

N. d'ordre: 1618

THÈSE

présentée devant

l'Université de Rennes 1
Institut de Formation Supérieure en Informatique et
Communication

Pour obtenir

le grade de Docteur de l'Université de Rennes 1
Mention INFORMATIQUE

par
François Noël

Sujet de la thèse

Spécialisation dynamique de code par évaluation partielle

Soutenue le 23 octobre 1996 devant la commission d'examen

MM.	Banâtre Jean-Pierre	Président
	Consel Charles	Examineur
	Danvy Olivier	Rapporteur
	Jones Neil D.	Rapporteur
	Le Métayer Daniel	Examineur

Remerciements

Je tiens à remercier :

Jean-Pierre Banâtre, de m'avoir fait l'honneur de présider ce jury.

Neil D. Jones et Olivier Danvy, de m'avoir fait l'honneur d'être les rapporteurs de cette thèse.

Charles Consel, de m'avoir fourni ce sujet en or ainsi que d'avoir dirigé mes recherches.

Daniel Le Métayer, de m'avoir fait l'honneur de participer à ce jury.

Tous ceux qui ont participé à l'élaboration du système Tempo et qui ont, de fait, permis la validation de l'approche présentée dans cette thèse.

Elisabeth, d'avoir relu et corrigé le style de ce document.

Marie, d'avoir supporté mes humeurs durant la rédaction de ce document.

Table des matières

1	Introduction	7
2	Grammaires d'arbres	11
3	Évaluation partielle	13
3.1	Présentation	13
3.1.1	Principes	14
3.2	Un exemple complet d'évaluateur partiel	17
3.2.1	Le langage PLI	17
3.2.2	Évaluation partielle en ligne de PLI	19
3.2.3	Évaluation partielle hors ligne pour PLI	22
3.2.4	La notion d'explicateur	30
4	Spécialisation dynamique	33
4.1	Systèmes généraux	34
4.1.1	DCG	34
4.1.2	Tick C	35
4.2	Compilateurs dynamiques	38
4.2.1	Fabius	39
4.2.2	Auslander, Philipose, Chamber, Eggers et Bershad	41
5	Notre approche	47
5.1	Extension génératrice	49
5.1.1	Grammaires de spécialisation	49
5.1.2	Spécialiseur dédié	51
5.2	Dérivation des patrons objets	52
5.2.1	Identification des patrons sources	52
5.2.2	Compilation des patrons	54
5.3	Production du spécialiseur dynamique	55

6	Approche formelle	57
6.1	Extensions du langage	57
6.1.1	Grammaires	57
6.1.2	Extension génératrices	58
6.2	Calcul de l'extension génératrice	60
6.3	Correction	62
7	Mise en œuvre	69
7.1	Tempo	69
7.1.1	Pré-spécialisation	70
7.1.2	Spécialisation statique	73
7.2	Spécialisation dynamique avec Tempo	73
7.2.1	Générateur d'extension génératrices	77
7.2.2	Générateur de patrons	78
7.2.3	Compilateur de patrons	81
7.2.4	Générateur d'extensions génératrices dynamiques	84
7.3	Conclusion	84
8	Conclusion	87
8.1	Résultats	87
8.2	Récapitulatif	87
8.3	Perspectives et directions futures	88

Chapitre 1

Introduction

La généricité des logiciels repose souvent sur une multitude d'options qui affectent leurs comportements. L'interprétation de ces options se traduit généralement par une inefficacité importante du programme. De fait, lors de la conception d'une application complexe, le programmeur doit trouver un compromis entre la généralité et l'efficacité de son système. Généralement, nombre de ces options sont constantes pendant une partie de l'exécution de l'application.

Pour tirer avantage de contextes d'exécution spécifiques, le programmeur peut prévoir toutes les spécialisations du code concerné ayant des structures différentes. Le programmeur écrit alors une procédure chargée d'aiguiller le flot de contrôle vers la spécialisation correspondante en fonction des valeurs des invariants d'exécution.

Cette technique a été étudiée par Pike, Locanthi et Reiser sur la procédure **BitBlit**, chargée de fusionner des plans graphiques [PLR85]. Cette procédure est très générale et, de fait, possède un grand nombre de paramètres¹ rendant difficile une mise en œuvre efficace. La procédure **BitBlit** est constituée d'un emboîtement de boucles et ses paramètres déterminent le flot de contrôle dans la boucle la plus interne. Comme les paramètres sont constants durant toute l'exécution de la procédure, il en est de même pour le flot de contrôle dans la boucle interne. La version prévoyant toutes les spécialisations de cette procédure fonctionne beaucoup plus vite que la version générique. Cependant, la taille du code obtenu est de l'ordre de 1Mo, soit environ 128 fois plus importante que celle de la procédure générique. Cette méthode n'est pas réaliste dès lors que le nombre de spécialisations devient trop grand.

La spécialisation dynamique consiste à ne produire le code des spécialisations qu'à l'exécution. Adapter une application à différents contextes d'exécution, sans pour autant produire toutes les spécialisations correspondantes à la compilation, est un élément crucial pour éviter une explosion de code. Cette approche a beaucoup été étudiée dans des domaines variés comme les systèmes d'exploitation [PMI88, MP89, PAB⁺95] et le graphisme [Loc87].

1. Ces paramètres sont entres autres : la tailles et l'alignements des plans, l'opérateur de fusion, etc.

Un aspect essentiel de la spécialisation dynamique de code est le coût de la production des spécialisations. En effet, puisque la génération de code est effectuée à l'exécution, ce coût est à prendre en considération si l'on désire valider objectivement une telle technique. Plus la production des spécialisations est chère, plus il faut utiliser le code spécialisé afin d'en amortir le coût de production.

Locanthi a expérimenté, manuellement, la spécialisation dynamique de code pour la procédure **BitBlit** [Loc87]. Il obtient un code exécutable fonctionnant environ 4 fois plus vite que la procédure générique. La taille du code chargé de produire les spécialisations à l'exécution est inférieure à celle de la procédure générique, ce qui rend son approche réaliste.

Massalin et Pu ont conçu un système d'exploitation qui inclut la spécialisation dynamique comme technique de base pour optimiser une grande variété des composants du système. Ils obtiennent des accélérations variant de 2 à 40 selon les parties du système considérées [MP89].

Pu *et al.* se sont intéressés à la spécialisation dynamique dans le cadre des systèmes d'exploitation [PMI88, MP89, PAB⁺95]. Par exemple, la gestion de fichiers peut grandement tirer partie de la spécialisation dynamique [PAB⁺95]. Lorsqu'un fichier est ouvert, des paramètres tels que le type du fichier ainsi que le périphérique sur lequel il se trouve restent constants durant toute l'ouverture du fichier. Un système d'exploitation pratiquant la spécialisation dynamique peut produire, lors de l'ouverture du fichier, des spécialisations des primitives de manipulation de fichiers, par rapport à ces invariants. Ils rapportent que la spécialisation dynamique permet d'éliminer les interprétations redondantes des structures de donnée et que les gains obtenus sont significatifs [PAB⁺95].

Cependant, la plupart de ces expériences sont manuelles et spécifiques à une application particulière [KEH91]. Généralement, le programmeur écrit manuellement des patrons, c'est-à-dire des fragments de code avec des trous. Le programmeur doit aussi écrire le spécialiseur dynamique chargé d'assembler et d'instancier les patrons avec les valeurs des invariants dynamiques [KEH93]. Pour limiter le coût de la production des spécialisations, les patrons sont généralement écrits directement en langage machine afin d'éviter l'utilisation d'un assembleur, voire même d'un compilateur à l'exécution.

L'écriture manuelle de patrons en langage machine est une tâche fastidieuse et peu fiable. De plus, les patrons ne sont pas portables et doivent être réécrits pour chaque architecture. Les mêmes problèmes apparaissent au niveau de la maintenance des applications. Ces approches nécessitent une trop grande intervention du programmeur. Elles ne semblent pas appropriées à l'écriture de spécialiseurs complexes.

Plus récemment, des systèmes généraux permettant de faire de la production dynamique de code sont apparus [EP94, EHK96]. Il ne s'agit pas encore de systèmes permettant de produire automatiquement des spécialiseurs dynamiques. Toutefois, ces systèmes permettent de spécifier le code dynamique de manière indépendante de la machine cible. Le passage d'une machine à une autre se fait simplement en procédant à un reciblage du système. Les clients n'ont pas à être modifiés.

Les compilateurs dynamiques sont des systèmes qui repoussent certaines phases de la

compilation de certaines régions du code [LL96, APC⁺96] à l'exécution. En particulier, la production de code exécutable, pour ces régions, est effectuée à l'exécution. Ainsi, disposant des valeurs, le spécialiseur dynamique peut mieux optimiser le code. Ces systèmes produisent automatiquement les spécialiseurs dynamiques à partir d'un programme et de quelques annotations. Ces dernières permettent la délimitation des zones que l'on désire compiler dynamiquement ainsi que la spécification des invariants d'exécution.

Leone et Lee ont proposé le compilateur dynamique, nommé Fabius, qui traite un sous-ensemble pur et du premier ordre du langage ML [LL93, LL94, LL96]. Les spécialiseurs dynamiques émettent les instructions une à une plutôt que d'utiliser des patrons. Auslander, Philipose, Chamber, Eggers et Bershad ont développé un compilateur dynamique pour le langage C. Ici, l'approche utilisant des patrons a été retenue.

Les compilateurs dynamiques procèdent à des optimisations fines des instructions ou des patrons, par rapport aux valeurs des invariants dynamiques. Le code obtenu est donc de bonne qualité et ces compilateurs permettent de bonnes accélérations. Toutefois, cela a pour effet d'augmenter, dans de grandes proportions, le coût de production des spécialisations. Par exemple les mesures effectuées par Auslander *et al.* montrent qu'il faut un minimum de 900 utilisations du code spécialisé pour en amortir la production [APC⁺96].

Nous présentons dans ce document une approche générale basée sur l'évaluation partielle de programmes. Le langage que nous traitons est le langage C. Le spécialiseur dynamique est automatiquement produit à partir du programme et d'une description des invariants d'exécution. Le spécialiseur dynamique utilise des patrons pour produire les spécialisations. Ces patrons sont automatiquement dérivés d'une grammaire caractérisant l'ensemble de toutes les spécialisations possibles du code considéré. Les patrons sont compilés statiquement pour produire des patrons objets, similaires à ceux des approches manuelles décrites ci-dessus. Contrairement aux compilateurs dynamiques, nous n'optimisons pas les patrons à l'exécution. Il ne s'agit pas d'une limitation mais d'un choix de conception permettant d'assurer un très faible coût de production des spécialisations. Le spécialiseur dynamique n'a que quelques opérations simples, en plus du calcul des invariants, à effectuer pour produire le code. Nous estimons qu'il faut, en moyenne, moins de 10 utilisations de la spécialisation pour en amortir le coût de production.

Plan

Le chapitre 2 présente les grammaires d'arbres sur lesquelles sont basées nos analyses. Le chapitre 3 introduit les concepts de base de l'évaluation partielle des langages impératifs. Le chapitre 4 expose les différents éléments relatifs à la spécialisation dynamique. Deux systèmes généraux de production dynamique de code ainsi que deux compilateurs dynamiques sont présentés. Les chapitres 5 et 6 présentent l'approche que nous avons choisie. Le chapitre 7 décrit l'intégration de nos travaux dans l'évaluateur partiel Tempo. Le chapitre 8 présente nos résultats, récapitule les contributions de notre travail et détaille des directions futures.

Chapitre 2

Grammaires d'arbres

Ce document utilise les grammaires d'arbres pour définir les différentes syntaxes abstraites dont nous avons besoin. Elles sont aussi un constituant essentiel de nos spécialiseurs dynamiques. Cette section présente les concepts élémentaires sur ces objets pour la lecture de ce document.

Soit $\Sigma = \cup_{n \geq 0} \Sigma_n$ un alphabet gradué de symboles. On définit alors l'ensemble des arbres (sans variables) sur Σ par :

Définition 1 *Arbres*

L'ensemble des arbres sur l'alphabet Σ est le plus petit ensemble F_Σ tel que :

- $\Sigma_0 \subset F_\Sigma$
- $t_1, \dots, t_n \in F_\Sigma \wedge f \in \Sigma_n \Rightarrow f(t_1, \dots, t_n) \in F_\Sigma$

Une grammaire d'arbres est un objet permettant de caractériser des ensembles (potentiellement infinis) d'arbres aussi appelés forêts. Nous nous intéresserons plus particulièrement aux grammaires régulières définies comme suit :

Définition 2 *Grammaires d'arbres régulières*

Une grammaire régulière sur l'alphabet Σ est un triplet (N, P, s) avec :

- N un ensemble fini, non vide, de symboles non terminaux tel que $N \cap \Sigma_0 = \emptyset$
- $P \subset N \times F_{\Sigma \cup N}$ un ensemble fini, non vide, de productions notées $n \rightarrow t$
- $s \in N$ un symbole de départ

Soit $\gamma = (N, P, s)$ une grammaire régulière et p, q deux arbres de $F_{\Sigma \cup N}$. On définit alors la relation \rightarrow_γ par $p \rightarrow_\gamma q$ si q est obtenu par le remplacement, dans p , d'une occurrence d'un

non terminal n par un arbre t avec $n \rightarrow t \in P$. On note \rightarrow_γ^* la fermeture réflexive et transitive de \rightarrow_γ . On définit alors le langage produit par une grammaire par :

Définition 3 *Langage*

La forêt produite par une grammaire d'arbres $\gamma = (N, P, s)$ en $n \in N$ est l'ensemble $L(\gamma, n) = \{t \in F_\Sigma \mid n \rightarrow_\gamma^ t\}$. La forêt produite par γ , notée $L(\gamma)$ est l'ensemble $L(\gamma, s)$.*

Chapitre 3

Évaluation partielle

L'évaluation partielle est une technique d'optimisation de programmes qui a été particulièrement étudiée dans le contexte des langages fonctionnels [JSS89, AC94, Con93, BW93]. Ce n'est que récemment que les analyses pratiquées dans les évaluateurs partiels ont été étendues aux langages impératifs. Aujourd'hui, l'évaluation partielle traite des langages impératifs complets [And92, And94, BGZ94, KKZG94, CHN⁺96]. Ce chapitre décrit les concepts élémentaires de l'évaluation partielle des langages impératifs. Ils sont, en effet, à la base des travaux présentés dans ce document.

3.1 Présentation

Le but de l'évaluation partielle est de spécialiser automatiquement un programme par rapport à une partie de ses entrées [JGS93, CD93]. La notion d'entrée est ici très générale et inclut les paramètres du programme, ses variables globales, etc. Le résultat obtenu, appelé programme résiduel, est un nouveau programme calculant la même fonction que le programme initial. Généralement, le programme résiduel est plus efficace que le programme initial car les calculs ne dépendant que des valeurs connues à la spécialisation ont été supprimés.

Le processus de production d'un programme résiduel à partir d'un programme et des valeurs connues peut être vu comme une interprétation non standard du programme. Cette dernière évalue normalement les constructions qui ne dépendent que des entrées connues et reproduit les autres.

Cette technique permet d'optimiser les programmes dans lesquels une partie des entrées reste constante durant toute l'exécution. C'est une situation fréquente si le programmeur prend soin d'écrire du code générique. De nombreux paramètres sont alors constants pour une exécution donnée. La spécialisation de l'application par rapport à ces valeurs permettra de retrouver l'efficacité d'un code «sur mesure». La spécialisation se fera d'autant mieux que le programme comporte une couche d'interprétation, liée à ces paramètres, bien distincte du

reste du code.

3.1.1 Principes

De manière générale, un programme écrit dans un langage impératif L peut être vu comme une continuation. Une continuation est une fonction qui associe un nouvel état mémoire à un état initial. Elle est calculée par la sémantique du langage à partir du texte du programme. Le résultat de l'exécution du programme est obtenu en appliquant la continuation à un état mémoire initial, construit à partir des entrées du programme considéré. Spécialiser un programme par rapport aux valeurs d'une partie de ses entrées revient donc à le spécialiser par rapport à un état mémoire partiellement défini. Cette définition est spécifique à la spécialisation des langages impératifs.

Un état mémoire est généralement modélisé par une fonction associant une valeur à chaque emplacement mémoire. Soit Loc un ensemble de cellules et Val l'ensemble des valeurs qu'elles peuvent contenir. L'état mémoire est alors représenté par une fonction σ de l'ensemble $Mem = Loc \rightarrow Val$.

Un état mémoire partiel est un état mémoire dans lequel il y a des inconnues. C'est à dire une fonction δ de l'ensemble $Mem^{ep} = Loc \rightarrow Val + \{\square\}$. Les éléments de Val correspondent à des valeurs connues, dites *statiques*. Le symbole « \square » correspond à des valeurs inconnues dites *dynamiques*.

On définit alors la notion de programme résiduel par :

Définition 4 *Programme résiduel*

Soit P un programme écrit dans un langage L et $\delta \in Mem^{ep}$ un état mémoire partiel. P_δ , écrit dans un autre langage T , est un programme résiduel de P par rapport à δ si

$$(\forall \sigma \in Mem) ((\forall l \in Loc) \delta(l) \in Val \Rightarrow \sigma(l) = \delta(l)) \Rightarrow \mathcal{T}[[P_\delta]]\sigma = \mathcal{L}[[P]]\sigma.$$

où \mathcal{L} et \mathcal{T} sont des interprètes pour les langages L et T respectivement.

Un *évaluateur partiel* est une fonction qui calcule un programme résiduel à partir d'un programme et d'un état mémoire partiel. L'obtention du programme résiduel consiste essentiellement à propager les valeurs statiques afin d'évaluer le maximum de constructions et de reproduire les autres. La propagation des constantes est aussi effectuée au travers des appels de procédures qui sont dépliées ou résidualisées. La notion de résidualisation de procédures est la même que pour les programmes. Les évaluateurs partiels, dit *polyvariants*, sont capables de produire plusieurs spécialisations d'une même procédure. À l'inverse, les évaluateurs partiels, dit *monovariants*, n'en produisent qu'une. Dans la suite de ce document, nous ne considérons que les évaluateurs partiels polyvariants.

On distingue essentiellement deux types de mise en œuvre de la sémantique définissant la spécialisation: les évaluateurs partiels *en ligne*, et les évaluateurs partiels *hors ligne* [JSS89, CD93, JGS93].

Évaluateurs partiels en ligne

Un évaluateur partiel en ligne est une mise en œuvre directe de la sémantique de la spécialisation. Le spécialiste évalue les constructions ne dépendant que des valeurs statiques et détermine le code résiduel «à la volée».

La spécialisation en ligne est très précise. En effet, les analyses chargées de déterminer les constructions que l'on peut évaluer sont effectuées en présence des valeurs de spécialisations fournies par l'état partiel.

Cependant, les évaluateurs partiels en ligne sont lents car il n'y a aucune réutilisation des calculs. Par exemple, les différentes spécialisations d'une même procédure, pour des états partiels définissant les mêmes valeurs dynamiques, engendrent des analyses distinctes alors que les opérations effectuées sont pratiquement les mêmes.

Évaluateurs partiels hors ligne

L'idée de la spécialisation hors ligne est de séparer le processus de spécialisation en deux phases distinctes afin de «compiler» les analyses effectuées [JSS89].

La première phase, que nous appellerons phase de *pré-spécialisation*, travaille sur une description abstraite de l'état partiel. Cette description associe, à chaque cellule mémoire, un *temps de liaison*, *statique* ou *dynamique*, indiquant si la valeur qu'elle contient sera ou non disponible pendant la deuxième phase. Une analyse, appelée *analyse de temps de liaison*, propage cette description dans tout le programme pour déterminer les constructions qui seront ou non évaluables lors de la deuxième phase [JSS89, Con93, JGS93]. Certains évaluateurs hors ligne effectuent ensuite une analyse, appelée *analyse d'actions*, pour déterminer, pour chaque construction du programme, une transformation permettant d'obtenir le code résiduel [CD90].

La deuxième phase effectue la spécialisation du programme à l'aide des valeurs concrètes de l'état partiel. Cette phase est guidée par les informations collectées durant la phase de pré-spécialisation.

La spécialisation hors ligne factorise les analyses ne dépendant que de la description de l'état partiel. Elle est donc plus efficace que la spécialisation en ligne. En revanche, elle est moins précise car l'absence de valeurs concrètes oblige la phase de pré-spécialisation à introduire des approximations.

Le reste de cette section présente plus en détail les différentes opérations effectuées par un évaluateur partiel hors ligne.

Analyse de temps de liaison. L'analyse de temps de liaison est une abstraction de la sémantique de spécialisation en ligne. Ne disposant pas des valeurs concrètes, elle doit procéder à des approximations. Par exemple, lors du traitement d'une conditionnelle dont le test est évaluable, l'analyseur de temps de liaison procède à l'analyse des deux branches. Il est incapable de déterminer la branche qui sera choisie à la spécialisation puisqu'il ne peut pas calculer

le test. Il doit donc joindre les résultats. Cette jonction introduit une perte de précision par rapport à l'évaluation partielle en ligne qui ne spécialise que la branche choisie. La précision de l'analyseur de temps de liaison détermine la qualité du code résiduel.

On peut classer les analyseurs de temps de liaison en fonction de leur précision. On donne ci-dessous deux critères qui sont à considérer pour cette évaluation.

- **Sensibilité au contexte.** Ce critère est également appelée polyvariance de l'analyse de temps de liaison qui ne doit pas être confondue avec la polyvariance du spécialiste. La sensibilité au contexte permet de déterminer différentes descriptions d'une même procédure suivant le contexte d'appel [Con93].

Si l'analyse de temps de liaison n'est pas sensible au contexte, elle fusionne les contextes d'appel et ne produit qu'une seule description par procédure. Ce regroupement est conjonctif: un paramètre formel n'est statique que si tous les contextes d'appel comportent une valeur statique pour l'argument correspondant. Certaines valeurs sont donc considérées comme dynamiques alors qu'elles sont, en réalité, statiques. Le programme spécialisé correspondant contiendra donc plus de constructions résiduelles que celui produit par un évaluateur partiel en ligne.

- **Sensibilité au flot.** La sensibilité au flot permet de déterminer le temps de liaison des variables en chaque point du programme.

Une analyse de temps de liaison, non sensible au flot, fusionne les différents temps de liaison d'une variable. La variable sera donc considérée dynamique à des endroits où elle est en fait statique. La conséquence est la même que dans le cas de la non sensibilité au contexte.

Analyse d'actions. Certains évaluateurs partiels hors ligne utilisent directement les informations fournies par l'analyse de temps de liaison pour produire le code résiduel. Il est toutefois intéressant d'utiliser les temps de liaisons pour déterminer, pour chaque construction du programme initial, une transformation de programme appelée *action* [CD90]. Les actions permettent d'obtenir les constructions résiduelles des constructions qu'elles annotent. Le résultat de l'analyse d'actions est un arbre syntaxique décoré, appelé *arbre d'actions*.

Il y a essentiellement quatre actions :

- L'action *Évalue* annote les nœuds pouvant être totalement réduits. Les sous-arbres ayant de tels nœuds comme racine n'apparaissent pas dans le programme résiduel.
- L'action *Identité* annote les nœuds qui sont totalement irréductibles. Les sous-arbres ayant de tels nœuds comme racine sont reproduits tels quels dans le programme résiduel.
- L'action *Réduit* annote les nœuds qui se réduisent mais qui contiennent des sous-arbres non réductibles. Ces nœuds disparaissent du programme résiduel.

- L'action *Reconstruit* annote les nœuds qui sont irréductibles mais qui contiennent des sous-arbres réductibles. Ces nœuds sont reproduits dans le programme résiduel.

Spécialisation. En l'absence d'analyse d'actions, la spécialisation est une interprétation non standard du programme initial. Cette interprétation évalue les constructions que l'analyse de temps de liaison a déterminées comme étant statiques et reproduit les autres.

Si la phase de pré-spécialisation comporte une analyse d'actions, la spécialisation consiste à interpréter l'arbre d'actions à l'aide de la sémantique des actions et de l'état partiel.

Extensions génératrices. La spécialisation hors ligne permet aussi la production d'un spécialiseur dédié à un programme et une description d'état partiel donnés [Ers77, JSS89, And92, BW93, JGS93, And94]. Un tel spécialiseur, également appelé *extension génératrice*, produit les spécialisations à partir des valeurs concrètes de l'état partiel.

L'extension génératrice peut être obtenue par spécialisation du spécialiseur par rapport au programme considéré et à la description de l'état partiel [JSS89]. Cette technique oblige à écrire l'évaluateur partiel dans le même langage que celui qu'il analyse.

Il est aussi possible d'obtenir l'extension génératrice en compilant les informations de l'analyse de temps de liaison [And92, And94] où encore l'arbre d'actions si l'évaluateur partiel effectue une analyse d'actions [CD90].

3.2 Un exemple complet d'évaluateur partiel

Dans cette section, nous décrivons le processus de spécialisation pour un petit langage impératif (PLI). Nous étudions, dans un premier temps, un évaluateur partiel en ligne, puis une version hors ligne avec analyse de temps de liaison et analyse d'actions.

3.2.1 Le langage PLI

Afin de simplifier la présentation, le petit langage impératif qui va nous servir de support est réduit à son strict minimum. La figure 3.1 décrit sa syntaxe. Un programme consiste en une instruction qui peut être une instruction vide, une séquence, une affectation ou une conditionnelle. Une expression consiste en une constante, une variable ou un appel de primitive (uniquement binaire). La figure 3.2 donne un exemple de ce que l'on peut écrire avec PLI.

La figure 3.3 décrit la sémantique de PLI. Les domaines sémantiques sont : les entiers, pour interpréter les constantes, les fonctions binaires d'entiers, pour interpréter les primitives, et la mémoire. Normalement, l'interprétation d'un identificateur se fait à l'aide d'un environnement lui associant une cellule mémoire. Cela permet de prendre en compte la structure de bloc des langages de programmation modernes en autorisant plusieurs liaisons pour un même identificateur selon les contextes considérés. La simplicité de notre langage assure que l'ensemble des

$i \in Ident$	Identificateurs	
$n \in Num$	Constantes	
$op \in OpBin$	Opérateurs binaires	
	Syntaxe abstraite	Syntaxe concrète
$c \in Com$	$:: =$	
	NOF	
	SEQ (c_1, c_2)	{ $c_1; c_2$ }
	AFF (i, e)	$i = e$
	COND (e, c_1, c_2)	si (e) alors c_1 sinon c_2
$e \in Exp$	$:: =$	
	CST (n)	n
	VAR (i)	i
	APP (op, e_1, e_2)	e_1 op e_2

FIG. 3.1 – La syntaxe de PLI

```

{
  l = 2 * x;
  si (l == 2) alors res = l * l + 2 * y sinon { l = y * 2; res = l * l };
  res = 2 * res
}

```

FIG. 3.2 – Un exemple de programme PLI

Int	Valeurs entières
$f \in Fun_2 = Int \times Int \rightarrow Int$	Fonctions entières binaires
$\sigma \in Mem = Ident \rightarrow Int$	Mémoire
$\mathcal{C} : Com \rightarrow Mem \rightarrow Mem$	
$\mathcal{C}[\mathbf{NOP}]$	$= \lambda\sigma. \sigma$
$\mathcal{C}[\mathbf{SEQ}(c_1, c_2)]$	$= \mathcal{C}[c_1] \circ \mathcal{C}[c_2]$
$\mathcal{C}[\mathbf{AFF}(i, e)]$	$= \lambda\sigma. \sigma[i \mapsto \mathcal{E}[e]\sigma]$
$\mathcal{C}[\mathbf{COND}(e, c_1, c_2)]$	$= \lambda\sigma. \text{if } \mathcal{E}[e]\sigma \text{ then } \mathcal{C}[c_1]\sigma \text{ else } \mathcal{C}[c_2]\sigma$
$\mathcal{E} : Exp \rightarrow Mem \rightarrow Int$	
$\mathcal{E}[\mathbf{CST}(n)]$	$= \lambda\sigma. \mathcal{N}[n]$
$\mathcal{E}[\mathbf{VAR}(i)]$	$= \lambda\sigma. \sigma(i)$
$\mathcal{E}[\mathbf{APP}(op, e_1, e_2)]$	$= \lambda\sigma. \mathcal{O}[op](\mathcal{E}[e_1]\sigma, \mathcal{E}[e_2]\sigma)$
$\mathcal{N} : Num \rightarrow Int$	
$\mathcal{O} : OpBin \rightarrow Fun_2$	

FIG. 3.3 – La sémantique de PLI

identificateurs et celui des emplacements mémoire sont en bijection. Comme nous ne manipulons pas d'adresses¹, il est possible de s'affranchir de l'environnement et de l'ensemble des cellules, en modélisant un état mémoire par une fonction σ de l'ensemble $Mem = Ident \rightarrow Int$.

Il est à remarquer que la simplicité de PLI interdit les effets de bord dans les expressions. C'est pourquoi l'évaluation d'une expression ne retourne pas d'état mémoire mais simplement une valeur.

3.2.2 Évaluation partielle en ligne de PLI

Dans cette section, nous décrivons un évaluateur partiel en ligne pour PLI. La figure 3.4 décrit une sémantique non standard pour PLI. Elle calcule les programmes résiduels à partir d'un programme et d'un état mémoire partiel. L'évaluation est réalisée à l'aide d'une mémoire partielle dont les états sont des éléments de l'ensemble $Mem^{ep} = Ident \rightarrow Int + \{\square\}$.

Le résultat de l'évaluation partielle d'une expression est une nouvelle expression.

Une constante s'évalue partiellement à elle-même. Une variable statique, donc liée à un entier, s'évalue partiellement à une constante. Une variable dynamique, donc liée à « \square », s'évalue partiellement à elle-même. L'évaluation partielle d'un appel de primitive consiste à évaluer partiellement ses deux opérandes. Si les deux opérandes s'évaluent partiellement à des constantes, et sont donc complètement évaluables, on leurs applique la primitive. Le résultat de l'évaluation partielle de l'appel est alors la constante correspondante. Sinon, le résultat est l'appel de

1. PLI ne dispose pas de pointeurs.

Int	Valeurs entières
$f \in Fun_2 = Int \times Int \rightarrow Int$	Fonctions entières binaires
$c \in Com$	Commandes
$e \in Exp$	Expressions
$\delta \in Mem^{ep} = Ident \rightarrow Int + \{\square\}$	États mémoire partiels

$$\mathcal{C}_{ep} : Com \rightarrow Mem^{ep} \rightarrow (Com \times Mem^{ep})$$

$$\mathcal{C}^{ep}[\mathbf{NOP}] = \lambda\delta. (\mathbf{NOP}, \delta)$$

$$\mathcal{C}^{ep}[\mathbf{SEQ}(c_1, c_2)] =$$

$$\lambda\delta. \mathbf{case } c'_1 \mathbf{ of}$$

$$\quad \mathbf{NOP} \rightarrow (c'_2, \delta'')$$

$$\quad - \rightarrow \mathbf{case } c'_2 \mathbf{ of}$$

$$\quad \quad \mathbf{NOP} \rightarrow (c'_1, \delta'')$$

$$\quad \quad - \rightarrow (\mathbf{SEQ}(c'_1, c'_2), \delta'')$$

$$\mathbf{where } (c'_1, \delta') = \mathcal{C}^{ep}[c_1]\delta$$

$$(c'_2, \delta'') = \mathcal{C}^{ep}[c_2]\delta'$$

$$\mathcal{C}^{ep}[\mathbf{AFF}(i, e)] = \lambda\delta. \mathbf{case } e' \mathbf{ of}$$

$$\quad \mathbf{CST}(n) \rightarrow (\mathbf{NOP}, \delta[i \mapsto \mathcal{N}[n]])$$

$$\quad - \rightarrow (\mathbf{AFF}(i, e'), \delta[i \mapsto \square])$$

$$\mathbf{where } e' = \mathcal{E}^{ep}[e]\delta$$

$$\mathcal{C}^{ep}[\mathbf{COND}(e, c_1, c_2)] =$$

$$\lambda\delta. \mathbf{case } e' \mathbf{ of}$$

$$\quad \mathbf{CST}(n) \rightarrow \mathbf{if } \mathcal{N}[n] \mathbf{ then } (c'_1, \delta') \mathbf{ else } (c'_2, \delta'')$$

$$\quad - \rightarrow (\mathbf{COND}(e', c'_1, c'_2), \mathit{joindre } \delta' \delta'')$$

$$\mathbf{where } e' = \mathcal{E}^{ep}[e]\delta$$

$$(c'_1, \delta') = \mathcal{C}^{ep}[c_1]\delta$$

$$(c'_2, \delta'') = \mathcal{C}^{ep}[c_2]\delta$$

$$\mathit{joindre} : Mem^{ep} \rightarrow Mem^{ep} \rightarrow Mem^{ep}$$

$$\mathit{joindre} = \lambda\delta_1 \lambda\delta_2 \lambda i. \mathbf{if } \delta_1(i) = \delta_2(i) \mathbf{ then } \delta_1(i) \mathbf{ else } \square$$

$$\mathcal{E}^{ep} : Exp \rightarrow Mem^{ep} \rightarrow Exp$$

$$\mathcal{E}^{ep}[\mathbf{CST}(n)] = \lambda\delta. \mathbf{CST}(n)$$

$$\mathcal{E}^{ep}[\mathbf{VAR}(i)] = \lambda\delta. \mathbf{if } \delta(i) \in Int \mathbf{ then } \mathbf{CST}(\mathcal{N}^{-1}(\delta(i))) \mathbf{ else } \mathbf{VAR}(i)$$

$$\mathcal{E}^{ep}[\mathbf{APP}(op, e_1, e_2)] =$$

$$\lambda\delta. \mathbf{case } e'_1 \mathbf{ of}$$

$$\quad \mathbf{CST}(n_1) \rightarrow \mathbf{case } e'_2 \mathbf{ of}$$

$$\quad \quad \mathbf{CST}(n_2) \rightarrow \mathbf{CST}(\mathcal{N}^{-1}(\mathcal{O}[op](\mathcal{N}[n_1], \mathcal{N}[n_2])))$$

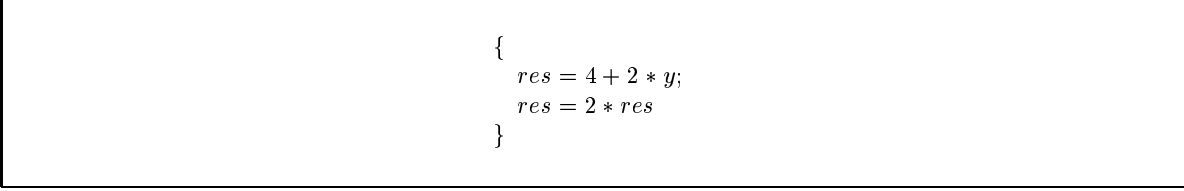
$$\quad \quad - \rightarrow \mathbf{APP}(op, e'_1, e'_2)$$

$$\quad - \rightarrow \mathbf{APP}(op, e'_1, e'_2)$$

$$\mathbf{where } e'_1 = \mathcal{E}^{ep}[e_1]\delta$$

$$e'_2 = \mathcal{E}^{ep}[e_2]\delta$$

FIG. 3.4 – Un évaluateur partiel en ligne pour PLI



```

{
  res = 4 + 2 * y;
  res = 2 * res
}

```

FIG. 3.5 – Le programme résiduel retourné par \mathcal{C}^{ep} pour notre exemple et δ_0

primitive construit avec les expressions résiduelles des deux opérandes.

L'évaluation partielle d'une instruction consiste en une instruction résiduelle et un nouvel état partiel. Puisque l'instruction peut contenir, elle-même, des instructions modifiant l'état partiel, sa propagation est nécessaire.

L'évaluation partielle d'une séquence consiste à évaluer partiellement les deux instructions en séquence. Si l'une des instruction s'évalue partiellement à l'instruction vide, et est donc totalement évaluable, le résultat de l'évaluation partielle de la séquence est celui de l'autre instruction. Sinon, le résultat est la séquence construite avec les deux instructions résiduelles des instructions. L'évaluation partielle d'une affectation consiste à évaluer partiellement l'expression en partie droite. Si l'expression s'évalue partiellement à une constante, on effectue l'affectation. Le résultat de l'évaluation partielle de l'affectation est alors l'instruction vide. Sinon, le résultat est l'affectation construite avec l'expression résiduelle de l'expression. L'évaluation partielle d'une conditionnelle consiste d'abord à évaluer partiellement le test. Si ce dernier s'évalue partiellement à une constante, le résultat de l'évaluation partielle de la conditionnelle est celui de la branche correspondant à la valeur du test. Sinon, on évalue partiellement les deux branches en parallèle (sans séquentialiser l'état partiel). Le résultat est alors la conditionnelle construite avec le test et les instructions résiduelles des branches. Les états issus de l'évaluation partielle des branches sont combinés par la fonction *joindre* afin de poursuivre l'analyse avec un état partiel cohérent. La jonction force une variable à être dynamique si les valeurs qui lui sont associées dans les deux états partiels sont en conflit. C'est-à-dire, si la variable est statique dans une branche et dynamique dans l'autre, ou si elle est statique dans les deux mais liée à des valeurs différentes.

Toutefois, la jonction des états mémoire lors du traitement d'une conditionnelle, sous contrôle dynamique, pose un problème supplémentaire. Cela ce produit quand une variable passe de l'état statique, à la fin d'une branche, à l'état dynamique, après la conditionnelle. Le code résiduel peut utiliser la variable après la conditionnelle puisqu'elle est dynamique. Cependant, cette dernière n'a pas de valeur significative dans le code résiduel puisqu'elle était statique à la fin de la branche. Une solution est de rajouter une affectation donnant une valeur à la variable dans le code résiduel [Mey91]. Ce problème étant commun aux analyses en ligne et hors ligne, il est traité globalement dans la section 3.2.4.

La figure 3.5 montre le résultat de la spécialisation de l'exemple de la figure 3.2 pour l'état

$$\begin{array}{ll}
tl \in Tl = \{Stat, Dyn\} & \text{Temps de liaison} \\
\gamma \in Mem^{tl} = Ident \rightarrow Tl & \text{Descriptions d'états partiels} \\
\\
\mathcal{C}^{tl} : Com \rightarrow Mem^{tl} \rightarrow (Tl \times Mem^{tl}) & \\
\mathcal{C}^{tl}[\mathbf{NOP}] & = \lambda\gamma. (Stat, \gamma) \\
\mathcal{C}^{tl}[\mathbf{SEQ}(c_1, c_2)] & = \lambda\gamma. ((tl_1 \sqcup tl_2, \gamma'') \\
& \quad \text{where } (tl_1, \gamma') = \mathcal{C}^{tl}[\![c_1]\!] \gamma \\
& \quad \quad (tl_2, \gamma'') = \mathcal{C}^{tl}[\![c_2]\!] \gamma' \\
\mathcal{C}^{tl}[\mathbf{AFF}(i, e)] & = \lambda\gamma. (tl, \gamma[i \mapsto tl]) \\
& \quad \text{where } tl = \mathcal{E}^{tl}[\![e]\!] \gamma \\
\mathcal{C}^{tl}[\mathbf{COND}(e, c_1, c_2)] & = \\
\lambda\gamma. \text{ case } tl \text{ of} & \\
\quad Stat \rightarrow (tl_1 \sqcup tl_2, \gamma' \sqcup \gamma'') & \\
\quad Dyn \rightarrow (Dyn, joindre^{tl} \gamma' \gamma'') & \\
\text{where } tl & = \mathcal{E}^{tl}[\![e]\!] \gamma \\
(tl_1, \gamma') & = \mathcal{C}^{tl}[\![c_1]\!] \gamma \\
(tl_2, \gamma'') & = \mathcal{C}^{tl}[\![c_2]\!] \gamma \\
\\
joindre^{tl} : Mem^{tl} \rightarrow Mem^{tl} \rightarrow Mem^{tl} & \\
\\
\mathcal{E}^{tl} : Exp \rightarrow Mem^{tl} \rightarrow Tl & \\
\mathcal{E}^{tl}[\mathbf{CST}(n)] & = \lambda\gamma. Stat \\
\mathcal{E}^{tl}[\mathbf{VAR}(i)] & = \lambda\gamma. \gamma(i) \\
\mathcal{E}^{tl}[\mathbf{APP}(op, e_1, e_2)] & = \lambda\gamma. \mathcal{E}^{tl}[\![e_1]\!] \gamma \sqcup \mathcal{E}^{tl}[\![e_2]\!] \gamma
\end{array}$$

FIG. 3.6 – Un analyseur de temps de liaison pour PLI

partiel $\delta_0 = \lambda i. \text{if } i = x \text{ then } 1 \text{ else } \square$.

3.2.3 Évaluation partielle hors ligne pour PLI

Dans cette section, nous présentons un évaluateur partiel hors ligne pour PLI. Sa phase de pré-spécialisation comporte une analyse de temps de liaison sensible au flot² ainsi qu'une analyse d'actions. Ensuite, nous présentons deux exploitations possibles des arbres d'actions obtenus : interprétation ou compilation.

Comme pour l'évaluation partielle en ligne, ce spécialisteur calcule un programme résiduel à partir d'un programme et d'un état partiel.

Analyse de temps de liaison

L'analyse de temps de liaison consiste à annoter un arbre syntaxique avec des temps de liaison à l'aide d'une description de l'état partiel. Les temps de liaison sont des éléments de l'ensemble $Tl = \{Stat, Dyn\}$. L'ensemble Tl est ordonné par $Stat \sqsubset Dyn$ et on note \sqcup l'opérateur délivrant la plus petite borne supérieure de deux éléments de Tl .

La figure 3.6 donne un algorithme calculant les temps de liaison en chaque nœud d'un arbre syntaxique pour une description abstraite d'état partiel donnée. Les descriptions d'états partiels sont des éléments de l'ensemble $Mem^{tl} = Ident \rightarrow Tl$. On étend l'opérateur \sqcup sur les descriptions d'état partiel par $\gamma_1 \sqcup \gamma_2 = \lambda i. \gamma_1(i) \sqcup \gamma_2(i)$.

L'analyse de temps de liaison, pour une instruction, consiste à calculer le temps de liaison de la construction ainsi qu'une nouvelle description de l'état partiel. On assure ainsi la sensibilité au flot. Le temps de liaison d'une instruction est statique chaque fois que l'évaluateur partiel en ligne retourne l'instruction vide pour cette instruction. Il est dynamique dans les autres cas.

Le temps de liaison d'une séquence est la plus petite borne supérieure des temps de liaison des instructions qui la composent. La description de l'état partiel est mis à jour en séquence. Le temps de liaison d'une affectation est celui de l'expression en partie droite. La description de l'état partiel est modifiée en conséquence. Le temps de liaison d'une conditionnelle, dont le test est statique, est la plus petite borne supérieure des temps de liaison des deux branches. L'approximation est nécessaire car il est impossible de calculer le test (nous n'avons pas les valeurs concrètes) et donc de déterminer la branche qui sera choisie à la spécialisation. On applique la même opération aux descriptions d'état partiel retournées. Le temps de liaison d'une conditionnelle, dont le test est dynamique, est aussi dynamique. La nouvelle description d'état partiel est alors calculée à l'aide de la fonction *joindre*^{tl} qui force toutes les variables modifiées dans les branches à devenir dynamiques et laisse les autres inchangées. Cette approximation s'explique par le fait que la valeur des variables est inconnue. De fait, il est impossible de savoir si les affectations contenues dans les branches sont en conflit. La fonction *joindre*^{tl} se résumant essentiellement à une analyse d'utilisation, nous n'en donnerons pas, ici, la définition.

Comme pour l'analyse en ligne, il convient d'ajouter des affectations explicitant les variables qui passent de l'état statique à l'état dynamique à cause de la jonction des états mémoire lors du traitement d'une conditionnelle. Contrairement à l'analyse en ligne, les conditionnelles sous contrôle statique sont aussi sujettes à ce traitement car elles nécessitent aussi une jonction des états. Ce point est détaillé dans la section 3.2.4.

Puisque les expressions de PLI ne font pas d'effet de bord, l'analyse de temps de liaison pour une expression consiste simplement à calculer son temps de liaison. Le temps de liaison d'une expression est statique chaque fois que l'évaluateur partiel en ligne retourne une constante pour cette expression. Il est dynamique dans les autres cas.

2. La notions de sensibilité au contexte n'a ici pas de sens puisque nous ne disposons pas de procédures

```

{
   $l = 2 * x;$ 
  si ( $l == 2$ ) alors  $res = l * l + 2 * y$  sinon {  $l = y * 2;$   $res = l * l$  };
   $res = 2 * res$ 
}

```

FIG. 3.7 – Les temps de liaison de notre exemple

$i \in Ident$	Identificateurs
$tl \in Tl$	Temps de liaison
$n \in Num$	Constantes
$op \in OpBin$	Opérateurs binaires
$c^{tl} \in Com^{tl} \quad : : =$	
	NOP
	SEQ(c_1^{tl}, c_2^{tl})
	AFF(i, e^{tl})
	COND($e^{tl}, c_1^{tl}, c_2^{tl}$)
$e^{tl} \in Exp^{tl} \quad : : =$	
	CST(n)
	VAR(i, tl)
	APP(op, e_1^{tl}, e_2^{tl})

FIG. 3.8 – La syntaxe décoré de temps de liaison

Le temps de liaison d'une constante est toujours statique. Celui d'une variable est obtenu à l'aide de la description de l'état partiel. Le temps de liaison d'un appel de primitive est la plus petite borne supérieure des temps de liaison de ses opérandes.

La figure 3.7 donne les temps de liaison pour notre exemple et la description d'état partiel $\gamma_0 = \lambda i. \mathbf{if} \ i = \llbracket x \rrbracket \ \mathbf{then} \ Stat \ \mathbf{else} \ Dyn$. Les constructions statiques sont soulignées alors que les autres sont laissées telles quelles.

Dans la pratique, l'algorithme de la figure 3.6 retourne aussi un arbre syntaxique annoté avec les temps de liaison calculés. Parce que notre évaluateur partiel dispose d'une analyse d'actions, seules les occurrences des identificateurs doivent être annotées. La syntaxe de ces arbres annotés est décrite dans la figure 3.8. Elle est identique à la syntaxe de PLI, sauf pour la règle définissant les occurrences des identificateurs.

Analyse d'actions

Notre spécialiseur effectue une analyse d'actions calculant un arbre d'actions à partir d'un arbre annoté de temps de liaison. On définit une nouvelle syntaxe pour les arbres d'actions.

$i \in Ident$	Identificateurs		
$n \in Num$	Constantes		
$op \in OpBin$	Opérateurs binaires		
	Syntaxe abstraite		Syntaxe concrète
$c^a \in Com^a$	$::= \mathbf{EV}(c)$ $ \mathbf{ID}(c)$ $ \mathbf{RED_SEQg}(c, c^a)$ $ \mathbf{RED_SEQd}(c^a, c)$ $ \mathbf{REC_SEQ}(c_1^a, c_2^a)$ $ \mathbf{REC_AFF}(i, e^a)$ $ \mathbf{RED_COND}(e, c_1^a, c_2^a)$ $ \mathbf{REC_COND}(e^a, c_1^a, c_2^a)$		c^{ev} c^{id} $\{c; c^a\}^{redg}$ $\{c^a; c\}^{redd}$ $\{c_1^a; c_2^a\}^{rec}$ $i =^{rec} e^a$ $si^{red}(e) \mathbf{alors}^{red} c_1^a$ $ \quad \mathbf{sinon}^{red} c_2^a$ $si^{rec}(e^a) \mathbf{alors}^{rec} c_1^a$ $ \quad \mathbf{sinon}^{rec} c_2^a$
$e^a \in Exp^a$	$::= \mathbf{EV}(e)$ $ \mathbf{ID}(e)$ $ \mathbf{REC_APP}(op, e_1^a, e_2^a)$		e^{ev} e^{id} $e_1^a \mathbf{op}^{rec} e_2^a$

FIG. 3.9 – La syntaxe des actions pour PLI

Cette syntaxe, décrite dans la figure 3.9 combine les constructeurs de la syntaxe initiale et les actions.

Les deux premières règles pour les instructions et les expressions correspondent aux actions génériques *Évalue* et *Identité* décrites précédemment. Les autres résultent de la combinaison des constructeurs de la syntaxe de PLI et des actions *Réduit* et *Reconstruit*.

Une séquence peut être réduite à gauche ou à droite si l'un des deux opérandes disparaît à la spécialisation. L'opérande qui disparaît n'est pas annotée car il est forcément statique. Sinon la séquence est reconstruite. Il n'y a qu'une action de reconstruction pour les affectations. Une affectation réduite est annotée avec l'action *Évalue*. Une conditionnelle est, soit réduite, soit reconstruite. Comme pour la séquence, le test d'une conditionnelle réduite n'est pas annoté car il est forcément statique.

Il n'y a qu'une seule action spécifique aux expressions: la reconstruction d'un appel de primitive. Tous les autres cas sont annotés avec les actions génériques *Évalue* ou *Identité*.

Les figures 3.10 et 3.11 donnent une sémantique permettant de calculer un arbre d'actions à partir d'un arbre annoté de temps de liaison. La fonction «-» supprime les annotations d'un arbre de la syntaxe annotée pour en faire un arbre de la syntaxe initiale. La définition de cette fonction est triviale et n'est pas précisée ici.

Une instruction vide est toujours annotée avec l'action *Évalue*. Une séquence dont les deux

$tl \in Tl = \{Stat, Dyn\}$	Temps de liaison
$c^a \in Com^a$	Commandes annotées d'actions
$e^a \in Exp^a$	Expression annotées d'actions
$\mathcal{C}^a : Com^{tl} \rightarrow Com^a$	
$\mathcal{C}^a[\mathbf{NOP}]$	$= \mathbf{EV}(\mathbf{NOP})$
$\mathcal{C}^a[\mathbf{SEQ}(c_1^{tl}, c_2^{tl})]$	$=$
case c_1^a of	
$\mathbf{EV}(_)$	\rightarrow case c_2^a of
$\mathbf{EV}(_)$	$\rightarrow \mathbf{EV}(\mathbf{SEQ}(\overline{c_1^{tl}}, \overline{c_2^{tl}}))$
-	$\rightarrow \mathbf{RED_SEQg}(c_1^a, c_2^a)$
$\mathbf{ID}(_)$	\rightarrow case c_2^a of
$\mathbf{EV}(_)$	$\rightarrow \mathbf{RED_SEQd}(c_1^a, \overline{c_2^{tl}})$
$\mathbf{ID}(_)$	$\rightarrow \mathbf{ID}(\mathbf{SEQ}(\overline{c_1^{tl}}, \overline{c_2^{tl}}))$
-	$\rightarrow \mathbf{REC_SEQ}(c_1^a, c_2^a)$
-	\rightarrow case c_2^a of
$\mathbf{EV}(_)$	$\rightarrow \mathbf{RED_SEQd}(c_1^a, \overline{c_2^{tl}})$
-	$\rightarrow \mathbf{REC_SEQ}(c_1^a, c_2^a)$
where $c_1^a = \mathcal{C}^a[c_1^{tl}]$	
$c_2^a = \mathcal{C}^a[c_2^{tl}]$	
$\mathcal{C}^a[\mathbf{AFF}(i, e^{tl})]$	$=$ case e^a of
$\mathbf{EV}(_)$	$\rightarrow \mathbf{EV}(\mathbf{AFF}(i, \overline{e^{tl}}))$
$\mathbf{ID}(_)$	$\rightarrow \mathbf{ID}(\mathbf{AFF}(i, e^{tl}))$
-	$\rightarrow \mathbf{REC_AFF}(i, e^a)$
where $e^a = \mathcal{C}^a[e^{tl}]$	
$\mathcal{C}^a[\mathbf{COND}(e^{tl}, c_1^{tl}, c_2^{tl})]$	$=$
case e^a of	
$\mathbf{EV}(_)$	\rightarrow case c_1^a of
$\mathbf{EV}(_)$	\rightarrow case c_2^a of
$\mathbf{EV}(_)$	$\rightarrow \mathbf{EV}(\mathbf{COND}(\overline{e^{tl}}, \overline{c_1^{tl}}, \overline{c_2^{tl}}))$
-	$\rightarrow \mathbf{RED_COND}(e^{tl}, c_1^a, c_2^a)$
-	$\rightarrow \mathbf{RED_COND}(\overline{e^{tl}}, c_1^a, c_2^a)$
$\mathbf{ID}(_)$	\rightarrow case c_1^a of
$\mathbf{ID}(_)$	\rightarrow case c_2^a of
$\mathbf{ID}(_)$	$\rightarrow \mathbf{ID}(\mathbf{COND}(\overline{e^{tl}}, \overline{c_1^{tl}}, \overline{c_2^{tl}}))$
-	$\rightarrow \mathbf{REC_COND}(e^a, c_1^a, c_2^a)$
-	$\rightarrow \mathbf{REC_COND}(e^a, c_1^a, c_2^a)$
-	$\rightarrow \mathbf{REC_COND}(e^a, c_1^a, c_2^a)$
where $e^a = \mathcal{C}^a[e^{tl}]$	
$c_1^a = \mathcal{C}^a[c_1^{tl}]$	
$c_2^a = \mathcal{C}^a[c_2^{tl}]$	

FIG. 3.10 – Un analyseur d'actions pour PLI (instructions)

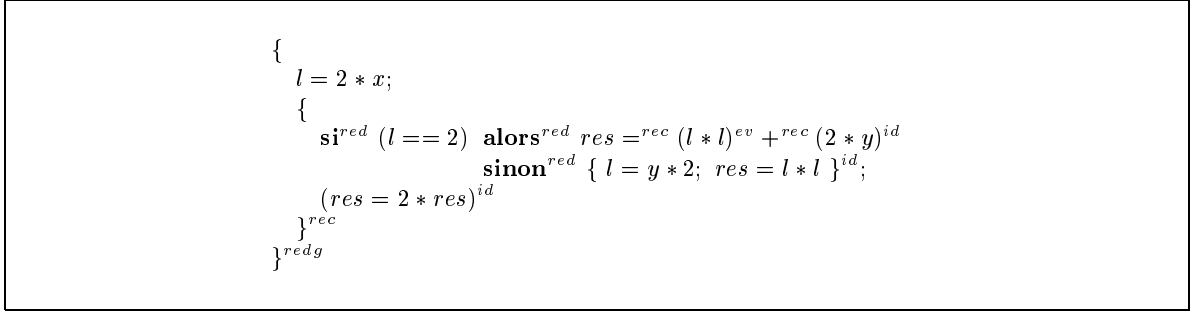
$\mathcal{E}^a : Exp^{tl} \rightarrow Exp^a$	
$\mathcal{E}^a[\mathbf{CST}(n)]$	$= \mathbf{EV}(\mathbf{CST}(n))$
$\mathcal{E}^a[\mathbf{VAR}(i, tl)]$	$= \mathbf{if } tl = Stat \mathbf{ then EV(VAR}(i)) \mathbf{ then ID(VAR}(i))$
$\mathcal{E}^a[\mathbf{APP}(op, e_1^{tl}, e_2^{tl})]$	$=$
case e_1^a of	
$\mathbf{EV}(_) \rightarrow$	case e_2^a of
$\mathbf{EV}(_) \rightarrow$	$\mathbf{EV}(\mathbf{APP}(op, \overline{e_1^{tl}}, \overline{e_2^{tl}}))$
$_ \rightarrow$	$\mathbf{REC_APP}(op, e_1^a, e_2^a)$
$\mathbf{ID}(_) \rightarrow$	case e_2^a of
$\mathbf{ID}(_) \rightarrow$	$\mathbf{ID}(\mathbf{APP}(op, \overline{e_1^{tl}}, \overline{e_2^{tl}}))$
$_ \rightarrow$	$\mathbf{REC_APP}(op, e_1^a, e_2^a)$
$_ \rightarrow$	$\mathbf{REC_APP}(op, e_1^a, e_2^a)$
where $e_1^a =$	$\mathcal{E}^a[e_1^{tl}]$
$e_2^a =$	$\mathcal{E}^a[e_2^{tl}]$

FIG. 3.11 – Un analyseur d'actions pour PLI (expressions)

instructions sont annotées avec l'action *Évalue* (respectivement *Identité*) est aussi annotée avec l'action *Évalue* (respectivement *Identité*). Si l'un des opérande est annoté avec l'action *Évalue*, alors la séquence est annotée avec une de ses actions de réduction. Sinon, elle est annotée avec son action de reconstruction. Si l'expression en partie droite d'une affectation est annotée avec l'action *Évalue* (respectivement *Identité*), alors l'affectation est aussi annotée avec l'action *Évalue* (respectivement *Identité*). Sinon, l'affectation est annotée avec son action de reconstruction. Une conditionnelle dont le test est annoté avec l'action *Évalue*, est annotée soit avec l'action *Évalue*, si les deux branches sont annotées avec l'action *Évalue*, soit avec l'action de réduction dans les autres cas. Une conditionnelle, dont le test est annoté avec l'action *Identité*, est annotée soit avec l'action *Identité*, si les deux branches sont annotées avec l'action *Identité*, soit avec l'action de reconstruction dans les autres cas. Une conditionnelle dont le test est annoté avec une action de reconstruction est aussi annotée avec l'action de reconstruction.

Une expression constante est toujours annotée avec l'action *Évalue*. Une occurrence d'identificateur est annotée, soit avec l'action *Évalue* soit avec l'action *Identité* en fonction du temps de liaison de la variable. Un appel de primitive est annoté avec l'action *Évalue* (respectivement *Identité*) si ses deux opérandes sont annotés avec l'action *Évalue* (respectivement *Identité*). Dans les autres cas, l'appel de primitive est annoté avec son action de réduction.

Il est possible d'optimiser les règles pour l'annotation des expressions. En effet, le fragment $2 * y$, dans lequel l'occurrence de l'identificateur y est dynamique, produit l'arbre d'actions $2^{ev} *^{rec} y^{id}$. Si on autorise les constantes à être annotées avec l'action *Identité* plutôt qu'avec l'action *Évalue*, en fonction des contextes d'apparition, nous pourrions obtenir l'arbre d'actions $(2 * y)^{id}$.

FIG. 3.12 – L'arbre d'actions retourné par \mathcal{C}^a pour notre exemple

La figure 3.12 donne l'arbre d'actions obtenu pour notre exemple en tenant compte de cette optimisation. Toutefois, nous ne décrivons pas cette optimisation plus en détail.

Spécialisation

Après l'analyse d'actions, les valeurs concrètes de l'état partiel sont utilisées pour produire la spécialisation correspondante. La figure 3.13 définit la sémantique des actions de PLI. Cette sémantique calcule le code résiduel à partir d'un arbre d'actions et d'un état mémoire initial. Ce dernier est construit à partir de l'état mémoire partiel en remplaçant les occurrences du symbole « \square » par 0^3 .

La spécialisation d'une instruction retourne une instruction résiduelle ainsi qu'un nouvel état mémoire. En effet, l'état évolue en fonction des affectations annotées avec l'action *Évalue* contenues dans l'instruction initiale.

Une instruction annotée avec l'action *Évalue* est passée à l'évaluateur standard et disparaît du code résiduel. Le nouvel état mémoire est celui retourné par l'évaluateur standard. Une instruction annotée avec l'action *Identité* est laissée telle quelle. L'état mémoire est inchangé. Pour une séquence se réduisant à gauche, on commence par évaluer l'opérande gauche avec l'évaluateur standard. Puis, on spécialise l'opérande droite avec l'état mémoire obtenu. La réduction à droite est similaire. La reconstruction d'une séquence ainsi que celle d'une affectation s'expliquent d'elles mêmes. Pour une conditionnelle réductible, on commence par évaluer le test avec l'évaluateur standard. Puis, on spécialise la branche choisie selon le résultat obtenu. Le seul problème causé par la reconstruction d'une conditionnelle est la gestion de l'état mémoire. Les deux branches sont spécialisées en parallèle. Nous obtenons donc deux nouveaux états mémoire. Ces deux états peuvent être en conflit pour certaines variables. Cependant, l'analyse de temps de liaison aura alors pris soin d'annoter les occurrences ultérieures de variables pouvant être en conflit comme dynamiques. Les états mémoire résultant de l'évaluation des deux branches sont donc équivalents. On choisit alors arbitrairement l'état mémoire

3. Les identificateurs dynamiques ne sont jamais référencés.

$c \in Com$	Commandes
$e \in Exp$	Expressions
$\sigma \in Mem = Ident \rightarrow Int$	États mémoire
$\mathcal{C}^{spec} : Com^a \rightarrow Mem \rightarrow (Com \times Mem)$	
$\mathcal{C}^{spec}[\mathbf{EV}(c)]$	$= \lambda\sigma. (\mathbf{NOP}, \mathcal{C}[c]\sigma)$
$\mathcal{C}^{spec}[\mathbf{ID}(c)]$	$= \lambda\sigma. (c, \sigma)$
$\mathcal{C}^{spec}[\mathbf{RED_SEQg}(c, c^a)]$	$= \lambda\sigma. \mathcal{C}^{spec}[c^a](\mathcal{C}[c]\sigma)$
$\mathcal{C}^{spec}[\mathbf{RED_SEQd}(c^a, c)]$	$= \lambda\sigma. (c', \mathcal{C}[c]\sigma')$
	where $(c', \sigma') = \mathcal{C}^{spec}[c^a]\sigma$
$\mathcal{C}^{spec}[\mathbf{REC_SEQ}(c_1^a, c_2^a)]$	$= \lambda\sigma. (\mathbf{SEQ}(c_1, c_2), \sigma')$
	where $(c_1, \sigma') = \mathcal{C}^{spec}[c_1^a]\sigma$
	$(c_2, \sigma'') = \mathcal{C}^{spec}[c_2^a]\sigma'$
$\mathcal{C}^{spec}[\mathbf{REC_AFF}(i, e^a)]$	$= \lambda\sigma. (\mathbf{AFF}(i, \mathcal{E}^{spec}[e^a]\sigma), \sigma)$
$\mathcal{C}^{spec}[\mathbf{RED_COND}(e, c_1^a, c_2^a)]$	$= \lambda\sigma. \text{if } \mathcal{E}[e]\sigma \text{ then } \mathcal{C}^{spec}[c_1^a]\sigma \text{ else } \mathcal{C}^{spec}[c_2^a]\sigma$
$\mathcal{C}^{spec}[\mathbf{REC_COND}(e^a, c_1^a, c_2^a)]$	$= \lambda\sigma. (\mathbf{COND}(\mathcal{E}^{spec}[e^a]\sigma, c_1, c_2), \sigma')$
	where $(c_1, _) = \mathcal{C}^{spec}[c_1^a]\sigma$
	$(c_2, \sigma') = \mathcal{C}^{spec}[c_2^a]\sigma$
$\mathcal{E}^{spec} : Exp^a \rightarrow Mem \rightarrow Exp$	
$\mathcal{E}^{spec}[\mathbf{EV}(e)]$	$= \lambda\sigma. \mathbf{CST}(\mathcal{N}^{-1}(\mathcal{E}[e]\sigma))$
$\mathcal{E}^{spec}[\mathbf{ID}(e)]$	$= \lambda\sigma. e$
$\mathcal{E}^{spec}[\mathbf{REC_APP}(op, e_1^a, e_2^a)]$	$= \lambda\sigma. \mathbf{APP}(op, \mathcal{E}^{spec}[e_1^a]\sigma, \mathcal{E}^{spec}[e_2^a]\sigma)$

FIG. 3.13 – La sémantique des actions de PLI

<pre> { x = 1; si (< Dyn >) alors {x = < Dyn >; ...} sinon {x = x + 1; ...}; e = x + 1 } </pre>	<pre> { x = 1; si (< Dyn >) alors {x = x + 2; ...} sinon {x = x + 1; ...}; e = x + 1 } </pre>
---	---

FIG. 3.14 – Deux exemples causant des problèmes

résultant de l'évaluation de la deuxième branche.

Une expression annotée avec l'action *Évalue* est passée à l'évaluateur standard et l'expression résiduelle est alors la constante correspondante. Une expression annotée *Identité* est laissée telle quelle. La reconstruction d'un appel de primitive consiste à reconstruire le nœud avec les expressions résiduelles des deux opérandes.

La spécialisation de l'exemple présenté dans la figure 3.2 produit alors le même résultat que le spécialiseur en ligne.

On montrera dans le chapitre 6 comment on peut construire un spécialiseur dédié à partir d'un arbre d'actions.

3.2.4 La notion d'explicateur

Comme nous l'avons précisé plus haut, un problème se pose chaque fois qu'une variable passe de l'état statique, à la fin d'une branche de conditionnelle, à l'état dynamique, après la conditionnelle à cause de la jonction des états mémoire. Après la conditionnelle, le code résiduel peut utiliser cette variable puisqu'elle devient dynamique. Cependant, cette variable n'a pas de définition dans le code résiduel car elle était statique à la fin de la branche.

Pour le spécialiseur en ligne, seules les conditionnelles sous contrôle dynamique sont concernées car les autres ne nécessitent pas de jonction. La jonction force une variable à être dynamique lorsque qu'elle est statique dans une branche et dynamique dans l'autre, ou lorsqu'elle est statique dans les deux branches mais possède des valeurs différentes. La figure 3.14 montre deux fragments de code illustrant ces situations.

La figure 3.15 donne le code résiduel correspondant, tel qu'il est retourné par notre spécialiseur en ligne (valable pour n'importe quel état partiel à cause de la première affectation). On constate aisément que ces deux programmes résiduels sont faux. Pour le premier, si l'on passe dans la branche fautive, la variable x n'a pas de valeur et pour le deuxième, il n'en a jamais.

Une solution à ce problème consiste à ajouter une affectation donnant une valeur à la variable dans le programme résiduel [Mey91]. Cette affectation est insérée à la fin de la branche

<pre> { si (< Dyn >) alors {x =< Dyn >; ...} sinon {...}; e = x + 1 } </pre>	<pre> { si (< Dyn >) alors {...} sinon {...}; e = x + 1 } </pre>
--	--

FIG. 3.15 – Le code résiduel retourné pour les exemples de la figure 3.14

<pre> { si (< Dyn >) alors {x =< Dyn >; ...} sinon {...; x = 2}; e = x + 1 } </pre>	<pre> { si (< Dyn >) alors {...; x = 3} sinon {...; x = 2}; e = x + 1 } </pre>
---	--

FIG. 3.16 – Le code résiduel de nos exemples avec des explicateurs

où le problème se pose. Ces affectations sont généralement appelées *explicateurs*. La figure 3.16 donne des programmes résiduels corrects pour les deux exemples ci-dessus en y ajoutant des explicateurs.

On procède de la même façon pour le spécialiste hors ligne en ajoutant aussi des explicateurs pour les conditionnelles sous contrôle statique. En effet, l'analyse de temps de liaison procède aussi à la jonction des états issus de l'évaluation partielle des branches d'une conditionnelle dynamique. Les explicateurs sont ajoutés, comme pour la spécialisation en ligne, lorsqu'une variable passe de l'état statique à l'état dynamique.

Hornof, Noyé et Consel ont montré qu'une analyse de temps de liaison sophistiquée permet de se passer des explicateurs [HNC96]. Elle est constituée de deux analyses couplées : une analyse avant et une analyse arrière. Elle permet à l'analyse d'actions de reconstruire et d'évaluer certaines affectations qui ne seraient normalement qu'évaluées. Ainsi, les variables concernées disposent de valeurs significatives dans le code résiduel.

Chapitre 4

Spécialisation dynamique

Les compilateurs effectuent traditionnellement des optimisations, soit indépendantes de la valeur réelle des variables, soit dépendantes des constantes du programme. Il est fréquent que certaines variables restent constantes durant toute l'exécution d'un fragment du programme, mais soient inconnues avant cette exécution. Ces variables sont appelées *constantes d'exécution*.

Un compilateur traditionnel, produisant tout le code statiquement, ne peut pas exploiter ces constantes d'exécution car il ne dispose pas de leurs valeurs. Par contre, si on peut produire du code dynamiquement, un fragment de programme peut alors être spécialisé par rapport à des constantes d'exécution. Nous appellerons, *spécialisation dynamique*, ce cas particulier de production dynamique de code et, *spécialiseur dynamique* le code chargé de la production des spécialisations.

À priori, la spécialisation dynamique semble simple à réaliser. Il suffit de produire le code source des spécialisations à l'exécution à l'aide de techniques comme l'évaluation partielle. Puis on invoque un compilateur pour obtenir le code exécutable des spécialisations. Toutefois la compilation est une opération coûteuse qui pourrait annuler l'effet de cette optimisation. Cette technique n'est donc utilisable que dans les cas où le code spécialisé est utilisé un grand nombre de fois.

Plusieurs paramètres entrent en jeu pour calculer le gain, ou la perte, engendrés par la spécialisation dynamique d'un fragment de code. Si l'exécution du fragment coûte un temps t et qu'il est exécuté n fois, le coût total de l'opération est $t * n$. Si la production de la spécialisation coûte t_{gen} et que l'exécution du code obtenu coûte t_s , le coût total avec spécialisation dynamique est $t_{gen} + n * t_s$. L'accélération est donc $a = \frac{t}{t_s}$ et l'amortissement, c'est-à-dire la valeur minimale de n pour laquelle l'opération est rentable, est $\frac{t_{gen}}{t-t_s}$, où encore, $\frac{t_{gen}}{t_s(a-1)}$.

L'amortissement est un facteur primordial car c'est lui qui détermine le champ d'application de la méthode. L'accélération détermine le gain une fois que la production du code dynamique est amortie. Un des objectifs essentiels des systèmes de spécialisation dynamique est donc de minimiser l'amortissement. Il faut produire rapidement du code spécialisé efficace.

Ces deux objectifs sont contradictoires car la production de code hautement optimisé est une opération coûteuse. Il faut donc trouver un compromis.

Certains systèmes, appelés *compilateurs dynamiques* [LL96, APC⁺96], repoussent certaines phases de la compilation à l'exécution afin de mieux optimiser le code en fonction des valeurs des constantes d'exécution. Le spécialiseur dynamique n'est pas un compilateur complet mais il inclut des analyses coûteuses qui se trouvent normalement dans les compilateurs. En ce sens, ces compilateurs dynamiques favorisent la qualité du code produit. Le problème est que le temps de production du code spécialisé augmente de façon non linéaire par rapport à sa qualité. On gagne beaucoup mais il y a encore plus à récupérer.

L'approche adoptée par les compilateurs dynamiques ne semble donc pas satisfaisante pour assurer de faibles amortissements. La thèse soutenue dans ce document est qu'il vaut mieux avoir de faibles temps de production même si le code spécialisé est de moins bonne qualité. Cette faiblesse devient insignifiante lorsque le degré de spécialisation du fragment de programme considéré est important. D'autre part, cette approche rend possible l'utilisation de la spécialisation dynamique dans les cas où le code spécialisé n'est exécuté qu'un petit nombre de fois. Nous proposons, dans le chapitre 5, une méthode basée sur ce principe. Elle ne nécessite aucune opération de compilation lors de la production des spécialisations car tout le code est compilé avant l'exécution. De fait, la production dynamique est très rapide et son amortissement très bas.

Le reste de ce chapitre présente deux systèmes généraux de production dynamique de code pouvant être utilisés pour écrire manuellement des spécialiseurs dynamiques ainsi que deux compilateurs dynamiques.

4.1 Systèmes généraux

Nous présentons ici deux systèmes généraux permettant de faire de la production dynamique de code. Bien qu'ils permettent d'écrire, manuellement, des spécialiseurs dynamiques, ils ne s'agit pas de systèmes capables de les produire automatiquement. Ils fournissent simplement les outils pour le faire indépendamment de la machine cible.

4.1.1 DCG

Le système DCG, proposé par Engler et Proebsting [EP94], permet au programmeur d'écrire des applications pratiquant la production dynamique de code indépendamment de la machine cible. Comme le compilateur LCC [FH91], sur lequel il est construit, DCG est rapidement reconfigurable. Cette opération ne nécessite aucune modification de ses clients puisque la spécification du code dynamique est indépendante de l'architecture cible. Malgré la présence d'un générateur de code à l'exécution, DCG est relativement rapide.

Le système DCG consiste en une bibliothèque de fonctions permettant à ses clients de produire du code dynamiquement. La procédure est la plus petite entité de code que l'on

puisse produire dynamiquement avec DCG. Le programmeur spécifie le code dynamique à l'aide d'une forêt d'expressions de la représentation intermédiaire (RI) du compilateur LCC. Il spécifie, de même la déclaration des paramètres et des variables locales de la procédure. Ces déclarations sont construites en invoquant certaines fonctions de DCG. DCG inclut aussi des fonctions pour créer facilement tous les nœuds légaux de la représentation intermédiaire de LCC. Il offre également des fonctions permettant de construire les objets fréquemment utilisés comme les constantes, les adresses et les références aux variables locales de la fonction. La forêt et la déclaration des paramètres et des variables locales sont passés au générateur de code de DCG qui compile la représentation intermédiaire. Finalement, le générateur de code retourne un pointeur sur le code exécutable dynamiquement créé.

DCG inclut un mécanisme d'allocation de registres indépendant de la machine cible. Le programmeur spécifie une priorité pour chaque candidat et DCG alloue alors les registres en fonction de cette spécification. S'il y a trop de candidats, les moins prioritaires sont maintenus dans la pile. L'avantage de cette technique est son indépendance vis-à-vis d'une architecture ainsi que sa rapidité. Cependant, elle demande beaucoup au programmeur.

La production de code commence par la sélection d'instructions qui utilisent des systèmes de réécritures ascendants (BURS) pour traduire rapidement la représentation intermédiaire en code machine [FA91]. Les optimisations globales sont du ressort de l'utilisateur et le générateur de code de DCG ne fait que quelques optimisations locales. Ce choix est grandement justifié par le coût excessif de certaines optimisations qui rendrait le système inutilisable car beaucoup trop lent. Ensuite, DCG résout les branchements et procède à l'émission du code binaire. Les auteurs estiment, qu'en moyenne, il faut 350 instructions pour en produire une.

Bien qu'indépendante de la machine cible, l'utilisation de DCG est laborieuse. En effet, le langage de spécification du code dynamique est de bas niveau et son utilisation directe comporte les mêmes risques (fiabilité, maintenance, etc.) que l'utilisation d'un langage machine. De plus, le programmeur doit faire lui-même l'allocation de registres ainsi que certaines optimisations. Parce que DCG invoque un générateur de code à l'exécution, le temps de production du code dynamique est relativement lent.

4.1.2 Tick C

Engler, Hsieh et Kaashoek ont proposé un langage, nommé Tick C [EHK96]. Il s'agit d'une extension au langage ANSI C permettant au programmeur de calculer des programmes à l'exécution. Le code dynamique est spécifié à l'aide d'un mécanisme de macros ressemblant à celui du langage LISP. La mise en œuvre initiale de Tick C produisait du code pour le système DCG. Poletto, Engler et Kaashoek ont développé ensuite un compilateur indépendant de DCG [PEK96].

Présentation

Tick C ajoute deux constructeurs de type à l'ANSI C : **cspec** et **vspec**. Ces constructeurs s'utilisent comme le constructeur de pointeurs et permettent de typer les spécifications de code dynamique. Le type auquel ils s'appliquent est celui du code spécifié. Comme un pointeur, une spécification de code dynamique peut être dérépérée pour passer de la spécification au code lui-même. La même règle que pour les pointeurs est alors utilisée pour typer le résultat obtenu.

Le code dynamique est spécifié à l'aide de l'opérateur **backquote**, noté «`'`». Cet opérateur peut être appliqué à une expression comme à une instruction. La spécification obtenue est alors de type τ **cspec**, où τ est le type du code spécifié. Par exemple les expressions

```
'4
'printf ("tick")
'{ int i; for (i = 1; i < 10; i++) printf ("coucou"); }
```

sont des spécifications légales. La première est de type **int cspec** alors que les deux autres sont de type **void cspec**. Ce mécanisme est à un niveau : le code dynamique ne peut pas spécifier, à son tour, du code dynamique. En d'autres termes, les opérateurs **backquote** ne peuvent pas être emboîtés. Le code dynamique est autorisé à contenir des variables libres et capture alors les variables du code statique englobant. Le mécanisme de liaison est lexical. L'utilisation d'une variable en dehors de sa portée conduit à des résultats aléatoires. Les instructions ne sont pas autorisées à transférer le flot de contrôle en dehors de la **backquote** qui les définit.

Tick C permet aussi de spécifier des variables (*left-values* de C) dynamiques. Elles permettent au code dynamique de disposer de paramètres. Une telle spécification est de type τ **vspec**, où τ est le type de la variable spécifiée. Par exemple, la déclaration :

```
int vspec i;
```

permet de spécifier une variable dynamique entière.

Le dérépérage des spécifications de code s'effectue à l'aide de l'opérateur @. Il permet d'insérer des spécifications dans d'autres spécifications. Son utilisation n'est valide qu'à l'intérieur d'une **backquote** expression. Par exemple, on peut écrire :

```
int cspec c0 = '4;
int cspec c1 = '8;
int cspec c2 = '@c0 + @c1;
```

La dernière déclaration est équivalente à :

```
int cspec c2 = '(4 + 8);
```

L'opérateur @ peut aussi s'appliquer à une fonction retournant un objet de type **cspec** ou de type **vspec**. Le résultat retourné par la fonction est alors inséré dans la spécification en cours.

```

typedef int vect[TAILLE_MAX];

void cspec construit_produit(vect u, int taille)
{
    vect vspec v = param (0, vect);
    int cspec prod = 0;

    for (i = 0; i < taille; i++)
        prod = '@prod + $(u[i]) * (@v)[$i];

    return 'return @prod;;
}

```

FIG. 4.1 – Un exemple de programme Tick C

L'opérateur \$ permet l'insertion de valeurs dynamiques dans le code dynamique. C'est-à-dire, des valeurs qui ne peuvent être calculées à la compilation statique mais seulement à l'exécution, lorsque l'on évalue les spécifications de code dynamique. Pour le compilateur, il s'agit donc d'un trou, ou encore d'une constante, qui ne sera connue qu'à l'exécution. L'opérateur \$ s'applique à toute expression qui n'est pas de type **cspec** ou de type **vspec** et ne doit pas référencer les variables du code dynamique. Par exemple, on peut écrire :

```

int cspec c;

x = g(...);
c = 'printf ("$x = ", $x);

```

Il est important de comprendre que l'expression est évaluée au moment de l'évaluation de la spécification. Ce n'est pas la même chose que l'utilisation de variables libres dans le code dynamique qui ne sont évaluées qu'au moment où le code dynamique est compilé.

La figure 4.1 donne un exemple de ce que l'on peut écrire avec Tick C. Il s'agit d'un spécialiseur dynamique pour la fonction classique calculant le produit cartésien de deux vecteurs d'entiers. Les constantes d'exécution sont le premier vecteur et sa taille. La première déclaration spécifie le paramètre de la fonction dynamique qui est le deuxième vecteur. Puis, on initialise une variable de type **cspec** avec la constante 0. La boucle construit la somme des produits des éléments du vecteur *u* (utilisés comme des constantes) avec les éléments du vecteur dynamique (indexés avec des constantes).

Tick C est aussi constitué d'une bibliothèque de fonctions permettant de compiler les spécifications de code dynamique en code machine exécutable, de créer des variables dynamiques (paramètres et locales), ainsi que d'autres fonctions de service.

Discussion

Bien que novateur dans ses fonctionnalités, Tick C ne se situe pas au même niveau que la méthode proposée dans ce document. Comme DCG, il ne permet pas de produire automatiquement des spécialiseurs dynamiques. Il donne seulement un moyen de les écrire au niveau du langage C. Malgré cette amélioration, il ne semble pas souhaitable d'écrire manuellement les spécialiseurs dynamiques dès lors que l'on s'intéresse à des algorithmes complexes.

Au moment où nous avons publié nos travaux [CN96], le compilateur Tick C produisait du code pour DCG. Comme DCG, Tick C était relativement lent. Depuis, le compilateur a été totalement réécrit et Tick C obtient des résultats nettement meilleurs [PEK96, Eng96].

Les extensions de C proposées par Tick C peuvent en fait être vues comme une interface à la génération de code dynamique. De fait, nous avons alors ajouté une option à notre système lui permettant d'utiliser ce langage C étendu pour la génération de spécialiseurs dynamiques.

4.2 Compilateurs dynamiques

L'approche adoptée par les compilateurs dynamiques est de différer certaines phases de la compilation à l'exécution, pour certaines régions des programmes sources. Ainsi, il est possible de mieux optimiser le code en fonction des valeurs des constantes d'exécution de ces régions.

Pour une région donnée, le compilateur dynamique commence par identifier les constructions *statiques*, qui ne dépendent que de ses constantes d'exécution, et les constructions *dynamiques* constituées du reste du code. Ensuite, il compile normalement les parties statiques et repousse certaines phases de la compilation, et plus particulièrement la production de code machine, pour les parties dynamiques. La part de la compilation effectuée statiquement pour les constructions dynamiques dépend du système considéré. Elle est généralement assez grande afin de disposer d'un spécialiseur dynamique raisonnablement rapide. Cependant, il subsiste des opérations coûteuses. Par exemple, Fabius [LL96] propose de faire de l'allocation de registres à l'exécution.

Les spécialiseurs dynamiques sont automatiquement produits à partir d'une description des constantes d'exécution de la région. Le code dynamique est produit, soit en recopiant des *patrons* [KEH91, KEH93], fragments de code binaire pré-compilés avec des trous pour les valeurs des expressions statiques [APC⁺96], soit en émettant les instructions machine une à une [LL96]. Généralement, le spécialiseur dynamique va procéder à des optimisations fines des patrons, ou des instructions, par rapport aux valeurs statiques. C'est pourquoi on parle de compilation dynamique.

Le reste de cette section présente deux compilateurs dynamiques, le premier pour un langage fonctionnel pur du premier ordre, le second traitant le langage C dans son intégralité.

```

fun   produit( $v_1, v_2$ ) = prod( $v_1, v_2, 0, \text{length } v_1, 0$ )

and   prod( $v_1, v_2, \text{index}, \text{taille}, \text{somme}$ ) =
        if  $\text{index} = \text{taille}$  then  $\text{sum}$ 
            else prod( $v_1, v_2, \text{index} + 1, \text{taille}, \text{somme} + v_1[\text{index}] * v_2[\text{index}]$ )

```

FIG. 4.2 – Un exemple de programme ML

4.2.1 Fabius

Leone et Lee ont proposé [LL93, LL94, LL96] un compilateur dynamique pour un sous-ensemble, du premier ordre et sans effet de bord, du langage ML. Ce compilateur, nommé Fabius, transforme presque automatiquement les programmes qui lui sont fournis en du code pouvant se spécialiser dynamiquement.

La figure 4.2 présente l'exemple utilisé dans [LL96] pour illustrer le fonctionnement de Fabius. Il s'agit de la fonction *produit* calculant le produit cartésien de deux vecteurs de même taille par accumulation. La fonction auxiliaire *prod* reçoit les deux vecteurs, l'index courant dans les vecteurs, la taille des vecteurs et le résultat de la somme des produits des éléments déjà considérés.

Étagement

La première tâche de Fabius est de distinguer les parties statiques, appelées précoces dans [LL96], des parties dynamiques, ou tardives. Il réalise cette opération à l'aide d'une analyse proche d'une analyse de temps de liaison non inter-procédurale propageant la déclaration des constantes d'exécution dans tout le programme.

Le programmeur doit fournir manuellement les temps de liaison des paramètres formels de chacune des fonctions du programme. Il exprime cette information en curryfiant ses fonctions. Il place d'abord les variables qui sont des constantes d'exécution, puis les autres. Fabius propage alors cette information dans le corps de la fonction pour identifier les parties statiques et dynamiques. Ce mécanisme syntaxique a l'avantage d'être simple mais se révèle limitatif car il est impossible de disposer de plusieurs descriptions différentes pour une même région de code. Toutes les spécialisations d'une même fonction doivent donc avoir les mêmes constantes d'exécution. Cela oblige à ignorer les constantes spécifiques à certaines spécialisations.

La figure 4.3 montre le résultat de cette analyse pour l'exemple de la figure 4.2 avec v_1 comme constante d'exécution. Les parties statiques sont soulignées alors que les autres sont laissées telles quelles.

```

fun   produit  $v_1$   $v_2 = prod(\underline{v_1}, \underline{0}, \underline{length\ v_1}) (v_2, 0)$ 

and    $prod (v_1, \underline{index}, \underline{taille}) (v_2, \underline{somme}) =$ 
        if  $\underline{index} = \underline{taille}$  then  $\underline{sum}$ 
        else  $prod(v_1, \underline{index} + 1, \underline{taille}) (v_2, \underline{somme} + v_1[\underline{index}] * v_2[\underline{index}])$ 

```

FIG. 4.3 – Le résultat de l'étagement pour notre exemple

```

       $prod :$    beq    $\underline{\$i}, \underline{\$n}, \underline{L1}$ 
               sll    $\underline{\$i_1}, \underline{\$i}, \underline{2}$ 
               addu   $\underline{\$p_1}, \underline{\$v_1}, \underline{\$i_1}$ 
               lw     $\underline{\$x_1}, (\underline{\$p_1})$ 
               sll    $\underline{\$i_2}, \underline{\$i}, \underline{2}$ 
               lw     $\underline{\$x_2}, \underline{\$i_2}(\underline{\$p_2})$ 
               mult   $\underline{\$prd}, \underline{\$x_1}, \underline{\$x_2}$ 
               add    $\underline{\$sum}, \underline{\$sum}, \underline{\$prd}$ 
               addi   $\underline{\$i}, \underline{\$i}, \underline{1}$ 
               j      $\underline{prod}$ 

       $L1 :$      move   $\underline{\$result}, \underline{\$sum}$ 
               j      $\underline{\$ra}$ 

```

FIG. 4.4 – Le code intermédiaire produit pour la fonction *prod*

Production du code intermédiaire

Après l'étagement, Fabius produit un code intermédiaire écrit dans un langage de machine à registres. Les instructions de ce langage sont annotées de temps de liaison. La figure 4.4 donne, à titre indicatif, le code obtenu pour notre exemple. Les instructions dynamiques doivent être vues comme des générateurs d'instructions machine. Ce code intermédiaire est donc un spécialiste dynamique dédié qui va produire un certain nombre de fois le corps de la boucle, pour les différentes valeurs de l'index et des éléments du vecteur v_1 , puis l'instruction de retour.

Production du code machine

Fabius se débarrasse de la représentation intermédiaire en compilant normalement les instructions statiques et en transformant les instructions dynamiques en quelques instructions capables de générer le code machine correspondant. On a donc bien une compilation de la représentation intermédiaire.

Il procède aussi à des optimisations dynamiques par rapport aux valeurs des constantes d'exécution. Il remplace certaines instructions dynamiques, ayant des opérandes statiques, par une séquence d'instructions machine capables de générer des codes différents en fonction des valeurs des arguments statiques. Par exemple, si la représentation intermédiaire contient une multiplication dynamique dont le premier opérande est statique, Fabius la remplace par une séquence d'instructions produisant directement zéro si l'argument statique vaut zéro, et par une séquence optimale de décalage et d'addition dans les autres cas. C'est de la sélection dynamique d'instructions.

Il est évident que l'on ne peut pas introduire trop d'optimisations à ce stade sans risquer de ralentir dramatiquement le spécialiseur dynamique. Limiter le test pour zéro n'a statistiquement que peu d'intérêt, sauf si on travaille sur des problèmes de matrices creuses. D'autre part, la transformation en décalages et additions est une opération coûteuse. Rien n'est dit dans [LL96] sur les optimisations réellement mises en œuvre dans Fabius.

Discussion

Pour valider réellement l'approche, il faudrait que Fabius compile un sous-ensemble de ML plus important. Leone annonce que Fabius utilise six instructions pour en produire une. Ces résultats n'incluent pas le temps utilisé à faire des optimisations dynamiques. Ce coût est seulement celui des instructions chargées de recopier le code de l'instruction à émettre dans la mémoire. Rien n'est dit sur le coût réel de la production dynamique de code.

De plus, peu de détails sont donnés sur la façon dont est compilé le code dynamique. Fabius procède à une allocation de registres séparée pour les variables statiques et les variables dynamiques.

Dans certains cas, le spécialiseur dynamique procède à une allocation de registres. Bien que restreinte, cette opération est une tâche supplémentaire pour le spécialiseur dynamique et le coût de la production de code augmente.

4.2.2 Auslander, Philipose, Chamber, Eggers et Bershad

Parallèlement à nos travaux, Auslander, Philipose, Chamber, Eggers et Bershad ont proposé [APC⁺96] un compilateur dynamique pour le langage C. Le programmeur décore ses programmes à l'aide de quelques annotations afin d'indiquer au compilateur que certaines parties du code (région) doivent être compilées dynamiquement.

Leur système est composé d'un compilateur statique et d'un spécialiseur dynamique. Le compilateur statique produit un ensemble de patrons pré-optimisés ainsi que des informations destinées à piloter le spécialiseur dynamique lors de la production des spécialisations. Le code dynamique est produit en assemblant les patrons et en les instanciant avec les valeurs dynamiques. Le spécialiseur dynamique optimise aussi le code des patrons.

Annotations

Le programmeur délimite les zones contenant du code devant être compilé dynamiquement avec la construction :

dynamicRegion (v_1, \dots, v_n) s

La liste des variables en argument est celle des constantes d'exécution de la région. L'instruction s constitue le code de la région. Afin de ne pas introduire de confusion avec les notions de parties statiques et dynamiques définies ci-dessus, nous parlons ici plutôt de *région de spécialisation*. Normalement, à une région, ne correspond qu'un seul code dynamique. Les valeurs des constantes d'exécution ne peuvent donc pas être changées. Une variante de cette construction permet de disposer de plusieurs codes différents pour diverses valeurs des constantes d'exécution.

La construction :

unrolled s

indique au compilateur que l'on souhaite dérouler dynamiquement la boucle s . L'opération n'est légale que si le test de la boucle ne dépend que des constantes d'exécution. Il serait possible de déléguer la décision au spécialiseur dynamique mais cela aurait pour effet de le ralentir. La méthode avec annotations semble la meilleure pour traiter le déroulage de boucles.

Compilateur statique

Le compilateur statique, résultat de la modification d'un compilateur¹ existant, compile et optimise normalement le code en dehors des régions de spécialisation. Les sections suivantes décrivent le traitement des régions de spécialisation.

Identification des expressions statiques et dynamiques

Une analyse détermine les variables et expressions statiques des régions de spécialisation à partir de leurs constantes d'exécution. Il s'agit d'une forme d'analyse de temps de liaison capable de traiter des graphes de contrôle non structurés. Les libertés que le langage C laisse au programmeur (branchements intempestifs, etc.) autorisent l'écriture de programmes engendrant de tels graphes. L'identification des expressions statiques et dynamiques est constituée de deux analyses distinctes fonctionnant en parallèle.

Une analyse détermine, pour chaque point de programme de la région, l'ensemble des variables qui sont des constantes d'exécution. Elle procède par propagation de cet ensemble, initialisé avec la liste des constantes d'exécution fournie par le programmeur, sur le graphe de contrôle de la région. Cet ensemble est alors mis à jour en chaque nœud à l'aide de règles telles que, «si x est une constante d'exécution il en est de même pour y après l'instruction

1. Il s'agit du compilateur optimisant « Multiflow compiler »

```

int f(int x, int y)
{
  dynamicRegion (x) {
    int l;
    l = 2 * x;
    if (l == 2) return l * l + 2 * y;
    else return 4 * y;
  }
}

```

FIG. 4.5 – Un exemple de calcul de constantes d'exécution

$x = y$ ». Le traitement des jonctions se complique par le fait qu'une variable peut être une constante d'exécution pour deux nœuds précédant une jonction et ne plus l'être à la jonction. Par exemple, considérons le graphe de l'instruction :

```
if (test) x = 1; else x = 2;
```

Si le test est dynamique, on ne peut pas ajouter la variable x à l'ensemble des constantes d'exécution (comme pour l'analyse de temps de liaison) car on ne sait pas quel chemin le flot de contrôle utilisera à l'exécution. Par contre si le test est statique, la variable x est alors une constante d'exécution.

Il n'est pas possible, dans le cas des graphes non structurés, de distinguer directement si la jonction est sous contrôle statique ou dynamique. Pour régler ce problème, une autre analyse, fonctionnant en parallèle, calcule des conditions d'accessibilité aux nœuds en fonction des valeurs des constantes d'exécution. L'exclusion mutuelle des conditions associées aux prédécesseurs de la jonction, indique alors qu'il n'existe qu'un seul chemin possible, pour une configuration donnée des valeurs des constantes d'exécution. La jonction est alors sous contrôle statique.

Les ensembles obtenus sont utilisés pour annoter les différentes parties de la région comme statique ou dynamique. La figure 4.5 donne un exemple de programme annoté. Les expressions identifiées comme statiques sont soulignées alors que les autres sont laissées telles quelles.

Code d'initialisation et patrons

Après cette analyse, le compilateur produit deux graphes à partir du graphe initial. Le premier, celui du *code d'initialisation*, retient les fragments permettant de calculer les expressions statiques et de stocker les résultats obtenus dans une table. Cette table est utilisée par le spécialiste dynamique pour instancier les patrons. L'autre graphe, celui des patrons, est constitué du reste du code de la région dans lequel les expressions constantes ont été remplacées par des trous.

```
t = allocateTable ();  
t[0] = t0 = 2 * x;  
t[1] = t1 = (t0 == 2);  
t[2] = t2 = t0 * t0;
```

FIG. 4.6 – Le code d’initialisation de notre exemple

L’article décrivant ces travaux donne peu de détails en ce qui concerne le traitement des conditionnelles. Il semble que les conditionnelles sous contrôle statique ne soient pas reproduites dans le code d’initialisation de la région. Par conséquent, ce dernier calcule des expressions qui ne seront pas utilisées.

La figure 4.6 montre le code d’initialisation de notre exemple en supposant que la conditionnelle n’y est pas reproduite.

Les patrons sont optimisés par le compilateur statique dans leur contexte englobant. Ainsi, des optimisations globales pourront avoir lieu sur le code des patrons. Toutefois, la nécessité de retrouver les trous contenus dans les patrons oblige à limiter quelque peu les optimisations effectuées (pas de déplacement de code hors des patrons, pas de duplication des références aux trous, etc.).

Production du code et directives d’assemblage

Le code exécutable des régions dynamiques est obtenu en terminant la compilation du code d’initialisation ainsi que celle des patrons. De plus, lorsqu’il procède à ces opérations, le compilateur produit une séquence de directives utilisées par le spécialiseur dynamique pour la production effective du code dynamique.

Spécialiseur dynamique

Examinons maintenant les opérations intervenant à l’exécution lorsque l’on souhaite produire le code exécutable d’une région dynamique à partir des valeurs de ses constantes d’exécution. Tout d’abord, le code d’initialisation de la région est exécuté afin de construire la table contenant les valeurs dynamiques. Ensuite, le spécialiseur dynamique interprète les directives produites par le compilateur statique avec la table précédemment construite et le code des patrons de la région. Cette interprétation procède à la recopie du code des patrons, à l’instanciation de leurs trous, à la sélection des branches des conditionnelles statiques, à la gestion du déroulage de boucles ainsi qu’au relogement de certains branchements relatifs au compteur ordinal. La stratégie d’instanciation des trous dépend de leurs types, les entiers sont directement emballés dans une instruction immédiate. Les objets plus gros sont, soit construits à l’aide de plusieurs instructions immédiates, soit chargés depuis une table.

Le spécialiseur dynamique procède aussi à certaines optimisations locales simples en fonction des valeurs des constantes dynamiques. Par exemple, la multiplication par une constante dynamique est remplacée par une suite de décalages et d'additions.

Les résultats obtenus montrent de bonnes accélérations au niveau du code dynamique (de 1, 2 à 1, 8). Toutefois, l'amortissement de la production des spécialisations est très important (de 916 à 4760).

Discussion

Ce système est sans doute, tant par ses objectifs que par ses choix de mise en œuvre, le plus proche du nôtre. Comme nous, le langage considéré est un langage impératif et le programmeur n'a qu'une intervention limitée pour disposer de code dynamique. Le système d'annotation est simple et une analyse propage automatiquement les déclarations de temps de liaisons.

Toutefois, certains aspects différencient ce système du nôtre. Notre système procède à une transformation des programmes ayant un graphe de contrôle non structuré en programmes à graphe structuré. Cette transformation est effectuée à l'aide de l'approche décrite dans [EH93]. L'avantage de traiter directement les graphes non structurés est d'éviter la perte d'efficacité introduite par une transformation. Cependant, leur analyse n'est pas inter-procédurale et ne propage donc pas d'information au travers des appels de fonctions. Le programmeur doit donc annoter chacune des régions dynamiques.

Comme nous, le code dynamique est produit en recopiant des patrons pré-compilés statiquement. Par contre, le code des patrons est optimisé dynamiquement par rapport aux valeurs des expressions constantes. Le code obtenu est de meilleure qualité mais il faut plus de temps pour le produire.

À l'inverse de nos travaux, le spécialiseur dynamique est général et non dédié à chaque région de spécialisation. Cette couche d'interprétation des directives produites par le compilateur statique le rend peu efficace. Il aurait sans doute été moins coûteux de produire différents compilateurs dynamiques dédiés, assemblant, instanciant et optimisant directement les patrons tout en calculant les valeurs des constantes d'exécution (il ne serait alors pas nécessaire de construire une table).

Chapitre 5

Notre approche

Nous avons élaboré une méthode pour construire automatiquement des spécialiseurs dynamiques effectuant le minimum d'opérations afin de produire rapidement le code exécutable des spécialisations. En particulier, notre méthode permet la compilation statique du code dynamique en un ensemble de patrons. Le spécialiseur se contente alors de recopier les patrons en instanciant leurs trous avec les valeurs des constantes d'exécution.

À l'opposé des compilateurs dynamiques existants, la seule opération que l'on s'autorise à faire sur les patrons est l'instanciation de leurs trous. Le code des patrons n'est donc pas optimisé, à l'exécution, par rapport aux valeurs des constantes d'exécution.

Comme les compilateurs dynamiques, il nous faut déterminer les parties statiques et dynamiques des programmes analysés à partir d'un ensemble de constantes d'exécution. C'est une opération totalement identique à l'analyse de temps de liaison d'un évaluateur partiel hors ligne. L'idée est d'utiliser directement certaines techniques de l'évaluation partielle hors ligne pour effectuer cette analyse. On dispose ainsi d'analyses inter-procédurales précises et performantes permettant de produire automatiquement des spécialiseurs de qualité.

Toutefois, l'évaluation partielle est classiquement une transformation source vers source. Les spécialiseurs dédiés qu'elle permet de construire produisent le texte du programme spécialisé. La spécialisation n'est utilisable qu'après sa compilation. Il faut donc étendre certaines techniques d'évaluation partielle afin de construire des spécialiseurs dédiés produisant directement du code machine. Les arbres d'actions émis par l'analyse d'actions d'un évaluateur partiel hors ligne constituent une interface idéale pour nos analyses.

L'arbre d'actions constitue le point de départ de nos algorithmes. Nous supposons que l'évaluateur partiel qui l'a produit, à partir d'un programme et d'une description d'état partiel, est correct ; c'est-à-dire que l'évaluation de l'arbre d'actions retourne des programmes résiduels corrects vis-à-vis de la sémantique du langage. Le but est de transformer l'arbre d'actions en un spécialiseur dynamique produisant les spécialisations correspondantes à l'aide des valeurs des constantes d'exécution.

Puisque nous souhaitons compiler statiquement le code dynamique nous avons besoin d'une représentation globale des diverses spécialisations que l'on peut produire avec l'arbre d'actions. Malmkjær a proposé dans [Mal94] de calculer une grammaire, par interprétation abstraite de l'extension génératrice, pour représenter l'ensemble des spécialisations possibles. L'extension génératrice est obtenue par auto-application¹

Nous avons retenu l'idée d'utiliser une grammaire pour représenter l'ensemble des spécialisations. Toutefois, nous la calculons sur un arbre d'actions plutôt que sur un spécialiseur dédié². De plus, comme l'ensemble des spécialisations est un ensemble d'arbres syntaxiques, nous avons choisi d'utiliser des grammaires d'arbres régulières plutôt que des grammaires de mots.

Les différentes phases de notre approche sont les suivantes :

- L'arbre d'actions est transformé en un spécialiseur dédié manipulant des règles de grammaire. Cette extension génératrice construit les spécialisations en procédant à des réécritures du non-terminal de départ suivant les règles de la grammaire.
- La grammaire de spécialisation est extraite de l'extension génératrice. Il s'agit seulement d'une extraction et non d'une analyse. Les règles de grammaire sont, en effet, directement disponibles.
- La grammaire de spécialisation est dérivée en un ensemble de fragments de code source avec des trous appelés *patrons sources*. Ils constituent les briques de base des spécialisations que l'on peut engendrer avec la grammaire.
- Les patrons sources sont alors compilés statiquement et de manière globale, afin d'obtenir un code de bonne qualité, pour donner un ensemble de *patrons objets*, similaires aux patrons de [APC⁺96].
- L'extension génératrice est modifiée en remplaçant les règles de grammaire par des instructions de recopie et d'instanciation des patrons objets. On procède aussi à l'insertion d'instructions permettant de reloger certains branchements relatifs au compteur ordinal. Le résultat obtenu est donc un spécialiseur dynamique produisant rapidement et directement du code exécutable.

Le reste de ce chapitre est organisé comme suit. Tout d'abord, on présente la forme des extensions génératrices que nous utilisons; la construction automatique des ces objets est décrite formellement dans le chapitre 6. Puis, nous montrons comment les patrons sources sont dérivés de la grammaire de spécialisation pour être finalement transformés en patrons objets par un compilateur traditionnel. Finalement, nous examinons les transformations à opérer sur l'extension génératrice pour obtenir le spécialiseur dynamique.

1. L'auto-application est la spécialisation du spécialiseur par rapport au programme et à la description de l'état partiel considéré.

2. Un arbre d'actions est de plus haut niveau qu'un spécialiseur dédié.


```

{
  l = 2 * x;
  si (l == 2) alors res = l * l + 2 * y sinon { l = y * 2; res = l * l };
  res = 2 * res
}

```

FIG. 5.1 – Une exemple de programme PLI

5.1 Extension génératrice

Les extensions génératrices que nous utilisons manipulent des grammaires d'arbres pour produire le code des spécialisations. Nous présentons ici la notion de *grammaire de spécialisation* ainsi que la structure de nos extensions génératrices.

5.1.1 Grammaires de spécialisation

La sémantique des actions permet d'obtenir une spécialisation donnée à partir de l'arbre d'actions et d'un état mémoire (cohérent avec la description d'état partiel considéré). Avant l'exécution, les valeurs de l'état mémoire sont inconnues. Il est donc impossible de disposer du texte du programme résiduel pour le compiler. Par contre, on peut calculer une caractérisation finie de l'ensemble de toutes les spécialisations possibles (pour la description d'état partiel considéré). Cette caractérisation est obtenue par interprétation abstraite de l'arbre d'actions. Une manière naturelle de représenter des ensembles infinis d'arbres est d'utiliser des grammaires d'arbres régulières. Nous appellerons, *grammaire de spécialisation* d'un arbre d'actions, la grammaire caractérisant l'ensemble des spécialisations qu'il engendre.

Toutefois, il n'est pas possible de caractériser exactement la forêt de toutes les spécialisations envisageables à l'aide d'une grammaire d'arbres régulière. Il est, en effet, indécidable de déterminer l'ensemble des valeurs retournées par une expression, ainsi que les chemins possibles pour le flot de contrôle. Par conséquent, la grammaires de spécialisation que nous calculons caractérise un sur-ensembles de la forêt des spécialisations. Les spécialiseurs dédiés que nous construisons ne sélectionnent jamais d'éléments illégaux.

Considérons maintenant l'exemple de programme PLI de la figure 5.1. La figure 5.2 donne l'arbre d'actions retourné par les algorithmes du chapitre 3 pour cet exemple avec la variable x comme unique constante d'exécution.

La figure 5.3a montre les deux formes de spécialisations que l'on peut obtenir avec l'arbre d'actions de notre exemple. Le symbole Int est une constante générique représentant un entier quelconque.

La figure 5.3b donne la grammaire de spécialisation de notre exemple. Elle permet de dériver toutes les spécialisations de la figure 5.3a. Le symbole I_0 est un terminal générique

$$\begin{array}{l}
 \{ \\
 \quad l = 2 * x; \\
 \quad \{ \\
 \quad \quad \mathbf{si}^{red} (l == 2) \quad \mathbf{alors}^{red} \quad res =^{rec} (l * l)^{ev} +^{rec} (2 * y)^{id} \\
 \quad \quad \quad \mathbf{sinon}^{red} \{ l = y * 2; \quad res = l * l \}^{id}; \\
 \quad \quad (res = 2 * res)^{id} \\
 \quad \quad \quad \}^{rec} \\
 \quad \quad \quad \}^{redg} \\
 \}
 \end{array}$$
FIG. 5.2 – L’arbre d’actions de notre exemple si la variable x est une constante d’exécution
$$\begin{array}{ll}
 \{ & \{ \\
 \quad res = Int + 2 * y; & \quad l = y * 2; \\
 \quad res = 2 * res & \quad res = l * l; \\
 \} & \quad res = 2 * res \\
 & \}
 \end{array}$$

(a) Les deux formes de spécialisation possibles

$$P \rightarrow \{ \\
 \quad C; \\
 \quad res = 2 * res \\
 \}$$

$$C \rightarrow res = I_0 + 2 * y$$

$$C \rightarrow \{ \\
 \quad l = y * 2; \\
 \quad res = l * l \\
 \}$$

(b) La grammaire de spécialisation

FIG. 5.3 – La forêt et la grammaire de spécialisation de notre exemple

```

sired (e1) alorsred res = yid
sinonred
  sired (e2) alorsred res = 2 * yid
  sinonred res = 3 * yid

```

FIG. 5.4 – Un exemple de programme produisant une grammaire de spécialisation trop générale

```

{
  P → { C; res = 2 * res };
  l = 2 * x;
  si (l == 2) { C → res = I0 + 2 * y; I0 → l * l }
  else C → { l = y * 2; res = l * l };
}

```

FIG. 5.5 – L'extension génératrice de l'exemple

permettant de représenter n'importe quel entier. Cette grammaire décrit également des spécialisations impossibles. Par exemple, toutes celles remplaçant I_0 par une valeur n'étant pas un carré parfait.

Des situations similaires apparaissent au niveau du flot de contrôle pour des arbres d'actions du type de celui présenté dans la figure 5.4. Les expressions e_1 et e_2 sont des expressions statiques quelconques. La grammaire de spécialisation correspondante permet de produire les trois combinaisons possibles. Cependant, il se peut que les expressions e_1 et e_2 soient mutuellement exclusives. Nous avons donc ici un autre type d'exemples pour lesquels la grammaire de spécialisation inclut des programmes résiduels impossibles à produire.

5.1.2 Spécialiseur dédié

L'arbre d'actions est transformé en un spécialiseur dédié construisant les instructions résiduelles en réécrivant des termes suivant les règles de grammaire. Le calcul des extensions génératrices est décrit formellement dans le chapitre 6. Le reste de cette section présente la structure de nos extensions génératrices.

Le langage initial est enrichi de constructions permettant de représenter les règles et de déclencher les substitutions de non-terminaux. La figure 5.5 montre le spécialiseur obtenu pour notre exemple. La sémantique du langage est aussi modifiée pour prendre en compte les nouvelles constructions. Tout d'abord, la mémoire peut maintenant contenir des parties droites de règles. Un identificateur, consacré à la construction des spécialisations, est initialement lié au non-terminal de départ de la grammaire (P dans notre exemple).

L'évaluation d'une règle de la forme $nt \rightarrow t$ consiste à remplacer les occurrences du non-terminal nt par le terme t dans le programme résiduel en cours de construction. L'évaluation de la première règle de notre exemple construit l'instruction :

$$\{ C; res = 2 * res \}$$

L'évaluation d'une instanciation de terminal générique est basée sur le même principe. Le résultat de l'évaluation de l'expression est substitué au terminal générique dans le programme en cours de construction.

Examinons maintenant la suite de l'évaluation du spécialiseur de la figure 5.5 la variable x ayant pour valeur 1. L'affectation

$$l = 2 * x$$

s'évalue normalement et la variable l est liée à l'entier 2. C'est donc la première branche qui s'exécute, provoquant le remplacement du non-terminal C par l'instruction

$$res = I_0 + 2 * y$$

Le code résiduel devient donc

$$\{ res = I_0 + 2 * y; res = 2 * res \}$$

Finalement, la dernière instruction instancie le terminal générique I_0 avec la valeur $2 * 2$ et la spécialisation obtenue est

$$\{ res = 4 + 2 * y; res = 2 * res \}$$

On retrouve la spécialisation présentée précédemment dans la figure 3.5.

5.2 Dérivation des patrons objets

Cette section présente l'identification des patrons sources et leur transformation en patrons objets.

5.2.1 Identification des patrons sources

L'obtention des patrons sources à partir de la grammaire de spécialisation consiste essentiellement à transformer les parties droites de règles en syntaxe concrète³ et à délimiter les patrons. La première opération est classique si ce n'est le traitement des terminaux génériques que l'on doit transformer en trous. La représentation concrète des trous dépend grandement du langage considéré ainsi que du compilateur utilisé pour compiler les patrons. Pour s'abstraire de ces dépendances, nous représenterons ici les trous par des identificateurs notés entre

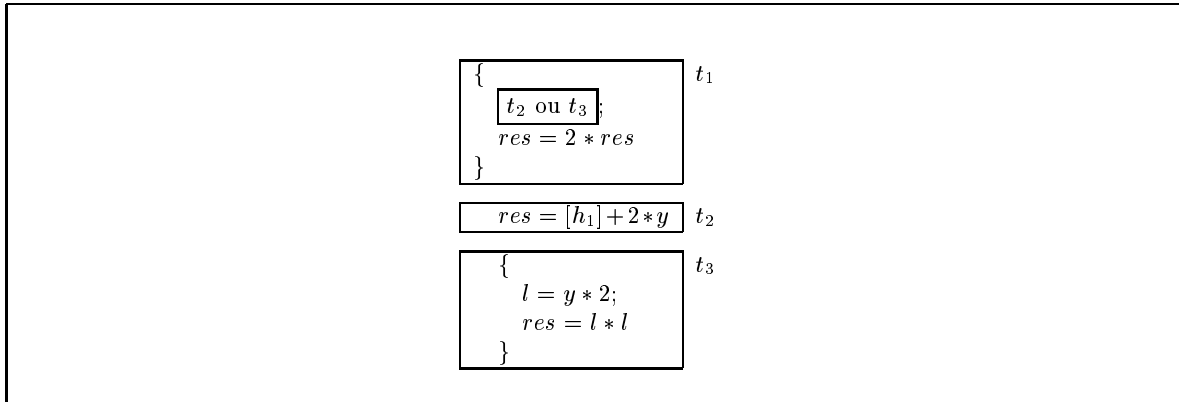


FIG. 5.6 – Les patrons sources de notre exemple (2)

crochets. Chaque trou a un nom unique. Plusieurs méthodes sont possibles pour délimiter les patrons. Nous en présentons deux.

La première consiste à créer un patron, pour chaque partie droite de règle dans la grammaire de spécialisation, en y remplaçant les occurrences de non-terminaux par des emplacements. Ces emplacements sont destinés à recevoir les patrons associés aux règles ayant ce non-terminal comme partie gauche. Sur notre exemple, nous obtenons trois patrons, un pour le programme à proprement parlé, et un pour chacune des alternatives de la conditionnelle comme le montre la figure 5.6. Le patron principal contient un emplacement pouvant recevoir les patrons t_2 ou t_3 .

Cette transformation a l'avantage d'être simple mais se révèle coûteuse, pour les spécialisateurs dynamiques. Il est impossible de prévoir, avant l'exécution, lequel des deux patrons va être inséré dans l'emplacement prévu à cet effet. Il est donc nécessaire de réserver une place de taille égale à celle du plus grand patron. Cela conduit à des spécialisations parfois trop grandes et dans le cas où le plus petit des patrons est sélectionné, il faut insérer un saut pour passer à la suite du programme spécialisé. De plus, si l'on désire dérouler des boucles, il est alors impossible de borner la taille des patrons et cette méthode n'est alors pas envisageable.

La deuxième méthode de délimitation des patrons consiste à éliminer les patrons imbriqués de la première méthode. Il faut «casser» la syntaxe du langage car les patrons ne sont plus des entités syntaxiques cohérentes. La technique consiste à créer un patron de parts et d'autres de chaque occurrence de non-terminaux dans les parties droites de règle. Une règle de grammaire n'engendre plus un patron unique mais plusieurs patrons. La figure 5.7 montre les patrons que l'on obtient pour notre exemple avec cette méthode. Un spécialisateur dynamique créé de cette façon n'a plus qu'à mettre les patrons bout à bout pour obtenir une spécialisation donnée. Nous utilisons cette méthode dans la suite de cette présentation.

3. Opération inverse de l'analyse syntaxique.

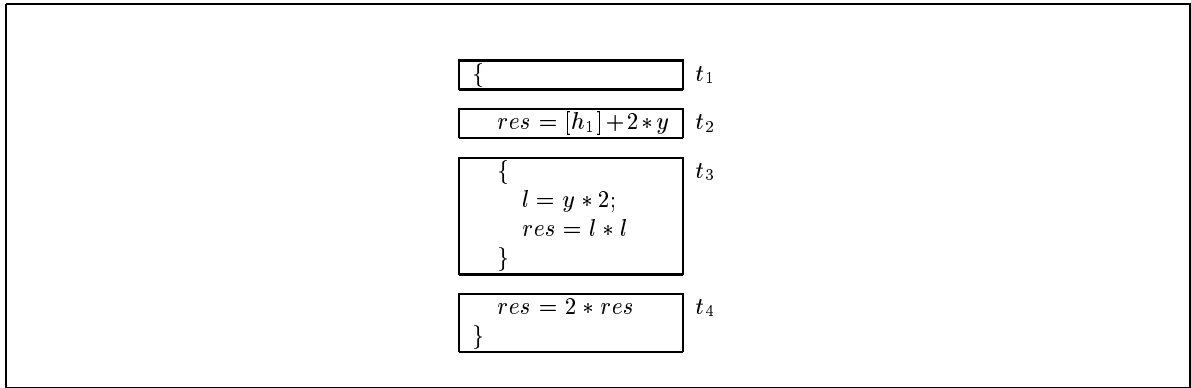


FIG. 5.7 – Les patrons sources de notre exemple (2)

5.2.2 Compilation des patrons

Lorsque les patrons sont délimités et transformés en syntaxe concrète, ils sont compilés afin d'obtenir des patrons objets. L'avantage est que les patrons sont disponibles avant l'exécution, et ainsi, aucune compilation à l'exécution n'est nécessaire pour produire les spécialisations. Les patrons ne sont pas compilables tels quels car ils ne constituent pas des entités syntaxiques cohérentes.

À ce stade, il est important de noter qu'il serait possible de développer un compilateur permettant de compiler les patrons. Il s'agirait d'un outil permettant de compiler des arbres syntaxiques incomplets incluant des constantes génériques (à cause des terminaux génériques comme I_0). Cependant, le développement d'un compilateur est une tâche longue et difficile si l'on désire rivaliser avec les excellents compilateurs existants.

Il serait aussi possible de modifier un compilateur existant pour lui permettre de telles compilations. Là encore, le travail requis est important car ces outils ne sont pas aussi modulaires qu'ils le prétendent. Les modifications nécessaires peuvent avoir des répercussions dans tout le compilateur.

Nous avons choisi d'utiliser un compilateur existant, sans le modifier. Il convient alors de transformer nos grammaires de spécialisation en un source syntaxiquement correct pour le compilateur utilisé. Évidemment, cette transformation dépend fortement du langage ainsi que du compilateur employé. Nous nous bornerons ici à des considérations générales indépendantes d'un langage ou d'un compilateur spécifique.

Jusqu'à maintenant, les patrons ont été présentés comme des entités indépendantes. Toutefois, si nous les compilons séparément, le code obtenu est de mauvaise qualité. En effet, il est alors impossible de tirer avantage du contexte dans lequel ils apparaissent dans les spécialisations. Certaines techniques de compilation, comme l'allocation de registres ou le séquençement d'instructions n'auraient alors aucun effet sur les patrons. Une solution est de combiner les

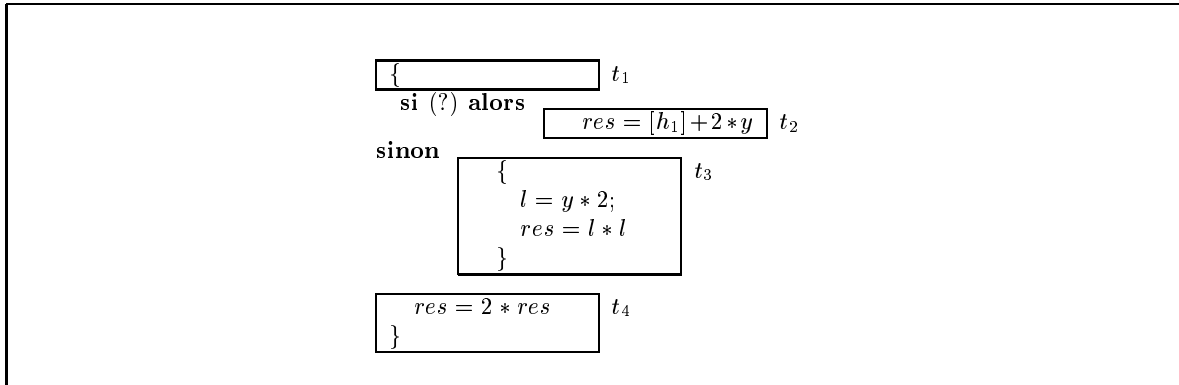


FIG. 5.8 – Le texte source utilisé pour compiler les patrons sources de l'exemple

patrons en un texte de programme unique exprimant pour chacun d'entre eux l'union de tous les contextes dans lesquels ils peuvent apparaître. Ce source suit la structure de la grammaire de spécialisation. On introduit une conditionnelle, portant sur un identificateur externe, afin de préserver l'incertitude sur la sélection des patrons. C'est ici que l'approximation causée par l'utilisation de grammaires régulières est la plus gênante. En effet les contextes considérés sont un peu trop généraux car certaines combinaisons de patrons que l'on peut dériver de la grammaire ne sont pas des spécialisations possibles.

La figure 5.8 montre le résultat de cette transformation pour notre exemple. Malgré la présence du «*?*» le compilateur est capable de traiter les patrons de manière globale car il connaît les différentes possibilités d'assemblage.

Dans la réalité, les patrons sont délimités par des marqueurs, de manière à pouvoir les extraire du code objet produit par le compilateur. La représentation des trous est dépendante du langage et du compilateur utilisé. Il faut une représentation qui permette de retrouver l'adresse des trous dans le code objet. Cette adresse est, en effet, nécessaire à l'instanciation des patrons. De plus, il faut utiliser un objet produisant une instruction immédiate afin que le trou se comporte comme une véritable constante.

Après compilation, on récupère les fragments de code qui constituent nos patrons objets ainsi que l'adresse de leurs trous. Il est aussi nécessaire d'identifier certains sauts qui passent au dessus de zones de code dont on ne connaît pas la taille. Ces sauts doivent, en effet, être relogés. Leur identification est effectuée en analysant le code produit par le compilateur pour repérer les sauts inter-patrons.

5.3 Production du spécialiseur dynamique

Nous décrivons ici les transformations à effectuer sur les extensions génératrices manipulant des règles de grammaire et produisant de la syntaxe abstraite pour obtenir des spécialiseurs

```

{
  copie_patron(t1);
  l = 2 * x;
  si (l == 2) alors {
    copie_patron(t2);
    instancie(t2, l * l)
  }
  sinon copie_patron(t3);
  copie_patron(t4)
}

```

FIG. 5.9 – Le spécialiseur dynamique de notre exemple

dynamiques manipulant des patrons objets et produisant du code machine.

Nous souhaitons que nos spécialiseurs dynamiques produisent le code exécutable uniquement par recopie des patrons objets. Nous ne voulons pas procéder à des insertions qui engendrent des déplacements de code déjà copié. La grammaire de spécialisation étant essentiellement emboîtée, il convient de l'aplatir pour suivre la structure de découpage des patrons. Une règle est transformée en autant d'instructions de recopie de patrons qu'elle en produit. Ces instructions sont correctement placées dans le code de l'extension génératrice.

La figure 5.9 montre l'effet de cette transformation sur l'extension génératrice de la figure 5.5. La première règle produit les instructions de recopie des patrons t_1 et t_4 alors que les deux autres engendrent respectivement celles des patrons t_2 et t_3 .

Les instructions d'instanciation des terminaux génériques sont simplement remplacées par des instructions de remplissage de trous. Remplir un trou consiste à ranger la valeur de l'expression associée dans l'instruction immédiate que le compilateur a produit pour le trou. Finalement, on ajoute des instructions permettant de reloger les sauts inter-patrons identifiés à la compilation.

Ainsi, nos spécialiseurs dynamiques produisent le code des spécialisations en recopiant le code des patrons objets, en instanciant leurs trous, et en relogant certains sauts. La simplicité des opérations effectuées assure un faible coût de production des spécialisations dynamiques.

Chapitre 6

Approche formelle

Ce chapitre présente la construction automatique des extensions génératrices présentées dans le chapitre 5 pour le langage PLI. Le point de départ est un arbre d'actions produit par l'évaluateur partiel hors ligne décrit dans le chapitre 3. Une première section décrit les extensions syntaxiques et sémantiques que l'on doit effectuer sur le langage pour permettre la représentation et la manipulation des grammaires par les extensions génératrices. Ensuite, nous donnons une nouvelle sémantique pour les actions calculant une extension génératrice à partir de l'arbre d'actions. Cette sémantique est le résultat de l'abstraction de la mémoire dans la sémantique des actions donnée dans la figure 3.13. La dernière section prouve la correction de la construction proposée.

6.1 Extensions du langage

Nos extensions génératrices produisent le texte des spécialisations en utilisant des grammaires d'arbres. Il est nécessaire d'enrichir la syntaxe et la sémantique de PLI pour manipuler ces objets.

6.1.1 Grammaires

La figure 6.1 donne la syntaxe des parties droites des règles de grammaire. Il s'agit de la syntaxe de PLI enrichie de deux règles. Une règle permet d'inclure des occurrences de non-terminaux. Les non-terminaux sont repérés par des éléments de l'ensemble $Nterm$. Une autre règle autorise l'utilisation de terminaux génériques dans les expressions. Les terminaux génériques sont repérés par des éléments de l'ensemble $Gterm$.

$i \in Ident$	Identificateurs
$n \in Num$	Constantes
$op \in OpBin$	Opérateurs binaires
$nt \in Nterm = nt_1, \dots, nt_2$	Non-terminaux
$tg \in Gterm = tg_1, \dots, tg_2$	Terminaux génériques
Syntaxe abstraite	
$c^{pd} \in Com^{pd}$	$::= \text{NOP}$ $ \text{SEQ}(c_1^{pd}, c_2^{pd})$ $ \text{AFF}(i, e^{pd})$ $ \text{COND}(e^{pd}, c_1^{pd}, c_2^{pd})$ $ \text{NTERM}(nt)$
$e^{pd} \in Exp^{pd}$	$::= \text{CST}(n)$ $ \text{VAR}(i)$ $ \text{APP}(op, e_1^{pd}, e_2^{pd})$ $ \text{GTERM}(tg)$
	Syntaxe concrète
	$ \{c_1^{pd}, c_2^{pd}\}$ $ i = e^{pd}$ $ \text{si } (e^{pd}) \text{ alors } c_1^{pd} \text{ sinon } c_2^{pd}$ $ nt$
	$ n$ $ i$ $ e_1^{pd} \text{ op } e_2^{pd}$ $ [tg]$

FIG. 6.1 – La syntaxe des parties droites de règle

6.1.2 Extension génératrices

Pour que les extensions génératrices puissent manipuler des règles de grammaire, on enrichit d'une deuxième manière la syntaxe de PLI. La figure 6.2 présente ces extensions. Deux règles sont ajoutées pour les instructions. Une règle permet aux extensions génératrices de produire le code correspondant à une règle de grammaire. Une autre règle autorise l'extension génératrice à instancier un terminal générique avec le résultat de l'évaluation d'une expression en amont de la spécialisation.

Comme le montre l'interpréteur d'actions de la figure 3.13, le traitement d'une conditionnelle reconstruite nécessite une évaluation spéculative des deux branches. Chacune des branches doit être évaluée avec le même état mémoire. Pour faciliter cette opération, on ajoute la construction **PROT** permettant d'évaluer une instruction sans modification de l'état mémoire.

La figure 6.3 décrit la sémantique permettant l'évaluation des extensions génératrices. Un identificateur spécifique est autorisé à contenir une commande de l'ensemble Com^{pd} ou une expression de l'ensemble Exp^{pd} . Au début de l'évaluation de l'extension génératrice, on initialise cet identificateur avec le non-terminal de la première règle de grammaire contenue dans l'extension génératrice. L'évaluation de l'extension génératrice va alors construire, pas à pas, le programme résiduel en effectuant les substitutions engendrées par les règles de grammaire.

Pour simplifier l'expression de la preuve de correction de l'algorithme de calcul des ex-

$i \in Ident$	Identificateurs
$n \in Num$	Constantes
$op \in OpBin$	Opérateurs binaires
$nt \in Nterm = nt_1, \dots, nt_2$	Non-terminaux
$tg \in Gterm = tg_1, \dots, tg_2$	Terminaux génériques

Syntaxe abstraite	Syntaxe concrète
$c^s \in Com^s \quad ::= \text{NOP}$	
$\text{SEQ}(c_1^s, c_2^s)$	$\{c_1^s, c_2^s\}$
$\text{AFF}(i, e)$	$i = e$
$\text{COND}(e, c_1^s, c_2^s)$	si (e) alors c_1^s sinon c_2^s
$\text{REGLE}(nt, c^{pd})$	$nt \rightarrow c^{pd}$
$\text{INST}(tg, e)$	$tg \rightarrow e$
$\text{PROT}(c^s)$	prot c^s

FIG. 6.2 – La syntaxe des extensions génératrices

Int	Valeurs entières
$f \in Fun_2 = Int \times Int \rightarrow Int$	Fonctions entières binaires
$\rho \in Mem^s = (Com^{pd} + Exp^{pd}) \times Mem$	Mémoire

$C^s : Com^s \rightarrow Mem^s \rightarrow Mem^s$	
$C^s[\text{NOP}]$	$= \lambda\rho. \rho$
$C^s[\text{SEQ}(c_1^s, c_2^s)]$	$= C^s[c_1^s] \circ C^s[c_2^s]$
$C^s[\text{AFF}(i, e)]$	$= \lambda(c^{pd}, \sigma). (c^{pd}, \sigma[i \mapsto \mathcal{E}[e]\sigma])$
$C^s[\text{COND}(e, c_1^s, c_2^s)]$	$= \lambda(c^{pd}, \sigma). \text{if } \mathcal{E}[e]\sigma \text{ then } C^s[c_1^s](c^{pd}, \sigma) \text{ else } C^s[c_2^s](c^{pd}, \sigma)$
$C^s[\text{REGLE}(nt, c^{pd})]$	$= \lambda(c_1^{pd}, \sigma). (c_1^{pd}[nt \leftarrow c^{pd}], \sigma)$
$C^s[\text{INST}(tg, e)]$	$= \lambda(c^{pd}, \sigma). (c^{pd}[tg \leftarrow \mathcal{E}[e]\sigma], \sigma)$
$C^s[\text{PROT}(c^s)]$	$= \lambda(c^{pd}, \sigma). (c_1^{pd}, \sigma)$
	where $(c_1^{pd}, _) = C^s[c](c^{pd}, \sigma)$

FIG. 6.3 – La sémantique étendue pour les extensions génératrices

```

{
  nt0 → { nt1; res = 2 * res };
  l = 2 * x;
  si (l == 2) alors { nt1 → res = [tg1] + 2 * y; tg1 → l * l }
  sinon nt1 → { l = y * 2; res = l * l }
}

```

FIG. 6.4 – Un exemple d’extension génératrice

tensions génératrices, nous avons choisit de séparer l’identificateur chargé de mémoriser le code résiduel en cours de construction des autres. Ainsi, les états mémoire manipulés par la sémantique étendue sont constitués d’un état de l’ensemble Mem ainsi que d’un élément de l’ensemble $Com^{pd} + Exp^{pd}$.

Évaluer une règle de grammaire consiste simplement à remplacer une occurrence de son non terminal dans le code résiduel en cours de construction, par la partie droite de la règle. Un mécanisme identique est utilisé pour remplacer les terminaux génériques par le résultat de l’évaluation de l’expression qui lui est associée. Ainsi, le programme résiduel est construit incrémentalement par le jeu des substitutions engendrées par les règles de grammaire contenues dans l’extension génératrice.

L’évaluation d’une instruction protégée consiste simplement à ne retenir que l’effet de cette instruction sur le programme résiduel en cours de construction. Le reste de la mémoire est inchangé.

La figure 6.4 montre un exemple d’extension génératrice écrite dans le langage étendu.

6.2 Calcul de l’extension génératrice

Le calcul de l’extension génératrice est effectué en calculant en parallèle la grammaire de spécialisation et le code du spécialiseur dédié.

La sémantique calculant les extensions génératrices est décrite dans la figure 6.5.

L’évaluation d’une instruction annotée consiste en une partie droite de règle de grammaire et en une instruction assurant la résolution de tout ses non-terminaux et terminaux génériques. Certaines actions engendrent des choix pour le code résiduel, on procède alors à la création d’une règle pour chacun des cas. Ces règles partagent toutes le même nouveau non-terminal en partie gauche. Leurs parties droites sont celles calculées pour chacune des alternatives. Le code résiduel produit par ces actions est alors le non-terminal qui vient d’être créé. Finalement, les règles nouvellement construites sont insérées dans le code du spécialiseur.

L’évaluation d’une instruction annotée avec l’action *Évalue* retourne une instruction résiduelle vide et un spécialiseur identique à l’instruction initiale. L’évaluation d’une instruction

$c^{pd} \in Com^{pd}$	Parties droites de règles
$e^{pd} \in Exp^{pd}$	Expressions correspondantes
$c^s \in Com^s$	Instructions pour les extensions
$e \in Exp$	Expressions
$\mathcal{P}^{sd} : Com^a \rightarrow (Com^{pd}, Com^s)$	
$\mathcal{P}^{sd}[[c^a]]$	$(\mathbf{NTERM}(nt), \mathbf{SEQ}(\mathbf{REGLE}(nt, c^{pd}), c^s))$ where $(c^{pd}, c^s) = \mathcal{C}^{sd}[[c^a]]$ nt est un nouveau non-terminal
$\mathcal{C}^{sd} : Com^a \rightarrow (Com^{pd} \times Com^s)$	
$\mathcal{C}^{sd}[\mathbf{EV}(c)]$	(\mathbf{NOP}, c)
$\mathcal{C}^{sd}[\mathbf{ID}(c)]$	(c, \mathbf{NOP})
$\mathcal{C}^{sd}[\mathbf{RED_SEQg}(c, c^a)]$	$(c^{pd}, \mathbf{SEQ}(c, c^s))$ where $(c^{pd}, c^s) = \mathcal{C}^{sd}[[c^a]]$
$\mathcal{C}^{sd}[\mathbf{RED_SEQd}(c^a, c)]$	$(c^{pd}, \mathbf{SEQ}(c^s, c))$ where $(c^{pd}, c^s) = \mathcal{C}^{sd}[[c^a]]$
$\mathcal{C}^{sd}[\mathbf{REC_SEQ}(c_1^a, c_2^a)]$	$(\mathbf{SEQ}(c_1^{pd}, c_2^{pd}), \mathbf{SEQ}(c_1^s, c_2^s))$ where $(c_1^{pd}, c_1^s) = \mathcal{C}^{sd}[[c_1^a]]$ $(c_2^{pd}, c_2^s) = \mathcal{C}^{sd}[[c_2^a]]$
$\mathcal{C}^{sd}[\mathbf{REC_AFF}(i, e^a)]$	$(\mathbf{AFF}(i, e^{pd}), c^s)$ where $(e^{pd}, c^s) = \mathcal{E}^{sd}[[e^a]]$
$\mathcal{C}^{sd}[\mathbf{RED_COND}(e, c_1^a, c_2^a)]$	$(\mathbf{NTERM}(nt), \mathbf{COND}(e, \mathbf{SEQ}(\mathbf{REGLE}(nt, c_1^{pd}), c_1^s), \mathbf{SEQ}(\mathbf{REGLE}(nt, c_2^{pd}), c_2^s)))$ where $(c_1^{pd}, c_1^s) = \mathcal{C}^{sd}[[c_1^a]]$ $(c_2^{pd}, c_2^s) = \mathcal{C}^{sd}[[c_2^a]]$ nt est un nouveau non-terminal
$\mathcal{C}^{sd}[\mathbf{REC_COND}(e^a, c_1^a, c_2^a)]$	$(\mathbf{COND}(e^{pd}, c_1^{pd}, c_2^{pd}), \mathbf{SEQ}(c^s, \mathbf{SEQ}(\mathbf{PROT}(c_1^s), c_2^s)))$ where $(e^{pd}, c^s) = \mathcal{E}^{sd}[[e^a]]$ $(c_1^{pd}, c_1^s) = \mathcal{C}^{sd}[[c_1^a]]$ $(c_2^{pd}, c_2^s) = \mathcal{C}^{sd}[[c_2^a]]$
$\mathcal{E}^{sd} : Exp^a \rightarrow (Exp^{pd} \times Com^s)$	
$\mathcal{E}^{sd}[\mathbf{EV}(e)]$	$(\mathbf{GTERM}(tg), \mathbf{INST}(tg, e))$ where tg est un nouveau terminal générique
$\mathcal{E}^{sd}[\mathbf{ID}(e)]$	(e, \mathbf{NOP})
$\mathcal{E}^{sd}[\mathbf{REC_APP}(op, e_1^a, e_2^a)]$	$(\mathbf{APP}(op, e_1^{pd}, e_2^{pd}), \mathbf{SEQ}(c_1^s, c_2^s))$ where $(e_1^{pd}, c_1^s) = \mathcal{E}^{sd}[[e_1^a]]$ $(e_2^{pd}, c_2^s) = \mathcal{E}^{sd}[[e_2^a]]$

FIG. 6.5 – Algorithme de calcul des extensions génératrices

annotée avec l'action *Identité* est similaire. Le code résiduel retourné par l'évaluation d'une séquence qui se réduit (à gauche ou à droite) est celui de l'instruction annotée. Le spécialiseur correspondant est une séquence constituée du spécialiseur associé à l'instruction annotée et de l'instruction évaluable. L'ordre dépend du côté où l'on réduit. L'évaluation d'une séquence reconstruite consiste à composer les résultats de l'évaluation des deux instructions. L'évaluation d'une conditionnelle réductible introduit un choix au niveau du programme résiduel. On exprime ce choix en créant deux règles, ayant le même nouveau non terminal en partie gauche et le code résiduel des branches en partie droite. Le code résiduel de la conditionnelle est alors le non-terminal fraîchement créé. Le spécialiseur associé est une conditionnelle. Le test est celui de la conditionnelle initiale et les branches sont le résultat de l'insertion des règles créés en tête du spécialiseur construit pour chacune des branches. L'évaluation d'une conditionnelle reconstruite consiste à composer les résultats de l'évaluation du test et des branches. Il convient toutefois de protéger le spécialiseur de la première branche. Ainsi, lors de la spécialisation, le spécialiseur de la deuxième branche disposera d'un état mémoire correct.

L'évaluation d'une expression annotée consiste en une expression résiduelle pouvant contenir des terminaux génériques ainsi que d'une instruction permettant de les instancier tous.

L'évaluation d'une expression annotée avec l'action *Évalue* retourne un terminal générique comme expression résiduelle. Le spécialiseur associé est alors une instruction instanciant le terminal générique à l'aide de l'expression initiale. L'évaluation d'une expression annotée avec l'action *Identité* retourne une expression résiduelle identique à l'expression initiale et un spécialiseur vide. L'évaluation d'un appel de primitive reconstruit consiste à composer les résultats retournés par l'évaluation des deux opérandes.

À la fin de l'évaluation de l'arbre d'actions, la fonction \mathcal{P}^{sd} transforme la partie droite, calculée pour l'instruction, en une règle. Cette règle est alors insérée dans le spécialiseur calculé pour l'instruction.

6.3 Correction

La correction de l'évaluateur partiel dépasse le cadre de ce document. Nous supposons donc qu'il est correct et que l'évaluation des arbres d'actions avec la sémantique de la figure 3.13 retourne un programme résiduel correct par rapport à la sémantique de PLI.

Prouver la correction de notre générateur d'extensions génératrices consiste à montrer qu'étant donné un arbre d'actions et des valeurs de spécialisation, l'extension génératrice construit le même programme résiduel que l'interprétation des actions. Formellement, cela s'exprime par le théorème suivant :

Théorème 1 $(\forall c^a \in Com^a)(\forall \sigma \in Mem)$
si $(c^{pd}, c^s) = \mathcal{P}^{sd}[[c^a]]$ **alors** $\mathcal{C}^s[[c^s]](c^{pd}, \sigma) = \mathcal{C}^{spec}[[c^a]]\sigma$

Avant de prouver ce théorème, il nous faut démontrer deux propositions. La première

exprime la correction des extensions génératrices au niveau de la construction des expressions. La deuxième fait de même pour les instructions.

Proposition 1 $(\forall e^a \in Exp^a)(\forall \sigma \in Mem)$
si $(e^{pd}, c^s) = \mathcal{E}^{sd}[[e^a]]$ **alors** $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (\mathcal{E}^{spec}[[e^a]]\sigma, \sigma)$

Proposition 2 $(\forall c^a \in Com^a)(\forall \sigma \in Mem)$
si $(e^{pd}, c^s) = \mathcal{C}^{sd}[[c^a]]$ **alors** $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = \mathcal{C}^{spec}[[c^a]]\sigma$

On prouve alors la proposition 1 par induction structurelle sur l'arbre d'actions :

- $e^a = \mathbf{EV}(e)$
 $(e^{pd}, c^s) = \mathcal{E}^{sd}[[e^a]] = (\mathbf{GTERM}(tg), \mathbf{INST}(tg, e))$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = \mathcal{C}^s[[\mathbf{INST}(tg, e)]](\mathbf{GTERM}(tg), \sigma)$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (\mathbf{GTERM}(tg)[tg \leftarrow \mathcal{E}[[e]]\sigma], \sigma)$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (\mathcal{E}[[e]]\sigma, \sigma)$
 D'où $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (\mathcal{E}^{spec}[[e^a]]\sigma, \sigma)$

- $e^a = \mathbf{ID}(e)$
 $(e^{pd}, c^s) = \mathcal{E}^{sd}[[e^a]] = (e, \mathbf{NOP})$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = \mathcal{C}^s[[\mathbf{NOP}]](e, \sigma)$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (e, \sigma)$
 D'où $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (\mathcal{E}^{spec}[[e^a]]\sigma, \sigma)$

- $e^a = \mathbf{REC_APP}(op, e_1^a, e_2^a)$
 $(e^{pd}, c^s) = \mathcal{E}^{sd}[[e^a]] = (\mathbf{APP}(op, e_1^{pd}, e_2^{pd}), \mathbf{SEQ}(c_1^s, c_2^s))$
 avec $\begin{cases} (e_1^{pd}, c_1^s) = \mathcal{E}^{sd}[[e_1^a]] \\ (e_2^{pd}, c_2^s) = \mathcal{E}^{sd}[[e_2^a]] \end{cases}$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = \mathcal{C}^s[[\mathbf{SEQ}(c_1^s, c_2^s)]](\mathbf{APP}(op, e_1^{pd}, e_2^{pd}), \sigma)$
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = \mathcal{C}^s[[c_2^s]](\mathcal{C}^s[[c_1^s]](\mathbf{APP}(op, e_1^{pd}, e_2^{pd}), \sigma))$
 Comme c_1^s (resp. c_2^s) n'agit que sur les terminaux génériques de e_1^{pd} (resp. e_2^{pd}), on à
 $\mathcal{C}^s[[c^s]](e^{pd}, \sigma'') = (\mathbf{APP}(op, e_3^{pd}, e_4^{pd}), \sigma)$
 avec $\begin{cases} (e_3^{pd}, \sigma') = \mathcal{C}^s[[c_1^s]](e_1^{pd}, \sigma) \\ (e_4^{pd}, \sigma'') = \mathcal{C}^s[[c_1^s]](e_1^{pd}, \sigma') \end{cases}$
 Par induction, on obtient $\begin{cases} (e_3^{pd}, \sigma') = (\mathcal{E}^{spec}[[e_1^a]]\sigma, \sigma) \\ (e_4^{pd}, \sigma'') = (\mathcal{E}^{spec}[[e_2^a]]\sigma', \sigma') \end{cases}$
 Donc $\mathcal{C}^s[[c^s]](e^{pd}, \sigma) = (\mathbf{APP}(op, \mathcal{E}^{spec}[[e_1^a]]\sigma, \mathcal{E}^{spec}[[e_2^a]]\sigma), \sigma)$

D'où $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (\mathcal{E}^{spec} \llbracket e^a \rrbracket \sigma, \sigma)$

La preuve la proposition 2 utilise le lemme ci-dessous qui exprime que l'exécution d'une instruction du langage initial ne modifie ni ne dépend du programme résiduel en cours de construction.

Lemme 1 $(\forall c \in Com)(\forall \sigma \in Mem)(\forall c^{pd} \in Com^{pd})$
 $\mathcal{C}^s \llbracket c \rrbracket (c^{pd}, \sigma) = (c^{pd}, \mathcal{C} \llbracket c \rrbracket \sigma)$

La preuve de ce lemme est triviale car seule les constructions spécifiques au langage Com^s utilisent et modifient le programme résiduel en cours de construction.

On prouve alors la proposition 2 par induction structurale sur l'arbre d'actions :

• $c^a = \mathbf{EVAL}(c)$

$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket = (\mathbf{NOP}, c)$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket \mathbf{NOP} \rrbracket (c, \sigma)$

D'après le lemme 1, $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (\mathbf{NOP}, \mathcal{C} \llbracket c \rrbracket \sigma)$

D'où $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c^a \rrbracket \sigma$

• $c^a = \mathbf{ID}(c)$

$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket = (c, \mathbf{NOP})$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket \mathbf{NOP} \rrbracket (c, \sigma)$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (c, \sigma)$

D'où $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c^a \rrbracket \sigma$

• $c^a = \mathbf{RED_SEQg}(c, c_2^a)$

$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket = (c_1^{pd}, \mathbf{SEQ}(c, c_1^s))$ avec $(c_1^{pd}, c_1^s) = \mathcal{C}^{sd} \llbracket c_2^a \rrbracket$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket \mathbf{SEQ}(c, c_1^s) \rrbracket (c_1^{pd}, \sigma)$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket c_1^s \rrbracket (\mathcal{C}^s \llbracket c \rrbracket (c_1^{pd}, \sigma))$

D'après le lemme 1 $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket c_1^s \rrbracket (c_1^{pd}, \mathcal{C} \llbracket c \rrbracket \sigma)$

Par induction, on obtient $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c_2^a \rrbracket (\mathcal{C} \llbracket c \rrbracket \sigma)$

D'où $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c^a \rrbracket \sigma$

• $c^a = \mathbf{RED_SEQd}(c_1^a, c)$

$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket = (c_1^{pd}, \mathbf{SEQ}(c_1^s, c))$ avec $(c_1^{pd}, c_1^s) = \mathcal{C}^{sd} \llbracket c_1^a \rrbracket$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket \mathbf{SEQ}(c_1^s, c) \rrbracket (c_1^{pd}, \sigma)$

$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket c \rrbracket (\mathcal{C}^s \llbracket c_1^s \rrbracket (c_1^{pd}, \sigma))$

Par induction, on obtient $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket c \rrbracket (\mathcal{C}^{spec} \llbracket c_1^a \rrbracket \sigma)$

Soit $(c_2^{pd}, \sigma_2) = \mathcal{C}^{spec} \llbracket c_1^a \rrbracket \sigma$

le lemme 1 donne $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (c_2^{pd}, \mathcal{C} \llbracket c \rrbracket \sigma_2)$
D'où $\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c^a \rrbracket \sigma$

• $c^a = \mathbf{REC_SEQ}(c_1^a, c_2^a)$

$$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket = (\mathbf{SEQ}(c_1^{pd}, c_2^{pd}), \mathbf{SEQ}(c_1^s, c_2^s))$$

$$\text{avec } \begin{cases} (c_1^{pd}, c_1^s) = \mathcal{C}^{sd} \llbracket c_1^a \rrbracket \\ (c_2^{pd}, c_2^s) = \mathcal{C}^{sd} \llbracket c_2^a \rrbracket \end{cases}$$

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket \mathbf{SEQ}(c_1^s, c_2^s) \rrbracket (\mathbf{SEQ}(c_1^{pd}, c_2^{pd}), \sigma)$$

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket c_2^s \rrbracket (\mathcal{C}^s \llbracket c_1^s \rrbracket (\mathbf{SEQ}(c_1^{pd}, c_2^{pd}), \sigma))$$

Comme c_1^s (resp. c_2^s) n'agit que sur les non-terminaux et les terminaux génériques de c_1^{pd} (resp. c_2^{pd}), on à

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (\mathbf{SEQ}(c_3^{pd}, c_4^{pd}), \sigma'')$$

$$\text{avec } \begin{cases} (c_3^{pd}, \sigma') = \mathcal{C}^s \llbracket c_1^s \rrbracket (c_1^{pd}, \sigma) \\ (c_4^{pd}, \sigma'') = \mathcal{C}^s \llbracket c_2^s \rrbracket (c_2^{pd}, \sigma') \end{cases}$$

$$\text{Par induction, on obtient } \begin{cases} (c_3^{pd}, \sigma') = \mathcal{C}^{spec} \llbracket c_1^a \rrbracket \sigma \\ (c_4^{pd}, \sigma'') = \mathcal{C}^{spec} \llbracket c_2^a \rrbracket \sigma' \end{cases}$$

$$\text{D'où } \mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c^a \rrbracket \sigma$$

• $c^a = \mathbf{REC_AFF}(i, e^a)$

$$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket = (\mathbf{AFF}(i, e_1^{pd}), c_1^s) \text{ avec } (e_1^{pd}, c_1^s) = \mathcal{E}^{sd} \llbracket e^a \rrbracket$$

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^s \llbracket c_1^s \rrbracket (\mathbf{AFF}(i, e_1^{pd}), \sigma)$$

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (\mathbf{AFF}(i, e_2^{pd}), \sigma') \text{ avec } (e_2^{pd}, \sigma') = \mathcal{C}^s \llbracket c_1^s \rrbracket (e_1^{pd}, \sigma)$$

La proposition 1 donne $(e_2^{pd}, \sigma') = (\mathcal{E}^{spec} \llbracket e^a \rrbracket \sigma, \sigma)$

$$\text{Et donc } \mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = (\mathbf{AFF}(i, \mathcal{E}^{spec} \llbracket e^a \rrbracket \sigma), \sigma)$$

$$\text{D'où } \mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) = \mathcal{C}^{spec} \llbracket c^a \rrbracket \sigma$$

• $c^a = \mathbf{RED_COND}(e, c_1^a, c_2^a)$

$$(c^{pd}, c^s) = \mathcal{C}^{sd} \llbracket c^a \rrbracket$$

$$(c^{pd}, c^s) = (\mathbf{NTERM}(nt), \mathbf{COND}(e, \mathbf{SEQ}(\mathbf{REGLE}(nt, c_1^{pd}), c_1^s), \mathbf{SEQ}(\mathbf{REGLE}(nt, c_2^{pd}), c_2^s)))$$

$$\text{avec } \begin{cases} (c_1^{pd}, c_1^s) = \mathcal{C}^{sd} \llbracket c_1^a \rrbracket \\ (c_2^{pd}, c_2^s) = \mathcal{C}^{sd} \llbracket c_2^a \rrbracket \end{cases}$$

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) =$$

$$\mathcal{C}^s \llbracket \mathbf{COND}(e, \mathbf{SEQ}(\mathbf{REGLE}(nt, c_1^{pd}), c_1^s), \mathbf{SEQ}(\mathbf{REGLE}(nt, c_2^{pd}), c_2^s)) \rrbracket (\mathbf{NTERM}(nt), \sigma)$$

$$\mathcal{C}^s \llbracket c^s \rrbracket (c^{pd}, \sigma) =$$

$$\begin{aligned}
& \text{if } \mathcal{E}[e]\sigma \text{ then } \mathcal{C}^s[\mathbf{SEQ}(\mathbf{REGLE}(nt, c_1^{pd}), c_1^s)](\mathbf{NTERM}(nt), \sigma) \\
& \quad \text{else } \mathcal{C}^s[\mathbf{SEQ}(\mathbf{REGLE}(nt, c_2^{pd}), c_2^s)](\mathbf{NTERM}(nt), \sigma) \\
\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & \\
& \text{if } \mathcal{E}[e]\sigma \text{ then } \mathcal{C}^s[\mathcal{C}_1^s](\mathcal{C}^s[\mathbf{REGLE}(nt, c_1^{pd})](\mathbf{NTERM}(nt), \sigma)) \\
& \quad \text{else } \mathcal{C}^s[\mathcal{C}_2^s](\mathcal{C}^s[\mathbf{REGLE}(nt, c_2^{pd})](\mathbf{NTERM}(nt), \sigma)) \\
\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & \text{if } \mathcal{E}[e]\sigma \text{ then } \mathcal{C}^s[\mathcal{C}_1^s](c_1^{pd}, \sigma) \text{ else } \mathcal{C}^s[\mathcal{C}_2^s](c_2^{pd}, \sigma) \\
\text{Par induction, il vient} & \\
\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & \text{if } \mathcal{E}[e]\sigma \text{ then } \mathcal{C}^{spec}[\mathcal{C}_1^a]\sigma \text{ else } \mathcal{C}^{spec}[\mathcal{C}_2^a]\sigma \\
\text{D'où } \mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & \mathcal{C}^{spec}[\mathcal{C}^a]\sigma
\end{aligned}$$

$$\begin{aligned}
& \bullet c^a = \mathbf{REC_COND}(e^a, c_1^a, c_2^a) \\
(c^{pd}, c^s) = \mathcal{C}^{sd}[\mathcal{C}^a](c^{pd}, c^s) = & (\mathbf{COND}(e_0^{pd}, c_1^{pd}, c_2^{pd}), \\
& \mathbf{SEQ}(c_0^s, \mathbf{SEQ}(\mathbf{PROT}(c_1^s), c_2^s)))
\end{aligned}$$

$$\text{avec } \begin{cases} (e_0^{pd}, c_0^s) = \mathcal{E}^{sd}[e^a] \\ (c_1^{pd}, c_1^s) = \mathcal{C}^{sd}[\mathcal{C}_1^a] \\ (c_2^{pd}, c_2^s) = \mathcal{C}^{sd}[\mathcal{C}_2^a] \end{cases}$$

$$\begin{aligned}
\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & \\
& \mathcal{C}^s[\mathbf{SEQ}(c_0^s, \mathbf{SEQ}(\mathbf{PROT}(c_1^s), c_2^s))](\mathbf{COND}(e_0^{pd}, c_1^{pd}, c_2^{pd}), \sigma) \\
\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & \\
& \mathcal{C}^s[\mathcal{C}_2^s](\mathcal{C}^s[\mathbf{PROT}(c_1^s)](\mathcal{C}^s[\mathcal{C}_0^s](\mathbf{COND}(e_0^{pd}, c_1^{pd}, c_2^{pd}), \sigma)))
\end{aligned}$$

Comme c_0^s (resp. c_1^s et c_2^s) n'agit que sur les non-terminaux et les terminaux génériques de e_0^{pd} (resp. c_1^{pd} et c_2^{pd}), on à

$$\begin{aligned}
\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = & (\mathbf{COND}(e_1^{pd}, c_3^{pd}, c_4^{pd}), \sigma''') \\
\text{avec } \begin{cases} (e_1^{pd}, \sigma') = \mathcal{C}^s[\mathcal{C}_0^s](e_0^{pd}, \sigma) \\ (c_3^{pd}, \sigma'') = \mathcal{C}^s[\mathbf{PROT}(c_1^s)](c_1^{pd}, \sigma') \\ (c_4^{pd}, \sigma''') = \mathcal{C}^s[\mathcal{C}_2^s](c_2^{pd}, \sigma'') \end{cases}
\end{aligned}$$

$$\text{La proposition 1 donne } \begin{cases} (e_1^{pd}, \sigma') = (\mathcal{E}^{spec}[e^a]\sigma, \sigma) \\ (c_3^{pd}, \sigma'') = \mathcal{C}^s[\mathbf{PROT}(c_1^s)](c_1^{pd}, \sigma) \\ (c_4^{pd}, \sigma''') = \mathcal{C}^s[\mathcal{C}_2^s](c_2^{pd}, \sigma'') \end{cases}$$

Soit encore $\mathcal{C}^s[\mathcal{C}^s](c^{pd}, \sigma) = (\mathbf{COND}(\mathcal{E}^{spec}[e^a]\sigma, c_3^{pd}, c_4^{pd}), \sigma''')$

$$\text{avec } \begin{cases} (c_3^{pd}, \sigma'') = \mathcal{C}^s[\mathbf{PROT}(c_1^s)](c_1^{pd}, \sigma) \\ (c_4^{pd}, \sigma''') = \mathcal{C}^s[\mathcal{C}_2^s](c_2^{pd}, \sigma'') \end{cases}$$

$$\text{D'où } \begin{cases} (c_3^{pd}, \sigma'') = (c_5^{pd}, \sigma) \\ (c_4^{pd}, \sigma''') = \mathcal{C}^s[\mathcal{C}_2^s](c_2^{pd}, \sigma) \end{cases}$$

$$\text{avec } (c_5^{pd}, _) = \mathcal{C}^s[\mathcal{C}_1^s](c_1^{pd}, \sigma)$$

$$\text{Par induction, il vient } \begin{cases} (c_5^{pd}, _) = \mathcal{C}^{spec}[\mathcal{C}_1^a]\sigma \\ (c_4^{pd}, \sigma''') = \mathcal{C}^{spec}[\mathcal{C}_2^a]\sigma \end{cases}$$

Soit encore $\begin{cases} (c_3^{pd}, -) & = \mathcal{C}^{spec}[[c_1^a]]\sigma \\ (c_4^{pd}, \sigma''') & = \mathcal{C}^{spec}[[c_2^a]]\sigma \end{cases}$

D'où $\mathcal{C}^s[[c^s]](c^{pd}, \sigma) = \mathcal{C}^{spec}[[c^a]]\sigma$

Chapitre 7

Mise en œuvre

Ce chapitre présente un évaluateur partiel pour le langage C, nommé Tempo, dans lequel nous avons intégré nos travaux. Tempo est la mise en œuvre d'une approche originale permettant de faire de la spécialisation statique et dynamique en partageant la plupart des analyses effectuées en amont de la spécialisation.

7.1 Tempo

Les techniques de l'évaluation partielle sont désormais suffisamment matures pour envisager leur application à des problèmes réalistes. Toutefois, la plupart des évaluateurs partiels actuels ne font aucune hypothèse sur le type des programmes qu'ils traitent. La précision des analyses, nécessaire à une telle généralité, interdit l'application de ces spécialiseurs à des programmes complexes. En revanche, la conception d'un évaluateur partiel pour une certaine classe de programmes permet de limiter la complexité des algorithmes de spécialisation en fonction des spécificités des programmes traités [CHN⁺96].

La complexité des langages traités par les évaluateurs partiels modernes oblige à structurer les analyses qu'ils pratiquent. Une architecture hors ligne avec analyse d'actions, comme celle décrite dans le chapitre 3, permet de maîtriser la richesse de ces langages. De plus, cette approche permet de traiter uniformément la spécialisation statique et dynamique [CHN⁺96].

Tempo est une mise en œuvre de cette approche pour le langage C. Il est spécialement conçu pour traiter du code système. L'examen de nombreuses lignes de code¹ a permis d'adapter la précision des analyses d'alias et de temps de liaison au traitement des programmes système. Cet examen a aussi permis la définition de l'ensemble des transformations de programme nécessaire à la spécialisation de tels programmes. La taille des programmes considérés a conduit à faire de Tempo un évaluateur partiel « orienté module ». En effet, contrairement à des

1. Ce travail a été réalisé en collaboration avec des équipes travaillant sur les systèmes d'exploitation.

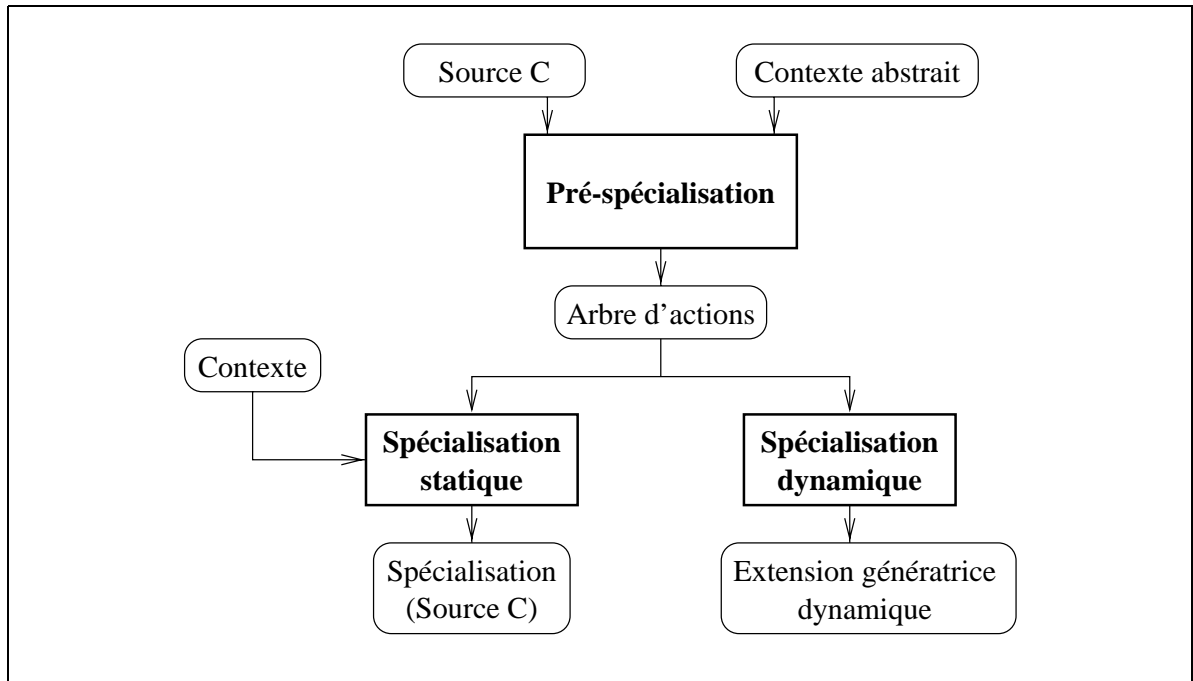


FIG. 7.1 – Une approche uniforme pour la spécialisation statique et dynamique.

spécialiseurs comme C-Mix [And92, And94], Tempo est capable de ne spécialiser qu’une partie du programme qui lui est fourni.

Comme le montre la figure 7.1, Tempo traite uniformément la spécialisation statique et dynamique. La phase de pré-spécialisation produit un arbre d’actions qui peut être utilisé pour produire, soit un programme résiduel à partir des valeurs de spécialisation, soit une extension génératrice dynamique. Cette dernière fournissant, à l’exécution, le code exécutable des spécialisations à partir des valeurs de spécialisation. Ainsi, la spécialisation dynamique est traitée de la même façon que la spécialisation statique. Seule l’exploitation de l’arbre d’actions est différente.

Les sections suivantes décrivent, plus en détails, les différentes analyses pratiquées lors de la pré-spécialisation ainsi que la branche concernant la spécialisation statique. La spécialisation dynamique est présentée en section 7.2.

7.1.1 Pré-spécialisation

Comme le montre la figure 7.2, la phase de pré-spécialisation de Tempo est constituée de quatre modules distincts. Le point de départ est un module écrit en langage C ainsi qu’un contexte abstrait. Ce contexte abstrait donne les temps de liaison des paramètres du point

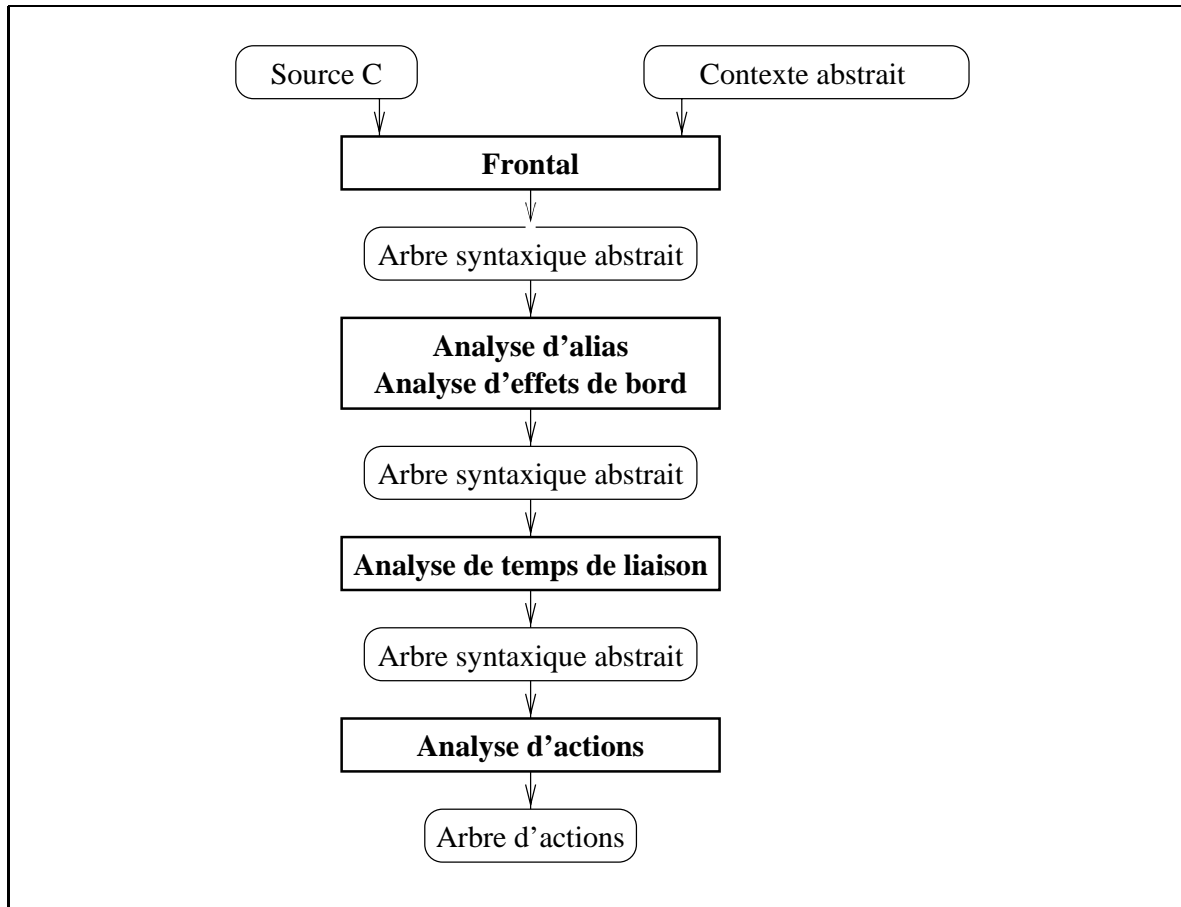


FIG. 7.2 – La phase de pré-spécialisation de Tempo.

d'entrée et des globales du module. La suite de cette section présente les différents modules de cette phase d'analyse.

Frontal

Cette première étape consiste à construire un arbre syntaxique abstrait à partir du module initial. Ce travail est réalisé à partir du frontal du système SUIF² légèrement modifié pour nos besoins. Le frontal de SUIF effectue l'analyse syntaxique du module et le réécrit dans un sous-ensemble du langage. Il est ainsi possible de traiter la quasi-intégralité du langage C en limitant les constructions que l'on doit analyser. Ainsi, la syntaxe abstraite de Tempo ne comporte, par exemple, qu'un seul type de boucles (**do ... while**) et une seule conditionnelle (**if ... then ... else ...**). Il n'y a ni expressions conditionnelles, ni déclarations de types imbriquées. Afin de ne pas avoir à traiter de programmes non-structurés, le frontal de Tempo procède ensuite à une élimination des instructions de branchement (**goto**) comme le décrivent Erosa et Hendren [EH93].

Analyse d'alias et d'effets de bord

Dans les langages comme C, les effets de bord ne sont pas toujours explicites. Il est possible de faire un effet de bord sur une variable via un pointeur. Pour permettre à l'analyse de temps de liaison de prendre correctement en compte les effets de bord, il convient d'effectuer une analyse déterminant l'ensemble des alias de chacune des variables du programme. L'analyseur d'alias de Tempo est inter-procédural, insensible au contexte mais sensible au flot [Ru95].

Après le calcul des alias, une analyse d'effets de bord détermine l'effet de chaque procédure sur la mémoire. Elle détermine aussi l'ensemble des procédures effectuant d'autres types d'effets de bord comme les entrées/sorties.

Analyse de temps de liaison

Comme l'analyse d'alias, l'analyse de temps de liaison de Tempo est tout spécialement adaptée à l'analyse de code système. Elle est sensible au contexte, sensible au flot et traite les structures de données partiellement statiques. Une structure de données partiellement statique est une structure possédant, à la fois, des champs statiques et des champs dynamiques. Cela conduit à permettre à un pointeur statique de référencer, à la fois, des objets statiques et dynamiques. Un tel pointeur peut donc être utilisé dans des contextes statiques ou des contextes dynamiques et possède donc des temps de liaison différents suivant les contextes d'utilisation. Comme pour la non sensibilité au flot, les évaluateurs partiels replient généralement ces contextes pour n'en faire qu'un [JGS93, And94]. De fait, cela se traduit par une perte de précision.

2. SUIF est un système conçu pour expérimenter l'optimisation de programmes parallèles [WFW⁺94].

Pour traiter ce problème, Hornof, Noyé et Consel ont introduit la notion de sensibilité aux références qui autorise différents temps de liaison pour une variable selon les contextes d'utilisation. Le lecteur trouvera une description complète de cette notion dans [HNC96].

Analyse d'actions.

Contrairement à la plupart des évaluateurs partiels, la spécialisation n'est pas effectuée directement à partir des informations fournies par l'analyse de temps de liaison. En effet, la phase de pré-spécialisation de Tempo se termine par une analyse d'actions similaire à celle présentée dans le chapitre 3. Cela permet une meilleure séparation entre le processus d'analyse et la production effective du code résiduel. C'est la clé de l'approche uniforme pour la spécialisation statique et dynamique.

7.1.2 Spécialisation statique

L'arbre d'actions produit par la phase de pré-spécialisation est interprété avec les valeurs de spécialisation afin d'obtenir le code source de la spécialisation. Comme nous l'avons montré dans la figure 3.13, l'interprétation d'un arbre d'actions utilise un interpréteur standard pour calculer les constructions évaluables. Pour éviter l'écriture ou la modification d'un interpréteur pour le langage C, les fragments évaluables de l'arbre d'actions sont emballés dans des procédures C [CHN⁺96]. Ces procédures sont compilées avec le compilateur GCC et le résultat obtenu est relié avec le spécialiseur.

7.2 Spécialisation dynamique avec Tempo

Cette section décrit la branche de Tempo chargée de produire des spécialiseurs dynamiques à partir des arbres d'actions issus de la phase de pré-spécialisation. Le compilateur que nous avons choisi pour compiler les patrons sources est le compilateur GCC et l'architecture cible est un processeur SPARC.

Tout en suivant la structure des modules de spécialisation dynamique décrits en figure 7.3, nous détaillons ici les points qui ne sont pas abordés dans le chapitre 5. Les points dépendants du compilateur ainsi que de la machine cible seront explicitement indiqués.

La figure 7.4 présente l'exemple d'une fonction C que nous utilisons pour illustrer nos transformations. Il s'agit d'une version simplifiée de la fonction **printf** du langage C. Une chaîne de contrôle, *ctrl_str*, définit le format d'impression des éléments du tableau *val*. Cette chaîne est constituée de caractères normaux, imprimés tels quels, ainsi que de quatre caractères spéciaux précédés du caractère « % ».

La figure 7.5 donne l'arbre d'actions pour la variable *ctrl_str* connue.

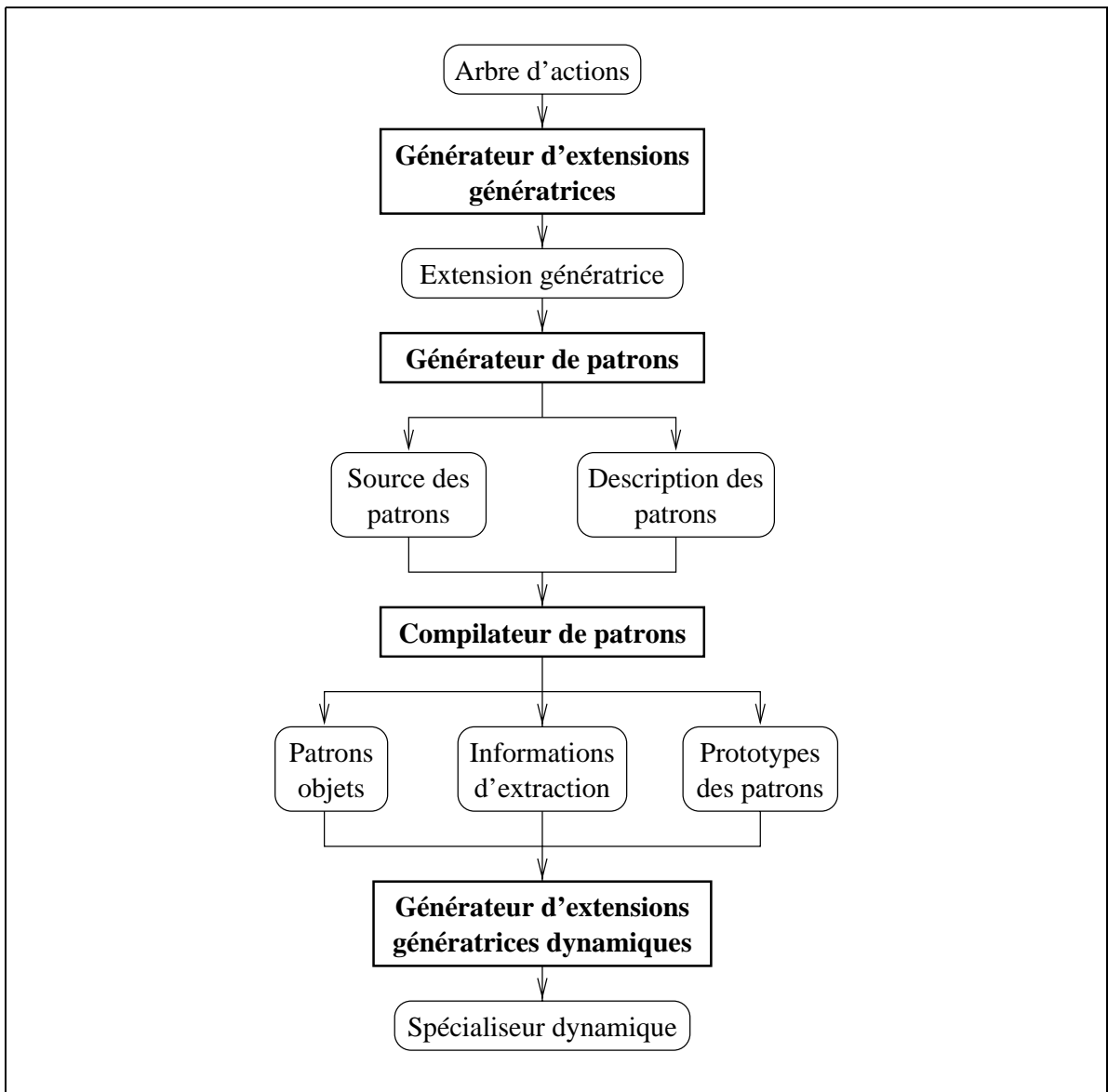


FIG. 7.3 – Organisation des modules de spécialisation dynamique

```
void print(char *ctrl_str, int *val)
{
    while (*ctrl_str)
    {
        if (*ctrl_str == '%')
        {
            ctrl_str = ctrl_str + 1;
            switch (*ctrl_str)
            {
                case 'd' :
                    putchar (*val + '0');
                    val = val + 1;
                    break;
                case 'c' :
                    putchar (*val);
                    val = val + 1;
                    break;
                case 't' :
                    putchar ('\t');
                    break;
                case 'n' :
                    putchar ('\n');
                    break;
            }
        }
        else putchar (*ctrl_str);
        ctrl_str = ctrl_str + 1;
    }
}
```

FIG. 7.4 – Une fonction d’affichage formaté

```

void print(char *ctrl_str, int *val)
{
  whilered (*ctrl_str){
    ifred (*ctrl_str == '%'){
      ctrl_str = ctrl_str + 1;
      switchred (*ctrl_str){
        case 'd' :
          {
            putchar (*val + '0');
            val = val + 1;
          }id
          break;
        case 'c' :
          {
            putchar (*val);
            val = val + 1;
          }id
          break;
        case 't' :
          putchar ('\t')id;
          break;
        case 'n' :
          putchar ('\n')id;
          break;
      }
    }redg
    elsered putcharrec ((*ctrl_str)ev);
    ctrl_str = ctrl_str + 1;
  }redd
}

```

FIG. 7.5 – L'arbre d'actions obtenu pour l'exemple de la figure 7.4 et la variable *ctrl_str* connue

7.2.1 Générateur d'extension génératrices

Le générateur d'extensions génératrices est une mise en œuvre directe des algorithmes du chapitre 6. Le résultat de ce module est un arbre abstrait.

Toutefois le langage traité par notre mise en œuvre est plus complet que le langage PLI. Le reste de cette section présente les points non abordés dans le chapitre 6.

Variables globales, pointeurs et structures de données. Ces constructions ne donnent pas lieu à des modifications conceptuelles dans la production des extensions génératrices. En effet, seules les analyses pratiquées lors de la phase de pré-spécialisation en sont affectées : introduction d'analyses d'alias et d'effets de bord, et adaptation des analyses de temps de liaison et d'actions.

Fonctions. Les programmes traités par Tempo ne sont pas limités à une seule fonction. Les arbres d'actions issus de la phase de pré-spécialisation sont donc constitués d'un ensemble d'arbres décrivant les différentes fonctions du programme. Comme, l'analyse de temps de liaison de Tempo est sensible au contexte, il peut y avoir plusieurs descriptions pour une même fonction. Chacune de ces descriptions porte un nom dérivés de celui de la fonction d'origine et des temps de liaison des globales utilisées par la fonction ainsi que ceux de ses paramètres ayant conduit à cette description. Le générateur d'extensions génératrices traite séparément les différentes descriptions de fonctions de l'arbre d'actions et produit une extension génératrice par description.

Appels de fonctions résiduelles. Le code résiduel produit pour un appel à une fonction résiduelle est simplement un appel à une fonction factice (car la véritable fonction ne sera construite qu'à l'exécution). Le spécialiseur correspondant est un appel à l'extension génératrice de la fonction appelée.

Boucles. Les boucles sont aussi traités par la branche dynamique de Tempo. Une boucle peut être soit résidualisée (i.e. reconstruite), soit déroulée (i.e. réduite). La reconstruction d'une boucle est traitée de la même façon que la reconstruction d'une conditionnelle. L'ensemble des spécialisations engendrées par le déroulage d'une boucle est celui des séquences des spécialisations du corps de la boucle. Il est possible de représenter cet ensemble par deux règles de grammaire du type suivant :

$$\begin{aligned} B &\rightarrow \mathbf{NOP} \\ B &\rightarrow \mathbf{SEQ}(c^{pd}, B) \end{aligned}$$

où c^{pd} est le code résiduel correspondant au corps de la boucle.

Cette technique a le désavantage d'introduire des récurrences dans les grammaire. Pour ne pas perdre, et avoir à retrouver, la structure du programme initial, nous avons choisit

d'emballer ces récurrences. Ainsi, l'ensemble des spécialisations de la boucle est représenté par la règle récursive $B \rightarrow^* \mathcal{E}^{p^d}$ ayant la même signification que les deux règles ci-dessus.

Le spécialiseur correspondant inclut alors une boucle chargé d'ajouter autant d'occurrence de \mathcal{E}^{p^d} qu'il est nécessaire.

La figure 7.6 donne l'extension génératrice calculée pour notre exemple.

7.2.2 Générateur de patrons

Ce module produit le texte combinant les patrons sources. La transformation de l'extension génératrice en un source combinant les patrons est décrite, pour le langage PLI, dans le chapitre 5. Nous décrivons ci-dessous les modification engendrées par l'ajout de fonctions et de boucle à notre langage.

Fonctions. Chaque extension génératrice est traitée séparément et est transformée en une fonction regroupant ses patrons.

Appels de fonctions résiduelles. Puisque les procédures résiduelles ne sont produites qu'à l'exécution, il n'est pas possible d'en connaître l'adresse lors de la génération des patrons. C'est une situation similaire à celle des trous produits par les fragments évaluable d'une expression. C'est pourquoi les appels de procédures résiduelles donnent lieu à des *trous d'appels*. Comme pour les trous, la représentation de ces objets dépend du compilateur utilisé pour compiler les patrons (nous présentons celle que nous avons choisit pour GCC ci-dessous). Il faut une représentation qui permet au compilateur de patrons de repérer ces tous d'appel et qui force GCC à produire une instruction d'appel. Lors de la production effective des spécialisations, le spécialiseur dynamique remplira les trous d'appel avec les adresses des procédures résiduelles dynamiquement construites.

Boucles. Comme pour le traitement des alternatives de la grammaire, une règle récursive, produite par le déroulage d'une boucle, est traduite par une boucle contrôlé par un identificateur externe. Ainsi, on préserve l'incertitude sur le nombre de fois que le corps de la boucle est répété.

En plus du code source des patrons, le générateur produit des informations indispensables à la récupération des patrons objets et au repérage de leurs trous. Ces informations consistent en une liste décrivant les patrons de chacune des fonctions de l'arbre d'actions. La description des patrons d'une fonction consiste en une liste donnant, pour chaque patron, son nom, la liste de ses trous ainsi que la liste de ses trous d'appels.

Les sections suivantes décrivent la manière dont les patrons sont délimités ainsi que la représentation des trous de valeurs et d'appels.

```

void gen_printSD(int *ctrl_str)
{
  F → int printSD(int *val){ B; }
  while (*ctrl_str){
    B →* C;
    if (*ctrl_str == '%'){
      C → S;
      ctrl_str = ctrl_str + 1;
      switch (*ctrl_str){
        case 'd' :
          S → { putchar (*val + '0'); val = val + 1; }
          break;
        case 'c' :
          S → { putchar (*val); val = val + 1; }
          break;
        case 't' :
          S → putchar ('\t');
          break;
        case 'n' :
          S → putchar ('\n');
          break;
      }
    }
    else
    {
      C → putchar (Ch0);
      Ch0 → *ctrl_str;
    }
    ctrl_str = ctrl_str + 1;
  }
}

```

FIG. 7.6 – L'extension génératrice de notre exemple

Représentation des trous de valeurs

La représentation des trous doit être telle que le compilateur produise des instructions immédiates pour ces trous. Ainsi, les trous se comporteront comme de véritables constantes. De plus, il doit être possible de retrouver les trous dans le code objet produit par le compilateur.

La première idée qui vient à l'esprit, pour représenter les trous, est d'utiliser des constantes particulières, une par trou, et différentes de toutes celles déjà présentes dans le code initial. Ainsi, un trou produit une instruction immédiate portant une constante donnée. Une analyse du code objet permet alors de la retrouver. Toutefois, cette représentation est impossible car les compilateurs, et en particulier GCC, optimisent généralement certaines opérations portant sur des constantes. Par exemple, la multiplication par une constante est remplacée par une séquence de décalages et d'additions. Il n'est donc pas possible de retrouver les trous après compilation.

Nous avons choisi de représenter les trous à l'aide de l'adresse d'un identificateur externe dont le nom est dérivé de celui du trou. Cette adresse est inconnue du compilateur car elle est déterminée lors de l'édition des liens. Ainsi, il ne pourra pas optimiser les opérations portant sur les trous. Pour ne pas rentrer en conflit avec le vérificateur de types de GCC, l'adresse est convertie dans le type du trou. Par exemple, l'expression

```
putchar (Ch0);
```

est transformée en

```
putchar ((char)&Ch0);
```

Comme chaque trou possède un nom unique, ils sont aisément retrouvés dans le code objet à l'aide des informations que le compilateur produit pour l'éditeur de liens.

La mise en œuvre actuelle ne supporte pas les trous dont le type nécessite plus de place que le stockage des adresses (32 bits pour la SPARC). Pour de tels objets, la meilleure solution consiste à utiliser une table. C'est d'ailleurs ainsi que les compilateurs traitent généralement ce type de constantes.

Représentation des trous d'appel

La représentation de ces objets est similaire à celle des trous de valeur. À chaque trou d'appel, on associe un identificateur externe dont le nom est dérivé de celui du trou. Le trou est alors représenté par l'adresse de cet identificateur forcé au type de la procédure résiduelle.

Délimitation des patrons

Cet aspect est dépendant du compilateur utilisé. GCC ajoute certaines extensions au langage C. En particulier, il est possible de prendre l'adresse d'une étiquette à l'aide de l'opérateur « && ». Le résultat est alors de type **void *** et pointe sur le code suivant l'étiquette. Nous

utilisons cette facilité pour permettre l'extraction des patrons du code objet retourné par le compilateur.

Chaque patron est délimité à l'aide de deux étiquettes dont les noms sont dérivés de celui du patron. Toutefois, comme il est impossible de mettre des étiquettes en dehors des fonctions, le premier patron d'une fonction (respectivement, le dernier) n'a pas d'étiquette de début (respectivement, de fin). Le début du premier (respectivement, la fin du dernier) patron de la fonction est repéré par la fonction elle-même (respectivement, par une fonction factice, placée juste après elle).

Pour permettre l'extraction des patrons du code objet, chaque fonction possède une table statique (au sens de C) initialisée avec l'adresse des étiquettes repérant ses patrons. Ainsi, l'examen de la zone de données initialisées permet de retrouver l'adresse de début et de fin de chaque patrons objet. Puisqu'il s'agit d'une table statique, elle ne perturbe en rien la compilation de la fonction.

La figure 7.7 montre le code source obtenu pour notre exemple.

7.2.3 Compilateur de patrons

Le compilateur de patrons commence par compiler le source des patrons à l'aide de GCC. Le résultat obtenu, augmenté d'un symbole par patron, constitue les patrons objets. Ces symboles repèrent le début des patrons et permettent une manipulation symbolique aisée de ces derniers par l'extension génératrice dynamique. Afin de rendre ces symboles accessibles, la compilation des patrons produit aussi les prototypes de ces symboles.

Avant de décrire les informations que le compilateur de patrons produit en plus du code objet, il convient de préciser la manière dont les extensions génératrices dynamiques produisent le code des spécialisations. Les spécialisations sont obtenues en recopiant le code des patrons, les uns à la suite des autres, dans une zone mémoire prévue à cet effet. Par exemple, le spécialiseur de notre fonction *print_{SD}* commence par recopier le code du patron *pat₀*. Ensuite, il interprète la chaîne de contrôle. Pour chaque caractère de cette chaîne, il recopie le code du patron correspondant. Finalement, il termine par la recopie du patron *pat₆*.

Toutefois, il convient de prendre certaines précautions. En effet, la recopie de code risque d'invalider les branchements. Sur une architecture SPARC, tous les branchements sont relatifs au compteur ordinal. Les branchements dont l'origine et la cible sont dans le même patron ne pose donc pas de problème car la différence des adresses reste la même.

Il est aussi possible d'avoir des branchements dont l'origine et la cible se trouvent dans des patrons différents. La figure 7.8 montre un exemple d'arbre d'actions engendrant ce type de situation ainsi que le source des patrons correspondant. La compilation du source des patrons produit un branchement du patron *pat₀* vers le patron *pat₃* (à cause de la conditionnelle).

Les branchements inter-patrons doivent être relogés car la taille du code entre l'origine et la cible du branchement n'est pas la même dans la fonction compilée par GCC et dans les spécialisations. De plus, on ne peut pas effectuer ce relogement lors de la compilation des

```

extern int N0, N1, N2;
extern int Ch;

void printSD(int *val)
{
    static void *patrons[] = { (void *)printSD, &&pat0_fin,
                                &&pat1_debut, &&pat1_fin,
                                ...,
                                &&pat6_debut, (void *)_printSD };

    pat0_fin :
    while (N0){
        if (N1){
            switch (N3){
                case 0 :
                    pat1_debut :
                    putchar (*val + '0'); val = val + 1;
                    pat1_fin :
                    break;
                case 1 :
                    pat2_debut :
                    putchar (*val); val = val + 1;
                    pat2_fin :
                    break;
                case 2 :
                    pat3_debut :
                    putchar ('\t');
                    pat3_fin :
                    break;
                case 3 :
                    pat4_debut :
                    putchar ('\n');
                    pat4_fin :
                    break;
            }
        }
        else {
            pat5_debut :
            putchar ((char)&Ch0);
            pat5_fin :
        }
        pat6_debut :
    }

    void _printSD(void){}

```

FIG. 7.7 – Le source des patrons fourni à GCC pour notre exemple

```

void f(...)
{
  ifrec (...)
    ifred (...) s1 elsered s2;
  elserec s3;
}

```

L'arbre d'actions

```

void f(...)
{
  static void *patrons[] = {...}

  ifrec (...)
  pat0-fin :
    if (N0){
      pat1-debut :
        s'1
      pat1-fin :
    }
    else {
      pat2-debut :
        s'1
      pat2-fin :
    }
  pat3-debut :
  elserec s'3;
}

```

Le source des patrons

FIG. 7.8 – Un exemple produisant un branchement inter-patron

patrons car la taille du code inséré entre l'origine et la cible du branchement n'est connue qu'à l'exécution. Sur l'exemple, on ne sait pas lequel des deux patrons pat_1 ou pat_2 sera sélectionné par l'extension génératrice dynamique.

Pour les mêmes raisons, les appels aux bibliothèques doivent aussi être relogés. Comme l'adresse d'implantation d'un patron n'est connue qu'à l'exécution, ce relogement ne peut pas être calculé lors de la compilation des patrons.

Le relogement de ces branchements incombe donc à l'extension génératrice dynamique. Pour qu'elle puisse faire ce travail, le compilateur de patrons identifie les branchements inter-patrons ainsi que les appels aux bibliothèques en analysant le code objet des patrons.

Une description des patrons objets du module complète la compilation des patrons. Elle est constituée d'une liste décrivant les patrons objets de chacune des fonctions. La description des patrons objets d'une fonction donne, pour chaque patron objet, son nom, sa taille, la liste des adresses de ses trous de valeurs et d'appels, la liste des branchements inter-patron et celle des appels aux bibliothèques.

Le compilateur de patrons est fortement dépendant du compilateur utilisé ainsi que de la machine cible. En effet, le format des instructions immédiates, le type des branchements, le format du code objet dépend du compilateur, de la machine et du système d'exploitation utilisé. Toutefois, la réécriture de ce module pour une autre architecture n'est pas une lourde tâche. En effet, le code source de notre compilateur de patrons s'élève à moins de 40Ko.

7.2.4 Générateur d'extensions génératrices dynamiques

Le code source de l'extension génératrice dynamique est obtenu en appliquant les transformations décrites dans le chapitre 5 à l'extension génératrice précédemment obtenue. Ensuite, les informations collectées lors de la compilation des patrons objets sont intégrées dans ce texte source. Cette opération est réalisée en remplaçant les instructions de copie de patrons et de remplissage des trous par des macros utilisant ces informations. Finalement, les listes de branchements inter-patrons ainsi que celle des appels aux bibliothèques sont utilisées pour insérer des macros assurant le relogement de ces branchements.

La figure 7.9 montre le texte source obtenu pour notre exemple. Les macros `reloge_appel()` sont insérées à cause des appels à la fonction de bibliothèque `putchar`.

Ensuite, ce texte source est compilé à l'aide de GCC. Le code objet obtenu est alors relié à celui des patrons objets. Le résultat de cette opération est le spécialisteur dynamique.

7.3 Conclusion

Tempo est donc capable de produire automatiquement un spécialisteur dynamique à partir d'un arbre d'actions. La production des spécialisations ne consiste qu'à recopier le code des patrons, à remplir leurs trous et à reloger quelques branchements. Ainsi, le sur coût engendré par la production dynamique de code est très faible. Il en résulte que l'amortissement est lui

```

void gen_printSD(int *ctrl_str)
{
  copie_patron (pat0, TAILLE_PAT0);
  while (*ctrl_str){
    if (*ctrl_str == '%'){
      ctrl_str = ctrl_str + 1;
      switch (*ctrl_str){
        case 'd' :
          copie_patron (pat1, TAILLE_PAT1);
          reloge_appel (pat1, PAT1_ADRESSE_APPEL0);
          break;
        case 'c' :
          copie_patron (pat2, TAILLE_PAT2);
          reloge_appel (pat2, PAT2_ADRESSE_APPEL0);
          break;
        case 't' :
          copie_patron (pat3, TAILLE_PAT3);
          reloge_appel (pat3, PAT3_ADRESSE_APPEL0);
          break;
        case 'n' :
          copie_patron (pat4, TAILLE_PAT4);
          reloge_appel (pat4, PAT4_ADRESSE_APPEL0);
          break;
      }
    }
    else
    {
      copie_patron (pat5, TAILLE_PAT5);
      reloge_appel (pat5, PAT5_ADRESSE_APPEL0);
      instancie (pat5, ADRESSE_CH0, *ctrl_str);
    }
    ctrl_str = ctrl_str + 1;
  }
  copie_patron (pat6, TAILLE_PAT6);
}

```

FIG. 7.9 – L'extension génératrice dynamique de notre exemple

aussi très faible et qu'il est possible d'utiliser la spécialisation dynamique même dans des cas où l'on utilise peu le code spécialisé.

Chapitre 8

Conclusion

Ce chapitre décrit brièvement les résultats de l'évaluation préliminaire de nos travaux. Ensuite, il récapitule les points importants de ce document et donne quelques pistes à explorer pour améliorer notre système.

8.1 Résultats

Ce n'est que récemment que la branche dynamique de Tempo est capable de procéder à l'analyse de programme de taille réelle¹. C'est pourquoi nous n'avons pas encore eu le temps d'évaluer notre approche sur des applications de taille importante.

Les exemples sur lesquels nous avons testé notre système de spécialisation dynamique sont des petits modules constitués de quelques lignes. Par exemple, nous avons spécialisé dynamiquement une fonction de formattage de chaînes de caractères. Nous avons aussi appliqué nos analyses à la multiplication de matrices creuses.

Les mesures préliminaires effectuées sur ces petit exemples ont montré des amortissements généralement inférieurs à 10 et des accélérations variant entre 1 et 4. Parfois, le code spécialisé est moins rapide. Par exemple, le déroulage excessif de boucles peut rendre le cache quasi-inopérant. Toutefois, ces cas sont rares et en général le bilan est positif.

8.2 Récapitulatif

L'évaluation partielle est une transformation de programme traditionnellement effectuée à la compilation. Elle permet aux applications de s'adapter à des contextes particuliers qui doivent être connus avant l'exécution. Il est possible d'utiliser directement ces techniques pour faire de la spécialisation dynamique en prévoyant statiquement toutes les spécialisations

1. En effet, les versions préliminaires n'acceptaient pas toutes les constructions du langage C

différentes (aux valeurs près) et en les sélectionnant, à l'exécution, en fonction des valeurs des constantes d'exécution. Toutefois, cette méthode n'est applicable que lorsque le nombre de spécialisations est faible. Dans les autres cas, on risque d'obtenir une explosion de code.

Au moment où nous avons débuté nos travaux, la plupart des expériences de spécialisation dynamique étaient spécifiques à des applications particulières. Certains systèmes généraux permettaient de spécifier proprement des spécialiseurs dynamiques. Cependant, à part Fabius, aucun système ne permettait de les produire automatiquement à partir d'un programme et d'une description des constantes d'exécution. Toutefois, Fabius ne s'applique qu'à un sous-ensemble pur et du premier ordre du langage ML ; aucun travaux n'existaient sur la spécialisation dynamique de programmes impératifs.

Nos travaux ont permis d'étendre l'évaluation partielle aux cas où les valeurs de spécialisation ne sont connues qu'à l'exécution. Les techniques de l'évaluation partielle statique nous ont permis de développer une méthode permettant de produire automatiquement des spécialiseurs dédiés dynamiques. L'implantation de nos travaux dans l'évaluateur partiel Tempo montre que la spécialisation statique et la spécialisation dynamique partagent toute la phase de pré-spécialisation.

Nous avons défini formellement la production des extensions génératrices afin d'en détailler le fonctionnement et d'en prouver la correction.

Comme le montre l'évaluation préliminaire de notre système, les amortissements obtenus sont faibles. Les opportunités d'application sont, de fait, nombreuses. La raison de ces faibles amortissements est le faible coût de la production dynamique de code qui est assurée par la simplicité des opérations pratiquées par le spécialiseur dynamique. Ces opérations sont très éloignées de celles que pratique un compilateur. De ce fait, Tempo n'est pas un compilateur dynamique.

Toutefois, le code des spécialisations peut être de moins bonne qualité que celui produit par un compilateur dynamique. En effet, le principe de base de ces compilateurs est l'optimisation du code dynamique en fonction des valeurs connues à l'exécution. En revanche, le coût de ces optimisations engendre des amortissements élevés réduisant, du même coup, le champ d'application de ces techniques.

La portabilité du système est aussi un critère important. L'approche que nous proposons est totalement indépendante de la machine cible jusqu'à la production de l'extension génératrice. Seul le compilateur de patrons, et une partie mineure du générateur d'extensions génératrices sont à réécrire pour porter le système sur une autre architecture.

8.3 Perspectives et directions futures

Comme le montre la section 8.1, nous n'avons évalué notre approche que sur un nombre réduit de cas simples. Il conviendrait de l'appliquer à des exemples plus réalistes de manière à l'évaluer plus précisément.

Pour l'instant, les patrons sont compilés avec GCC sans optimisation. GCC est considéré comme un très bon compilateur. Toutefois, il produit un code de mauvaise qualité si on n'utilise pas les optimisations qu'il propose. Il n'est pas possible de compiler les patrons avec les optimisations sans prendre des précautions. Par exemple, certaines optimisations risquent de déplacer des morceaux de code hors de leur patron. Le compilateur de patrons ne pourra alors pas récupérer les patrons objets. D'autres optimisations risquent de propager des références aux trous hors de leurs patrons. Le remplissage des trous, tel que nous l'avons présenté, est alors impossible.

Une solution est d'utiliser un compilateur qui optimise localement à un bloc de base et non pas globalement. Cette granularité d'optimisation convient pour les patrons car ces derniers sont toujours constitués de blocs complets. Nous avons exploré cette voie avec le compilateur LCC et les résultats obtenus sont encourageants bien que la portée des optimisations soit limitée. Les expériences que nous avons faites avec GCC, avec optimisation, montrent que les étiquettes placées autour des patrons semblent interdire aux optimiseurs de bouger du code au delà de ces barrières. De plus, il semble possible d'introduire un traitement spécifique permettant d'autoriser les optimisations propageant des références aux trous hors de leur patron. Il serait souhaitable de poursuivre plus loin ces investigations de manière à permettre la compilation optimisante des patrons à l'aide de GCC.

Une autre direction consisterait à abstraire l'architecture cible du compilateur de patrons. Il serait en effet possible d'utiliser un mécanisme de description de machines similaire à celui sur lequel repose GCC pour la production du code machine. Cette abstraction serait très limitée car seulement le format du code objet, ainsi qu'une description des instructions de branchement seraient nécessaires.

Il serait aussi intéressant de définir des transformations de programmes sources permettant d'exprimer des optimisations relatives aux valeurs des constantes d'exécution. Contrairement aux compilateurs dynamiques, ces transformations resteraient au niveau du source. Il serait ainsi possible de faire des optimisations relatives aux valeurs de spécialisation sans faire d'optimisations dynamiques.

Bibliographie

- [AC94] J. M. Ashley and C. Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1431–1448, 1994.
- [And92] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, San Francisco, CA, USA, June 1992. Yale University, Hew Haven, CT, USA. Technical Report YALEU/DCS/RR-909.
- [And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.
- [APC⁺96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI96 [PLD96], pages 149–159.
- [BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In PEP94 [PEP94], pages 119–132.
- [BW93] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, Computer Science Department, University of Copenhagen, 1993. Research Report 93/22.
- [CD90] C. Consel and O. Danvy. From interpreting to compiling binding times. In N.D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 1990.
- [CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

- [CHN⁺96] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.-N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
- [CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [POP96], pages 145–156.
- [Con93] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.
- [EH93] A.M. Erosa and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. ACAPS Technical Memo 76, School of Computer Science, McGill University, Montreal, Canada, September 1993.
- [EHK96] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [POP96], pages 131–144.
- [Eng96] D.R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In PLDI96 [PLD96], pages 160–170.
- [EP94] D.R. Engler and T.A. Proebsting. DCG: An efficient retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 263–273. ACM Press, November 1994.
- [Ers77] A.P. Ershov. On the essence of translation. *Computer Software and System Programming*, 3(5):332–346, 1977.
- [FA91] Henry R. H. Fraser, C. W. and Proebsting T. A. Burg - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1991.
- [FH91] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software - Practice and Experience*, 21(9):963–988, 1991.
- [HNC96] L. Hornof, J. Noyé, and C. Consel. Accurate partial evaluation of realistic programs via use sensitivity. Rapport de recherche, Inria, Rennes, France, June 1996.
- [JGS93] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

- [JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.
- [KEH91] D. Keppel, S. Eggers, and R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science, University of Washington, Seattle, WA, 1991.
- [KEH93] D. Keppel, S. Eggers, and R. Henry. Evaluating runtime compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science, University of Washington, Seattle, WA, 1993.
- [KKZG94] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. In U. Meyer G. Snelting, editor, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.
- [LL93] M. Leone and P. Lee. Deferred compilation: The automation of run-time code generation. Technical Report CMU-CS-93-225, School of Computer Science, Carnegie Mellon University, December 1993.
- [LL94] M. Leone and P. Lee. Lightweight run-time code generation. In PEP94 [PEP94].
- [LL96] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI96 [PLD96], pages 137–148.
- [Loc87] B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.
- [Mal94] K. Malmkjaer. *Abstract Interpretation of Partial Evaluation algorithms*. PhD thesis, Yale University, March 1994.
- [Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105, Hew Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).
- [MP89] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *ACM Symposium on Operating Systems Principles*, pages 191–201, 1989.
- [PAB⁺95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *ACM Symposium on Operating Systems Principles*, 1995. To appear.

- [PEK96] M. Paoletto, D.R. Engler, and M.F. Kaashoek. tcc: A template-based compiler for 'c. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, Tucson, AZ, USA, February 1996.
- [PEP94] *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Technical Report 94/9, University of Melbourne, Australia, 1994.
- [PLD96] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 31(5), May 1996.
- [PLR85] R. Pike, B. N. Locanthi, and J.F. Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software - Practice and Experience*, 15(2):131–151, 1985.
- [PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *ACM Computing Systems*, 1(1):11–32, 1988.
- [POP96] *Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.
- [Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22. ACM SIGPLAN Notices, 30(6), June 1995.
- [WFW⁺94] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 94.

Résumé

La spécialisation de programmes en fonction d'invariants connus à l'exécution est une technique d'optimisation qui améliore notablement les performances. Elle permet aux applications de s'adapter à des contextes d'utilisation qui ne sont connus qu'à l'exécution et dont la spécificité ne peut donc pas être prise en compte à la compilation.

Cette technique est activement étudiée dans des domaines très variés. Par exemple, d'importants projets de recherche utilisent la spécialisation à l'exécution pour optimiser des systèmes d'exploitation hautement extensibles et paramétriques.

Cette thèse décrit une méthode générale permettant de spécialiser des programmes à l'exécution. À partir d'un programme et d'une description de ses invariants, nous produisons automatiquement à la compilation des fragments de code source incomplets, dans la mesure où les invariants ne seront connus qu'à l'exécution. Ces derniers sont ensuite transformés de manière à pouvoir être traités par un compilateur standard. À l'exécution, il ne reste plus qu'à sélectionner et copier les fragments, insérer les valeurs dynamiques et reloger certains sauts pour obtenir une spécialisation donnée.

Ces opérations sont simples et permettent donc un processus de spécialisation très efficace, qui ne nécessite qu'un petit nombre d'exécutions du programme spécialisé avant d'être amorti.

Les contributions de notre méthode sont les suivantes : elle permet une dérivation automatique des fragments à partir du programme et de ses invariants ; elle est portable ; elle est formellement définie ; enfin, elle est efficace comme le montre notre mise en œuvre pour le langage C.

Mots clés

Spécialisation dynamique, spécialisation statique, évaluation partielle, optimisation de programmes, compilation.