble()$	Arguments : G un graphe (V, E). A un sous-ensemble de V. Valeur de retour : l'intervalle double issu de A dans le graphe G.
$Linear()$	Argument : a un attribut sommet de G. Valeur de retour : toute la partie P de G issue de a telle que pour tout b dans P, b n'a qu'un seul précédent, c'est-à-dire la partie du graphe où l'on a un ordre total.
$Partitionne()$	Argument : un ensemble E de graphes de parallélisation. Valeur de retour : un ensemble de couples (p, S_p) correspondant au partitionnement de E.

FIG. 3.11 - *Définition préliminaire à l'algorithme de parallélisation*

la partie chemin L issue de ce maximum.

Nous appelons récursivement l'algorithme sur le reste du sous-graphe. Cette fois, nous avons deux maxima, il faut donc calculer les intervalles doubles $C1$ et $C2$. Chacun d'eux étant un chemin, l'appel récursif renvoie à chaque fois l'identité. Nous cherchons maintenant à les regrouper en tâches sœurs (définition 3.2.3). $C1$ et $C2$ ont le même suivant, elles sont sœurs, et par conséquent, la partition est $\mathcal{P} = \{(\{C1, C2\}, S\_0)\}$ qui est un singleton.

Nous ajoutons par conséquent des nœuds `COBEGIN`/`COEND` autour de $C1$ et $C2$. Nous traitons ensuite l'intervalle double issu de S_0. Celui-ci est chemin (restreint à un seul point). Il faut le connecter au `COEND`. Puis le graphe c de la figure 3.14 est retourné à la procédure appelante. Celle-ci connecte la partie chemin L au graphe récupéré pour construire le graphe d de la même figure. Ce graphe constitue la valeur retournée par l'algorithme comme parallélisation du sous-graphe C.

La seconde partie consiste à partitionner les tâches constituées par les sous-graphes parallèlisés à partir de A, B, C, D, E. Pour cela, nous appelons l'algorithme de partitionnement. Celui-ci retourne $\mathcal{P} = \{(\{A, B, C\}, S_1), (\{D, E\}, S_2)\}$. Ceci signifie que les tâches A, B, *Transfo*(C) sont sœurs, ainsi que D et E.

$transfo(G = (V, A))$
$First := Max(V)$ /*attributs qui ne dépendent de rien */
Si $card(First) = 1$ **alors** /*On note a l'attribut de First */
$(NewGraph := (V', A')) = linear(a, G)$ /* On recopie la partie du graphe G issue de a qui est chemin. */
si $NewGraph \neq G$ **alors**
$NewGraphRestant := transfo(G' = (V - V', A/(V - V')))$
$NewGraph := Connect(NewGraph, NewGraphRestant)$
fin si
retourner $NewGraph$
sinon
$Tasks := \emptyset$
Pour chaque a de $First$
faire
$(D_a = (V_a, A_a)) := IntervalleDouble(a, V)$/* Intervalle double issu de a dans V.*/
$NewGraph_a := transfo(D_a)$ /*Détection du parallélisme sur l'intervalle double */
$Tasks := Tasks \cup \{NewGraph_a\}$
fin pour
faire
$NewTasks := \emptyset$
$\mathcal{P} := Partitionne(Tasks)$ /* *partitionne* fait correspondre à chaque $p \in \mathcal{P}$ son ensemble de suivants immédiats S_p.*/
pour chaque partie d'intervalle double $(p, S_p) \in \mathcal{P}$ **faire**
si $card(p) > 1$ **alors**
$E_p := Ajouter\_Cobegin\_Coend(p)$
$G_p := IntervalleDouble(S_p)$
$NewGraph_p := transfo(G_p)$
$NewGraph_p := Connect(E_p, newGraph_p)$
fin si
$NewTasks := NewTasks \cup \{NewGraph_p\}$
fin pour
$Tasks := Newtasks$
jusqu'à $Card(Tasks) = 1$
retourner $T \in Tasks$ /* retourner l'unique élément de $Tasks$ */
fin si
Fin $transfo$

FIG. 3.12 - *Algorithme de parallélisation*

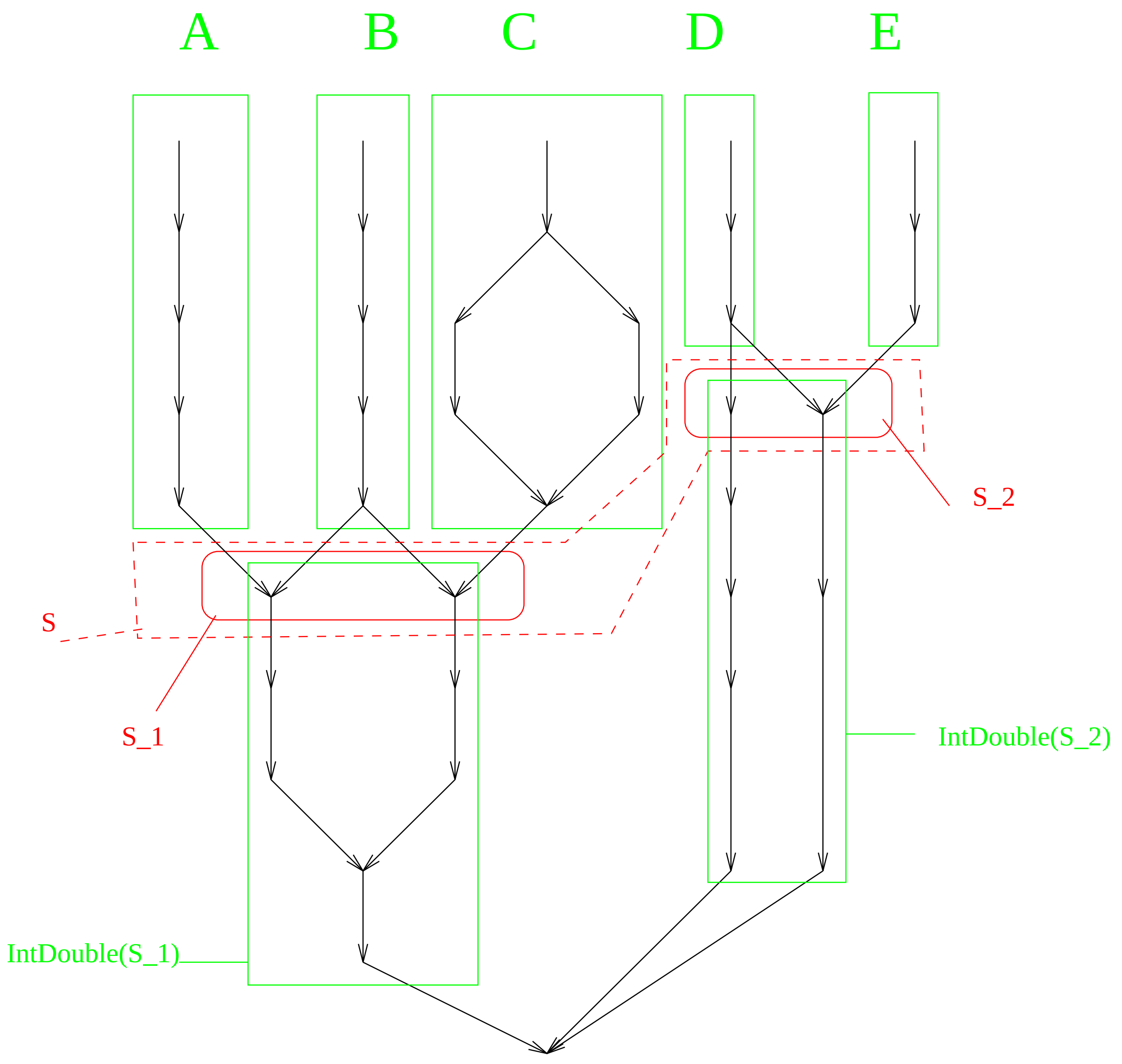


FIG. 3.13 - *Exemple : graphe initial*

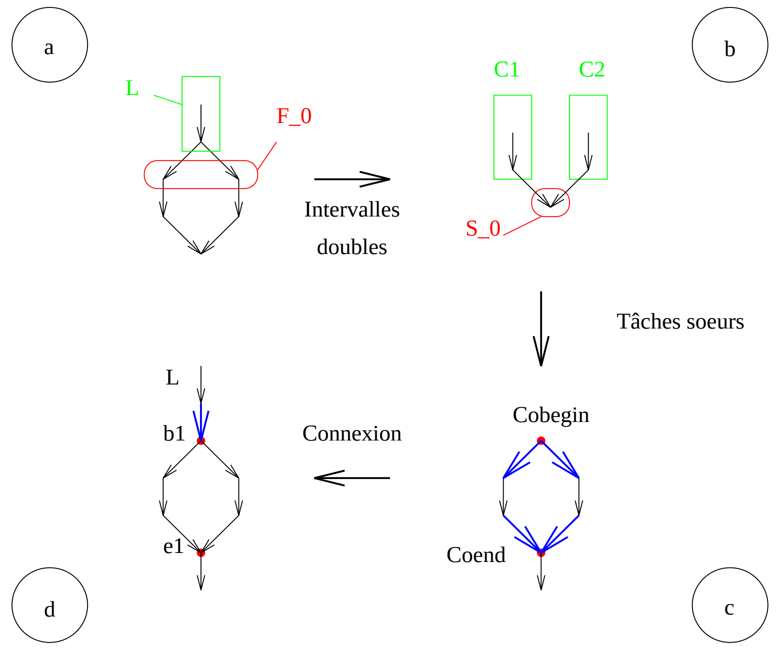


FIG. 3.14 - *Exemple : A*

La figure 3.15 montre la résolution de la parallélisation de $\{A, B, C\}$: ajout de `COBEGIN`/-`COEND` (action 1) puis calcul de l'intervalle double issu de S_1 (ensemble des suivants de T_ABC). L'action 2 donne le résultat de la parallélisation de cet intervalle. Nous connectons ensuite les deux graphes (action 3). La tâche issue de A, B et C est maintenant terminée, le graphe T_ABC est retourné comme résultat.

La partie $\{D, E\}$ se traite de la même façon, voir la figure 3.16.

Après avoir traité les deux parties, nous avons les deux tâches T_ABC et T_DE qu'il faut à présent traiter. Avec une nouvelle itération sur le partitionnement, nous obtenons une seule classe de sœurs contenant ces deux tâches. Nous ajoutons les nœuds `COBEGIN`/`COEND` autour d'elles, et nous calculons l'intervalle double issu des suivants. Un appel sur cet intervalle double de l'algorithme retourne un graphe linéaire qu'il ne nous reste plus qu'à connecter comme le montre la figure 3.17.

3.3 Parallélisation

Une fois le parallélisme détecté, nous pouvons encore choisir d'utiliser ou non l'indépendance des attributs. Souvent, en effet, le parallélisme trouvé induit des tâches minuscules, d'où un surcoût important. Dans cette section, nous allons étudier un certain nombre de méthodes pour tenter d'éviter une parallélisation conduisant à la création excessive de

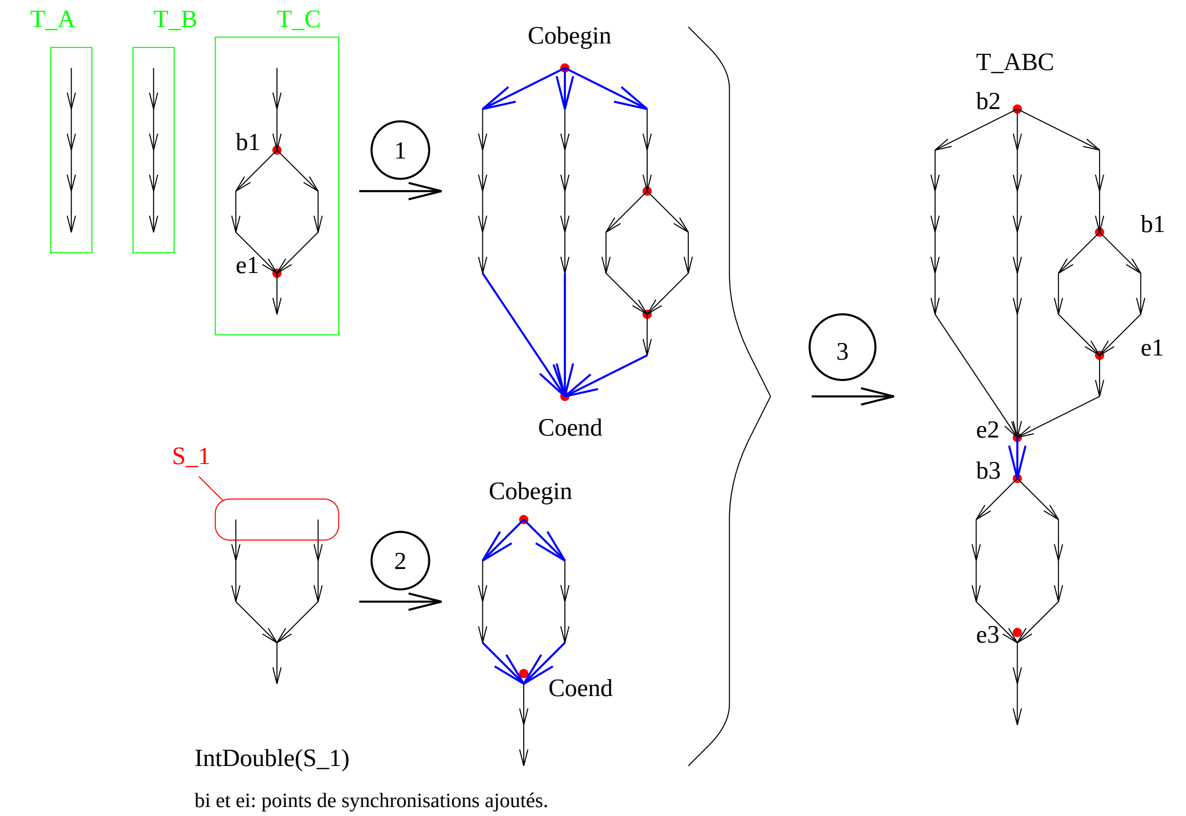


FIG. 3.15 - *Exemple : B*

tâches. Ce ne sont que des heuristiques. Nous en avons testées certaines, d'autres sont proposées à titre d'information. La plupart des travaux à ce sujet propose une approche "manuelle", c'est donc à l'écrivain de la grammaire attribuée de donner les nœuds où il désire une parallélisation. Cette approche est évidemment la plus simple à mettre en œuvre et, paradoxalement, c'est souvent aussi la plus efficace. Cette section essaie de proposer des méthodes ayant pour but de décharger l'utilisateur de la mise en œuvre du parallélisme.

3.3.1 Heuristique dynamique

Une opération intéressante consiste à limiter le parallélisme par rapport à la hauteur de l'arbre. Par exemple, nous pouvons refuser de créer une nouvelle tâche parallèle si elle a déjà n aïeuls ($n > 0$). L'avantage est d'empêcher la création imbriquée d'un trop grand nombre de tâches et, par conséquent, de créer des tâches trop près des feuilles, celles-ci étant par définition très courtes.

Avantages : Les tâches créées sont importantes donc le surcoût est "minimum". La méthode est facile à implanter. Les résultats obtenus avec cette heuristique sont très satisfai-

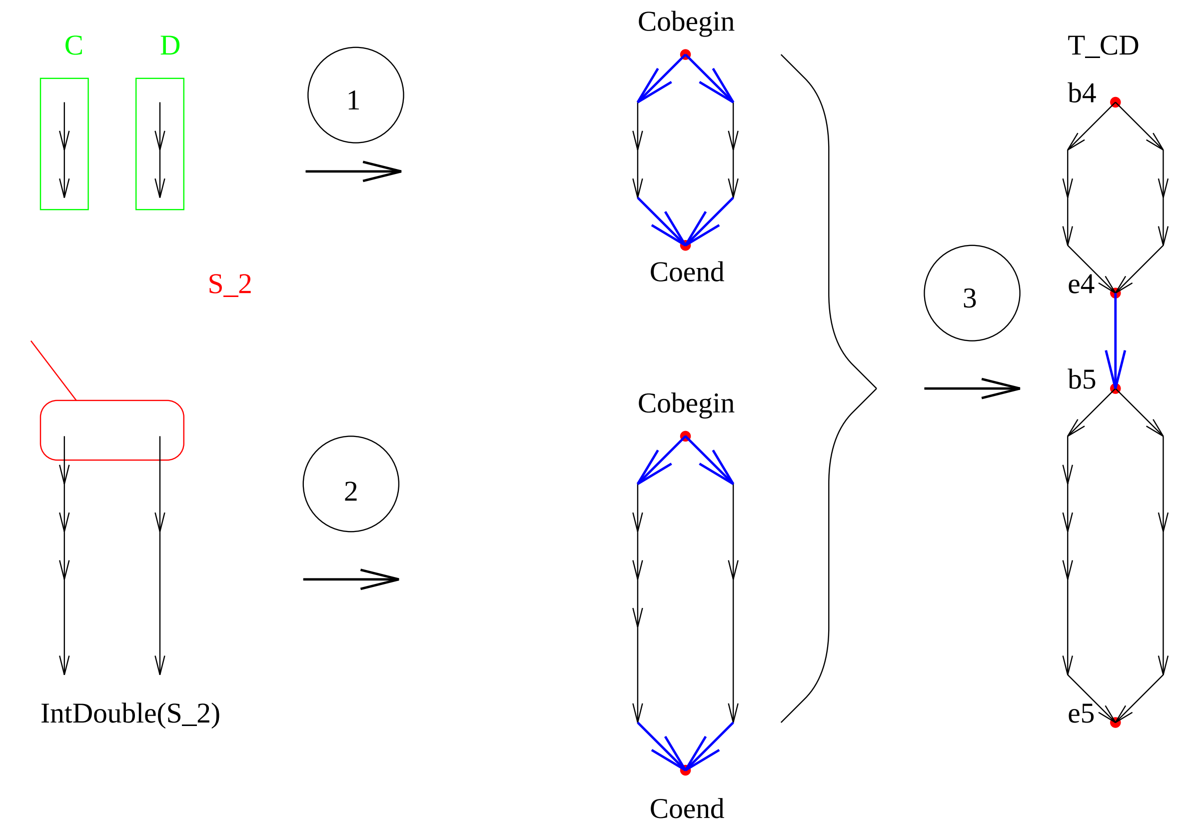


FIG. 3.16 - *Exemple : C*

sants.

Inconvénients : L'inconvénient majeur tient au fait que c'est justement une méthode dynamique. Les techniques d'optimisation mémoire que nous allons décrire par la suite s'appliquent de manière théorique. Cependant, la taille de l'évaluateur double car, à chaque séquence de visite, nous devons associer au mieux une version séquentielle et une version parallèle. Dans le cas d'une optimisation plus fine, la taille de l'évaluateur peut augmenter dans des proportions plus grandes.

3.3.2 Heuristiques statiques

Le graphe de parallélisation décrit dans ce chapitre détecte tout le parallélisme utilisable par notre méthode. Dans les cas pratiques, c'est encore souvent trop parallèle (trop de tâches pour peu de travail effectif), et les tâches sont souvent petites. Une première heuristique consiste à étudier le graphe et à supprimer la parallélisation de tâches qui n'effectueront qu'un `EVAL`, pour ne garder que des tâches qui auront à visiter au moins un nœud. Cette heuristique permet ainsi de supprimer les tâches qui sont naturellement courtes (une

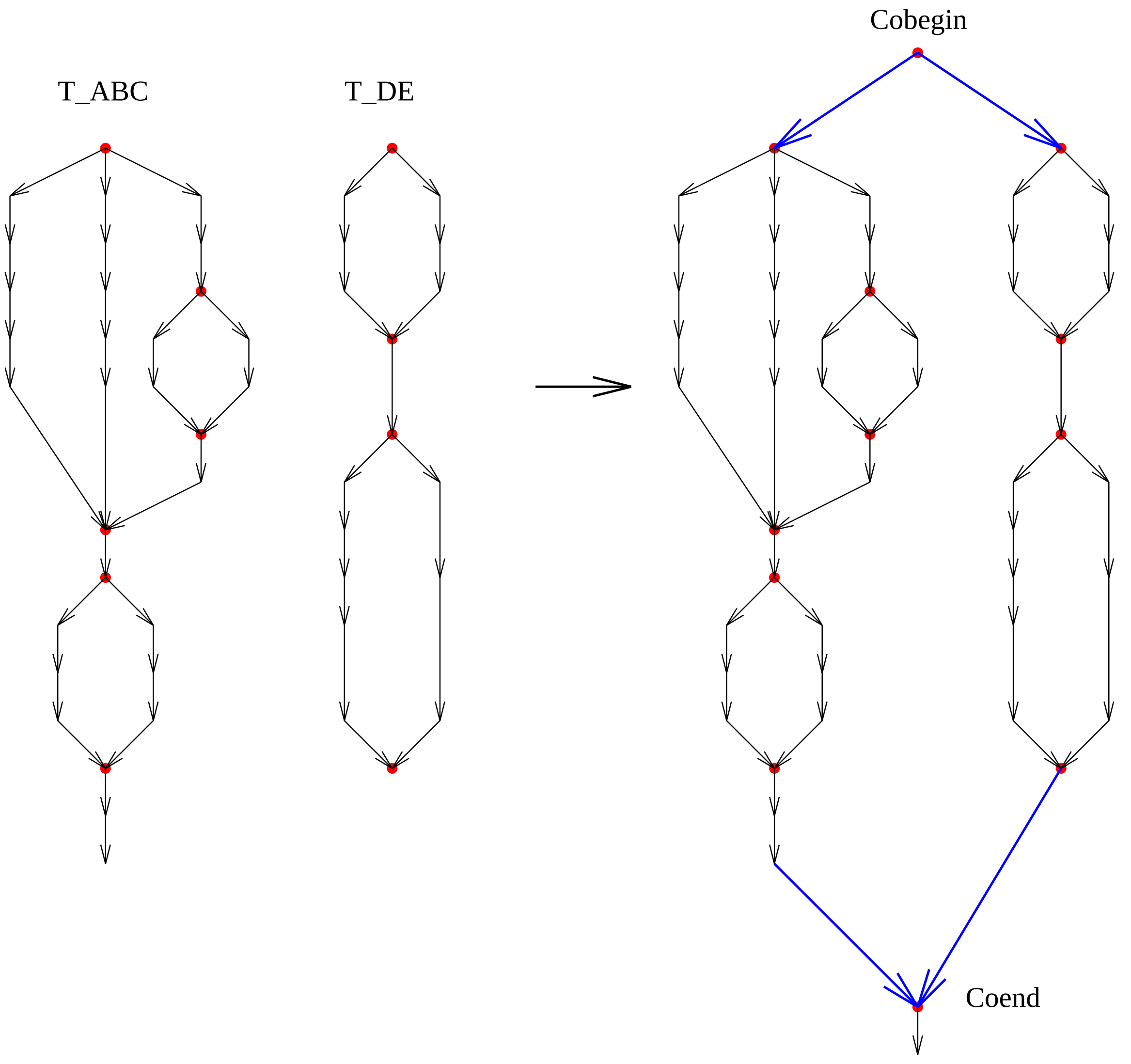


FIG. 3.17 - *Exemple : D*

ou plusieurs fonctions d'évaluation) et de ne conserver que des tâches suceptibles d'effectuer un parcours de sous-arbre. Cette heuristique n'est pas capable d'approximer la longueur d'une tâche. Elle donne seulement un poids de zéro aux attributs à évaluer directement dans la production à l'aide d'une fonction d'évaluation, et un poids de un aux attributs de la production à évaluer par une visite sur une autre production. Il n'y a aucune différentiation entre les différentes fonctions d'évaluation, ni entre deux visites. Notamment, une visite en bas de l'arbre (donc sur un sous-arbre obligatoirement restreint) est de même importance au regard de cette heuristique que la visite sur un nœud très haut dans l'arbre (où le sous-arbre est potentiellement important). Cette heuristique peut conduire à plusieurs méthodes d'attribution de poids aux différents nœuds du graphe. Mais seule l'heuristique présentée ci-dessus à été implantée.

Une deuxième heuristique consiste à utiliser les graphes de grammaire [MöW82].

On définit sur l'ensemble des non-terminaux une relation d'équivalence. Soit $\mathcal{R}$ la relation d'équivalence définie sur $N \times N$ telle que :

$$A \;\mathcal{R}\; B \Leftrightarrow \{\, A \stackrel{*}{\longrightarrow} B \text{ et } B \stackrel{*}{\longrightarrow} A \,\} \;.$$

On définit le graphe G ayant pour sommet les classes d'équivalence de $\mathcal{R}$. Il existe un arc $(\overline{A}, \overline{B})$ si $\exists a \in \overline{A}, b \in \overline{B}$ tel que $a \stackrel{*}{\longrightarrow} b$

Propriété : Ce graphe est acyclique et vérifie la propriété de la définition 1.1.15 ($\overline{Z}$ dérive dans toutes les autres classes). Par conséquent, nous disposons de la notion de profondeur.

Une heuristique consiste donc à refuser de paralléliser les nœuds qui appartiennent à une classe qui se trouve trop profond dans le graphe G.

Cette méthode permet de paralléliser les concepts de la grammaire qui se situent au plus haut niveau de celle-ci et qui semblent de bons candidats à la parallélisation, comme la notion de blocs dans les langages.

Avantage : Comme énoncé plus haut, c'est une méthode statique, ce qui n'impose aucun surcoût particulier. Les méthodes d'optimisation statique présentées au chapitre 6 fonctionnent.

Inconvénients : Nous pouvons trouver des arbres (dans des cas pratiques) pour lesquels l'heuristique donne de mauvais résultats. D'autre part, délimiter la profondeur à laquelle on ne parallélise plus est une décision arbitraire.

3.4 Les séquences de visite parallèle.

Le résultat dont nous disposons actuellement est un graphe. Ce graphe est formé de nœuds étiquetés pour le parallélisme et de nœuds étiquetés par les attributs de la production. L'évaluateur, nous l'avons vu dans le chapitre 1, se comporte comme un automate dont les labels de transitions sont `VISIT`, `EVAL` et `LEAVE`. Cette section va décrire brièvement le passage d'un graphe de parallélisation aux séquences de visites parallèles.

Définition 3.4.1 (Séquence de visite parallèle) *Une séquence de visite parallèle est un graphe dont les nœuds sont étiquetés* `COBEGIN`*(n),* `COEND`*(n),* `VISIT`*(i,v),* `EVAL`*(a),* `LEAVE`*. Il est obtenu directement à partir du graphe de parallélisation calculé par l'algorithme de détection du parallélisme. Un nœud* `COBEGIN`*(n) (resp.* `COEND`*(n)) du graphe de parallélisation se traduit par un nœud* `COBEGIN`*(n) (resp.* `COEND`*(n)) dans la séquence de visite. Un attribut d'entrée a du $i^{ème}$ fils de la production (calculé dans la $v^{ème}$ visite au fils) se traduit par un* `VISIT`*(i,v) et un attribut a de sortie de la production se traduit par un* `EVAL`*(a). Enfin les attributs hérités du père sont ignorés. Les arcs sont conservés par la transformation.*

Le graphe de parallélisation contenant tous les attributs de la production, nous en obtenons plusieurs séquences de visite.

Toutefois, le graphe d'une séquence de visite est isomorphe au graphe de parallélisation restreint aux attributs calculés par cette séquence de visite. Par la suite, nous ferons référence indifféremment au graphe de parallélisation ou aux séquences de visite, le contexte indiquant le graphe concerné.

À la différence du séquentiel, les séquences de visite ne sont plus linéaires, mais deviennent des graphes.

3.5 Conclusion

La détection du parallélisme présentée dans ce chapitre permet de sélectionner une certaine quantité de parallélisme. Les contraintes imposées sur le graphe de parallélisation nous assurent qu'il n'y a jamais de communication nécessaire entre deux tâches. Ce résultat implique que le nombre de synchronisations est restreint, à peu près une synchronisation par production parallèle. Même si cela semble être important sur certains exemples choisis en comparaison avec une parallélisation avec communication, dans le cas général, le nombre de synchronisations est à l'avantage de notre méthode. Notre sélection a l'avantage que le nombre de synchronisations est en rapport avec le nombre des non-terminaux, et non pas avec le nombre d'attributs de la grammaire.

La certitude de ne pas avoir de conflit d'accès grâce aux ordres l-ordonnés nous conduit également à penser que notre méthode va donner de bons résultats.

Chapitre 4

Implantation

Passer de la théorie à la pratique pour l'évaluation parallèle des attributs n'est pas chose facile. Autant il est aisé de trouver beaucoup de parallélisme dans les grammaires attribuées, autant pouvoir l'implanter efficacement relève du défi. La granularité rencontrée dans l'évaluation d'attributs est très fine, et d'importantes précautions doivent être prises afin de rendre le surcoût parallèle le plus petit possible. D'autre part, nous nous heurtons assez facilement à des systèmes parallèles qui ont des fonctionnalités plus adaptées au calcul scientifique.

L'exemple extrême consiste à utiliser le modèle de Fang [Fan72], qui assigne un processus par attribut, et nous voyons tout de suite les résultats catastrophiques auxquels cela peut conduire.

Notre implantation a donc deux buts, d'une part elle s'est voulue comparative, jusqu'à un certain point, et d'autre part elle veut prouver que l'on peut paralléliser efficacement l'évaluation d'attributs. L'implantation sur de véritables machines parallèles, et non pas seulement par simulateurs, démontre que notre modèle est valide. Nous considérons que l'implantation doit avoir une part importante en grammaires attribuées, car ces dernières représentent avant tout un procédé opérationnel. Il est donc important de valider que ce procédé peut être parallélisé, et ceci d'une manière qui n'est pas seulement théorique.

Pour répondre au but de comparaison, nous avons réalisé plusieurs implantations de notre modèle, en utilisant des techniques de parallélisation différentes à chaque fois. La section 4.4 donne une description de chacune des implantations effectuées.

Pour valider notre modèle, plusieurs machines parallèles ont été choisies qui seront brièvement décrites dans la section suivante .

4.1 Les machines parallèles utilisées

Toutes les machines sur lesquelles nous avons effectué une implantation sont des machines à mémoire partagée. Nous ne nous occuperons absolument pas des problèmes issus d'un autre type d'architecture. Nous avons fait l'hypothèse que la localisation des données à l'intérieur de cette mémoire partagée était sans importance au regard de l'efficacité. Bien évidemment, ce principe est faux en pratique, comme nous le verrons pour certaines machines. Cette section se propose d'étudier les différentes architectures sur lesquelles nous allons implanter nos algorithmes. Nous examinerons les points importants pour nous, c'est-à-dire l'implantation de la mémoire partagée, les types de processus disponibles sur les systèmes et le *scheduling* en général.

4.1.1 Balance 8000

Dans le Balance de Sequent, la mémoire partagée est implantée dans le système comme un fichier. Pour des raisons d'efficacité, ce dernier a son image en mémoire. Toutefois, lorsque la taille des données est importante, il est écrit sur le disque, ce qui peut provoquer une chute de performance pour certains gros exemples.

Sur cette machine, nous ne disposons que du *fork* Unix standard, c'est-à-dire qu'une création de processus provoque une recopie intégrale des informations du processus père. En conséquence, un *fork* sur cette machine est relativement coûteux et ne peut pas être employé pour exécuter de petites tâches.

Le *scheduling* utilisé pour l'ordonnancement des processus est celui utilisé pour les processus Unix, avec priorité...

Dernier point, la mémoire privée, c'est-à-dire la mémoire accessible seulement pour un processus, est implantée directement par le compilateur grâce à une directive particulière.

4.1.2 Multimax

Le Multimax de Encore Computer comporte 14 processeurs avec 64 Mo de mémoire, architecture bus. Les cartes processeurs et les cartes mémoires sont connectées à un même bus. Chaque processeur dispose d'un cache. Nous avons travaillé sous le système Mach.

Au niveau des processus, le Multimax dispose des processus Unix standard, mais également d'une librairie de processus légers *threads* non conformes à la norme POSIX. L'utilisation de *threads* permet un surcoût moins important au moment de la création d'un processus, car seuls les descripteurs systèmes du processus créateur sont dupliqués. Un *thread* consiste en un état du processeur, une pile d'exécution et une quantité limitée d'espace mémoire pour les données. Les *threads* partagent l'accès à la mémoire et héritent des droits d'accès Unix de leur père. La librairie fournit les fonctions de verrouillage nécessaires pour protéger les sections critiques.

Dans cette machine, la librairie des *threads* fournit la possibilité d'avoir une mémoire privée en laissant un champ de la structure des *threads* en accès à l'utilisateur. Il suffit par conséquent d'allouer une zone de mémoire partagée et d'affecter le champ du *thread* à l'adresse de cette zone. Cela crée quelques problèmes pour générer un code vraiment portable...

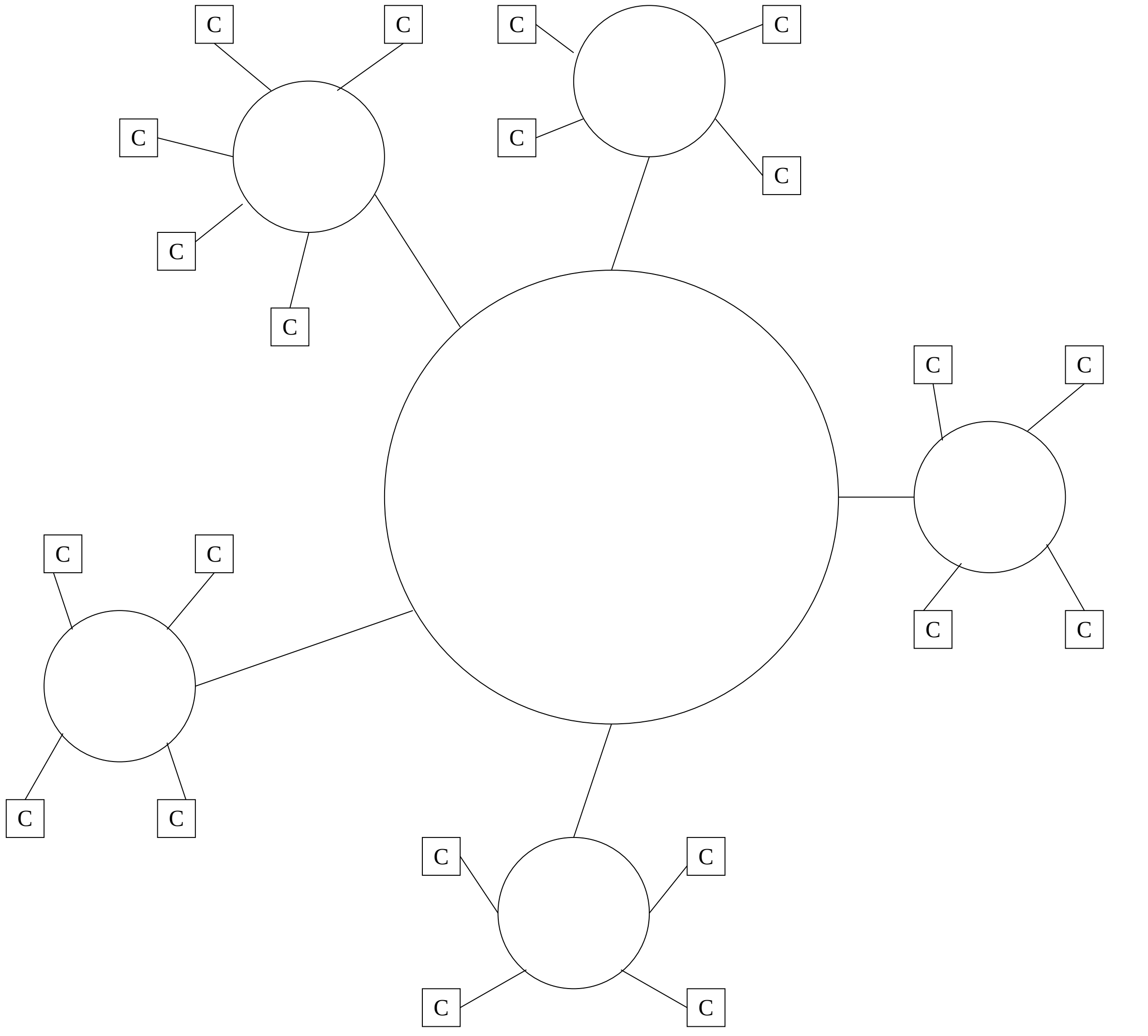


Fig. 4.1 - *Architecture de la KSR1*

4.1.3 KSR1

La KSR1-72 de Kendall Square Research, avec 72 cellules (processeurs KSR1) et 32 Mo de mémoire par cellule, n'est pas à proprement parler une machine à mémoire partagée. Pour cette machine, on parle de mémoire virtuellement partagée. La figure 4.1 montre l'organisation de cellules dans la KSR1. Chaque cellule comporte une quantité de mémoire importante. Cette mémoire est en adressage partagé, c'est-à-dire que tout processeur peut adresser la totalité de la mémoire. Le matériel se charge de rapatrier les données d'un processeur à un autre. L'organisation des caches fait que les communications sont très optimisées et la KSR se comporte relativement bien quant à notre impératif : la localisation des données ne doit pas changer les performances.

Là encore, nous disposons de *threads* (fondés sur le projet de norme IEEE Posix) avec toutes les fonctionnalités pour gérer la synchronisation, les sections critiques et même pour donner une priorité particulière à un *thread*. L'implantation des *threads* sur la KSR1 est beaucoup plus complète que sur le Multimax.

Le compilateur C fourni par KSR implante, comme pour le Balance, la possibilité de déclarer des identificateurs privés.

4.1.4 Les primitives parallèles

Pour être le plus général possible, nous avons fait des hypothèses quant aux primitives parallèles à utiliser dans notre implantation. De cette façon, nous avons exclu toute spécificité aux machines sur lesquelles nous avons porté notre programme. Nous sommes conscients que cette attitude va rendre nos implantations moins efficaces et plus rigides par rapport à ce qu'elles auraient pu être si nous avions utilisé pleinement toutes les possibilités offertes par une machine donnée. Mais d'un autre côté, le portage sur de nouvelles machines aurait été d'autant plus compliqué. Par ailleurs, nous ne souhaitons aucunement prouver que telle ou telle machine parallèle est la meilleure ou la plus adaptée à l'évaluation d'attributs. Il était donc indispensable que les outils utilisés sur chaque machine soient les mêmes ou, au pire, qu'ils soient équivalents en ce qui concerne les performances du système à leur égard.

Notre système a besoin de connaître les opérations suivantes (où t et f sont les valeurs booléennes *vrai* et *faux*) :

lock(m)	Si $m = t$ alors blocage sinon $m \leftarrow t$, en une opération atomique ;
unlock(m)	Si $m = f$ alors erreur sinon $m \leftarrow f$ et réveil de processus en attente ;
try_lock(m)	Si $m = t$ alors retourne f sinon $m \leftarrow t$, en une opération atomique ;
fork	Création d'un nouveau processus (léger ou non).

4.2 Le contexte

Le projet `Chloé` de l'`Inria` développe un générateur d'évaluateur de grammaires attribuées. Ce système, appelé FNC-2, accepte en entrée des grammaires de la classe FNC. Après une éventuelle transformation, le générateur d'évaluateurs récupère toujours une grammaire l-ordonnée. Ce système est écrit en C et utilise le système SYNTAX pour construire l'arbre d'entrée. Le code, où nous avons inséré le parallélisme, est exclusivement écrit en C. La raison pour laquelle nous avons fait l'hypothèse 4.1 devient évidente, nous ne sommes pas responsables de la stratégie de stockage ni de la construction des différentes structures accédées par le système.

La figure 4.2 représente le système FNC-2 dans son ensemble. Le lecteur pourra en trouver une description dans [JoP89].

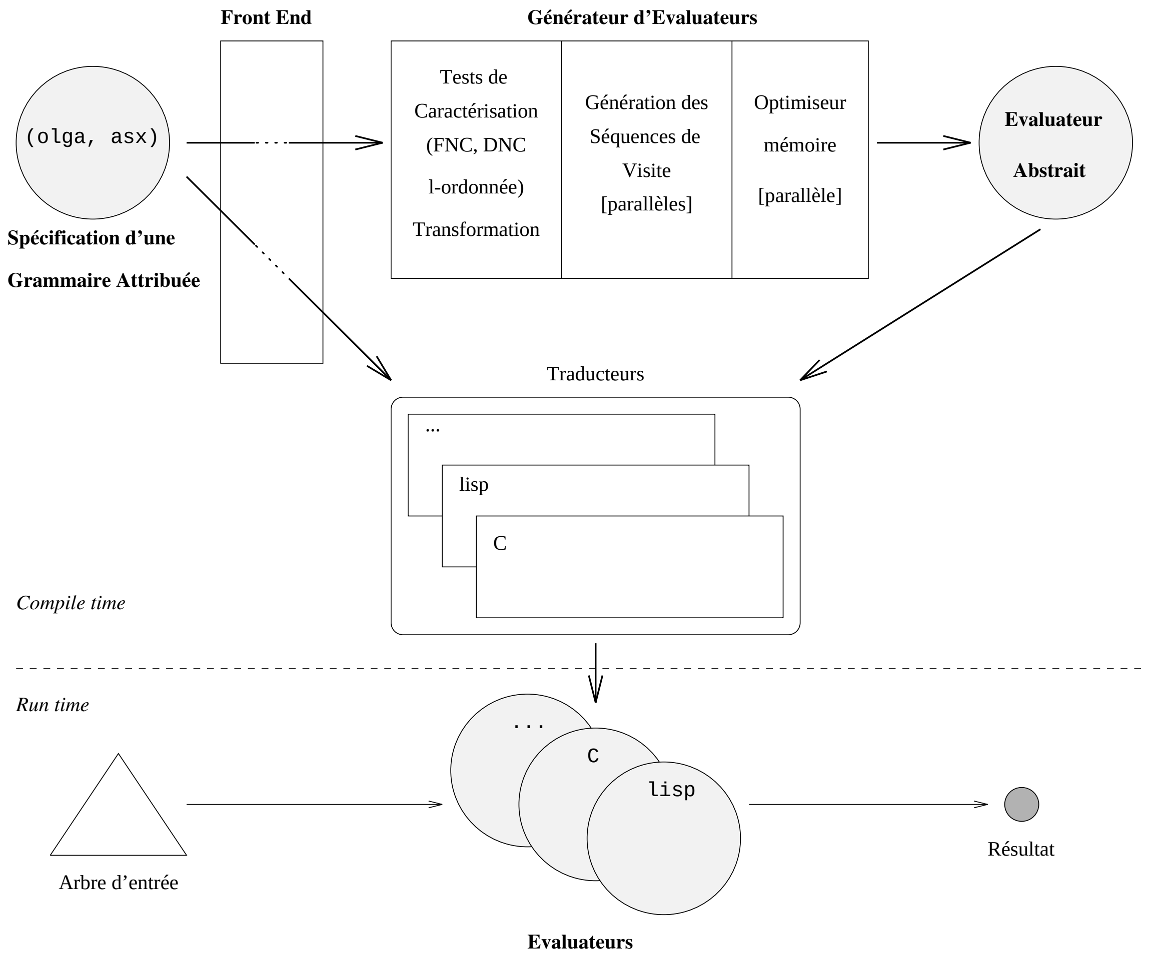


FIG. 4.2 - *Le système* FNC-2

L'insertion dans un système déjà existant apporte un certain nombre de contraintes quant à la façon d'implanter un nouveau concept : apprentissage du fonctionnement interne du système, respect de l'esprit de conception et, le plus important pour nous, l'accès aux fonctions sémantiques est déjà déterminé. En effet, c'est le *front-end* OLGA qui s'occupe des fonctions sémantiques et la partie de traduction de ces fonctions d'évaluation n'a pas été modifiée pour la version parallèle.

4.2.1 Les éléments de FNC-2 modifiés

Pour l'implantation de l'évaluation parallèle, nous avons souhaité modifier le moins possible le code FNC-2. Il a été nécessaire de modifier le générateur de séquences de visite. Le nouveau générateur implante les différents algorithmes proposés dans le chapitre 3.

La deuxième partie modifiée concerne le traducteur en C des séquences de visites précédemment construites. Chaque traducteur est décrit par un fichier C.

4.2.2 Les tests

Les tests des différentes versions sur nos architectures parallèles ont été faits avec le langage jouet Simproc. Ce langage est un sous-ensemble de Pascal et ses règles sémantiques sont des plus simples. Un compilateur Pascal étant en développement, nous souhaitons effectuer des tests avec ce langage. Nous sommes persuadés que les résultats seront encore meilleurs avec des grammaires réelles car, les règles sémantiques étant plus complexes, la durée de chaque tâche se trouve allongée.

4.3 Implantation du générateur de séquences de visite

Nous passerons assez vite sur ce point puisque ce n'est pas l'implantation du générateur qui nous intéresse mais celle du programme généré. Le générateur de visite prend en entrée les graphes de dépendance des productions ainsi que les ordres DNC, FNC et l-ordonnés. L'implantation qui en est faite calque assez bien la théorie décrite dans le chapitre 3. Elle ne s'occupe que de la détection du parallélisme et n'a par conséquent rien à voir ni avec le code généré ni avec les machines cibles. Le but de ce module est de construire le graphe des séquences de visite parallèles. Les structures associées à ce graphe contiennent toutes les informations nécessaires à l'élaboration du code chargé de l'évaluation.

4.4 Implantation du traducteur

La partie la plus intéressante de notre implantation est celle concernant le traducteur des séquences de visite en C. C'est l'efficacité du code généré par ce traducteur qui valide ou non notre approche. Plusieurs implantations de ce traducteur ont été effectuées. En plus du traducteur séquentiel déjà existant, qui traduit chaque visite par une procédure, nous avons écrit un autre traducteur séquentiel qui génère non plus des visites procédurales récursives mais un automate dont les transitions sont approximativement celles de l'automate théorique. Cette version a été développée surtout afin de permettre une comparaison entre la version parallèle et la version séquentielle qui correspondent à un même type d'implantation.

Les implantations parallèles effectuées peuvent se classifier suivant deux critères :

- *scheduling* système ou *scheduling* personnel ;
- séquences de visite procédurales ou par automate.

4.4.1 L'ordonnancement

Tout le parallélisme est exprimé dans nos algorithmes sous forme de tâches synchrones. Plusieurs solutions apparaissent pour *scheduler* ces tâches ; il est possible de lancer un processus système par tâche, ou bien d'effectuer soi-même leur gestion.

Ordonnancement système

Le système offre comme possibilité de processus, les processus "normaux" d'UNIX, obtenus par l'appel système *fork*, et comme nous l'avons vu dans la section 4.1, des processus "légers" obtenus par une librairie de *threads*.

Avant même d'implanter, nous nous sommes demandés si le *fork* UNIX était raisonnable. Nous avons comparé le temps de mise en œuvre d'un nouveau processus avec le temps de chaque tâche. Le résultat est que le *fork* coûte beaucoup trop cher pour organiser un parallélisme de grain aussi fin que celui auquel nous sommes confrontés. Il était par conséquent inutile d'implanter une version du traducteur générant un évaluateur utilisant un *fork* Unix pour chaque tâche parallèle. Aussi avons-nous implanté un ordonnancement système seulement avec des *threads*.

D'après les résultats que nous présenterons par la suite, nous avons également décidé d'utiliser, pour d'autres versions des programmes générés, un *scheduler* personnel simple. La seule chose dont nous ayons besoin, c'est un modèle du type producteur/consommateur de tâches. Chaque processus est à la fois producteur et consommateur. Quand il ne peut plus produire, il consomme. Le *scheduler* ainsi écrit ne s'occupe ni de priorité entre tâches ni d'équité.

Ordonnancement personnel

Notre *scheduler* est très simple. À sa base, une liste de séries de tâches à faire est partagée par tous les processus (voir figure 4.3). Cette liste est structurée en FIFO. Quand un processus veut envoyer une nouvelle série de tâches (sœurs) en parallèle, il verrouille la liste, ajoute un élément comprenant chacune des tâches sœurs, puis déverrouille la liste. Un élément de la liste contient donc plusieurs tâches à effectuer.

Quand un processus veut consommer une tâche, il verrouille la liste et prend la première tâche qui ne soit pas marquée "effectuée" du premier élément de la liste des tâches en attente. Puis il marque cette tâche comme effectuée. Si toutes les tâches de la série sont marquées effectuées, il retire l'élément de la liste des tâches en attente pour la mettre dans la liste des tâches en attentes de *join*.

L'avantage de mettre plusieurs tâches dans un même élément de liste est de verrouiller la liste moins longtemps lors d'une création de tâche. Le père prépare d'abord l'élément à mettre dans la liste et lorsqu'il verrouille la liste, c'est pour y ajouter toutes les tâches d'un seul coup.

En pratique, et pour des raisons d'efficacité, le *scheduler* ne fonctionne pas exactement ainsi. En effet, d'après le fonctionnement indiqué ci-dessus, chaque processus verrouille la liste dès qu'il veut ajouter une tâche, ou bien en retirer une. En gros, le *scheduler* décrit est section critique. Il est évident que cela n'est pas acceptable. Pour l'optimiser, nous avons créé une liste pour chaque processus Unix (qui correspond, dans notre modèle, à un processeur physique). Quand un processus produit, il remplit sa propre liste. Quand il consomme, il consomme en priorité sa liste, puis celle des autres processus si cette dernière est vide. De cette façon, nous diminuons la contention entre les processus.

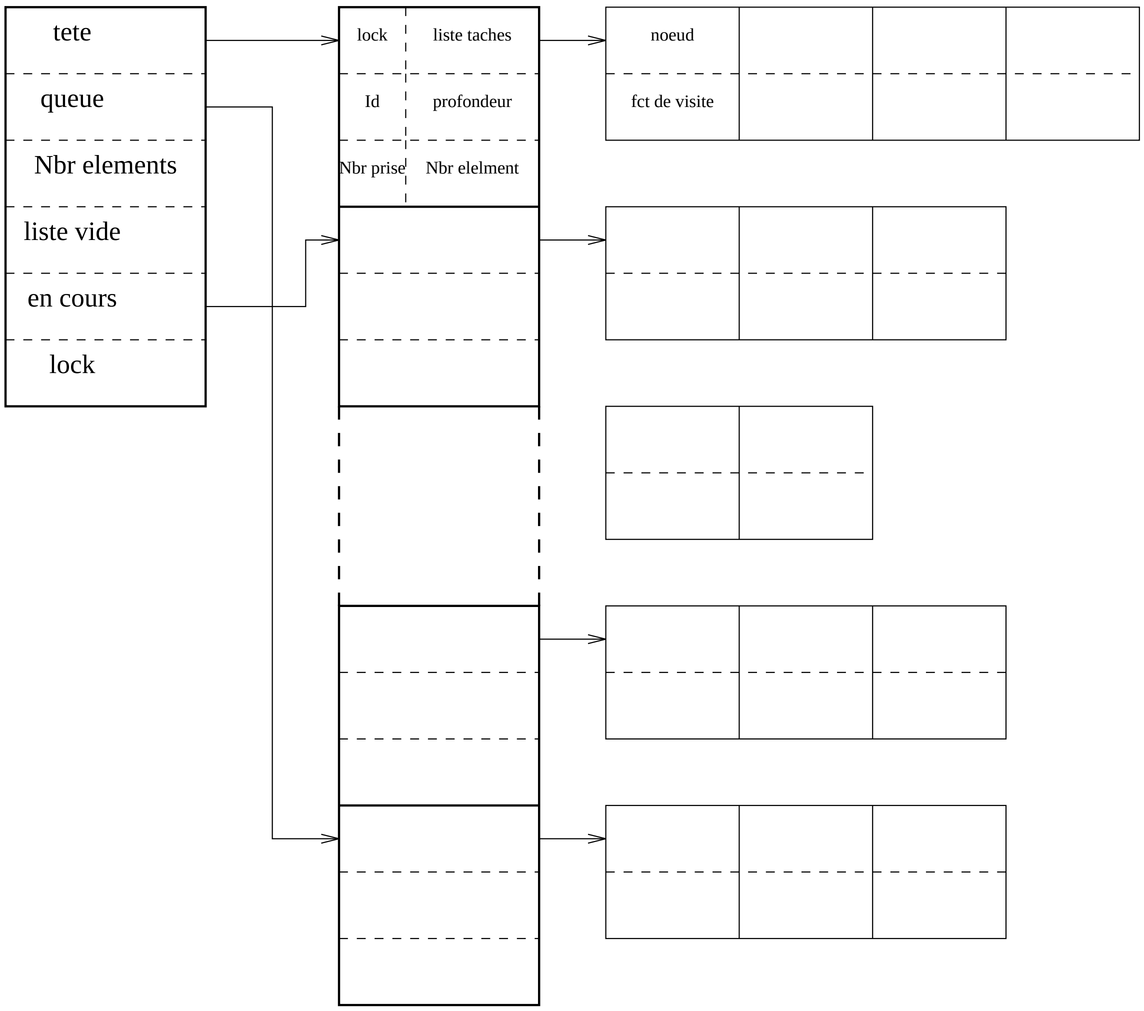


FIG. 4.3 - *Structure du scheduler*

D'autres optimisations de la gestion de la liste ont été faites afin de minimiser la taille des zones critiques.

Le *scheduler* ne verrouille jamais une file d'attente vide. De cette façon, tout processus qui n'a pas de travail ne "gêne" pas les autres processus.

4.4.2 Allocation mémoire

Un problème apparemment séparé de notre recherche concerne l'allocation dynamique de mémoire par l'appel de *malloc()* de la bibliothèque standard *C*. Curieusement, sur les machines parallèles que nous avons testées, il semblerait que la bibliothèque d'allocation soit particulièrement mal implantée. Les résultats obtenus par ce procédé sont illustrées par la figure 4.4, alors que la même méthode avec une allocation non blocante donne des résultats bien meilleurs (figure 4.5). La différence est significative lorsque le nombre de processus augmente. Plus il y a de processus, plus le blocage sur *malloc* est fréquent, et par conséquent, moins le code est performant. Une partie de notre implantation a donc consisté à écrire un module d'allocation mémoire pour chaque machine qui autorise plusieurs processus à allouer de la mémoire de façon concurrente. Chaque processus s'alloue une zone mémoire de grande taille, puis il effectue les allocations dynamiques à l'intérieur de cette zone. Quand la zone est pleine, il se réalloue une autre zone. Pour que cela soit valide, il faut bien entendu que cette zone soit dans la mémoire partagée par tous les processus, pour que chacun d'entre eux puisse accéder aux zones communes allouées par les autres processus. Le mécanisme est des plus simples car nous avons éliminé la possiblité de libérer l'espace mémoire.

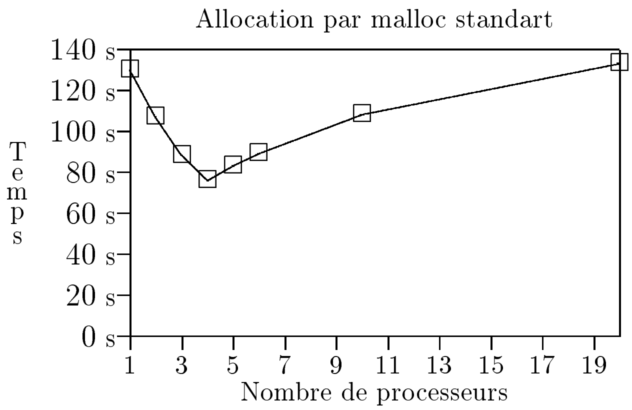


FIG. 4.4 - *Allocation mémoire de la bibliothèque standard*

4.4.3 Évaluateur procédural avec *scheduling* système

Les visites de l'évaluateur peuvent être vues comme des appels récursifs à une procédure, auquel cas, le `LEAVE` correspond au retour de l'exécution à la procédure appelante. Ce type d'implantation est très efficace en séquentiel car il permet de coder le parcours de l'arbre directement dans le code. C'est ce type d'implantation que l'on trouve actuellement dans

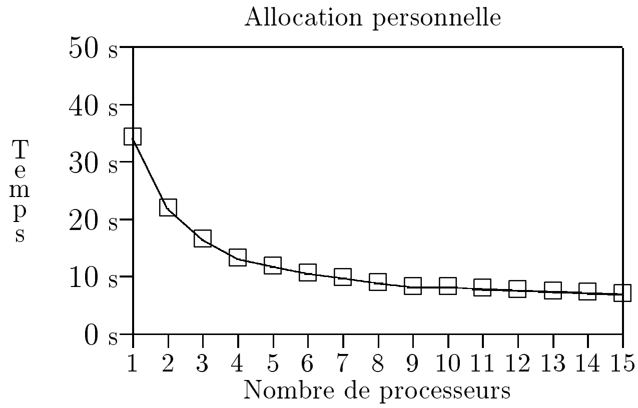


FIG. 4.5 - *Allocation mémoire personnelle*

le système séquentiel de FNC-2. Pour une comparaison avec les résultats séquentiels, une implantation de ce type a été effectuée (figure 4.6)

La traduction d'un *fork* se fait de la façon suivante :

```
/* lancement de 4 tâches */
type_tache tableau[4];
pour i de 1 à 4 faire
    tableau[i] = fork(tache(i))
fin pour
pour i de 1 à 4 faire
    attendre tableau[i]
fin pour
```

Dans cette implantation par *threads*, l'évaluateur se comporte de façon tout à fait naturelle : un processus envoie des tâches sœurs en parallèle, puis exécute un *wait* sur ses fils. Dès que tous les fils ont terminé, le père reprend l'exécution et termine sa tâche. Le modèle procédural correspond donc tout à fait à l'implantation par *threads*.

Les résultats de cette implantation sont particulièrement mauvais, comme le montre la figure 4.6. En fait, les *threads* sont des processus encore trop lourds à gérer. Comme nous le craignions, un *scheduling* personnel s'avère indispensable. Il est également très difficile de faire des mesures cohérentes car le *sheduling* système rend le temps d'exécution très aléatoire, et très dépendant de la charge totale de la machine.

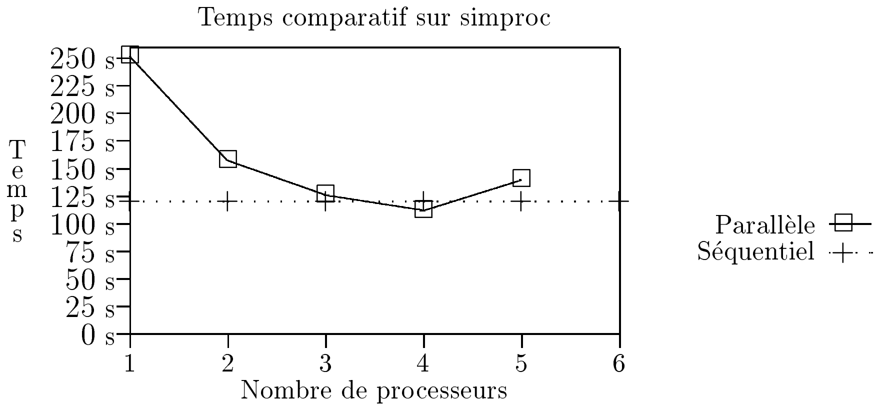


FIG. 4.6 - *Résultats sur KSR de la version procédurale*

4.4.4 Évaluateur procédural avec *scheduling* personnel

La figure 4.7 montre la communication entre l'évaluateur et le *scheduler* personnel. Il est à remarquer que le *scheduler* n'est pas un processus à part mais un sous-programme de chaque évaluateur. Plusieurs *schedulers* peuvent travailler en même temps. Le *scheduler* a deux rôles : d'une part c'est lui qui contrôle si le *join* pour lequel il a été appelé peut être effectué, d'autre part, en cas d'impossibilité d'exécuter le *join*, il décide d'effectuer une nouvelle tâche.

Pour la version *schedulée* :

/ lancement de 4 tâches */*
série_tache = créer_série_tache(4)
pour i de 1 à 4 **faire**
    ajoute tache(i) à série_tache
fin pour
ajouter série_tache à la liste des tâches.
scheduler() */* pendant l'attente des fils, on peut prendre une tâche */*

Dans la version procédurale du programme, le *scheduler* ne se comporte pas de façon satisfaisante : au niveau de l'exécution, un même processus peut effectuer deux tâches l'une à la suite de l'autre, ceci sans terminer la première. Ce cas se rencontre fréquemment lorsque le processus travaille à une tâche et lance des sous-tâches. Il doit alors attendre que ces sous-tâches se terminent. Il ne reste pas sans rien faire, donc il prend une deuxième tâche. Au niveau des processus, la pile d'exécution des appels récursifs contient donc quelque chose de la forme :

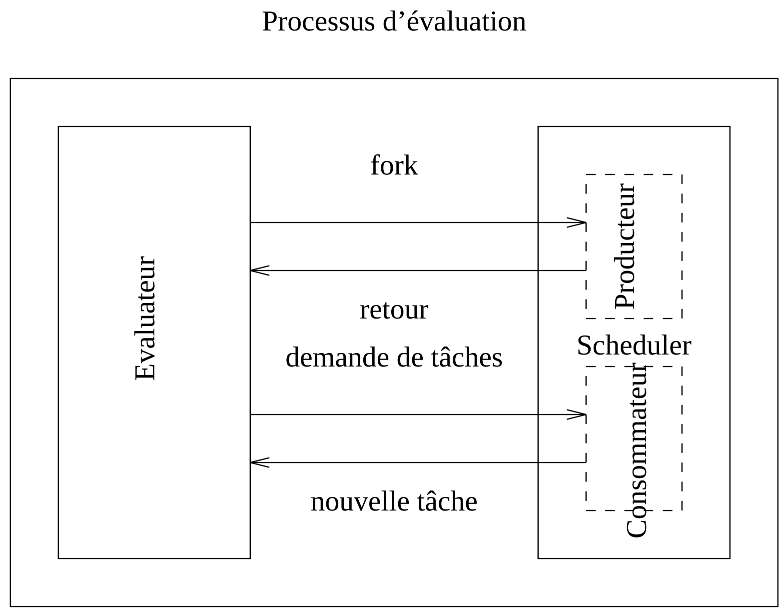


FIG. 4.7 - *Communication Évaluateur/scheduler*

tâche1
scheduler
tâche2
scheduler
tâche3

De cette façon, l'exécution de la tâche 2 ne peut se terminer que si la tâche 3 est terminée, et l'exécution de la tâche 1 se finira seulement lorsque les tâches 2 et 3 seront toutes deux achevées. Le mécanisme de retour du *scheduler*, qui correspond au retour d'un *wait* des tâches, ne correspond pas à la définition formelle du *wait*. Le procédé utilisé ici est très limité puisqu'il induit des dépendances dynamiques entre les tâches. Malgré tout, les résultats que nous obtenons ne sont pas aussi mauvais qu'on pourrait le croire au premier abord, comme le montrent les figures 4.8 à 4.11. Et le *speedup* à l'arrivée est encore acceptable. La figure 4.8 donne les résultats obtenus sur la KSR sans limitation de lancement de tâches. Nous voyons que la contention est réduite au minimum puisque l'ajout important de processus n'induit pas d'inefficacité, comme le montre la figure 4.9. Les résultats sont quasiment identiques sur toutes les machines. Pourtant le *speedup* pourrait être amélioré par l'utilisation de l'heuristique de limitation de la "profondeur" des tâches. Les figures 4.12, 4.13 et 4.14 montrent les résultats obtenus en limitant cette profondeur à trois.

la figure 4.15 montre l'efficacité de la parallélisation

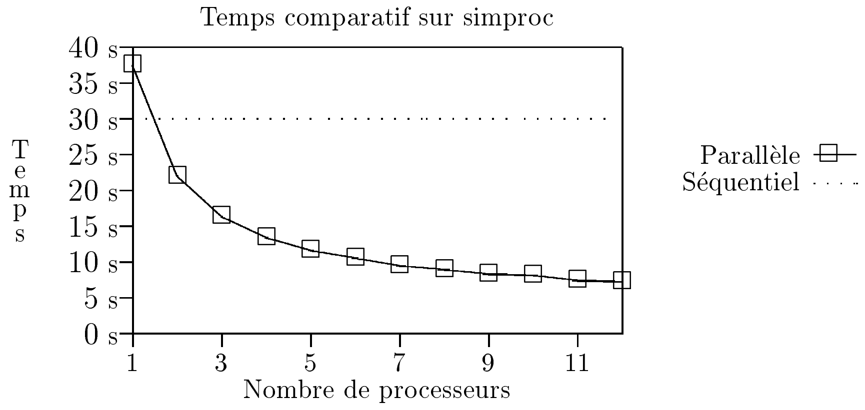


FIG. 4.8 - *KSR : version procédurale schedulée*

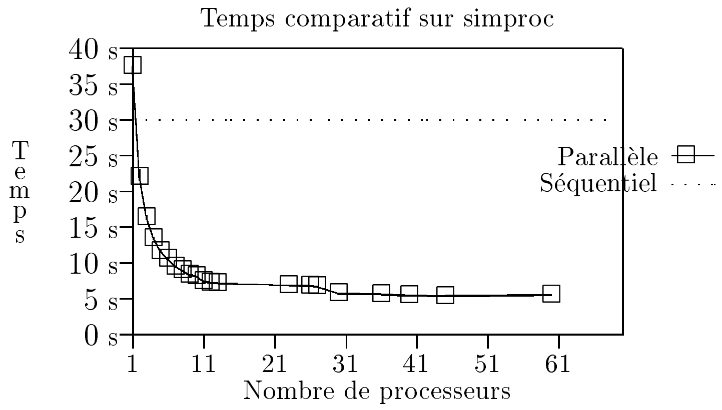


FIG. 4.9 - *KSR : version procédurale schedulée*

Un aspect plus désagréable de l'empilement excessif dans la pile d'exécution est qu'à l'analyse d'un gros programme, nous risquons le débordement de pile, ce qui provoque une erreur système alors que rien n'est théoriquement faux. Le seul remède est de redimensionner la pile à une taille supérieure et ce, manuellement. Aucune estimation de la hauteur de la pile ne peut être donnée.

Cela nous amène à concevoir une autre implantation où ce problème n'apparaît pas. Cette version est basée sur une gestion des visites par automate.

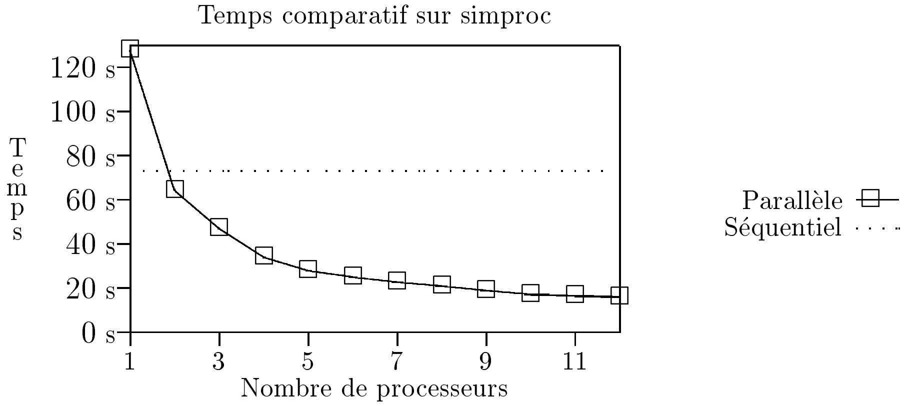


Fig. 4.10 - *Multimax: version procédurale schedulée*

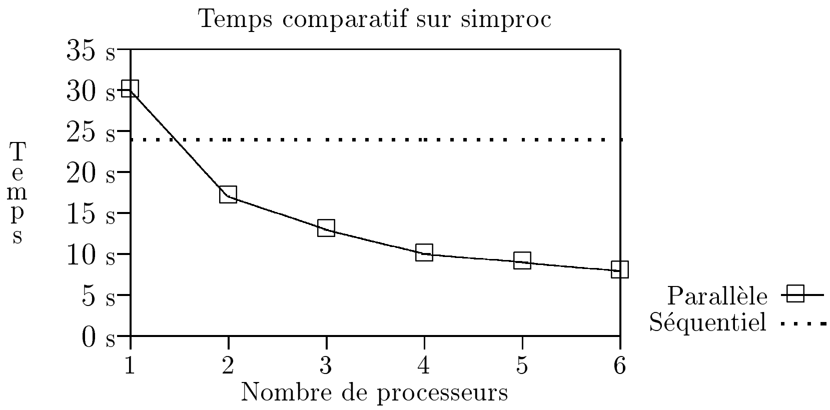


Fig. 4.11 - *B8000: version procédurale schedulée*

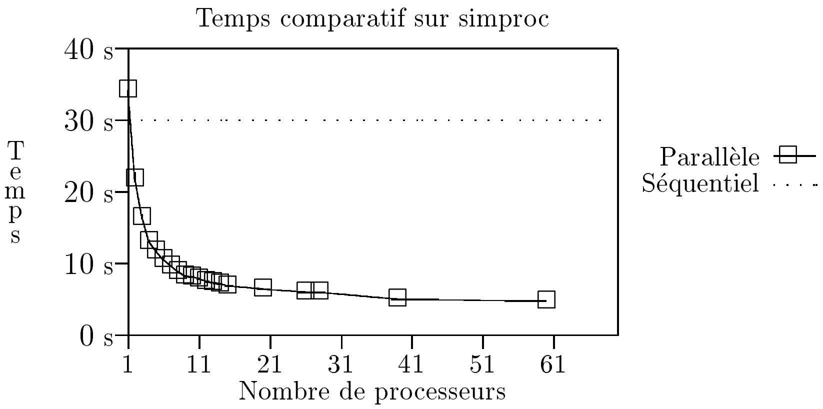


Fig. 4.12 - *KSR : version procédurale schedulée, profondeur 3*

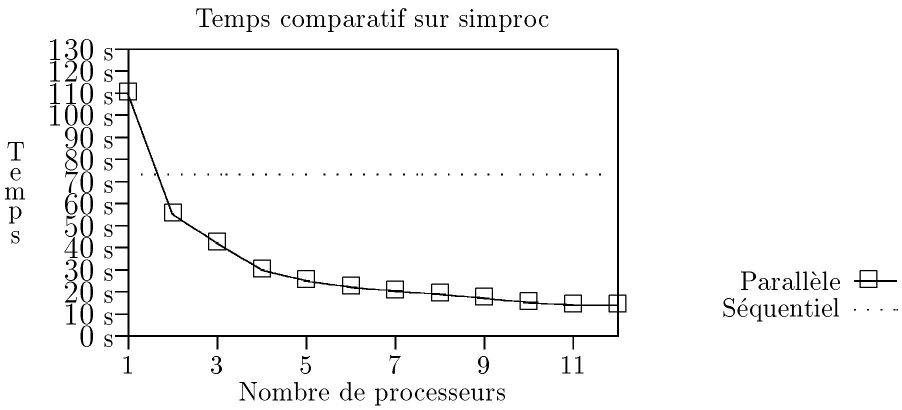


Fig. 4.13 - *Multimax : version procédurale schedulée, profondeur 3*

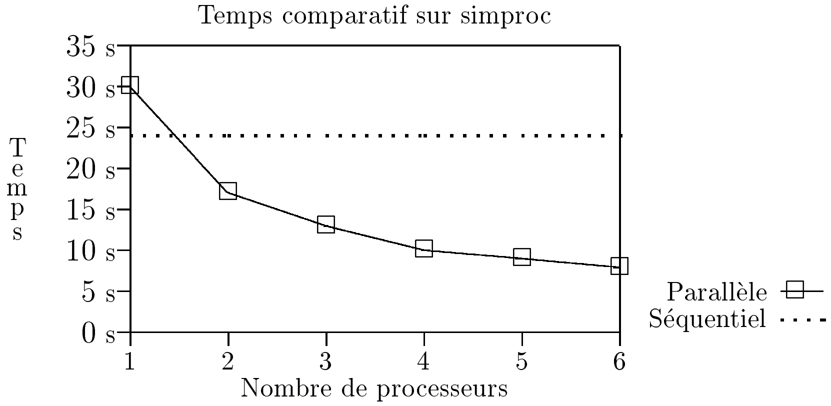


FIG. 4.14 - *B8000 : version procédurale schedulée, profondeur 3*

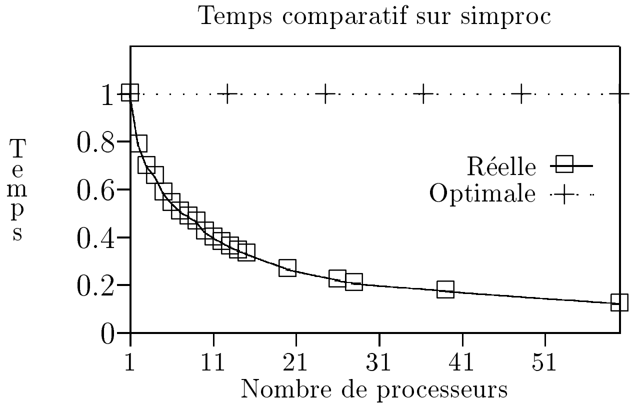


FIG. 4.15 - *Speedup pour la version procedurale schedulée*

4.4.5 Évaluateur par automate

Comme nous l'avons vu dans le chapitre 1, l'évaluateur séquentiel peut se décrire comme un automate dont les actions sont `EVAL`, `VISIT` et `LEAVE`. Le chapitre 3 a montré que nous pouvions étendre cet automate au cas parallèle en ajoutant les actions `COBEGIN` et `COEND`. Pour pallier au problème rencontré dans le *scheduler* personnel avec l'implantation procédurale, nous avons choisi d'implanter l'automate tel quel. Le principal avantage de cette méthode est de permettre à n'importe quel processus de terminer une tâche.

Le principe est simple : un processus obtient une tâche, il y travaille jusqu'à ce qu'il la termine, ou qu'il tombe sur un `COBEGIN`. Dans le cas d'une instruction parallèle, il lance les tâches sœurs et continue sa tâche jusqu'au *wait* dénoté par le `COEND`. Arrivé au `COEND`, il regarde s'il peut effectuer le *join*. Si c'est le cas, il continue. Sinon, il demande un nouveau travail. Pendant ce temps, d'autres processus (voire lui-même) auront effectué les tâches sœurs. Parmi ceux-ci, il en est un qui arrivera en dernier sur le `COEND`. Celui-ci continuera la tâche. Cette action est rendue possible car il n'y a pas d'appel récursif de procédure au niveau de l'implantation. La pile d'appel est toujours de même niveau :

le *scheduler* l'automate [fonctions d'évaluation]

Évidemment, le gain en souplesse au niveau du *join* se paie par la gestion complète de la pile du parcours d'arbre. Ce qui était fait automatiquement par la pile des paramètres doit maintenant être géré en association avec la liste des tâches. L'automate à pile se charge donc de gérer sa pile d'état. Le problème est d'arriver à faire co-exister plusieurs automates en parallèle, et par conséquent plusieurs piles. Mais une pile n'est pas complètement indépendante d'une autre. En effet, étudions l'automate initial ; il empile ses états dans sa pile. Lors d'une parallélisation, il doit créer autant de tâches (c'est-à-dire autant d'automates) que de branches parallèles. Chacune de ces tâches doit avoir sa propre pile d'état. De plus, chacune des tâches parallèles peut, si elle est la plus lente à s'exécuter, terminer la tâche initiale (autrement dit récupérer la pile d'état de la tâche initiale). Nous voyons donc qu'une tâche correspond à un état séquentiel d'automate, plus la pile d'état de la tâche appelante.

La figure 4.16 donne la structure d'une telle pile arborescente.

En fait, nous pouvons représenter la totalité des piles d'automates à un instant donné par une sorte de pile arborescente. Chaque automate voit son sommet de pile. Tant que nous sommes dans un cas séquentiel, l'automate peut empiler et dépiler comme il le veut. Lors d'un `COBEGIN`, il suffit d'empiler un élément spécial dans la pile. Cet élément contiendra le nombre de branches créées à cet étage de la pile. Nous assignerons un automate par branche créée. L'élément spécial ne pourra être dépilé que si tous les éléments qui lui sont supérieurs ont été dépilés. Ceci correspond au fait que nous ne pouvons effectuer un *join* que si toutes les tâches parallèles lancées à ce niveau sont terminées.

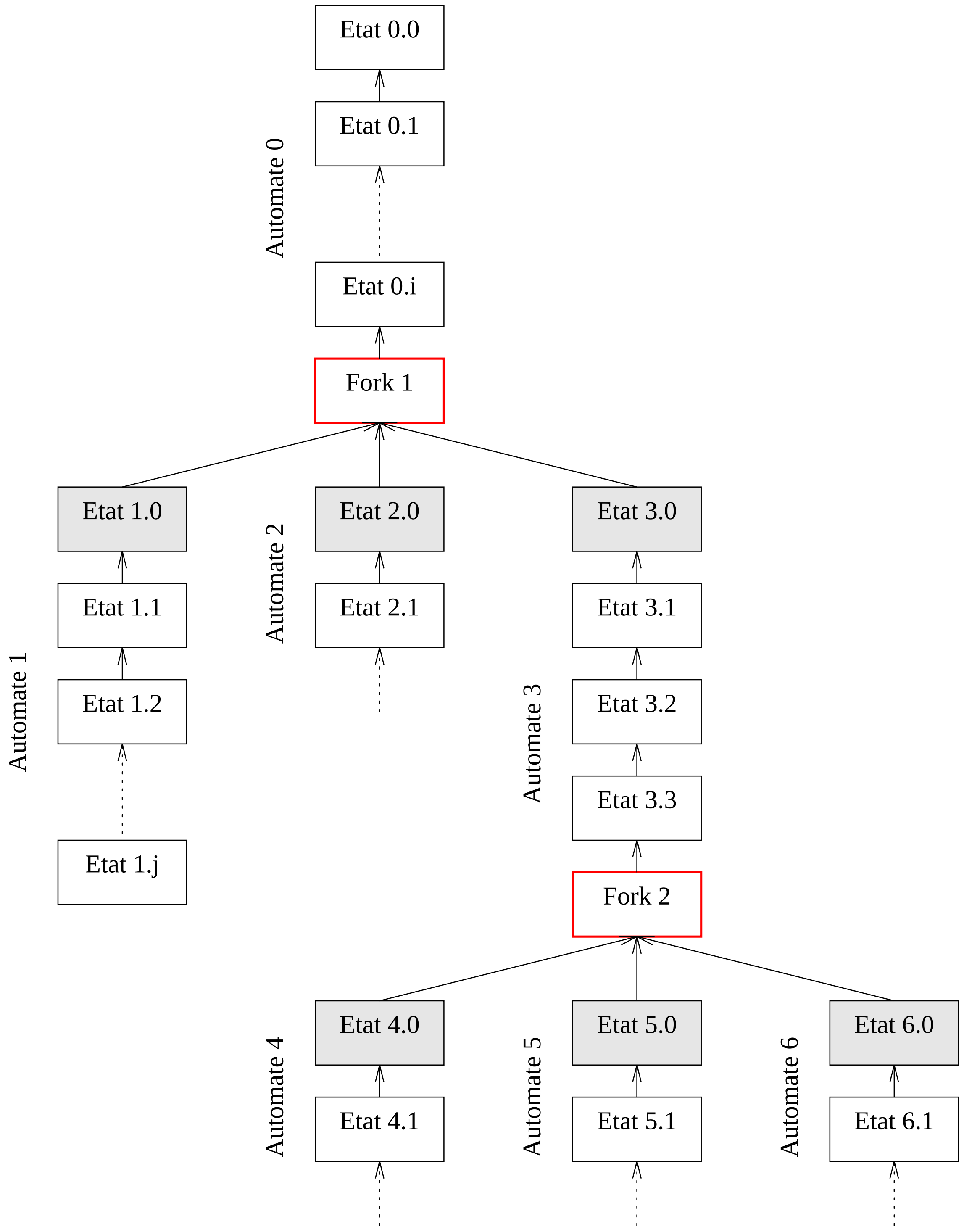


FIG. 4.16 - *Pile d'état de l'automate.*

Pour implanter cette pile, nous avions une contrainte assez précise. Les éléments qui suivent un élément spécial correspondent à des tâches. La liste des tâches à effectuer se doit donc de les référencer. Notre implantation a directement réuni ces éléments au sein d'une liste de tâches, comme le montre la figure 4.17.

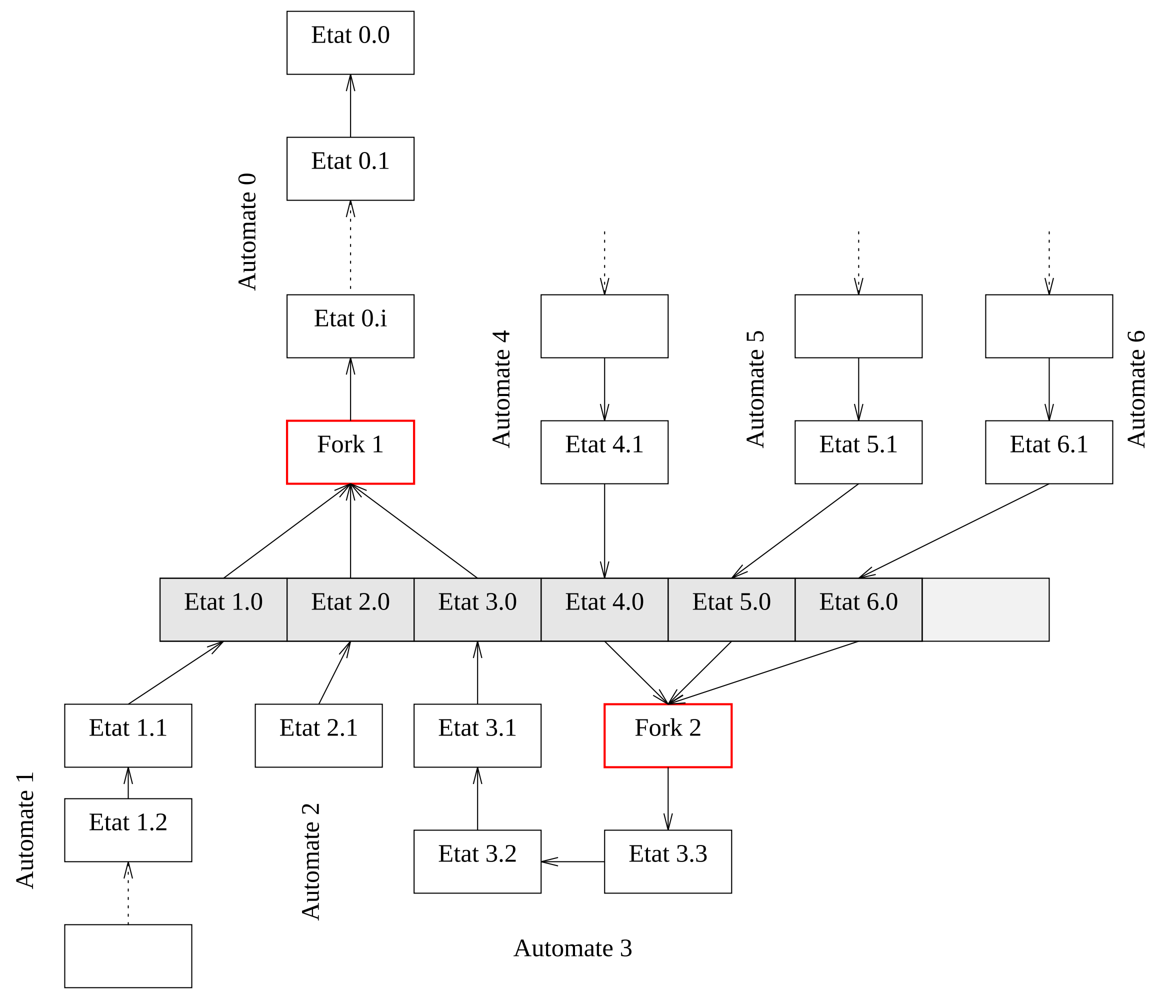


FIG. 4.17 - *Pile d'état de l'automate associée à la liste des tâches.*

Une tâche est constituée :

– d'un état initial,

– d'une pile d'état pour effectuer le *join*.

Un état est constitué :

– d'un compteur de programme,

– d'un nœud père de production,

– d'un nœud fils.

Cette version donne d'excellents résultats quant au *speedup* mais il faut toutefois émettre une remarque : la méthode de l'automate est moins efficace si on la fait tourner sur un seul processeur. Nous avons introduit en séquentiel une gestion qui n'a vraiment d'intérêt qu'en parallèle. Pourtant, il faut remarquer que le gain par processus est très satisfaisant. Malheureusement, le temps d'exécution de la version séquentielle est largement plus grand que pour la version procédurale et la différence de *speedup* ne réussit pas à rendre la version interprétée véritablement attractive.

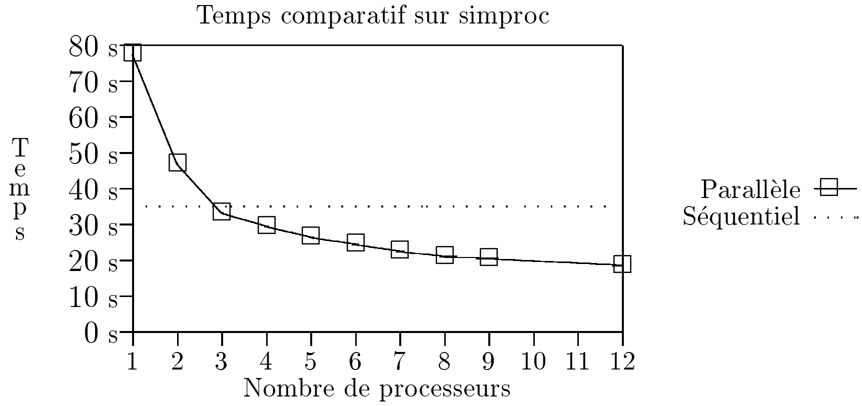


FIG. 4.18 - *Version interprétée parallèle pour la KSR1*

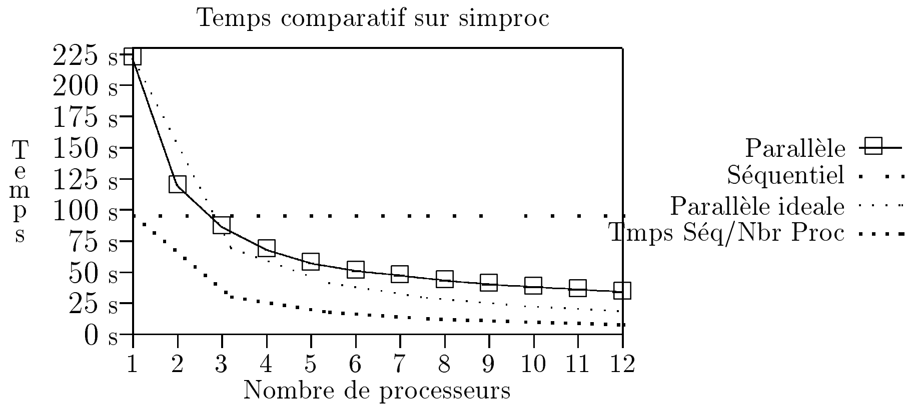


FIG. 4.19 - *Version interprétée parallèle pour le Multimax*

Sur le Multimax, la figure 4.19 nous montre que la version interprétée est plus efficace à mesure que nous ajoutons des processeurs, contrairement à la KSR1 où nous notons une stabilisation aux alentours de 10 processeurs. Cela semble être du à la communication entre les processus. Dans la KSR1, nous n'avons pas fixé les processus sur des processeurs particuliers et le temps nécessaire à un transfert d'un processeur à un autre, cumulé avec le temps de rechargement des caches, semble être pénalisant.

4.5 Conclusion

Ce chapitre a montré différentes implantations de l'évaluation parallèle d'attributs. D'une part des versions où les séquences de visite sont traduites par des procédures et, d'autre part, les versions interprétées où l'évaluateur est un automate à pile. Nous avons vu que les processus système et le *scheduling* associé sont bien trop lourds pour le peu de travail effectué par nos tâches. La présence d'un *scheduler* spécifique est donc indispensable.

Les résultats en *speedup* sont à l'avantage de la version interprétée, mais malheureusement, le handicap en efficacité dû à l'automate rend cette implantation guère utilisable. La version qui, au vue des résultats, nous semble la plus efficace reste donc paradoxalement la version procédurale *schedulée*. Le problème d'empilement ne semble pas diminuer grandement les performances. De plus, le problème de débordement de pile ne se produit que pour des programmes de taille gigantesque (50 Mo de code).

Pour améliorer ces performances déjà intéressantes, la simple heuristique de limiter le parallélisme dans l'arbre augmente de façon significative le *speedup*, ce qui prouve bien que de nombreuses tâches sont particulièrement petites. Ainsi, toute heuristique applicable statiquement serait encore plus avantageuse et améliorerait d'autant les performances.

Nos implantations n'étant pas spécifiques aux caractéristiques de certaines machines, nous pouvons en conclure que l'utilisation de machines multiprocesseurs à mémoire partagée comme machines cibles pour les évaluateurs n'est pas vaine. Nous sommes par conséquent capables, avec les grammaires attribuées de générer du code parallèle efficace à partir d'une spécification où tout concept parallèle est masqué à l'utilisateur.

Le chapitre 6 va étudier une optimisation supplémentaire du code parallèle généré, qui n'a rien à voir avec la sélection du parallélisme, mais qui permet d'appliquer en parallèle les techniques d'optimisation mémoire déjà connues dans le cas séquentiel.

Chapitre 5

Optimisation mémoire : Le cas séquentiel

Ce chapitre donne un rappel de l'optimisation mémoire séquentielle developpée dans la thèse de C. Julié [Jul89]. Le lecteur désirant en savoir plus dans ce domaine pourra s'y reporter.

5.1 Le principe

Lors de l'évaluation sur un arbre d'entrée, il y a un très grand nombre d'instances d'attributs à calculer. L'encombrement mémoire qui en résulte est très important si on tient à stocker toutes les instances d'attribut dans l'arbre. Pourtant, ces instances ne sont pas toutes actives à un instant donné. En effet, un grand nombre d'attributs n'ont pas de valeur avant un certain point de l'évaluation. Réciproquement, la valeur de certaines instances n'est plus jamais utilisée à partir d'un point de l'évaluation. Cette notion, pour l'instant informelle, correspond au concept de durée de vie défini plus loin dans ce chapitre.

Le but de l'optimisation mémoire est de faire partager un même espace mémoire hors de l'arbre à plusieurs instances d'attribut, en prenant en compte les durées de vie de ces instances. De nombreux auteurs se sont attaqué à cet épineux problème. La méthode présentée dans ce chapitre est celle de C. Julié [Jul89], qui intègre un certain nombre de propositions en matière d'optimisation mémoire [Kas84, FaY86]. Elle s'applique à tout évaluateur construit sous forme de séquences de visite. L'hypothèse forte de cette méthode est **l'existence d'un ordre total d'évaluation**, afin de connaître **statiquement** l'ordre de calcul des attributs, indépendamment du contexte courant, c'est-à-dire indépendamment de l'arbre d'entrée. Cette hypothèse est vérifiée pour la classe des grammaires l-ordonnées traitée par le système FNC-2.

C. Julié propose deux types de structures pour l'optimisation des attributs : d'une part des variables globales pour les attributs qui ont une seule instance active à un instant donné et, d'autre part des piles, pour certains attributs ayant plusieurs instances actives au même moment.

Pour clarifier ces choix, nous préciserons tout d'abord la notion d'instance **active** et nous rappellerons les définitions pour les **attributs temporaires**. Nous donnerons enfin les conditions suffisantes pour mettre un attribut en pile ou en variable, sans qu'il y ait perte d'informations.

Définition 5.1.1 (Calcul d'évaluateur) *Soit E l'évaluateur d'une grammaire attribuée G. Soit t un arbre d'entrée de G. On appelle calcul de E sur t, la suite $c = A_1 A_2 \dots A_p$ d'instances d'attributs telle que :*

- *c contient toutes les instances d'attributs de t exactement une fois.*
- *Si il existe une instance A_j qui dépend d'une instance A_i, alors $i < j$.*

Définition 5.1.2 (Durée de vie) *Soient c un calcul sur un arbre t, A_i une instance d'attribut, A_j l'instance d'attribut dont le calcul fait référence à A_i pour la dernière fois.*

On appelle **durée de vie** *de A_i la séquence de calcul comprise entre A_i et A_j.*

La notion de durée de vie correspond à l'intervalle de temps compris entre l'instant de la définition d'une instance d'attribut et l'instant où cette instance n'est plus utilisée. En ce qui concerne l'occupation mémoire, c'est la durée pendant laquelle la valeur de l'instance d'attribut doit rester accessible.

Définition 5.1.3 (Attributs temporaires) *Une occurence d'attribut est dite temporaire si sa définition et sa dernière utilisation ont lieu dans la même visite.*

Un attribut est temporaire si toutes ses occurences sont temporaires.

Les attributs temporaires sont fortement intéressants pour l'optimisation mémoire car ils ne sont utiles que pour une seule visite. Ainsi, la caractérisation de la structure à employer pour leur stockage est beaucoup plus simple que pour les attributs non-temporaires. De plus, l'expérience montre que la plupart des attributs sont temporaires. La section suivante montre comment implanter ces attributs en variable ou en pile.

5.2 Implantation en variable globale

5.2.1 Le problème

Soit $X.a$ un attribut temporaire. Est-il possible de stocker toutes les instances associées à l'attribut a dans une seule variable globale, sans affecter la valeur sémantique d'un arbre de dérivation, c'est-à-dire en conservant la correction du résultat qu'est la valeur des attributs synthétisés de la racine ?

Comme nous faisons cela statiquement, cela signifie qu'il est nécessaire de prouver que, quel que soit l'arbre d'entrée, il n'existe pas deux instances de $X.a$ actives au même instant. Autrement dit, il est nécessaire que leurs durées de vie soient disjointe.

Une condition suffisante pour que cette propriété soit vérifiée est que [KHZ82] :

– Pour un attribut hérité : Soit $X.a$ défini avant la $k^{ème}$ visite de X et plus jamais référencé après cette visite. $X.a$ ne peut pas être implanté en variable si, entre la première évaluation de la $k^{ème}$ visite et la dernière référence à $X_0.a$:

  1. soit une occurrence de $X.a$ est définie en partie droite de la production ;
  2. soit un nœud Y est visité tel que Y puisse dériver dans X.

– Pour un attribut synthétisé : Soit $X_j.a$ défini pendant la $k^{ème}$ visite de X_j et plus jamais référencé après le retour à l'ancêtre de ce nœud. $X.a$ ne peut pas être implanté en variable si, entre la première évaluation consécutive à la $k^{ème}$ visite de X_j et la dernière référence à $X_j.a$:

  1. soit une occurrence de $X.a$ est définie en partie gauche de la production ;
  2. soit un nœud Y est visité tel que Y puisse dériver dans X.

La différenciation du cas synthétisé et du cas hérité provient du fait que lorsqu'on regarde une production, l'utilisation d'un attribut hérité (resp. synthétisé) ne peut se faire que lorsque celui-ci est en partie gauche (resp. droite) de la production. L'attribut hérité (resp. synthétisé) est alors défini au-dessus (resp. en dessous) de la production considérée.

La signification de cette définition est qu'un attribut hérité peut être mis en variable si, entre le moment de la définition d'une de ces instances et celui de la dernière utilisation de cette instance, aucune autre instance n'est définie dans la même production sur un non-terminal en partie droite et, de plus, qu'aucun non-terminal en partie droite ne dérive sur le non-terminal portant l'attribut.

Le cas synthétisé s'explique pareillement.

Cette propriété a été affinée par [Jul89], en remarquant que la clause 2) est trop restrictive. En effet, la visite à un nœud Y qui dérive dans un nœud X n'implique pas forcément une nouvelle définition de l'attribut considéré. Il faut regarder quelle visite sur Y est effectuée par la visite au nœud X. Pour déterminer cette condition, C. Julié introduit la notion de grammaire de visite.

5.2.2 La solution

Une grammaire de visite représente l'imbrication des différentes visites sur les non-terminaux. La grammaire de visite permet d'exprimer le fait que la $i^{ème}$ visite sur un non-terminal A déclenche la $j^{ème}$ visite sur un non-terminal B. Cette information est indispensable pour répondre plus précisément au problème du stockage en variable.

Grammaires de visites : définition

En séquentiel, à partir de l'évaluateur, on définit les grammaires de visite de la façon suivante :

$Gv = (Nv, Tv, Zv, Pv)$
avec

- $Nv = \{(X,k), X \in N$ et $k \in [1, \phi(X)]\}$
  où $\phi(X)$ représente le nombre de visites sur le non-terminal X.

- $Tv = \emptyset$

- $Zv = (Z, 1)$

- - Soit $p : X_0 \leftarrow X_1, X_2, \ldots, X_{n_p} \qquad p \in P$
  - Soit $vs_p = vs_{p,1}, \ldots, vs_{p,\phi(X_0)}$ la séquence de visite globale associée à p.
    On note $vs_{p,i} = vs_{p,i,1}, \ldots, vs_{p,i,n_{p_i}}$les instructions `EVAL` `VISIT` associées à $vs_{p,i}$
  - On définit
    $Pv = \{p(l) | p \in P$ et $l \in [1, \phi(X_0)]$ avec $p(l) : (X_0, l) \leftarrow (X_1, k_1), \ldots, (X_m, k_m)\}$
    1. $\forall i \in [1,m], \exists j | vs_{p,l,j} = \texttt{VISIT}(X_i, k_i)$
    2. les (X_i, k_i) sont ordonnés suivant leur ordre d'apparition dans $vs_{p,i}$

Si nous reprenons l'exemple du chapitre 1, la grammaire de visite correspondante est donnée par la figure 5.1.

Grammaires des durées de vie

La grammaire de visite nous donne l'imbrication des visites. Il nous faut encore calculer les durées de vie associées aux instances d'attribut. Pour cela, nous ajoutons des terminaux à la grammaire de visite. Ces terminaux représentant la définition et la dernière utilisation d'une instance.

Pour déterminer les durées de vie associées à un attribut a défini à la $k^{ème}$ visite d'un non-terminal X, on définit la grammaire des durées de vie par :
$D_{DV}(X.a) = \{N', T', P', Z'\}$ avec

- $N' = Nv$: les non-terminaux sont ceux de la grammaire de visite.

- $T' = \{D, A\}$: D et A spécifient respectivement un point de **D**éfinition de $X.a$ et un point dernier **A**ccès à $X.a$.

- L'ensemble P' est construit à partir de Pv et des séquences de visite, en ajoutant sur certaines productions les terminaux D et A. La modification est la suivante : pour toute production $p(l) \in Pv$ qui contient au moins une occurrence de (X,k), on considère vs la séquence de visite associée à $p(l)$. Pour toute instruction $\texttt{EVAL}(B)$ de vs, on ajoute un terminal A pour chaque dernier accès fait à une instance de $X.a$ et on ajoute un terminal D pour chaque nouvelle définition de $X.a$

- $Z' = Zv$.

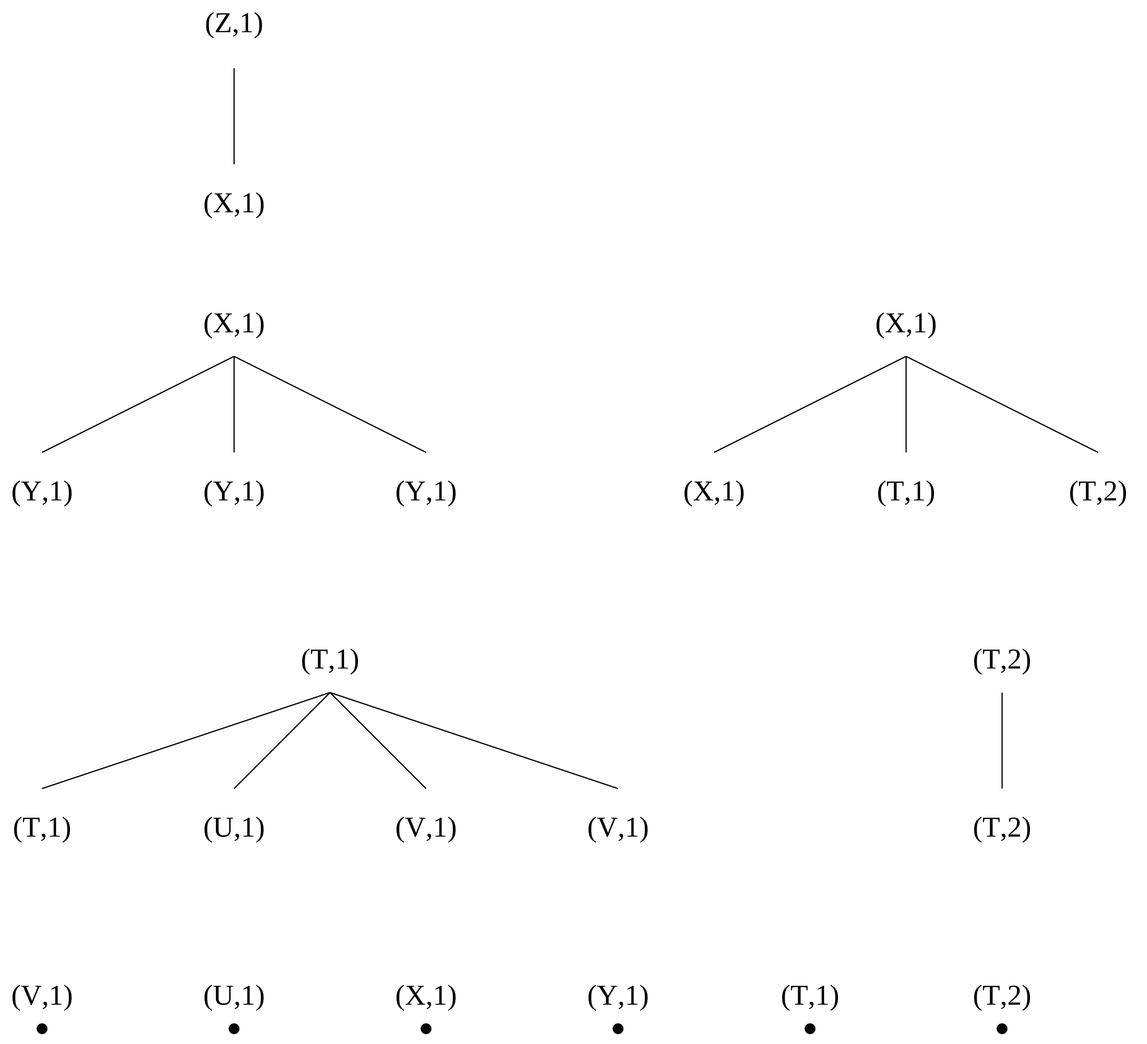


FIG. 5.1 - *Grammaire de visite*

Pour savoir si un attribut peut être implanté en variable, il suffit d'étudier le langage engendré par la grammaire des durées de vie.

Dans la grammaire $G_{DV}(X.a)$, chaque couple (D, A) représente la durée de vie d'une instance de $X.a$, même si ces deux symboles ne sont pas présents dans la même production.

Proposition 5.2.1 *Un attribut $X.a$ peut être implanté en variable si et seulement si sa grammaire des durées de vie vérifie :* $\mathcal{L}(G_{DV}(X.a)) = (DA)^*$

La preuve de cette proposition est donnée dans [Jul89]. La figure 5.2 montre un exemple d'attribut pouvant être implanté en variable, alors que la figure 5.3 montre un attribut qui ne peut pas l'être.

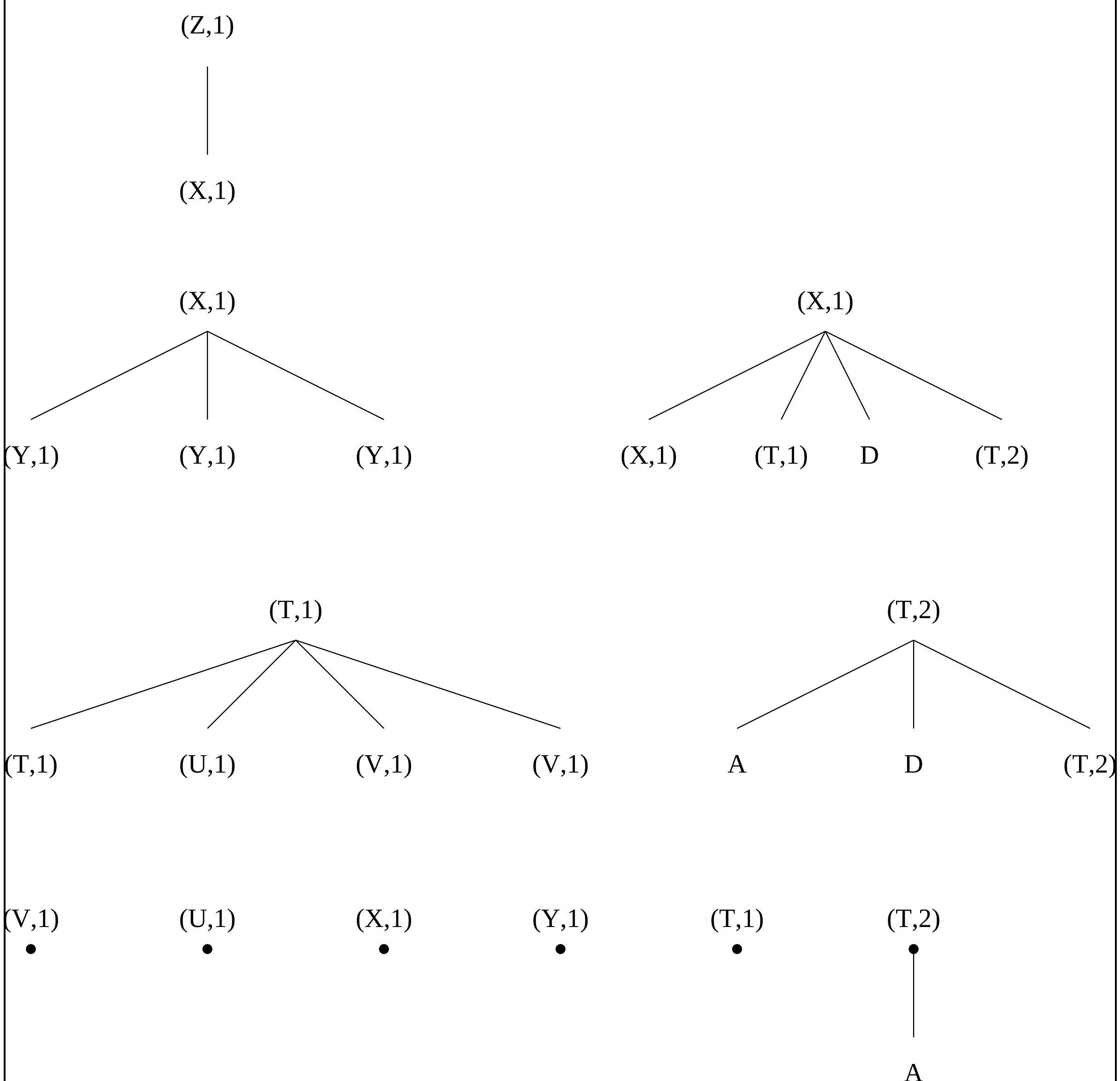
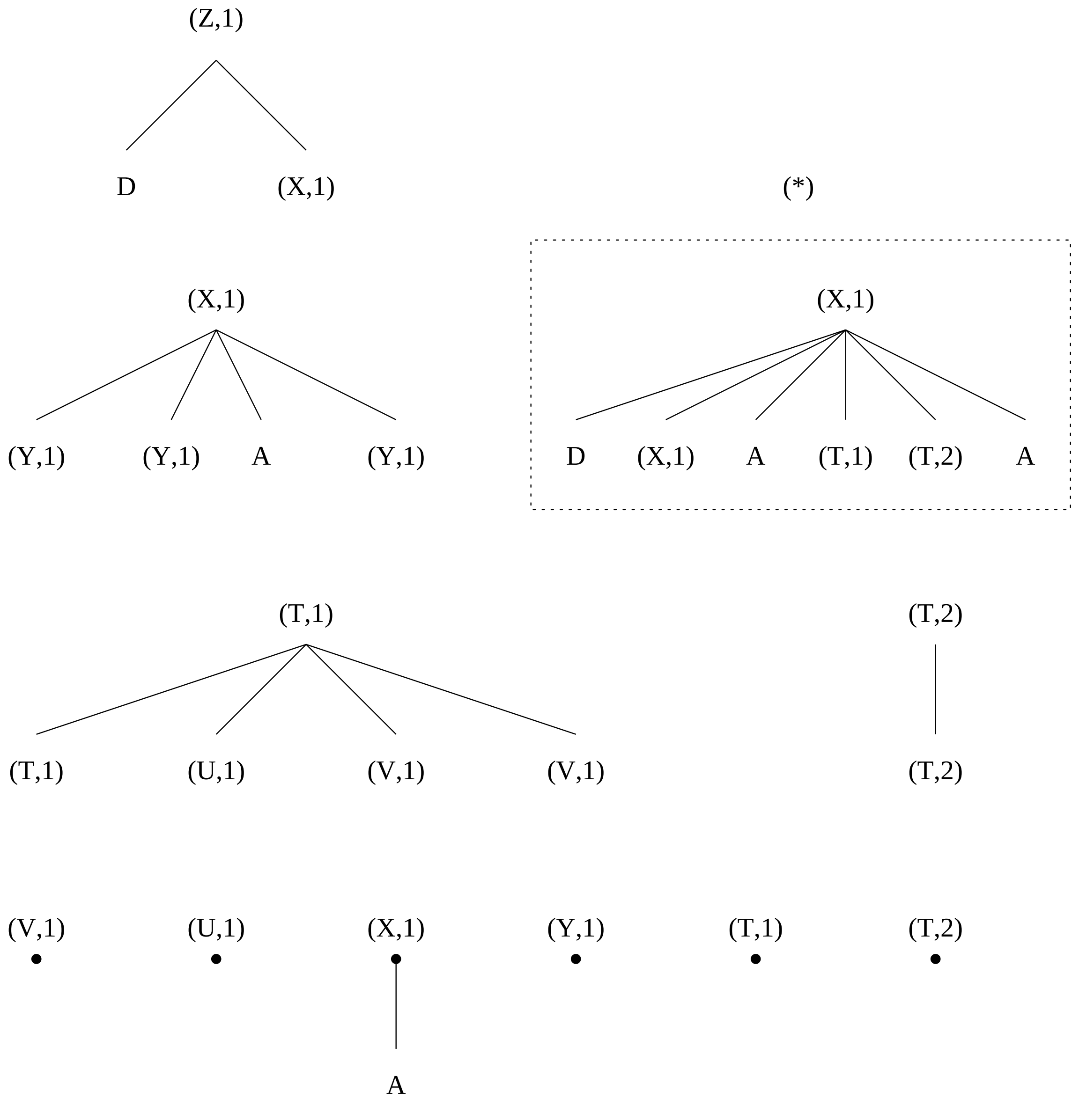


FIG. 5.2 - *Attribut en variable*



La production (*) introduit des phrases du type ...DAA... qui n'appartiennent pas a' (DA)*

FIG. 5.3 - *Attribut non en variable*

5.3 Implantation en pile globale

Un attribut dont les durées de vie ne sont pas deux à deux strictement disjointes ne peut pas être implanté en variable, sous peine d'introduire des erreurs dans l'évaluation. Pour ce type d'attribut, il convient donc de trouver une autre structure, compatible avec les durées de vie. En fait, nous savons qu'un attribut temporaire peut toujours être stocké en pile, ne serait-ce que dans la pile d'exécution. En effet, un attribut temporaire se comporte comme un paramètre pour la fonction qui effectue la visite sur l'arbre.

De façon théorique, il est toutefois intéressant de comprendre ce qui se passe quand un attribut ne peut pas être implanté en variable. Quand c'est le cas, c'est qu'il existe deux instances de cet attribut dont les durées de vie sont non disjointes ; cela signifie que l'une est incluse dans l'autre, ou bien qu'elles se chevauchent. S'il y a inclusion, alors nous obtenons tout de suite la gestion de la pile. En cas de chevauchement, pour voir les durées de vie calquées sur le fonctionnement d'une pile, il suffit de rallonger artificiellement certaines durées de vie.

5.4 Élimination des règles de copie

Les règles de copie, qui se traduisent par une redondance d'occupation mémoire et un ralentissement de l'exécution, sont l'une des causes de la non-efficacité des évaluateurs, d'autant qu'elles peuvent concerner des attributs complexes comme une table des symboles.

Il est parfois possible d'éliminer un certain nombre de règles de copie, notamment quand il s'agit de copie entre deux instances du même attribut. De plus cette suppression est d'autant plus simple qu'avec notre optimisation mémoire, deux instances d'un attribut partagent la même ressource mémoire.

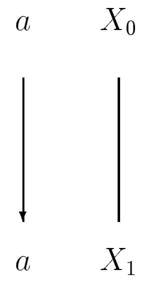
Pour comprendre l'élimination des règles de copie dans l'évaluateur, il est nécessaire de comprendre comment sont gérées les piles. Côté évaluateur, la gestion des piles requiert l'ajout de deux instructions spéciales, `PUSH` et `POP` pour indiquer à l'évaluateur d'empiler ou dépiler une valeur pour l'attribut considéré. `PUSH` empile une nouvelle valeur dont la durée de vie est incluse dans celle de la valeur actuelle en sommet de pile, `POP` dépile une valeur d'attribut lorsque la fin de la durée de vie est atteinte.

5.4.1 Les variables

Quand un attribut est implanté en variable, une règle de copie entre deux instances est systématiquement supprimée. Concrètement, cela revient à supprimer l'instruction `EVAL` de l'évaluateur dans les séquences de visite.

5.4.2 Les piles

Le problème des piles est un peu plus compliqué. Le cas le plus simple est donné par la figure 5.4. Dans cet exemple, nous pouvons non seulement éviter l'évaluation, mais aussi supprimer les instructions de gestion de pile introduites auparavant, puisqu'il est ridicule de dépiler une valeur pour la remettre immédiatement dans la pile.

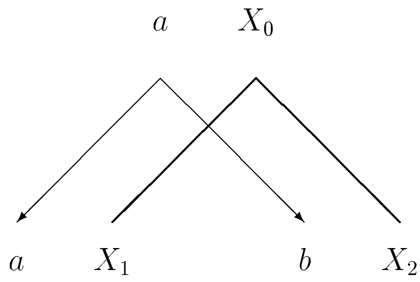


vs_p : `EVAL`($X_1.a$)
`POP`($X_0.a$)
`PUSH`($X_1.a$)
`VISIT`($X_1.1$)
...

FIG. 5.4 - *Suppression de règles de copie pour les piles, cas de durées de vie disjointes*

Proposition 5.4.1 *Toute règle de copie entre deux instances d'attribut stockées dans la même pile peut être éliminée si les durées de vie de ces instances sont strictement disjointes. Dans ce cas, les trois instructions* `EVAL`, `POP` *et* `PUSH` *peuvent être supprimées.*

Quand les durées de vie des deux instances ne sont pas disjointes, il est nécessaire de prendre des précautions.



vs_p : `EVAL`($X_1.a$)
`PUSH`($X_1.a$)
`VISIT`($X_1.1$)
`EVAL`($X_2.b$)
`POP`($X_0.a$)
...

FIG. 5.5 - *Suppression de règles de copie pour les piles, cas de durées de vie non-disjointes*

La figure 5.5 montre une telle situation. Dans un tel cas, il est possible de supprimer l'instruction `EVAL` correspondant à la règle de copie, mais il n'est pas possible de supprimer la duplication de l'information à l'intérieur de la pile.

5.4.3 Conclusion

Un grand nombre de règles de copie est supprimé, aussi bien pour les variables que pour les piles. Toutefois, pour le cas des piles, le gain n'est pas toujours significatif puisque nous remplaçons la règle de copie par une instruction de duplication de l'espace mémoire, ce qui revient à peu près au même.

5.5 Les attributs non temporaires

Par extension des grammaires de visite pour tenir compte des contextes supérieurs, [Jul89] permet la mise en variable globale de certains attributs non temporaires. Cette extension, bien que marginale par rapport au nombre des attributs touchés, est intéressante puisqu'elle permet de sortir très facilement ces non-temporaires de l'arbre d'entrée, ce qui demande en général une gestion assez poussée, comme avec des *binding trees* [VSK90].

5.6 Regroupement

À partir du résultat sur le stockage des attributs temporaires en pile ou en variable, nous pouvons effectuer une autre optimisation. Celle-ci consiste à regrouper plusieurs attributs dans la même structure. Par exemple, deux attributs $X.a$ et $Y.b$ étant chacun dans une variable globale, il est peut être possible d'utiliser la même variable pour ces deux attributs. Cette optimisation consiste à calculer un graphe d'incompatibilité entre les attributs et à faire de l'allocation de registre. Après cette optimisation, quelques règles de copies peuvent donc encore être supprimées.

5.7 Conclusion

L'optimisation séquentielle a pour fondement principal l'existence d'un ordre total sur les instructions de l'évaluateur. Seule la donnée de cet ordre permettra de calculer statiquement les durées de vie des attributs. Le résultat de l'optimiseur est le choix de structures adaptées pour le stockage de ces attributs. Le but est bien sûr de mettre un maximum d'attributs en variables globales, puisque c'est la structure qui est la plus économique en ce qui concerne l'encombrement mémoire. Mais pour certains attributs, une pile sera nécessaire.

En pratique, nous constatons que 60% des attributs temporaires sont stockés en variables et que 40% sont en piles. L'optimisation mémoire est donc très efficace en ce qui concerne l'encombrement nécessaire au stockage des attributs hors de l'arbre. En ce qui concerne les règles de copie, là encore, les résultats obtenus sont fort encourageants.

Il est, par conséquent, indispensable d'étendre cette théorie au cas de l'évaluation parallèle, qui est encore plus gourmande en espace mémoire que la version séquentielle (chaque processus a son propre lot de variables privées). Le chapitre suivant donne donc les moyens et conditions nécessaires à l'extension des théories séquentielles qui ont été brièvement présentées dans ce chapitre.

Chapitre 6

Optimisation mémoire : le cas parallèle

6.1 Présentation

L'optimisation mémoire présente un intérêt certain pour les grammaires attribuées. Tout d'abord, elle conduit à une consommation mémoire moins importante puisque certaines instances d'attributs partagent la même structure. Deuxièmement, elle permet l'élimination des règles de copie, qui représentent, comme nous l'avons vu dans le cas séquentiel, une grande partie des règles sémantiques d'une grammaire. Il était indispensable d'étendre l'optimisation séquentielle au parallélisme, afin de construire des compilateurs performants sur des architectures parallèles. Dans ce chapitre, nous allons donner une méthode permettant d'optimiser la consommation mémoire pour l'évaluation parallèle d'attributs. Cette méthode réutilise les différents concepts mis en œuvre pour le cas séquentiel. Elle se situe toujours dans le cadre d'une détermination statique.

Le chapitre 3 a donné une méthode de détection du parallélisme afin que le traducteur puisse utiliser tout le parallélisme possible sur une grammaire attribuée. Il n'a jamais été formellement question de la façon de distribuer le calcul associé aux tâches (tel que défini dans le chapitre 3), sur des tâches au sens opérationnel. La seule restriction que nous avons imposée avec le graphe de parallélisation et le modèle synchrone est que la communication ne se produit qu'entre des tâches ayant une relation fille/père, et seulement au moment de leur création et/ou de leur terminaison. En fait, cela signifie simplement que deux tâches parallèles au sens du chapitre 3 sont effectuées par deux tâches différentes au sens opérationnel. Maintenant, l'organisation des tâches entre elles peut être faite de plusieurs façons. Nous pouvons en effet décider qu'une tâche qui rencontre un `COBEGIN` peut lancer autant de filles qu'elle a de branches dans ce `COBEGIN`. Une autre solution consiste à faire travailler le père sur une des branches, tandis qu'une tâche est lancée sur chacune des autres branches. Nous pouvons également envisager qu'une des tâches filles continue le travail du père plutôt que l'inverse.

Choisir une méthode de distribution sur les tâches correspond à un partitionnement du graphe de parallélisation. À l'intérieur d'une tâche, le travail est effectué séquentiellement.

Nous allons voir que la méthode d'optimisation parallèle repose sur les mêmes bases de partitionnement du graphe de parallélisation, ce qui va nous amener à choisir le même partitionnement pour les deux problèmes. L'idée est la suivante. Les méthodes d'optimisation séquentielle travaillent sur un ordre total d'évaluation afin de déterminer l'interaction des attributs entre eux. Si l'ordre total change, l'optimisation séquentielle est modifiée. Or pour une famille d'ordres l-ordonnés sur les non-terminaux de la grammaire, l'ordre résultant sur les graphes de dépendance de chaque production n'est pas total. Lors de la création des séquences de visite, c'est le générateur d'évaluateur séquentiel qui choisit un ordre de façon plus ou moins arbitraire, cet ordre est déterminant pour l'optimiseur dans son calcul des durées de vie, puisque celui-ci travaille justement avec les séquences de visite. La question de l'influence du choix de l'ordre total sur le résultat de l'optimisation (plus de variables, plus de piles, élimination de règles de copie supplémentaires. . .) est encore une question ouverte. En mode parallèle, le générateur d'évaluateur ne fait pas ce choix d'un ordre total. En effet, quand il y a un choix à faire, il correspond à une opportunité de parallélisation. L'optimisation séquentielle ne peut donc pas s'appliquer telle quelle, car nous ne générons pas de séquences de visite séquentielles. Puisque les techniques d'optimisation séquentielles ont besoin d'un ordre total, nous avons pensé à éclater l'évaluateur parallèle en plusieurs sous-évaluateurs purement séquentiels (travaillant sur des ordres totaux) qui communiqueraient à certains niveaux mais qui seraient indépendants en ce qui concerne l'optimisation mémoire.

Notre méthode consiste tout d'abord à caractériser les régions du graphe de parallélisation où nous allons décomposer l'évaluateur parallèle. Pour des raisons pratiques, nous imposerons également ce découpage pour les tâches parallèles. De ce fait, nous aurons équivalence entre évaluateurs et tâches parallèles (au sens opérationnel).

Le type de partitionnement choisi est celui déjà cité plus haut, c'est-à-dire qu'une tâche mère va prendre en charge l'évaluation des attributs d'une des branches parallèles. C'est un avantage au niveau de l'optimiseur car cette démarche augmente la taille des sous-évaluateurs. De cette façon, nous diminuons le nombre de sous-évaluateurs, et d'autre part, l'optimisation s'effectue sur un nombre d'attributs qui n'est pas réduit au minimum. En effet, le nombre d'évaluateurs n'est pas une donnée négligeable puisqu'il n'y a, dans notre modèle, aucune optimisation prévue entre deux sous-évaluateurs. Augmenter le nombre d'évaluateurs peut forcer un attribut à occuper une cellule mémoire pour chaque sous-évaluateur, alors qu'il n'en aurait occupé qu'une seule s'il n'y avait eu qu'un seul sous-évaluateur. De la même façon que la taille des sous-évaluateurs influence le résultat de l'optimisation, le choix de la branche parallèle effectuée par la tâche mère n'est pas sans conséquence sur le résultat.

L'optimiseur mémoire séquentiel n'a besoin que d'une chose pour effectuer son travail : connaître l'évaluateur, c'est-à-dire l'ensemble des séquences de visite. À partir de là, il est capable de calculer les durées de vie, ainsi que les grammaires de visite.

Les définitions suivantes vont nous permettre d'expliciter les sous-évaluateurs et ce sur quoi ils travaillent. D'autre part, nous donnerons une méthode pour calculer les grammaires de visite pour chacun des sous-évaluateurs.

6.2 Définitions

Soit $G = (V, \mathcal{O})$ un *graphe de parallélisation.*

Définition 6.2.1 (Sous-séquence de visite) *Soit $G = (V, \mathcal{O})$ un graphe de parallélisation.*

$G' = (V', \mathcal{O}')$ est une sous-séquence de visite si et seulement si :

- *G' est un sous-graphe,*
- *$\mathcal{O}'$ définit sur V' un ordre total.*

La figure 6.1 donne des exemples de sous-séquences de visite.

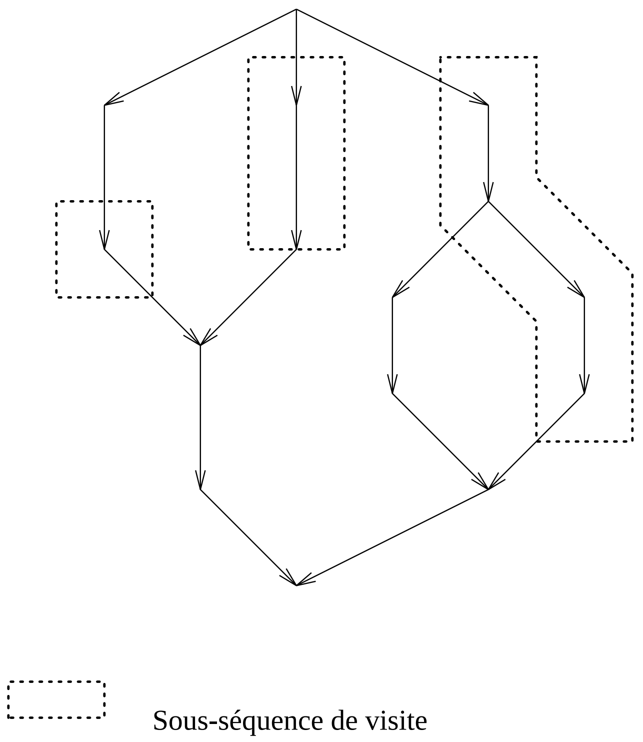


FIG. 6.1 - *Sous-séquences de visites*

La sous-séquence de visite représente une partie du graphe qui est purement séquentielle au regard de l'évaluation, c'est-à-dire qui est totalement ordonnée. Mais cette notion ne suffit pas, car nous voyons qu'une partie réduite à un seul nœud entre dans le cadre de la définition. Notre but n'étant pas de partitionner nœud par nœud (ce qui reviendrait à une parallélisation à la Fang), il faut introduire une notion qui va permetre de prendre une partie la plus "grande" possible. La définition suivante caractérise cette idée de maximalité.

Définition 6.2.2 (Sous-séquence de visite maximale) *Nous dirons que $G' = (V', \mathcal{O}')$ est une sous-séquence de visite maximale par rapport à $G = (V, \mathcal{O})$ si et seulement si elle est maximale au sens de l'inclusion.*

Lemme 6.2.1 *Le nœud minimum et le nœud maximum d'un graphe de parallélisation appartient à toutes les séquences de visite maximales par rapport à ce graphe.*

La figure 6.2 donne des exemples (parmi d'autres) de sous-séquences de visite maximales pour un graphe donné. Le choix d'une sous-séquence de visite n'est pas unique.

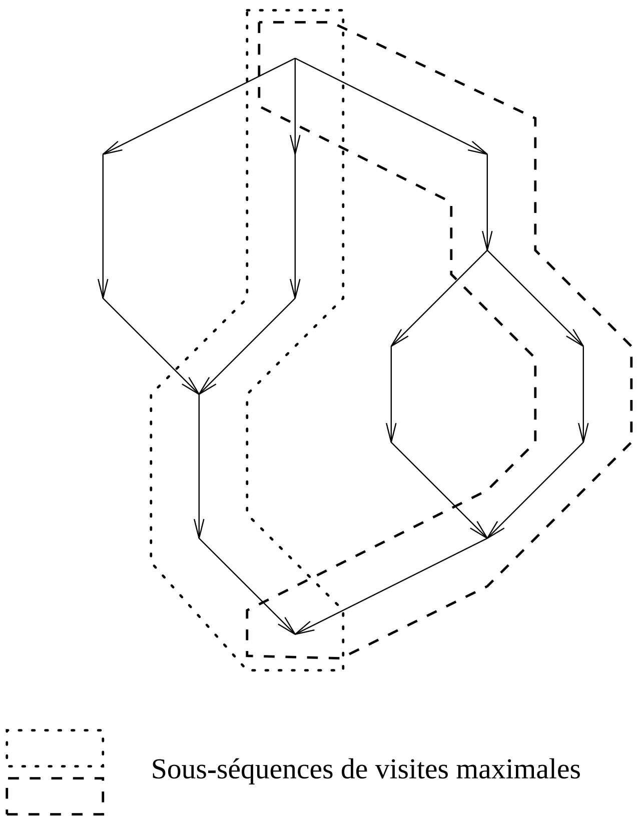


FIG. 6.2 - *Exemples de sous-séquences de visite maximales*

Avec une sous-séquence de visite maximale G', nous disposons d'une partie totalement ordonnée du graphe. Cette partie correspond à un sous-évaluateur pour cette production. Mais tous les nœuds du graphe G ne sont pas dans G', or nous voulons absolument que tous les nœuds de G soient pris en compte. Pour cela, nous introduisons la notion de partitionnement parallèle du graphe.

Définition 6.2.3 (Partitionnement parallèle) *On appelle partitionnement parallèle de* $(V, \mathcal{O})$ *une suite finie* $P = \{(V_i, \mathcal{O}_i)\}_{i \geq 0}$ *telle que :*

- $\bigcup_{i \geq 0} V_i = V$;
- $\forall i \neq j \; V_i \cap V_j = \emptyset$;

– $(V_0, \mathcal{O}_0)$ *est une sous-séquence de visite maximale par rapport à* $(V, \mathcal{O})$ *;*

– $(V_i, \mathcal{O}_i)$ *est une sous-séquence de visite maximale par rapport à* $(V - \bigcup_{i>j\geq 0} V_j, \mathcal{O}/V - \bigcup_{i>j\geq 0} V_j)$.

La figure 6.3 donne des exemples de partitionnements parallèles.

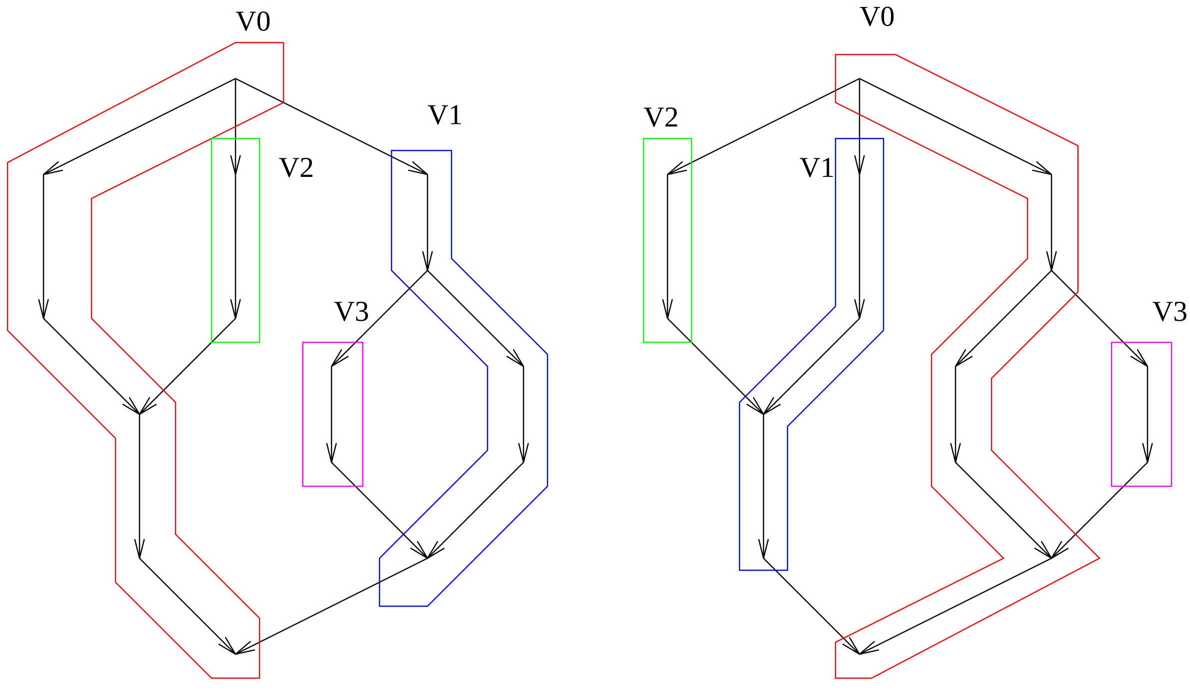


FIG. 6.3 - *Exemples de partitionnements parallèle*

Un partitionnement parallèle est un ensemble de chemin. Sur chacun de ces chemins, nous montrerons que les techniques d'optimisations mémoire séquentielle s'appliquent.

Lemme 6.2.2 V_0 *contient le min et le max de* V

Remarques :

– pour un $(V, \mathcal{O})$ donné, le partitionnement n'est pas unique ;

– le partitionnement influence l'efficacité de l'optimisation.

– le partitionnement n'influence pas l'efficacité de la parallélisation ;

Ces remarques nécessitent quelques explications : la non-unicité du partitionnement découle du choix d'un plus long chemin, effectué à chaque étape, et de sa non-unicité. Elle est illustrée par la figure 6.3 qui montre deux partitionnements possible d'un même graphe.

L'influence sur l'optimisation est due au fait que l'ordre total global sur chaque sous-évaluateur va changer, c'est-à-dire que les séquences de visite associées à chaque sous-évaluateur sont différentes selon le partitionnement. De ce fait, l'optimiseur n'a plus le même comportement, et les résultats sont par conséquent différents.

En ce qui concerne le comportement vis-à-vis de la parallélisation, le modèle synchrone nous assure que le chemin critique dans le graphe de parallélisation (chemin de plus longue durée) reste toujours le même, quel que soit le partitionnement. La seule modification qui puisse altérer l'efficacité, par rapport au partitionnement, se situe au niveau de l'implantation. C'est, par exemple, le coût du passage des paramètres aux sous-évaluateurs. Dans ce cas, il est souhaitable que le père prenne comme tâche fille à exécuter lui-même, celle nécessitant le plus de paramètres.

Ainsi le choix du partitionnement en ce qui concerne la parallélisation est surtout une question d'implantation. Pourtant, au regard de l'optimisation mémoire, on ne peut pas décider, lors de la construction, qu'un partitionnement est plus intéressant qu'un autre. En effet, la méthode d'optimisation mémoire possède une vision globale de l'évaluateur ; changer seulement une séquence de visite modifie le résultat de l'optimisation. Par conséquent, il ne sert à rien de choisir un partitionnement pour une production selon un critère précis alors que tous les autres partitionnements des autres productions ne sont pas calculés.

6.3 Les séquences de visites

À partir des partitionnements des graphes de parallélisation, il est maintenant possible de créer un ensemble de séquences de visite correspondant à chaque élément de la partition de chaque graphe de parallélisation. L'algorithme de la figure 6.4 nous montre comment sont générées toutes les séquences de visite des évaluateurs.

Les informations permettant d'identifier les séquences de visite sont :

- l'identificateur de la production visitée ;
- le numéro de visite.

Avec le graphe de dépendances, nous disposons de ces deux informations. Le graphe est associé à une production, et les ordres l-ordonnés nous indiquent le numéro de visite de chaque attribut.

6.3.1 Algorithme de construction des séquences de visite

Soit $p \in P$ une production de la grammaire, nous notons G_p le graphe de parallélisation associé et $\mathcal{P}_p$ le partitionnement parallèle de G_p.

Nous supposons l'existence d'une fonction $id(T)$ qui associe un identificateur unique à chaque élément de chaque graphe de parallélisation.

La fonction *type_sh(a)=Synthétisé* (resp. hérité) suivant que l'attribut a est synthétisé (resp. hérité).

Le rôle de cet algorithme est de préparer les données d'entrées pour l'optimiseur mémoire, en conséquence, il ne tient pas compte des nœuds parallèles.

Les séquences de visite obtenues forment l'union des séquences de visite de tous les sous-évaluateurs. Pour obtenir les séquences qui définissent un sous-évaluateur, il faut prendre une des séquences de visite "racine", c'est-à-dire les séquences obtenues par le Begin_visite(i,id(T)). Ensuite on prend toutes celles atteignables à partir de celle-ci. L'évaluateur principal (et initial) commence à la séquence de visite $(Z, 0)$ et contient également toutes les séquences de visite atteignables à partir de $(Z, 0)$.

La définition de ces évaluateurs par leur séquence de visite nous permet ainsi d'appliquer directement l'optimiseur mémoire séquentiel.

Dans la section suivante, nous donnons une définition des grammaires de visite associées à chaque évaluateur, ceci indépendamment de l'optimiseur séquentiel. Cette définition peut être utilisée pour la construction directe des grammaires de visite, ce qui évite un travail redondant au niveau de l'optimiseur.

6.4 Grammaires de visite

Soient

- $p \in P,\ p : X_0 \leftarrow X_1, \ldots, X_{n_p}$
- $v_{sp} = v_{sp_1}, \ldots, v_{sp_{\phi_v(X_o)}}$ la famille de graphes de visite associés à P
- $\{\mathcal{P}_{p,j}\}_{j\in\{1,\ldots,\phi_v(X_0)\}}$ un partitionnement parallèle pour $v_{sp,j}$.

On note $\mathcal{P}_{p,j} = (\mathcal{V}_j^0, \ldots, \mathcal{V}_j^m)_p$.
Soit $N_{GV} = \{(X, i)\}_{X\in N, i\in\{1,\ldots,\phi_v(X)\}}$
et $N_{//} = \{(Z_{p,v,n}, 1)\}_{p\in P, v\in\{1,\ldots,\phi_v(X_0)\}, n\in\{1,\ldots,card(\mathcal{P}_{p,j})\}}$.
À chaque $p \in P$, à chaque $l \in [1, \phi_v(X_0)]$, on associe

1. $p(l, 0) : (X_0, l) \rightarrow (X_{i_1}, l_1), \ldots, (X_{i_{n'}}, l_{n'})$ avec
   - $\forall j \in \{1, \ldots, n'\} : \exists I \in \mathcal{V}_l^0 | I = \texttt{VISIT}(X_{i_j}, l)$
   - les (X_{i_j}, l_j) sont ordonnés suivant l'ordre des $\texttt{VISIT}(X_{i_j}, l_j)$ sur $\mathcal{V}_l^0$
2. $p(l, i) : (Z_{p,l,i}, 1) \leftarrow (X_{i_1}, l_1), \ldots, (X_{i'_n}, l_{n'})$ avec
   - $\forall j \in \{1, \ldots, n'\} : \exists I \in \mathcal{V}_l^i | I = \texttt{VISIT}(X_{i_j}, l)$
   - les (X_{i_j}, l_j) sont ordonnés suivant l'ordre des $\texttt{VISIT}(X_{i_j}, l_j)$ sur $\mathcal{V}_l^i$

On appelle grammaires de visite parallèles les sous-grammaires de $(N_{GV}, Z, p_i(l_j, k))$ ayant comme axiome $(Z_{p,l,i}, 1)$.

```
int num_visit = 1;
pour p ∈ P
    pour T ∈ 𝒫_p
        si T est la tache principale alors
            Begin_visite(p,num_visit,0)
        sinon
            Begin_visite(p,num_visit,id(T))
        fin si
        pour a ∈ T          /* a est un attribut */
            position = pos(a,p);
            si position=0 alors
                si type_sh(a) = synthétisé alors
                    EVAL(a)
                sinon /* attribut hérité : Ce cas ne se rencontre que pour la tâche principale */
                    si num_visite(a) > num_visit alors
                        LEAVE(num_visit)
                        num_visit = num_visite(a)
                        Begin_visite(num_visit,0)
                    fin si
                fin si
            sinon /* position > 0 */
                si type_sh(a) = synthétisé alors VISIT(pos(a))
                sinon /* attribut hérité */
                    EVAL(a)
                fin si
            fin si
        fin pour
        LEAVE(num_visit)
    fin pour
fin pour
```

FIG. 6.4 - *Algorithme de construction des séquences de visite*

Remarques :

1. L'évaluateur restreint à une grammaire de visite parallèle est par construction séquentiel.
2. (Corollaire à 1) Les techniques d'optimisation séquentielles s'appliquent.
3. L'optimiseur parallèle fonctionne pour le cas séquentiel. Le cas séquentiel peut être vu comme le cas particulier où l'on choisit un partitionnement singleton.

6.5 Résultats théoriques

En appliquant les techniques d'optimisation séquentielle sur chacun des évaluateurs, nous obtenons des informations pour chaque attribut. Ces informations nous indiquent qu'un attribut $X.a$ est stocké dans une pile (ou dans une variable), et ainsi que la pile (ou variable) dans laquelle nous devons le placer.

Définition 6.5.1 (Schéma d'allocation) *Nous appellerons* **schéma d'allocation** *d'un évaluateur les informations obtenues grâce à l'optimiseur sur cet évaluateur. Le schéma d'allocation indique comment sont stockés les attributs.*

Définition 6.5.2 (Modèle d'allocation) *L'espace mémoire correspondant à un schéma est appelé modèle d'allocation.*

Toutes les instances parallèles d'un même évaluateur travaillent avec le même schéma. Il est par conséquent aisé de générer le code de chaque évaluateur. Chaque évaluateur possède son propre code, correspondant à chaque séquence de visite. Cette façon de générer le code augmente la taille de l'évaluateur puisque les séquences de visite qui appartiennent à plusieurs évaluateurs sont dupliquées autant de fois que nécessaire. Le problème le plus important à régler reste la communication entre les sous-évaluateurs, c'est-à-dire régler le parallélisme.

6.5.1 La communication entre évaluateurs

Dans un graphe de parallélisation, les différents sous-évaluateurs correspondent aux partitions de la définition 6.2.3. La communication se fait entre deux partitions ayant un lien de dépendance entre elles. Ces liens se font, par définition, toujours au niveau des instructions `COBEGIN/COEND`.

`COBEGIN` Au moment de la génération de code, la traduction d'une instruction `COBEGIN`, consiste tout d'abord à identifier les sous-évaluateurs associés à chaque tâche parallèle.

Pour chaque tâche, le code produit alloue un espace mémoire correspondant au schéma d'allocation associé au sous-évaluateur. Nous appellerons cet espace mémoire l'**instance du modèle d'allocation**. Chaque schéma d'allocation est propre à la tâche. Il n'y a pas de partage de schéma d'allocation. Cela signifie que pour une traduction tâches/processus, le schéma d'allocation est une variable privée au processus qui l'utilise. Un attribut en variable aura par conséquent une cellule mémoire dans chaque instance de shéma. L'étape suivante consiste à copier les attributs de la tâche courante dans l'instance du modèle associé à la tâche. Une tâche est donc constituée d'un évaluateur, d'un nœud de départ et d'une instance de modèle d'allocation. Le sous-évaluateur stockera toute les instances d'attributs (temporaires) dans l'instance de modèle d'allocation qui lui est associée.

`COEND` Lors d'une instruction `COEND`, la tâche principale dispose de toutes les instances de modèles de tâches qui viennent de se terminer. Elle peut donc recopier les valeurs d'attributs résultats des sous-tâches, dans sa propre instance modèle d'allocation.

Remarque : Cette implantation fonctionne même dans le cas de l'automate où la tâche qui lance les sous-tâches n'est pas forcément la même que celle qui effectue le *join*. Dans ce cas, il faut tout de même effectuer un changement de contexte au niveau du `COEND`. C'est-à-dire que la sous-tâche qui doit terminer la tâche principale va devoir se comporter comme si elle était effectivement la tâche principale. Elle va ainsi changer d'évaluateur et d'instance de modèle d'allocation.

6.5.2 Résultat

Une question intéressante est de savoir si l'évaluation parallèle donne de "meilleurs" résultats que l'optimiseur séquentiel. Par meilleurs, nous entendons :

1. Le pourcentage d'attributs non temporaires baisse-t-il ?
2. Le nombre de variables globales augmente-t-il ?

6.5.3 Les attributs non temporaires

Un attribut $X.a$ est dit non temporaire quand il est défini lors d'une visite i et qu'il est encore utilisé dans une visite $j > i$. Pour expliciter la notion de non temporaire, il faut distinguer le cas hérité du cas synthétisé.

1. **Instance d'attribut hérité non temporaire :** Soit a un attribut de $HER(X_0)$ tel que $\exists p \in P\,; p : X_0 \rightarrow X_1 \ldots X_{n_p}$.

   Soit α une instance de l'attribut a, définie **avant** la $k^{ème}$ visite d'un nœud x_0 tel que $label(x_0) = X_0$ et $prod(x_0) = p$.

   L'instance α est non temporaire si et seulement si elle est référencée lors d'une $j^{ème}$ visite de x_0 tel que $j > k$.

2. **Instance d'attribut synthétisé non temporaire :** Soit a un attribut de $SYN(X_i)$ tel que $\exists p \in P\,; p : X_0 \rightarrow X_1 \ldots X_i \ldots X_{n_p}$.

```
Variable globale au processus
type Modele : modele_courant ;

/* visite du non-terminal X sur la production p 1ere visite */
visite_X_p_1(nœud)
    type Nœud nœud ;
    type Tâche tache[n] ;
{
    type Modele tache_1_modele,. . . ,tache_n_modele ;
    ⋮
    /* lancement de la première tâche */
    tache_1_modele = allocation_modele(Cte_schema_de_tache_1) ;
    recopie_attribut(tache_1_modele,modele_courant) ;
    tache[1]=lance_tache(
        visite_para_p_cte_id(tache_1),
        fils(cte, nœud),
        tache_1_modele
        )
    ⋮
    /* lancement de la dernière tâche */
    tache_n_modele = allocation_modele(Cte_schema_de_tache_n) ;
    recopie_attribut(tache_n_modele,modele_courant) ;
    tache[n]=lance_tache(
        visite_para_p_cte_id(tache_n),
        fils(cte, nœud),
        tache_n_modele
        )
    /* code pour la tache du pere */
    ⋮
    /* Code pour le COEND */
    pour i de 1 à n
        attendre(tache[i])
    fin pour
    recopie_resultat(modele_courrant,tache_1_modele) ;
    ⋮
    recopie_resultat(modele_courrant,tache_n_modele) ;
    /* suite de la visite */
    ⋮
}
```

FIG. 6.5 - *Traduction d'une séquence de visite parallèle : utilisation des modèles*

Soit α une instance de l'attribut a, définie **pendant** la $k^{ème}$ visite d'un nœud x_0 tel que $label(x_0) = X_0$ et $prod(x_0) = p$.

L'instance α est non temporaire si et seulement si elle est référencée lors d'une $j^{ème}$ visite de x_0 tel que $j > k$.

Les attributs non temporaires détectés dans le cas séquentiel restent non temporaires dans la plupart des évaluateurs parallèles. Toutefois, certains évaluateurs peuvent considérer un attribut comme temporaire alors que d'autres le considèrent comme non temporaire. Nous allons étudier ici la condition nécessaire sur un attribut séquentiellement non temporaire pour qu'il soit traité comme temporaire dans un évaluateur parallèle.

Si un sous-évaluateur considère qu'un attribut est temporaire, il faut étudier les communications avec d'autres sous-évaluateurs. Si un autre sous-évaluateur a besoin de la valeur de l'attribut lors d'une visite ultérieure à sa définition, alors cet attribut sera en fait non temporaire.

Nous remarquons que nous avons cassé la grammaire et par conséquent la temporarité, et maintenant, afin de contrôler cette temporarité, nous sommes obligés de faire en quelque sorte machine arrière. Mais cette démarche est naturelle et provient du fait que la non temporarité d'un attribut est une notion séquentielle locale à une ou plusieurs productions. Le cas séquentiel globalise l'information à toute la grammaire, alors qu'en parallèle, nous essayons de garder cette information, et de la faire remonter seulement aux sous-évaluateurs concernés.

Nous pouvons déjà formuler une première remarque : le comportement des attributs hérités non temporaires dans le cas séquentiel ne va pas changer. Dès qu'un tel attribut est évalué par un sous-évaluateur, la ou les productions qui provoquent sa non temporarité sont toutes dans l'évaluateur en question. Ainsi, soit nous aurons une utilisation de l'attribut lors d'une visite ultérieure dans le même sous-évaluateur, soit il y aura une communication avec un autre sous-évaluateur. Ceci est dû au fait que toutes les productions liées à un non-terminal sont traitées par un évaluateur dès le moment où l'une d'entre elles l'est.

Pour les synthétisés, le schéma est différent. En effet, la (ou les) production responsable de la non temporarité d'un attribut synthétisé n'est pas forcément présente dans tous les évaluateurs qui ont en charge le non-terminal portant l'attribut.

C'est donc parmi ceux-ci que nous allons pouvoir trouver des attributs temporaires, localement à un sous-évaluateur.

Soit a un attibut séquentiellement non temporaire synthétisé d'un non-terminal X.

Soit $p : X_0 \leftarrow X_1 \ldots X \ldots X_n$ une production responsable de la non temporarité de $X.b$

Exemples : soit la production p de la figure 6.6 :

Si $X \not\Rightarrow Z$ alors toute grammaire contenant X et pas Z pourra considérer $X.b$ comme temporaire alors que la production p le rend non temporaire dans toute grammaire qui contient Z.

Proposition 6.5.1 *Soit $X.a \in A$ un attribut synthétisé séquentiellement non temporaire de la grammaire G.*

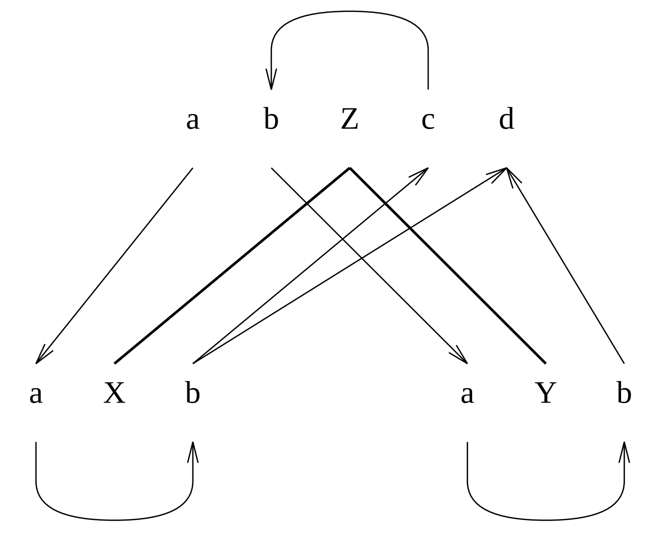


FIG. 6.6 - *Exemple de production avec attribut non temporaire*

Soit $\{p_i\}_{i \in N_{X.a}} \subset P$ *l'ensemble des productions qui trouve* $X.a$ *non temporaire dans leur graphe de dépendance. On note* $p_i : X_i \rightarrow Y_1 \ldots Y_{n_{p_i}}$.

Soit $G_{//} = ((N_{G_{//}}, T_{G_{//}}, P_{G_{//}}, Z_{G_{//}}), A_{G_{//}}, F_{G_{//}})$ *une sous-grammaire construite par l'optimiseur parallèle.*

Si

- $X.a \in N_{G_{//}}$
- $\forall i \in N_{X.a},\ X_i \notin N_{G_{//}}$
- $\forall i \in N_{X.a},\ X \not\xrightarrow{*} X_i$

alors $X.a$ *est temporaire dans la sous grammaire* $G_{//}$

6.6 Conclusion

Dans ce chapitre, nous avons exposé une méthode pour exploiter les résultats de l'optimisation mémoire séquentielle dans le cas parallèle. Pour cela nous avons décomposé l'évaluateur parallèle en plusieurs sous-évaluateurs séquentiels. À partir de là, les résultats de l'optimiseur séquentiel nous permettent déjà de nous préoccuper des attributs temporaires et de les stocker dans des structures identiques à celles rencontrées en séquentiel, c'est-à-dire des piles ou des variables. Chaque sous-évaluateur a son propre schéma d'allocation et chaque instance du sous-évaluateur possède sa propre instance de schéma pendant l'évaluation d'un arbre.

Nous avons ainsi gagné un grand nombre de cellules mémoire puisque, au niveau des temporaires, l'espace mémoire est limité à la taille du plus gros schéma fois le nombre de tâches actives. Cet espace mémoire est évidemment beaucoup plus important que celui consommé en séquentiel, vu que le nombre de tâches actives à un instant donné peut être très important. Toutefois, il est également évident que le partage de structure entre attributs diminue fortement la consommation mémoire par rapport à un évaluateur parallèle non optimisé.

Pour un attribut, la transformation de la structure de pile (pour le séquentiel) en une structure de variable (pour le parallèle), qui se produit relativement souvent, à défaut d'être un gain mémoire (chaque étage de la pile se retrouve dans une variable de plusieurs sous-évaluateurs), permet tout au moins une gestion moins coûteuse. Il en va de même pour les attributs séquentiellement non temporaires qui sont, pour certains sous-évaluateurs, vus comme temporaires.

L'implantation de cet optimiseur n'a pas été effectuée, mais comme nous récupérons tous les résultats séquentiels, nous sommes persuadé de l'efficacité d'un tel outil.

Conclusion

Les travaux présentés dans cette thèse portent sur les grammaires attribuées et le parallélisme. Notre but était de valider l'utilisation des grammaires attribuées comme mécanisme de spécification de problèmes qui se prètent à une résolution parallèle. Nous savions d'ores et déjà, grâce aux nombreux travaux antérieurs effectués dans ce domaine, que cela était théoriquement possible. Malheureusement, peu d'implantations réelles avaient été réalisées et les résultats n'étaient pas toujours très encourageants. Enfin, il est aisé de construire des algorithmes et des applications très adaptées à un type de machines parallèles particulier mais dont les performances s'effondrent au passage sur une autre machine.

Pour parvenir à notre objectif, nous nous sommes déjà fixé la classe de grammaire que nous souhaitions accepter. Notre choix s'est porté sur la classe des grammaires l-ordonnées. D'une part cette classe permet de calculer statiquement un ordre d'évaluation, et d'autre part, la transformation FNC vers l-ordonnée nous permet en fait d'accéder à l'ensemble des grammaires de la classe FNC. De plus, l'ordre total sur chaque non-terminal de la grammaire assure qu'il n'y a jamais conflit d'accès sur un nœud de l'arbre entre deux processus.

Notre méthode consiste à détecter le parallélisme sur la grammaire, puis de sélectionner statiquement, par certaines heuristiques, le parallélisme que nous souhaitons effectivement garder. La parallélisation se fait sous la forme *fork/join* afin de limiter au mieux les communications entre processus.

L'utilisation de machines parallèles à mémoire partagée nous a permis d'éviter les problèmes de distribution de l'arbre d'entrée sur les différents processeurs. La construction de l'arbre à travers un réseau imposant une optimisation de la distribution des nœuds pour limiter les communications entre les processus.

Au regard des résultats, nous pouvons être satisfait de nos implantations puisque nous exécutons certains programmes presque 5 fois plus vite qu'en séquentiel.

Nos expériences ont montré qu'il était nécessaire de modifier l'allocateur de mémoire dynamique pour assurer un minimum d'efficacité. L'utilisation de *fork* Unix standard est à proscrire, mais il n'est pas intéressant non plus d'utiliser un processus léger (*thread*) par tâche. L'ordonnancement est donc crucial pour l'efficacité de l'évaluation, et doit être le moins coûteux possible. De toute façon, le manque d'information sur la durée des tâches ne permet pas de construire un *scheduler* intelligent. Entre les versions procédurales et interprétées, le choix est encore difficile. Le temps d'exécution est en faveur de la première version, mais une implantation plus optimisée de la version interprétée permettrait peut-être de mettre ces deux versions sur un même plan.

Au niveau des machines parallèles, notre approche a été validée puisque toutes donnent des résultats globalement similaires. Nos algorithmes sont par conséquent facilement implantable sur toute machine parallèle à mémoire partagée.

Le dernier aspect important de notre travail a consisté à montrer qu'il était possible d'étendre l'optimisation mémoire séquentielle au cas parallèle. Ainsi, nous parvenons à optimiser le code parallèle, tant au niveau de la place mémoire nécessaire à l'exécution qu'au niveau du temps d'exécution, puisque nous supprimons un certain nombre d'évaluations.

Tous ces résultats tentent à prouver que les grammaires sont, comme nous le pensions au départ, particulièrement bien adaptées à la spécification de programme pour le parallélisme. L'utilisateur spécifie seulement un schéma de calcul et l'évaluateur se charge de paralléliser l'évaluation sur l'arbre. Couplé avec l'optimisation mémoire, nous déchargeons l'utilisateur de tâches relativement pénibles : l'optimisation et la parallélisation.

Quant aux prolongements de notre travail, le premier point est sans doute l'implantation réelle de l'optimiseur mémoire parallèle. Il serait effectivement très intéressant d'avoir des résultats pratiques de la théorie que nous avons exhibée.

Une sujet de recherche qui nous tient particulièrement à cœur consiste à étudier le lien entre la méta-composition (et la méta-union) et le parallélisme. En effet, lors de la méta-composition (et de la méta-union) de deux grammaires, le parallélisme se trouvant dans chacune des grammaires est fusionné au point peut-être de disparaître de la grammaire (l-ordonnée) résultante. Il serait par conséquent très intéressant d'exploiter des informations sur le parallèlisme, calculées avant la méta-composition, pour paralléliser des calculs d'attributs d'une même visite, avec toutefois une connaissance plus approfondie qu'avec des grammaires FNC quelconques.

Annexe A

Code de l'interface parallèle

Cette annexe contient le fichier d'entête qui s'occupe d'interfacer le reste du code avec les différentes machines parallèles. Quatre machines sont actuellement décrites : Le Balance 8000 de Sequent, le multimax de Encore Computer et la Ksr1 de Kendall pour les architechture parallèle ; le Sun 4 pour les machines séquentielles.

On remarquera que les définitions des macros sont les plus simple possibles et que le code n'a pas été optimisé en fonction des différentes spécificités des architechtures. La présence de telle macros étaient nécessaire puisque les différents systèmes ont des incompatibilités au niveau des noms des appels systèmes (ou bibliothèque) et de leur prototype.

```
#ifndef __INTERFACE_PARA__
#define __INTERFACE_PARA__

/* Suivant la machine parallele on definit les macros suivantes */
/* actuellement definies pour  */
/*  mmax  = MACH =multimax de encore */
/* ksr = KSR  = ksr de Kandall  */
/*  sequent = balance 8000 de sequent computer */
/*  sequentiel pour sun4 */

#ifdef PARA

/* ============================================================ */
/* definition pour le sequent : pas de conditions variables  */
/* ============================================================ */
#ifdef SEQUENT
```

```
#include <parallel/parallel.h>

#define BM_lock(l)      S_LOCK(l)
#define BM_try_lock(l)   S_CLOCK(l)
#define BM_unlock(l)    S_UNLOCK(l)
#define BM_init_lock(l)  S_INIT_LOCK(l)
typedef slock_t     BM_lock_t;

#define BM_para_init()
#define BM_my_number()   getpid()
#define BM_fork(t,f,p)   if ((t=fork())==0) { f(p);exit(0);}
#define BM_join(p)       fprintf(stderr,
                             "pas de join sur sequent\n"),exit(-1)
#define BM_exit(a)       exit(a)
#define BM_detach(p)
#define BM_fork_and_detach(f,p) if (fork==0) { f(p);exit(0);}
#define BM_thread int
#define BMkill()           kill_para()
#define BM_set_master()         numero_du_pere = 0;
#define BM_is_master()   (SeqNumero == 0)
#define BM_private  private
#define set_mon_numero(num) SeqNumero = num
extern  BM_thread numero_du_pere;
extern  BM_private int SeqNumero;
#endif sequent


/* ============================================================== */
/* definition pour le multimax */
/* ============================================================== */

#ifdef MACH

#include <cthreads.h>

#define BM_lock(l)   mutex_lock(l)
#define BM_try_lock(l)  mutex_try_lock(l)
#define BM_unlock(l)  mutex_unlock(l)
#define BM_init_lock(l) mutex_init(l)
typedef struct mutex BM_lock_t;

#define BM_para_init() conti_init_para()
```

```
#define BM_my_number() cthread_self()
#define BM_fork(t,f,p) t = cthread_fork(f, p)
#define BM_join(p) cthread_join(p)
#define BM_exit(a) cthread_exit()
#define BM_detach(p) cthread_detach(p)
#define BM_fork_and_detach(f,p) cthread_detach(cthread_fork(f,p))

typedef cthread_t    BM_thread;

#define BMkill()
#define set_mon_numero(num) cthread_set_data(cthread_self(),(void *) num)
#define BM_set_master() numero_du_pere = BM_my_number()
#define BM_is_master() (numero_du_pere == BM_my_number())
#define BM_private
#define BM_sbrk(c) sbrk(c)

#define SeqNumero ((int) cthread_data(cthread_self()))
extern  BM_thread numero_du_pere;

#endif MACH

/* ============================================================ */
/* definition pour la ksr */
/* ============================================================ */

#ifdef KSR
#include <mach.h>
#include <pthread.h>

#define BM_lock(l)   pthread_mutex_lock(l)
#define BM_try_lock(l)  pthread_mutex_trylock(l)
#define BM_unlock(l)  pthread_mutex_unlock(l)
#define BM_init_lock(l) pthread_mutex_init(l,pthread_mutexattr_default)
typedef pthread_mutex_t BM_lock_t;

#define BM_para_init()
#define BM_my_number() pthread_self()
#define BM_fork(t,f,p) pthread_create(&t,pthread_attr_default,\
       f,(void *)p)
#define BMkill()

#define BM_join(p) { void *stat ; pthread_join(p,&stat); }
```

```
#define BM_exit(stat) pthread_exit(stat)
#define BM_detach(p) pthread_detach(&p)
#define BM_fork_and_detach(f,p) \
               { pthread_t t;\
         pthread_create(&t,pthread_attr_default,f,p);\
                 pthread_detach(&t);\
               }
typedef pthread_t    BM_thread;

#define BM_set_master() numero_du_pere = 0
#define BM_is_master() (SeqNumero == 0)

#define BM_private    __private

#define set_mon_numero(num) SeqNumero = num
extern  BM_private int SeqNumero;
extern  BM_thread numero_du_pere;
#endif __ksr__

#else PARA


/* ============================================================ */
/* definition cas sequentiel : sun 4 */
/* ============================================================ */

#define BM_lock(l)
#define BM_try_lock(l)  TRUE
#define BM_unlock(l)
#define BM_init_lock(l)
#define BM_lock_t int

#define BM_para_init()
#define BM_my_number() 0
#define BM_fork(t,f,p) f(p)
#define BMkill()
#define BM_join(p)
#define BM_exit(r) return r
#define BM_detach(p)
#define BM_fork_and_detach(f,p)
#define BM_thread int

#define BM_set_master()
```

```
#define BM_is_master() TRUE
#define BM_private


#endif PARA

#endif __INTERFACE_PARA__
```

Annexe B

Caractéristiques de la KSR

Les informations présentées dans cette annexe sont principalement extraites de [McB92]

Architecture

La KSR1 est une machine hautement parallèle prévue pour contenir des centaines de processeurs, tout en préservant la simplicité d'utilisation de la mémoire partagée. Un système KSR1 peut contenir de 8 à 1088 processeurs avec des performances de l'ordre de 320 à 43.520 Mflops. Les processeurs partagent un espace d'adressage pouvant aller jusqu'à un million de mégabytes (2^{40} bytes).

Ce modèle a été possible grâce à une nouvelle technique appelée "mémoire ALLCACHE".

Logiciel

Le système de la KSR1 est une extension d'OSF/1(Mach). KSR OS est compatible avec 4.3BSD UNIX.

L'environnement est complet, X11, Motif, produits GNU. Au niveau des langages de programmation, on trouve Fortran (avec parallélisation automatique), C avec librairie de threads POSIX , IBM-compatible COBOL, un *debugger* parallèle.

Processeurs

Le processeur de la KSR1 est un ensemble de 4 puces. Une de ces puces, la principale, est appelée *Cell Execution Unit* (CEU). Le CEU est l'unité de contrôle du processeur. À chaque cycle d'horloge, le CEU *fetch* deux instructions. Certaines de ces instructions (loads, stores, branches, address arithmetic) sont directement traitées à l'intérieur du CEU.

Les instructions opèrent sur 40 bits d'adresse. Cette caractéristique est indispensable afin de pouvoir adresser la mémoire dans le cas d'un nombre important de processeurs. La seconde génération travaille sur 64 bits. Le CEU possède 32 registres d'adresses de 40 bits chacun.

Le CEU travaille en coopération avec trois co-processeurs :

- Floating Point Unit : 64 registres de 64 bits
- Integer and logical Operations Unit : 32 registres de 64 bits
- I/O Channel Unit : 30 Mbytes/sec.

Réseaux

La KSR1 accepte les principaux protocoles de communication :

- TCP/IP, NFS, DCE, SNA-3270,3770,LU6.2/PU2.1, X25,X29, X28, X3 ;
- Ethernet, Token Ring, HiPPI and FDDI.

Références

[Alb91a] H. Alblas, "The Trouble with Parallel Attribute Evaluation: an Explanation by Example", oral presentation at IFIP WG 2.4 meeting, Grassau, Jan. 1991.

[Alb91b] ———, "Introduction to Attribute Grammars", in *Attribute Grammars, Applications and Systems*, Prague, H. Alblas & B. Melichar, réds., 1–15, Lect. Notes in Comp. Sci. **545**, Springer-Verlag, New York–Heidelberg–Berlin, Juin 1991.

[Bar82] K. Barbar, "Étude comparative de différentes classes de grammaires d'attributs ordonnées", thèse de 3ème cycle, Univ. de Bordeaux I, Juin 1982.

[Bar84] ———, "Classification des grammaires d'attributs ordonnées", rapport 8412, Univ. de Bordeaux I, Avr. 1984.

[Boc76] G. V. Bochmann, "Semantic Evaluation from Left to Right", *Comm. ACM* **19**, 2 (Fév. 1976), 55–62.

[BöZ87] H-J. Böhm & W. Zwanepoel, "Parallel Attribute Grammar Evaluation", in *7th Int. Conf. on Distributed Computing Systems*, Berlin, R. Popescu-Zeletin, G. Le Lann & K. H. Kim, réds., 347–354, Sept. 1987.

[CoH79] R. Cohen & E. Harry, "Automatic Generation of Near-Optimal Linear-Time Translators for Non-Circular Attribute Grammars", in *6th ACM Symp. on Principles of Progr. Languages*, San Antonio, TX, 121–134, Jan. 1979.

[CoF80] B. Courcelle & P. Franchi-Zannettacci, "On the Equivalence Problem for Attribute Systems", rapport 8026, Univ. de Bordeaux I, 1980.

[CoF82] ———, "Attribute Grammars and Recursive Program Schemes", *Theoret. Comput. Sci.* **17**, 2 and 3 (1982), 163–191 and 235-257. See also: rapport 8008, Univ. de Bordeaux I (Avr. 1980)..

[Der84] P. Deransart, "Validation des grammaires d'attributs", thèse d'État, Univ. de Bordeaux I, Oct. 1984.

[DJL84] P. Deransart, M. Jourdan & B. Lorho, "Speeding up Circularity Tests for Attribute Grammars", *Acta Inform.* **21** (Déc. 1984), 375–391. See also: rapport RR 211, INRIA, Rocquencourt (Mai 1983)..

[DJL88] ———, *Attribute Grammars: Definitions, Systems and Bibliography*, Lect. Notes in Comp. Sci. **323**, Springer-Verlag, New York–Heidelberg–Berlin, Août 1988.

[Eng84] J. Engelfriet, "Attribute Grammars: Attribute Evaluation Methods", in *Methods and Tools for Compiler Construction*, B. Lorho, réd., 103–138, Cambridge Univ. Press, Cambridge, 1984.

[**EnF82**] J. Engelfriet & G. Filé, "Simple Multi-Visit Attribute Grammars", *J. Comput. System Sci.* **24**, 3 (Juin 1982), 283–314. See also: memorandum 314, Onderafdeling der Informatica, Tech. Hogeschool Twente (Août 1980)..

[**Fan72**] I. Fang, "FOLDS, a Declarative Formal Language Definition System", PhD thesis, report STAN-CS-72-329, Comp. Sc. Dept., Stanford Univ., Déc. 1972. Abstract in: *Séminaires Structure et Programmation des Calculateurs 1973*, ed. M. Kronental and Bernard Lorho, INRIA, Rocquencourt, pp. 275-290 (1973)..

[**FaY86**] R. Farrow & D. M. Yellin, "A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators", *Acta Inform.* **23**, 4 (1986), 393–427. See also: Tech. Rep., Dept. of Comp. Sc., Columbia Univ., New York, NY (Jan. 1985)..

[**Fil83**] G. Filé, "Interpretation and Reduction of Attribute Grammars", *Acta Inform.* **19** (1983), 115–150. See also: memorandum 359, Onderafdeling der Informatica, Tech. Hogeschool Twente (1981)..

[**Fil86**] ———, "Classical and Incremental Attribute Evaluation by Means of Recursive Procedures", in *11th Coll. on Trees in Algebra and Programming (CAAP '86)*, Nice, P. Franchi-Zannettacci, réd., 112–126, Lect. Notes in Comp. Sci. **214**, Springer-Verlag, New York–Heidelberg–Berlin, Mars 1986.

[**Fra82**] P. Franchi-Zannettacci, "Attributs sémantiques et schémas de programmes", thèse d'État, Univ. de Bordeaux I, Mars 1982.

[**Fra83**] J. L. Frankel, "The Architecture of Closely-coupled Distributed Computers and their Language Processors", PhD thesis, Dept. of Applied Maths., Harvard Univ., 1983.

[**GZZ89**] T. Gross, A. Zobel & M. Zolg, "Parallel Compilation for a Parallel Machine", in *ACM SIGPLAN '89 Conf. on Progr. Languages Design and Implementation*, Portland, OR, numéro spécial de *ACM SIGPLAN Notices* **24**, 7 (Juil. 1989), 91–100.

[**Jal83**] F. Jalili, "A General Linear-Time Evaluator for Attribute Grammars", *ACM SIGPLAN Notices* **18**, 9 (Sept. 1983), 35–44.

[**JOR75**] M. Jazayeri, W. F. Ogden & W. C. Rounds, "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars", *Comm. ACM* **18**, 12 (Déc. 1975), 679–706.

[**Jou82**] M. Jourdan, "Un évaluateur efficace pour les grammaires attribuées fortement non-circulaires", rapport 82-39, Laboratoire d'Informatique Théorique et Programmation, Paris, Sept. 1982.

[**Jou84a**] ———, "Recursive Evaluators for Attribute Grammars: an Implementation", in *Methods and Tools for Compiler Construction*, B. Lorho, réd., 139–164, Cambridge Univ. Press, Cambridge, 1984.

[**Jou84b**] ———, "Strongly Non-Circular Attribute Grammars and Their Recursive Evaluation", in *ACM SIGPLAN '84 Symp. on Compiler Construction*, Montréal, numéro spécial de *ACM SIGPLAN Notices* **19**, 6 (Juin 1984), 81–93.

[**Jou84c**] ———, "Les grammaires attribuées: implantation, applications, optimisations", thèse DDI, Univ. Paris VII, Mai 1984.

[**Jou91**] ———, "A Survey of Parallel Evaluation Methods", in *Attribute Grammars, Applications and Systems*, Prague, H. Alblas & B. Melichar, réds., 234–255, Lect. Notes in Comp. Sci. **545**, Springer-Verlag, New York–Heidelberg–Berlin, Juin 1991.

[**JoP89**] M. Jourdan & D. Parigot, *The FNC-2 System User's Guide and Reference Manual* release 0.4, INRIA, Rocquencourt, Fév. 1989. This manual is periodically updated..

[**Jul89**] C. Julié, "Optimisation de l'espace mémoire pour l'évaluation des grammaires attribuées", thèse, Dépt. d'Informatique, Univ. d'Orléans, Sept. 1989.

[**KaK86**] S. M. Kaplan & G. E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-based Environments", in *5th ACM Symp. on Principles of Distributed Computing*, Calgary, 121–130, Août 1986. See also: report UIUCDCS-R-86-1294, Univ. of Illinois at Urbana-Champaign (Sept. 1986)..

[**Kas80**] U. Kastens, "Ordered Attribute Grammars", *Acta Inform.* **13**, 3 (1980), 229–256. See also: Bericht 7/78, Institut für Informatik II, Univ. Karlsruhe (1978)..

[**Kas84**] ———, "The GAG-System—A Tool for Compiler Construction", in *Methods and Tools for Compiler Construction*, B. Lorho, réd., 165–182, Cambridge Univ. Press, Cambridge, 1984.

[**Kas91**] ———, "Attribute Grammars in a Compiler Construction Environment", in *Attribute Grammars, Applications and Systems*, Prague, H. Alblas & B. Melichar, réds., 380–400, Lect. Notes in Comp. Sci. **545**, Springer-Verlag, New York–Heidelberg–Berlin, Juin 1991.

[**KHZ82**] U. Kastens, B. Hutt & E. Zimmermann, *GAG: A Practical Compiler Generator*, Lect. Notes in Comp. Sci. **141**, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

[**Kat84**] T. Katayama, "Translation of Attribute Grammars into Procedures", *ACM Trans. Progr. Languages and Systems* **6**, 3 (Juil. 1984), 345–369.

[**KeR79**] K. Kennedy & J. Ramanathan, "A Deterministic Attribute Grammar Evaluator based on Dynamic Sequencing", *ACM Trans. Progr. Languages and Systems* **1**, 1 (Juil. 1979), 142–160.

[**KeW76**] K. Kennedy & S. K. Warren, "Automatic Generation of Efficient Evaluators for Attribute Grammars", in *3rd ACM Symp. on Principles of Progr. Languages*, Atlanta, Ge, 32–49, Jan. 1976.

[**KlG92**] A. Klaiber & M. Gokhale, "Parallel Evaluation of Attribute Grammars", in *IEEE transaction on parallel and distributed systems* **3 2**, Mars 1992.

[**Kle91**] E. Klein, "Ein Modell zur Generierung paralleler Attributauswerter", Dissertation, Fakultät für Informatik, Univ. Karlsruhe, 1991.

[**Kle92**] ———, "Parallel Ordered Attribute Grammars", in *IEEE*, 1992.

[**Knu68**] D. E. Knuth, "Semantics of Context-free Languages", *Math. Systems Theory* **2**, 2 (Juin 1968), 127–145. Correction: *Math. Systems Theory* **5**, 1, pp. 95-96 (Mars 1971)..

[**Kui89**] M. F. Kuiper, "Parallel Attribute Evaluation", PhD thesis, Faculteit Wiskunde en Informatica, Rijksuniv. Utrecht, Nov. 1989.

[**KuS90**] M. F. Kuiper & S. D. Swierstra, "Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism", in *Attribute Grammars and their Applications (WAGA)*, Paris, P. Deransart & M. Jourdan, réds., 61–75, Lect. Notes in Comp. Sci. **461**, Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990.

[**Lip79**] D. E. Lipkie, "A Compiler Design for Multiple Independent Processor Computers", PhD thesis, Dept. of Comp. Sc., Univ. of Washington, Seattle, WA, 1979.

[**Lor74**] B. Lorho, "De la définition à la traduction des langages de programmation: méthode des attributs sémantiques", thèse d'État, Univ. Paul Sabatier, Toulouse, Nov. 1974.

[**Lor77**] ———, "Semantic Attributes Processing in the System DELTA", in *Methods of Algorithmic Language Implementation*, A. Ershov & C. H. A. Koster., réds., 21–40, Lect. Notes in Comp. Sci. **47**, Springer-Verlag, New York–Heidelberg–Berlin, 1977.

[**Mad80**] O. L. Madsen, "On Defining Semantics by means of Extended Attribute Grammars", in *Semantics-Directed Compiler Generation*, N. D. Jones, réd., 259–299, Lect. Notes in Comp. Sci. **94**, Springer-Verlag, New York–Heidelberg–Berlin, 1980. See also: report DAIMI PB-109, Comp. Sc. Dept., Aarhus Univ. (Jan. 1980)..

[**Mar90**] B. Marmol, "Évaluateurs d'attributs parallèles sur multi-processeurs à mémoire partagée", rapport de DEA, Univ. d'Orléans, Sept. 1990.

[**McB92**] O. A. McBryan, *Parallel Computers: Current Systems and Capabilities*, Dept. of Comp. Sc., Univ. of Colorado at Boulder, Déc. 1992.

[**MöW82**] U. Möncke & R. Wilhelm, "Iterative Algorithms on Grammar Graphs", in *Conf. on Graph-theoretic Concepts in Computer Science (WG '82)*, Neunkirchen a.Br., H. J. Schneider & H. Göttler, réds., 177–194, Hanser Verlag, München, Juin 1982.

[**MöW91**] ———, "Grammar Flow Analysis", in *Attribute Grammars, Applications and Systems*, Prague, H. Alblas & B. Melichar, réds., 151–186, Lect. Notes in Comp. Sci. **545**, Springer-Verlag, New York–Heidelberg–Berlin, Juin 1991.

[**Par87**] D. Parigot, "Mise en œuvre des grammaires attribuées: transformation, évaluation incrémentale, optimisations", thèse de 3ème cycle, Univ. de Paris-Sud, Orsay, Sept. 1987.

[**Rep84**] T. Reps, *Generating Language-based Environments*, MIT Press, Cambridge, MA, 1984. Reprint of PhD thesis, report TR 82-514, Dept. of Comp. Sc., Cornell Univ., Ithaca, NY (Août 1982)..

[**Sch79**] R. M. Schell, "Methods for Constructing Parallel Compilers for Use in a Multiprocessor Environment", PhD thesis, report UIUCDCS-R-79-0958, Dept. of Comp. Sc., Univ. of Illinois at Urbana-Champaign, Fév. 1979.

[**SWJ88**] V. Seshadri, D. B. Wortman, M. D. Junkin, S. Weber, C. P. Yu & I. Small, "Semantic Analysis in a Concurrent Compiler", in *ACM SIGPLAN '88 Conf. on Progr. Languages Design and Implementation*, Atlanta, GA, numéro spécial de *ACM SIGPLAN Notices* **23**, 7 (Juil. 1988), 233–239.

[**Van88**] M. T. Vandevoorde, "Parallel Compilation on a Tightly-coupled Multiprocessor", SRC report 26, DEC Systems Research Center, Palo Alto, CA, Mars 1988.

[**VSK90**] H. H. Vogt, S. D. Swierstra & M. F. Kuiper, "On the Efficient Incremental Evaluation of Higher Order Attribute Grammars", report RUU–CS–90–36, Utrecht Univ., Déc. 1990.

[**Zar90**] A. K. Zaring, "Parallel Evaluation in Attribute Grammar-based Systems", PhD thesis, Tech. Rep. 90-1149, Dept. of Comp. Sc., Cornell Univ., Août 1990.