N° d'ordre: 2734

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Anne-Françoise Le Meur

Équipe d'accueil : COMPOSE
École Doctorale : MATISSE
Composante universitaire : IFSIC/IRISA

Titre de la thèse :

Approche déclarative à la spécialisation de programmes C

soutenue le 13 Décembre 2002 devant la commission d'examen

M. :	Jean-Pierre	Banâtre	Président
MM. :	Pierre	Cointe	Rapporteurs
	Olivier	Danvy	
MM. :	Krzysztof	Czarnecki	Examinateurs
	Thomas	Jensen	
	Charles	Consel	

Remerciements

Je tiens tout d'abord à remercier les membres du jury :
- Jean-Pierre Banâtre, professeur des Universités à Rennes I, qui m'a fait l'honneur de présider ce jury.
- Pierre Cointe, professeur à l'École des Mines de Nantes et responsable du projet Obasco et Olivier Danvy, professeur à l'Université d'Aarhus (Danemark), qui ont accepté la lourde tâche de rapporteur.
- Krzysztof Czarnecki, professeur assistant à l'Université de Waterloo (Canada) et Thomas Jensen, chargé de recherche au CNRS et responsable du projet Lande à l'IRISA, qui ont bien voulu faire partie de ce jury et examiner en détail ce document.
- Charles Consel, professeur des Universités à l'ENSEIRB et responsable du projet Compose, qui m'a donné l'opportunité de réaliser cette thèse. Il m'a fait bénéficier de ses compétences et de sa rigueur. Son dynamisme et son optimisme de tous les jours m'ont permis de mener ce travail à bien et je lui en serai pour toujours reconnaissante.

Je tiens aussi à remercier vivement Julia L. Lawall qui a été présente à de nombreuses occasions durant ces trois dernières années. Sa collaboration à ce travail a été décisive et des plus enrichissante. De plus, le soutien sans faille qu'elle m'a apporté pendant la rédaction de ce document a été des plus précieux. Enfin, je tiens à souligner la disponibilité, la patience et l'excellence dont elle fait preuve au quotidien.

Je remercie également tous les membres du projet Compose. Je pense tout particulièrement à Gilles, Renaud, Ulrik, Luciano, Benoît, Patrice, Xavier. Sans oublier bien sûr Laurent, mon ancien collègue de bureau, que je remercie pour son soutien, son amitié et tous nos petits moments de délire qui ont agrémenté notre quotidien. Je fais aussi un petit clin d'oeil à toutes les secrétaires du groupe : Catherine, Sabine, Nicole et Simone.

Je voudrais de plus mentionner plusieurs personnes rencontrées tout au long du chemin qui m'a mené à l'obtention de cette thèse. Je pense notamment à Jacques Noyé qui m'a fait faire mes premiers pas dans le monde de la recherche ; à Steve Beattie, Ashvin Goel, Lois Delcambre et Shirley Kapsch, ainsi que tous les autres qui ont compté lors de mon passage à OGI. De plus, je remercie Josiane Guégan ainsi que Rémi et Michou Le Guet qui m'ont fait l'amitié d'assister à ma soutenance.

Enfin et surtout, je remercie mes parents et ma soeur Catherine qui sont depuis toujours mes piliers. Je remercie mes parents pour toutes les opportunités qu'ils m'ont offertes et pour leur soutien inconditionnel. Je remercie ma soeur pour tous ses encouragements et pour nos moments de complicité qui me tiennent tant à coeur. C'est à eux trois que je dédie cette thèse.

Résumé

L'évaluation partielle est une transformation de programmes qui permet de spécialiser automatiquement un programme pour un contexte d'utilisation donné. Cette technique de spécialisation a suscité beaucoup d'attention ces vingt dernières années et a donné lieu à de nombreuses avancées aussi bien théoriques que pratiques. Ainsi, la spécialisation de programme a été étudiée pour les langages fonctionnels, impératifs, logiques et à objets. Plusieurs spécialiseurs ont été développés pour des langages à taille réelle tels que C ou Java et ont été utilisés avec succès pour de nombreuses applications dans des domaines aussi variés que les systèmes d'exploitation, le calcul scientifique et le graphisme.

En dépit des nombreux succès de la spécialisation de programmes, cette technique n'est pas encore accessible à des programmeurs non-experts. Une raison de cette non-accessibilité est qu'il est difficile de décrire les opportunités de spécialisation. Il est de ce fait courant qu'un programme soit trop ou insuffisamment spécialisé. Le programmeur doit alors faire face au délicat problème de comprendre pourquoi le programme spécialisé ne correspond pas à ce qu'il attendait.

Nous avons développé un langage de déclarations haut niveau qui permet au programmeur de préciser quels sont les fragments de code et les invariants qui doivent être pris en considération par le processus de spécialisation. Ces déclarations sont distinctes du code et sont écrites par le programmeur lors développement du programme. La syntaxe des déclarations est similaire à la syntaxe du langage traité, dans notre cas C. Les déclarations sont vérifiées avant la phase de spécialisation afin de renseigner le programmeur quant à la faisabilité de la spécialisation désirée.

Cette approche permet de rendre le processus de spécialisation prévisible par rapport aux déclarations et offre ainsi la possibilité d'envisager la spécialisation automatique et systématique de composants logiciels. Nous avons utilisé notre approche dans le cadre de nombreux exemples et notamment lors du développement d'un composant spécialisable dans le domaine des codes correcteurs d'erreurs.

Nous avons conçu et implémenté un compilateur pour le langage de déclarations, ainsi qu'un environnement graphique. Cet environnement offre aux programmeurs des outils d'aide au développement de composants spécialisables et permet aux utilisateurs de ces composants de les spécialiser en fonction de leur besoin.

Mots clés

Spécialisation de programmes, génie logiciel.

Abstract

Partial evaluation is a program transformation that aims at automatically specializing a program with respect to a given usage context. This specialization technique has been studied intensively for the past twenty years, and has now reached a mature state, both theoretically and practically. Many variations have been explored with respect to language paradigms and features. These studies have led to the design and implementation of specializers for real-sized languages such as C and Java.

Despite successful application in areas such as graphics, operating systems, and scientific computing, specializers have yet to be widely used. One obstacle to a wider use is the difficulty for a non-expert programmer to adequately describe specialization opportunities. Indeed, under-specialization or over-specialization often occurs, without any direct feedback to the programmer as to the cause of the problem.

We have developed a high-level, module-based declaration language allowing the programmer to guide the choice of both the code to specialize and the invariants to exploit during the specialization process. These declarations are provided, on the side, by the programmer, as the program is being developed. The syntax of this language is similar to the declaration syntax of the language in which the program is written. The declarations are checked prior to specialization to provide feedback to the programmer.

This approach allows the specialization process to be predictable with respect to the declarations, enabling the automatic and systematic specialization of software components. Our approach has been applied in the context of various examples such as the development of a specializable forward error correction encoder component.

We have designed and implemented a compiler for our declarative language, as well as a graphical environment both to assist a component programmer in the creation of a specializable component and to enable a component user to tailor a component to a given application.

Keywords

Program specialization, software engineering

Table des matières

Table des figures

Chapitre 1

Introduction

La spécialisation de programmes est une technique de transformation qui, étant donnés un programme générique et un contexte d'exécution particulier, produit un programme équivalent au programme générique mais dédié au contexte d'exécution. Intuitivement, la spécialisation a pour but d'instancier un programme par rapport à certains de ses paramètres d'entrée, permettant ainsi d'obtenir un code adapté pour un contexte d'utilisation donné. Plus concrètement, la spécialisation réalise une propagation de constantes agressive et inter-procédurale qui inclut entre autre des pliages de constantes, ainsi que des optimisations telles que le dépliage de fonction et le déroulage de boucle.

Pour illustrer l'intérêt d'une telle transformation, prenons l'exemple d'un interprète de filtres de paquets, l'interprète des "Berkeley Packets Filters" (BPF) [MJ93]. Cet interprète est utilisé dans les systèmes d'exploitation pour filtrer les paquets arrivant du réseau avant de les transmettre à une application donnée. Seuls les paquets qui satisfont certains critères sont transmis. Ces critères prennent la forme d'un filtre, c'est-à-dire un programme de filtrage, qui est interprété. C'est précisément cette couche d'interprétation qui rend l'interprète BPF générique et qui lui permet d'être utilisé par n'importe quelle application. Cependant cette généricité a un coût en terme de performance. En effet, pour une session donnée, le programme de filtrage est toujours le même, et est donc interprété des milliers de fois afin d'examiner les paquets circulant sur le réseau. Pour éviter ces interprétations successives, il est intéressant de spécialiser l'interprète BPF pour un filtre donné. Cette spécialisation a pour but d'instancier l'interprète pour le filtre donné, c'est-à-dire d'effectuer tous les calculs qui dépendent du filtre. Le résultat de cette spécialisation doit être un programme dédié où toute la couche d'interprétation a disparu et donc où la généricité qui dépendait du filtre dans le programme original n'apparaît plus. Si effectivement le programme obtenu présente cette caractéristique alors la spécialisation peut être considérée comme réussie.

Plusieurs spécialiseurs ont été développés et utilisés avec succès pour de nombreuses applications dans des domaines aussi variés que les systèmes d'exploitation, le calcul scientifique et le graphisme. Cependant, en dépit des nombreux succès de la spécialisation de programmes, cette technique n'est pas encore accessible à tous, c'est-à-dire

qu'il est souvent difficile pour des programmeurs non-experts en spécialisation de programmes d'obtenir une spécialisation réussie. Ainsi, il est fréquent que la spécialisation d'un programme par rapport à certains de ses paramètres d'entrée produise un programme sous ou sur-spécialisé. Dans le premier cas, non seulement la généricité qui dépendait des paramètres connus a disparu mais d'autres simplifications, non désirées, ont aussi été effectuées. Dans le second cas, toute la généricité qui dépendait des paramètres connus n'a pas été éliminée. Le programmeur doit dans les deux cas faire face au délicat problème de comprendre pourquoi le programme obtenu ne correspond pas à ce qu'il attendait.

Cette non-accessibilité est en partie due au fait que la recherche dans ce domaine s'est principalement intéressée à l'outil de transformation plutôt qu'à l'intégration de la spécialisation de programmes dans le processus du développement logiciel. Ainsi, la complexité du moteur de spécialisation a été accrue pour permettre aux spécialiseurs de traiter des applications de plus en plus compliquées, rendant par la même occasion la paramétrisation de tels spécialisateurs difficile à effectuer.

De plus, jusqu'à maintenant, la *spécialisabilité* d'un programme, c'est-à-dire la présence d'opportunités de spécialisation dans un programme, a toujours été envisagée *a posteriori*, c'est-à-dire une fois le programme développé. Cependant, par expérience, la spécialisabilité d'un programme n'est pas prévisible car elle dépend des structures de contrôle du programme, de l'agencement des données et du spécialiseur utilisé. Cette imprévisibilité n'est pas propre à la spécialisation de programmes. Par exemple, le degré d'optimisation effectuée par un compilateur, pour un programme donné, est lui aussi imprévisible : il dépend de la présence ou non dans le programme de motifs qui correspondent à des cas d'optimisation prévus. Pour éviter les incertitudes, il est donc essentiel que le programmeur prenne en compte la spécialisation durant le développement d'un programme et qu'il obtienne de la part du spécialiseur un retour d'information concernant le degré de spécialisation qui peut être effectuée.

1.1 Thèse

La thèse présentée dans ce document propose une approche déclarative à la spécialisation de programmes C qui a pour but de rendre cette technique plus accessible aux non-experts.

Notre approche repose sur un langage déclaratif qui permet de spécifier les opportunités de spécialisation d'un programme. Les déclarations sont écrites par le programmeur lors du développement du code. Non seulement les déclarations offrent un cadre de raisonnement qui permet au programmeur de réfléchir à la spécialisibilité de son programme mais elles correspondent aussi aux contraintes à respecter afin de garantir que toutes les opportunités de spécialisation sont bien prises en compte pendant le processus de spécialisation. Ainsi la vérification de ces contraintes par le spécialiseur renseigne le programmeur quant à la faisabilité de la spécialisation désirée.

Notre approche permet donc de rendre le processus de spécialisation prévisible par rapport aux déclarations, et offre ainsi la possibilité d'envisager la spécialisation auto-

matique et systématique de composants logiciels.

1.2 Contributions

Les contributions de cette thèse sont doubles. Dans un premier temps, nous présentons notre approche. Nous avons développé un langage de déclarations qui permet de spécifier les opportunités de spécialisation d'un programme, et une stratégie de vérification. Dans un deuxième temps, nous commençons à étudier comment la spécialisation peut s'intégrer dans un processus de développement logiciel. Nous décrivons dans ce but le développement, avec notre approche, d'un composant spécialisable pour le domaine des codes correcteurs d'erreurs. Enfin, nous présentons un environnement graphique que nous avons développé pour assister le développement et la spécialisation de composants logiciels.

Langage de déclarations Nous facilitons l'accès à la spécialisation de programmes C en proposant un langage haut niveau, proche de la syntaxe du langage de programmation C, qui abstrait les concepts internes de la spécialisation de programmes par des déclarations intuitives et faciles à utiliser. Nous avons développé un compilateur pour notre langage qui permet de simplifier l'utilisation du spécialiseur pour C nommé Tempo.

Vérifications L'utilisation de notre langage permet de rendre le processus de spécialisation prévisible par rapport aux déclarations : le programmeur spécifie les opportunités de spécialisation d'un programme et les vérifications s'assurent qu'elles sont bien prises en compte par le processus de spécialisation. Nous avons implémenté notre stratégie de vérification dans le spécialiseur Tempo.

Composants spécialisables Nous avons utilisé notre approche pour développer un composant logiciel spécialisable pour le domaine des codes correcteurs d'erreurs (CCE). Les CCE permettent d'éviter la perte de données lors de communications numériques en transmettant des informations redondantes. Un encodeur CCE présente plusieurs caractéristiques qui sont amenées à changer selon son contexte d'utilisation. Notre approche permet de générer automatiquement autant d'encodeurs spécifiques qu'il y a de contextes d'utilisation.

Environnement graphique Nous avons développé deux interfaces graphiques. Une interface permet au programmeur de visualiser les déclarations de spécialisation associées à un composant et de générer un composant spécialisable c'est-à-dire, un spécialiseur dédié pour un composant donné. L'autre interface joue le rôle d'une boîte noire à travers laquelle les utilisateurs peuvent obtenir des implémentations spécialisées du composant spécialisable.

1.3 Organisation du document

Ce document est organisé en trois parties. Nous présentons tout d'abord la problématique dans laquelle s'inscrit la spécialisation de programmes. Puis, nous décrivons notre approche. Nous proposons ensuite une intégration de notre approche dans un processus de développement logiciel.

Contexte Le chapitre 2 donne un aperçu de la spécialisation de programmes. Nous y détaillons en particulier le spécialiseur Tempo. Les raisons de la non-accessibilité de la spécialisation de programmes sont présentées dans le chapitre 3.

Approche proposée Le chapitre 4 donne une description globale de notre approche. Les détails du langage de déclarations sont présentés dans le chapitre 5 et les vérifications dans le chapitre 6. Pour terminer la présentation de notre approche, le chapitre 7 décrit tout d'abord comment le langage de déclarations peut permettre d'adresser les problèmes de spécialisation tels que la sous ou la sur-spécialisation, puis fait le point sur l'expressivité du langage et enfin présente des travaux connexes permettant au programmeur de contrôler le processus de spécialisation.

Dimension génie logiciel Le chapitre 8 propose une intégration de la spécialisation de programmes dans le développement de composants logiciels. Nous appliquons notre approche pour développer un encodeur CCE spécialisable. Ensuite, dans le chapitre 10, nous décrivons notre environnement graphique d'aide au développement de composants logiciels spécialisables. Pour conclure, le chapitre 11 récapitule nos contributions et donne les directions futures de nos travaux.

Première partie

Contexte

Chapitre 2

Spécialisation de programmes

L'évaluation partielle est une technique de transformation de programmes automatique qui a pour but de spécialiser un programme pour un contexte d'utilisation donné. Cette technique de spécialisation a suscité beaucoup d'attention ces vingt dernières années et a donné lieu à de nombreuses avancées théoriques et pratiques. Ainsi, la spécialisation de programmes a été étudiée pour les langages fonctionnels [Bon90, Con93], impératifs [And94, BGZ94, CHN+96a, KKZG94], logiques [LS91] et à objets [SLCM99]. De plus, des évaluateurs partiels permettant de traiter des langages de taille réelle tels que Fortran [BGZ94], C [And94, CHN+96a, GMP+00] et Java [SLCM99], ont été développés. Ces spécialiseurs ont été utilisés avec succès sur des applications qui couvrent des domaines aussi variés que les systèmes d'exploitation [KM00, MWP+01, MMV+98], le calcul scientifique [BS94, Law98, BGZ94] et le graphisme [And96].

2.1 Présentation

L'évaluation partielle prend en entrée un programme, ainsi que l'ensemble des paramètres de ce programme qui ont une valeur connue au moment de la spécialisation et produit un nouveau programme dit *résiduel*. Le programme résiduel est généralement plus efficace que le programme initial. Ce gain d'efficacité est le résultat de l'élimination de tous les calculs *invariants*, c'est-à-dire de ceux qui dépendent seulement de valeurs connues. Le programme résiduel ne contient ainsi plus que des calculs dépendants de valeurs indisponibles au moment de la spécialisation.

2.1.1 Exemple

Considérons la fonction `dotproduct` qui prend en arguments deux vecteurs d'entiers `u` et `v`, et leur taille `size` et qui calcule le produit scalaire des deux vecteurs.

```
int dotproduct (int *u, int *v, int size) {
  int i ;
  int sum = 0 ;
  for(i = 0 ; i < size ; i++)
    sum = sum + u[i] * v[i] ;
  return sum ;
}
```

Un scénario possible est de spécialiser la fonction `dotproduct` lorsque la taille et les données du vecteur `u` sont connues. Le résultat de cette transformation pour les valeurs de spécialisation `size` = 3 et `u` = {2,5,7} est la fonction `dotproduct_spe` suivante :

```
int dotproduct_spe (int *v) {
  int sum = 0 ;
  sum = 2 * v[0] ;
  sum = sum + 5 * v[1] ;
  sum = sum + 7 * v[2] ;
  return sum ;
}
```

Tous les calculs qui dépendent de `u` et de `size` dans le programme original ont été évalués et ont disparu du programme résiduel. En effet, le test de la boucle `for` pouvant être évalué à chaque itération, la boucle a pu être déroulée. De plus, chaque `u[i]` a été remplacé par sa valeur. Le programme résiduel ainsi obtenu est moins générique que le programme original puisque les valeurs de certains paramètres ont été figées, mais il est plus efficace. Notons aussi que les exécutions de `dotproduct({2,5,7},{1,2,3},3)` et de `dotproduct_spe({1,2,3})` produisent le même résultat c'est-à-dire 33.

2.1.2 Concepts

L'évaluation partielle réalise une propagation de constantes agressive et inter-procédurale qui correspond entre autres à des pliages de constantes, dépliages de fonctions et déroulages de boucles. On distingue deux types d'évaluateurs partiels : les évaluateurs partiels *en ligne* et les évaluateurs partiels *hors ligne*. Il est de plus possible d'effectuer la spécialisation non seulement à la compilation mais aussi à l'exécution.

2.1.2.1 Approche en ligne ou hors ligne

Un évaluateur partiel en ligne peut être vu comme un interprète non standard qui évalue les calculs invariants directement à partir des valeurs de spécialisation et qui résidualise le reste du code [JGS93, Mey91, WCRS91].

Un évaluateur hors ligne divise quant à lui le processus de spécialisation en deux phases : une phase d'analyse et une phase de spécialisation. Contrairement à l'approche en ligne, la phase d'analyse ne travaille pas directement à partir des valeurs concrètes des paramètres d'entrée du programme mais à partir d'une *description abstraite* de ces

paramètres. Cette description abstraite précise le *temps de liaison* à associer à chaque paramètre d'entrée. Ainsi, un paramètre qui a une valeur connue au moment de la spécialisation est dit *statique* ; dans le cas contraire il est *dynamique*. Par extension, les calculs qui dépendent de paramètres statiques sont statiques et les autres sont dynamiques. La phase d'analyse correspond à une analyse de *temps de liaison* qui propage les temps de liaison des paramètres d'entrée à travers le programme afin de déterminer quelles sont les constructions statiques qui seront évaluables à la spécialisation et quelles sont les constructions dynamiques qui elles devront être reconstruites. À partir des informations collectées pendant la phase d'analyse et des valeurs de spécialisation (c'est-à-dire les valeurs concrètes des paramètres d'entrée statiques) la phase de spécialisation effectue la spécialisation du programme proprement dite.

L'approche en ligne a l'avantage de manipuler directement les valeurs de spécialisation et peut donc déterminer très précisément quelles transformations effectuer. Cependant cette approche est lente car elle ne permet pas la réutilisation des calculs. Ainsi, les spécialisations distinctes d'une fonction pour un même ensemble d'arguments connus mais pour des valeurs de spécialisation différentes, nécessitent une analyse distincte des traitements à effectuer même si la majorité des opérations effectuées sont pratiquement identiques.

L'approche hors ligne identifie quant à elle les constructions statiques du programme avant même que l'étape de spécialisation ne débute. La phase de spécialisation n'a donc pas besoin d'analyser le code et peut se contenter d'interpréter les informations fournies par la phase d'analyse, ce qui rend la spécialisation plus rapide. De plus le résultat de la phase d'analyse reste valide tant que la description abstraite des paramètres d'entrée du programme est inchangée. Il est ainsi possible de spécialiser un programme pour des valeurs de spécialisation différentes sans avoir à refaire les analyses. Néanmoins, l'analyse réalisée par l'approche hors ligne est moins précise que celle en ligne puisqu'elle manipule des valeurs abstraites au lieu de valeurs concrètes et doit donc effectuer des approximations.

2.1.2.2 Spécialisation à la compilation ou à l'exécution

L'évaluation partielle est souvent considérée comme une transformation de source à source. On parle alors de *spécialisation à la compilation*. Le programme spécialisé est produit à partir du programme original et des valeurs des invariants qui sont connues avant l'exécution du programme. Il est ainsi possible de spécialiser un programme et de le compiler avant de l'exécuter. Le temps passé à spécialiser le code, même s'il est important, n'introduit aucun coût supplémentaire à l'exécution.

Cependant il arrive que les valeurs des invariants ne soient connues que pendant l'exécution du programme. Les invariants sont alors appelés *quasi-invariants* et il faut alors envisager une *spécialisation à l'exécution*. Néanmoins, la spécialisation à l'exécution n'est intéressante que si le temps mis pour générer le code spécialisé peut être amorti par le temps gagné en utilisant le programme spécialisé au lieu du programme original.

2.2 Tempo, un spécialiseur pour les programmes C

Tempo est un spécialiseur hors ligne pour le langage C qui a été développé par le projet Compose [CHL+98, CHN+96b]. Tempo permet de spécialiser un programme à la compilation ou à l'exécution.

2.2.1 Architecture de Tempo

L'architecture de Tempo a été guidée par une volonté d'uniformiser les approches de spécialisation à la compilation et à l'exécution, c'est-à-dire que les deux approches utilisent les mêmes analyses comme le montre la Figure 2.1. De plus, le spécialiseur a été spécialement conçu dans l'optique de traiter des applications système de taille réelle. Cette particularité a eu des répercussions sur le choix et la précision des analyses mises en œuvre.

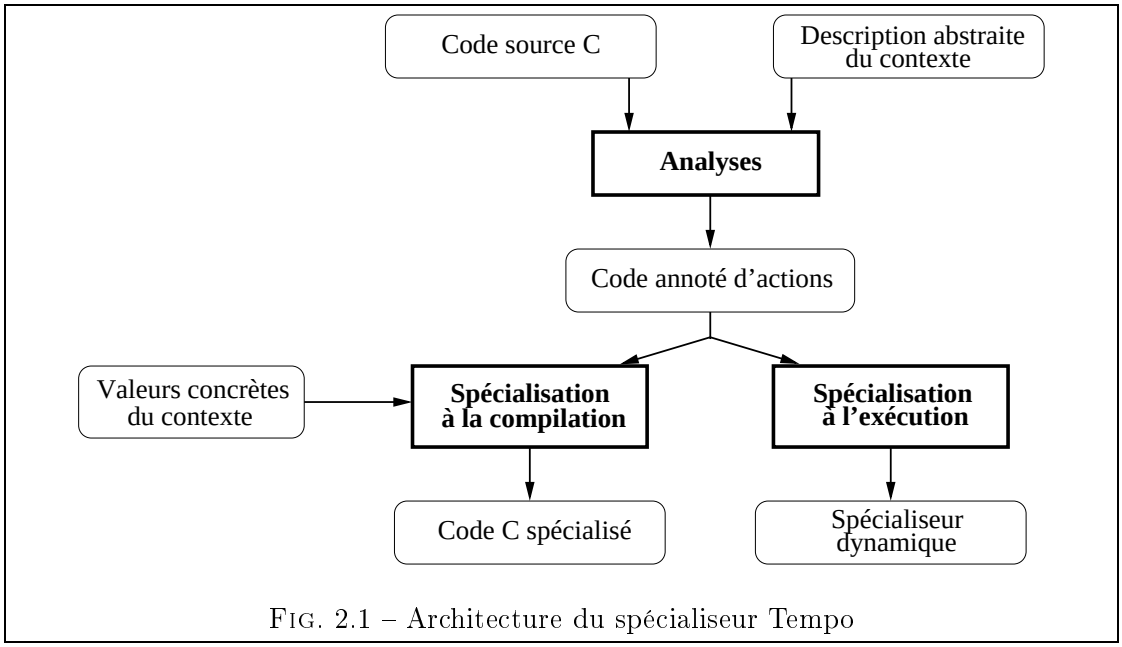


FIG. 2.1 – Architecture du spécialiseur Tempo

Pour spécialiser un programme, il faut fournir à Tempo le code source de ce programme ainsi qu'une description abstraite du contexte de spécialisation. Le programme est alors analysé à partir de la description abstraite du contexte. Une fois le programme analysé, il est possible d'opter pour une spécialisation à la compilation ou à l'exécution. Dans le premier cas, les valeurs de spécialisation doivent être précisées a priori et le résultat de la spécialisation est un programme spécialisé. Dans le cas de la spécialisation à l'exécution, un spécialiseur dynamique dédié est produit et pourra être appelé à l'exécution permettant ainsi la spécialisation du code dès que les valeurs des quasi-invariants seront disponibles. Nous décrivons maintenant les étapes d'analyse et de spécialisation à la compilation et à l'exécution.

2.2.1.1 Analyses

La phase d'analyse de Tempo est la première étape du processus de spécialisation. Elle prend en entrée le programme source et une description abstraite du contexte de spécialisation. Le programme est tout d'abord transformé en un arbre de syntaxe abstraite à partir duquel les analyses vont pouvoir s'effectuer. Tempo possède de nombreuses analyses qui sont dans l'ordre d'exécution : l'analyse d'alias, l'analyse de temps de liaison, l'analyse de temps d'évaluation et l'analyse d'actions (Figure 2.2).

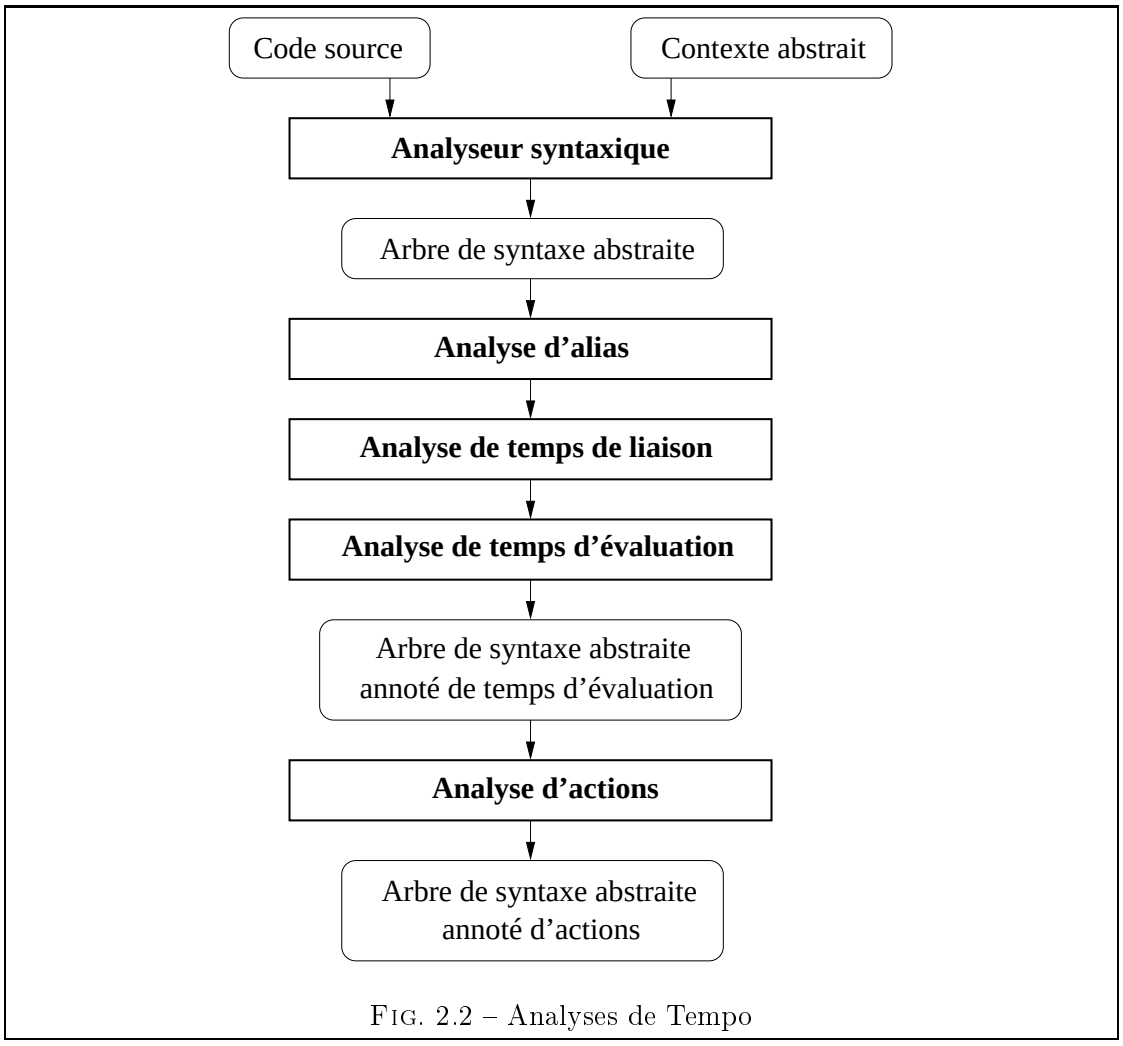


FIG. 2.2 – Analyses de Tempo

ANALYSE D'ALIAS

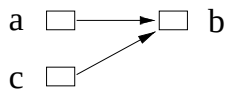
Le langage C possédant la notion de pointeur, il est nécessaire d'effectuer une analyse d'alias pour déterminer l'ensemble des emplacements mémoire associés à chaque expression de type pointeur.

Sans ces informations, l'analyse de temps de liaison serait trop approximative puis-

qu'elle ne prendrait pas en compte tous les emplacements mémoire concernés par un effet de bord pour calculer les temps de liaison des variables d'un programme.

Un alias est créé lorsque deux ou plusieurs noms font référence au même emplacement mémoire. Voyons concrètement la notion d'alias à travers un exemple :

```
int *a, b, *c ;
a = &b ;
c = a ;
```



La première affectation fait pointer la variable `a` sur l'emplacement mémoire correspondant à l'entier `b`. Après cette affectation, `*a` et `b` font donc référence au même emplacement mémoire. On dit alors que `*a` est un alias de `b`. La deuxième affectation fait que la variable `c` pointe sur le même emplacement mémoire que celui auquel fait référence `a` c'est-à-dire `b` ; `*c` devient donc un alias de `*a` et de `b`.

En C, les opérateurs `&` et `*` permettent de créer de nouveaux alias à n'importe quel point d'un programme. Par exemple, lors d'un appel de fonction avec passage des paramètres par référence, des alias sont créés en associant les arguments de la fonction avec les paramètres formels. Il faut aussi prendre les pointeurs sur fonctions, les références de tableau *etc.* L'analyse d'alias implémentée pour Tempo est basée sur le modèle "pointe-sur" ("points-to") [EGH94]. Ainsi, par exemple, la relation d'alias créée par l'affectation `a = &b` est représentée par $a \mapsto \{b\}$. Dans le cadre d'une analyse "pointe-sur", on détermine pour chaque expression de type pointeur, une approximation sûre de l'ensemble des alias correspondants, c'est-à-dire de l'ensemble des emplacements mémoire que l'expression peut représenter.

De manière générale, plusieurs propriétés caractérisent le comportement et la précision d'une analyse de programmes. Ainsi, la sensibilité au flot de contrôle permet à des variables à différents points d'un programme d'avoir différentes descriptions d'analyse. De plus, si une analyse est effectuée inter-procéduralement, c'est-à-dire si elle propage les informations non seulement dans la fonction analysée mais aussi à travers les fonctions appelées, il faut préciser si elle est sensible au contexte d'appel. Nous illustrons maintenant ces propriétés dans le cadre d'une analyse d'alias.

Sensibilité au flot de contrôle Une analyse qui prend en considération le flot de contrôle est dite sensible au flot. Cette propriété permet à un pointeur d'avoir un ensemble d'alias différent aux divers points de programme où il est impliqué. Dans le cas contraire, on parle d'analyse insensible au flot. Considérons l'exemple suivant :

	sensible au flot	*insensible au flot*
(1) `int a, b, *p ;`		
(2) `p = &a ;`		
(3) `foo1(p) ;`	`p` $\mapsto$ `{a}`	`p` $\mapsto$ `{a,b}`
(4) `p = &b ;`		
(5) `foo2(p) ;`	`p` $\mapsto$ `{b}`	`p` $\mapsto$ `{a,b}`

Pour cet exemple, une analyse sensible au flot assume que `p` pointe sur `a` après l'affectation au point de programme (2) et qu'il pointe sur `b` après celle du point de programme (4). Une analyse insensible au flot retient que `p` peut pointer sur `a` ou sur `b` à n'importe quel point du programme. Ainsi, à l'appel de `foo1`, une analyse insensible au flot assume que `p` pointe sur `a` ou `b` alors qu'en fait il pointe sur `a`. Une analyse insensible au flot est donc moins précise qu'une analyse sensible au flot.

Sensibilité au contexte d'appel La sensibilité au contexte d'appel de fonction est une propriété à considérer dans le cadre d'une analyse inter-procédurale. En effet, une analyse inter-procédurale propage les informations dans les fonctions appelées alors qu'une analyse intra-procédurale ne se soucie du flot de données que dans le corps de la fonction analysée.

Une analyse insensible au contexte associe un unique ensemble d'alias à chaque fonction. Une analyse sensible au contexte associe conceptuellement, quant à elle, autant d'ensembles d'alias qu'il y a de contextes d'appel différents pour une fonction. Dans la pratique, le nombre d'appel est limité pour des raisons d'efficacité. Considérons l'exemple suivant :

```
int a, b;                          int * incr(int * x) {
... incr(&a);                        *x = *x + 1;
... incr(&b);                        return x;
...                                }
```

Une analyse sensible au contexte crée deux variants pour la fonction `incr` : un variant pour le cas où `*x` correspond à `a` et un autre où `*x` correspond à `b`. Une analyse insensible au contexte analyse la fonction `incr` une seule fois avec un contexte d'appel où `x` pointe sur `a` ou `b`.

L'analyse d'alias de Tempo est une analyse inter-procédurale sensible au flot de contrôle mais insensible au contexte d'appel de fonction. D'une manière générale, le choix d'une analyse d'alias insensible au contexte peut paraître paradoxal du fait de l'imprécision des informations collectées. Cependant, l'utilisation d'une analyse insensible au contexte par Tempo est justifiée par le constat que, dans le cadre de la spécialisation de programmes systèmes, les calculs statiques dépendent de variables qui typiquement suivent un schéma régulier. Il est donc possible de se contenter dans ce cas d'informations imprécises. Il s'avère de plus que, dans le contexte d'une utilisation plus générale, les bénéfices d'une analyse d'alias sensible au contexte restent encore à être prouvés [Ruf95].

Après avoir défini les caractéristiques d'une analyse de programmes, nous pouvons maintenant considérer les choix d'implémentation concernant la granularité des objets manipulés.

Approximations La granularité a un impact sur la précision de l'analyse. Nous avons jusqu'à maintenant utilisé le terme *emplacement mémoire*. Un emplacement mémoire

est une représentation abstraite de l'ensemble des cellules mémoires utilisées pour stocker un objet à l'exécution. Ainsi pour représenter un pointeur d'entier, l'analyse associe un emplacement mémoire pour le pointeur et un autre pour l'entier. Cependant il est possible d'avoir besoin à l'exécution d'un nombre potentiellement très grand d'emplacements, notamment dans le cas de tableaux. Pour des raisons d'efficacité, une analyse ne peut pas prendre en considération un nombre trop important d'emplacements mémoire ; il faut donc faire un choix quant à la représentation des objets à utiliser. Dans le cadre de Tempo, un emplacement mémoire est soit :

- une variable scalaire, c'est-à-dire qu'un emplacement mémoire est utilisé pour représenter une variable de type entier, flottant, pointeur, *etc.*
- un tableau, c'est-à-dire qu'un seul emplacement représente tous les éléments du tableau.
- un champ de structure de données, c'est-à-dire qu'un seul emplacement est utilisé pour représenté un champ donné de toutes les instances d'une structure de données.

Nous illustrons l'impact de ce choix de conception avec deux exemples : un exemple avec un tableau d'entiers et un autre qui utilise des structures de données.

Tableau d'entiers Considérons l'exemple suivant :

```
int f(int a, int b) {
  int *tab[2] ;                    /* alias : tab ↦ {}    */
  tab[0] = &a ;                    /* alias : tab ↦ {a,b} */
  tab[1] = &b ;                    /* alias : tab ↦ {a,b} */
  return *tab[0] ;                 /* alias : tab ↦ {a,b} */
}
```

Les affectations ont pour effet de faire de `a` et `b` des alias de `tab`. Ainsi, pour l'analyse d'alias, `*tab[0]` peut correspondre soit à `a` soit à `b`, alors qu'il est clair que `*tab[0]` correspond en fait à `a`. Ce résultat s'explique par l'abscence d'analyse d'intervales. Il est en effet difficile de prendre en compte les indices d'un tableau car leurs valeurs ne sont souvent pas connues a priori. Une approximation est donc faite en n'associant qu'un seul emplacement mémoire à l'ensemble du tableau.

Structures de données Étudions maintenant le cas des structures de données. L'analyse d'alias utilisée par Tempo est monovariante par rapport aux structures de données, c'est-à-dire que lorsqu'un pointeur pointe sur le champ d'une structure, il a accès à ce champ pour toutes les instances de cette structure. L'exemple qui suit illustre cette situation.

```
struct vector {int size; int * data;};
struct vector v1, v2;
int *p;
p = &v1.size;
printf("la taille du vecteur v1 est : %i", *p);
```

Ainsi l'ensemble d'alias associé à `p` est {`vector.size`}, ce qui signifie que pour n'importe quelle instance de `vector` le champ `size` est un alias possible et donc ici, `p` $\mapsto$ {`v1.size`,`v2.size`} même si dans le programme seul `v1.size` est un alias de `p`.

ANALYSE DE TEMPS DE LIAISON

Nous décrivons maintenant l'analyse de temps de liaison de Tempo. Cette analyse est à la base de l'approche hors ligne. Elle détermine à partir de la description abstraite du contexte d'appel du programme et des informations d'alias fournies par l'analyse d'alias, le temps de liaison de chaque construction, c'est-à-dire des variables, des expressions, des instructions, *etc.*

Pour donner une idée plus précise du résultat produit par l'analyse de temps liaison de Tempo, nous reprenons l'exemple de `dotproduct` et montrons ci-dessous, les temps de liaison déterminés pour cette fonction. Pour différencier les constructions statiques et dynamiques, nous soulignons les constructions statiques.

```
int dotproduct (int *u, int *v, int size) {
  int i, sum;
  sum = 0;
  for(i = 0; i < size; i++)
    sum = sum + u[i] * v[i];
  return sum;
}
```

La fonction `dotproduct` est analysée à partir des informations de temps de liaison associées à ses arguments qui correspondent au contexte de spécialisation suivant : `u` est statique et pointe sur des données statiques, `v` est complètement dynamique et `size` est statique. La première affectation rend `sum` statique puisqu'une valeur constante lui est affectée. Les paramètres de l'expression de contrôle de la boucle `for` sont tous connus : `i` est initialisée avec une valeur constante et est donc statique, le test de contrôle dépend de `i` et de `size` qui sont statiques, enfin l'expression exécutée après chaque itération est une fonction qui incrémente `i` de `1` et donc `i` reste statique à chaque itération. Dans l'affectation du corps de la boucle, `u[i]` est statique, ainsi que l'indice `i` de `v[i]`. Les termes `sum` sont dynamiques[1] puisque `sum` est calculé à partir des valeurs de `v` qui sont

[1] Dans Tempo, l'expression à gauche de l'affectation est dénotée comme statique car pour Tempo `sum` correspond alors à l'emplacement mémoire de la variable `sum` et l'analyse suppose que l'emplacement mémoire est toujours connu dans cette situation. Cependant, pour simplifier, nous considérons ici la valeur de la variable et non son adresse.

inconnues. À la fin de la fonction, l'expression de retour est dynamique car la valeur de `sum` est inconnue.

Notons que l'analyse doit parfois effectuer des approximations car elle ne dispose pas des valeurs concrètes. Par exemple, lors du traitement d'une conditionnelle possédant un test statique, les deux branches sont analysées. Puis, comme l'analyse ne peut pas déterminer quelle branche sera choisie, elle combine les résultats d'analyse des deux branches ce qui introduit une imprécision.

Nous présentons maintenant les caractéristiques de l'analyse de temps de liaison de Tempo. Cette analyse est sensible au flot de contrôle et au contexte d'appel de fonction. De plus une propriété de sensibilité aux valeurs de retour a été mise en œuvre pour permettre une meilleure exploitation des valeurs de retour statiques. Nous illustrons avec des exemples ces propriétés.

Sensibilité au flot de contrôle Une analyse de temps de liaison sensible au flot de contrôle permet aux variables lues et écrites dans un programme d'avoir des temps de liaison qui varient d'un point de programme à un autre. Typiquement, le temps de liaison d'une variable est modifié lors d'une affectation : il devient statique si on affecte une valeur connue à la variable, et il devient dynamique dans le cas contraire. Une analyse insensible au flot n'a qu'un état de temps de liaison global pour tout le programme. Ainsi une variable est considérée dynamique pour tout le programme s'il existe un point du programme où on lui affecte une valeur dynamique. Considérons l'exemple suivant :

```
void f(int a, int b, int *p) {
  p = &a;                                  /* alias : p ↦ {a} */
  a = a + *p;
  a = b;                                   /* alias : p ↦ {a} */
  a = a + *p;                              /* alias : p ↦ {a} */
}
```

Dans le programme ci-dessus, la variable `a` est lue et écrite plusieurs fois. Pour une analyse sensible au flot, la variable `a` est statique jusqu'au moment où on lui affecte la valeur dynamique de `b`. La variable `a` reste dynamique dans la suite du programme. Une analyse insensible considérerait quant à elle `a` comme dynamique dans tout le programme, même dans les deux premières affectations. Une analyse sensible au flot a été choisie pour Tempo car la précision résultante est essentielle pour traiter avec succès des programmes réalistes où il est fréquent qu'une variable ait différents temps de liaison.

Sensibilité au contexte d'appel Une analyse sensible au contexte d'appel permet qu'une fonction soit analysée en fonction de ses différents contextes d'appel. On parle alors de *polyvariance*. Considérons l'exemple suivant :

```
int c ;
void f(int b) {
  g(b) ;
  g(b) ;
}
```

```
void g(int x) {
  c = c + x ;
}
```

Supposons que le contexte d'appel de la fonction `f` soit le suivant : le paramètre `b` est dynamique et la variable globale `c` est statique. Dans ce cas, une analyse sensible au contexte d'appel crée deux variants `g_1` et `g_2` pour la fonction `g` :

```
void g_1(int x) {
  c = c + x ;
}
```

```
void g_2(int x) {
  c = c + x ;
}
```

Dans le variant `g_1`, `x` est dynamique et `c` est statique. Dans l'autre variant, `x` et `c` sont tous les deux dynamiques. Du fait de la polyvariance, certaines descriptions de fonction peuvent se retrouver dupliquées de nombreuses fois (autant de fois qu'il existe de contextes de temps de liaison différents) mais cette approche à l'avantage d'exploiter au maximum les valeurs statiques dans les calculs. Une analyse insensible au contexte d'appel aurait pour notre exemple produit un seul variant correspondant à `g_2`, et aucun bénéfice aurait été tiré du fait que la valeur de `c` était connue lors du premier appel à `g`.

Sensibilité aux valeurs de retour Une analyse sensible aux valeurs de retour permet l'exploitation du résultat statique d'une fonction bien que certains de ses effets de bords soient résidualisés. Ainsi une analyse sensible aux valeurs de retour calcule un temps de liaison pour la valeur de retour d'une fonction et un temps de liaison pour les effets de bord de cette fonction. Une analyse insensible aux valeurs de retour ne calcule quant à elle qu'un seul temps de liaison pour les deux. Ainsi la valeur de retour d'une fonction ne peut pas être statique si toute la fonction ne peut pas être totalement évaluée.

Nous illustrons l'insensibilité aux valeurs de retour avec l'exemple suivant :

Insensible au retour

```
int c, d ;
void f(int a) {
  c = (g(a) * c) + d ;
}

int g(int x) {
  d = d + c ;
  return x + 1 ;
}
```

Spécialisation pour a=2 et c=5

```
int c, d ;
void f_spe() {
  c = (g_spe() * 5) + d ;
}

int g_spe() {
  d = d + 5 ;
  return 3 ;
}
```

Une analyse insensible au retour ne calcule qu'un seul temps de liaison. Par conséquent, les parties dynamiques de `g` qui sont résidualiées interdisent que la valeur de retour

statique soit évaluée. L'instruction `return` est résidualisée et l'opération de multiplication `g(a) * c` n'est pas évaluée. Voyons maintenant le même exemple avec une analyse sensible au retour :

Sensible au retour

```
int c, d;
void f(int a) {
  c = (g(a) * c) + d;
}


int g(int x) {
  d = d + c;
  return x + 1;
}
```

Spécialisation pour a=2 et c=5

```
int c, d;
void f_spe () {
  g_spe();
  c = 15 + d;
}

int g_spe() {
  d = d + 5;
}
```

Pour notre exemple, une analyse sensible au retour détermine que le temps de liaison de la valeur de retour de la fonction `g` est statique. Cela permet à l'expression `g(a) * c` d'être complètement statique et donc d'être évaluée à la spécialisation.

L'utilisation d'une analyse sensible aux valeurs de retour dans Tempo permet d'exploiter d'avantage les informations statiques en empêchant le temps de liaison d'une fonction d'interférer par effet de bord sur le temps de liaison de son résultat. Cette propriété s'est avérée très utile pour spécialiser des applications systèmes. En effet, les fonctions systèmes retournent souvent une indication de réussite ou d'échec, ou la taille des données modifiées et ces informations peuvent souvent être calculées sans que tous les calculs de la fonction ne soient effectués.

Maintenant que nous avons vu en détail les propriétés de l'analyse de temps de liaison de Tempo, nous présentons les approximations qui sont effectuées par cette analyse du fait des choix d'implémentation liés à la granularité des objets manipulés.

Approximations Concrètement, l'analyse de temps de liaison de Tempo considère trois types d'emplacement mémoire :

- emplacements de variables : un temps de liaison est associé à chaque variable qui n'est pas de type structure. Les informations de temps de liaison sont plus ou moins complexes en fonction du type de la variable. Ainsi, pour une variable qui a un type "pointeur sur entier", l'analyse détermine le temps de liaison du pointeur et de l'entier pointé. Par exemple, on peut avoir un pointeur statique sur un entier dynamique. Pour les tableaux, un temps de liaison est donné à l'adresse du tableau et un autre temps de liaison (un seul) est associé à l'ensemble des éléments.
- emplacements de retour : des temps de liaison sont associés au type de retour de chaque fonction. Cela permet de mettre en œuvre la propriété de sensibilité aux valeurs de retour.

– emplacements de champs de structures : des temps de liaison sont associés à chaque champ d'une structure de données, au lieu d'un temps de liaison global pour toute la structure. Cela permet d'avoir des structures de données partiellement statiques, et donc, de traiter les structures de données de manière précise. Par défaut, l'analyse de temps de liaison de Tempo est monovariante pour les structures, c'est-à-dire que le temps de liaison d'un champ de structure est le même pour toutes les instances de cette structure. Cependant, une analyse polyvariante a aussi été implémentée et peut être utilisée.

ANALYSE DE TEMPS D'ÉVALUATION

L'analyse de temps de liaison de Tempo ne suffit pas pour produire des annotations de temps de liaison correctes. Une analyse de temps d'évaluation est nécessaire. Cette analyse implémente la propriété de sensibilité au contexte d'utilisation que nous présentons ci-dessous.

Il arrive qu'une variable soit utilisée à la fois dans des calculs statiques et dynamiques ce qui peut avoir un impact sur son temps de liaison. En effet, une variable statique utilisée dans un contexte dynamique est évaluée à la spécialisation et est remplacée par sa représentation textuelle. Il n'existe cependant pas toujours une représentation textuelle pour cette variable notamment lorsqu'il s'agit de pointeurs, de tableaux ou de structures. Dans ces cas, il est nécessaire de rendre dynamique l'utilisation de cette variable.

Considérons l'exemple suivant pour illustrer cette propriété :

```
int a[20], *p, s, d;

a[11] = 1234;
p = a;
...
... = *((p + s) + 10);
... = *((p + d) + 20);
```

Si ce programme est analysé pour `a` et `s` statiques et `d` dynamique, alors la valeur du pointeur `p` dépend de la valeur statique de `a` et est utilisé à la fois dans un contexte dynamique et un contexte statique. Comme la valeur du pointeur ne peut pas être réifiée pendant la spécialisation, l'utilisation de `p` dans le contexte dynamique doit être considérée dynamique.

Une analyse insensible au contexte d'utilisation oblige toutes les utilisations d'une variable à avoir le même temps de liaison, si bien que pour notre exemple, toutes les utilisations du pointeur `p` sont rendues dynamiques :

Code annoté de temps d'évaluation

```
a[11] = 1234 ;
p = a ;
...
... = *((p + s) + 10) ;
... = *((p + d) + 20) ;
```

Spécialisation pour s = 1

```
a[11] = 1234 ;
p = a ; ...
...= *((p + 1) + 10) ;
...= *((p + d) + 20) ;
```

Ainsi la résidualisation de l'utilisation de `p` dans un contexte dynamique empêche l'utilisation de `p` dans le contexte statique d'être évaluée et par la même occasion empêche les expressions d'addition statiques de l'être aussi.

La sensibilité au contexte d'utilisation évite ce problème. Une analyse sensible au contexte d'utilisation permet aux utilisations d'une variable d'avoir des temps de liaison différents. Ainsi, si l'utilisation d'une variable devient dynamique du fait d'un contexte dynamique, cela n'a pas d'impact sur les autres utilisations de cette variable. Dans cette situation, l'analyse associe à l'initialisation de cette variable un temps de liaison dit *statique et dynamique*, ce qui permet d'exploiter sa valeur dans les expressions statiques tout en forçant sa résidualisation dans les autres calculs. Les termes statiques et dynamiques sont soulignés deux fois.

Code annoté de temps d'évaluation

```
a[11] = 1234 ;
p = a ;
...
... = *((p + s) + 10) ;
... = *((p + d) + 20) ;
```

Spécialisation pour s = 1

```
a[11] = 1234 ;
p = a ;
...
...= 1234 ;
...= *((p + d) + 20) ;
```

Cette fois-ci, l'utilisation de `p` dans le contexte statique est évaluée pendant la spécialisation, ce qui permet aux additions d'être elles aussi évaluées.

Notons qu'une analyse sensible au flot est moins précise qu'une analyse sensible au contexte d'utilisation. En effet, une analyse sensible au flot associe à une variable un temps de liaison à chaque fois que la variable est affectée, tandis qu'une analyse sensible au contexte associe un temps de liaison à cette variable à chaque fois qu'elle est utilisée.

ANALYSE D'ACTIONS

Dans Tempo, contrairement à la plupart des évaluateurs partiels, la spécialisation ne se fait pas directement à partir du programme annoté de temps d'évaluation. En effet, la phase d'analyse de Tempo se termine par une analyse d'actions qui a pour but de déterminer les transformations qui devront être effectuées pendant la phase de spécialisation.

L'analyse d'actions prend en entrée un arbre annoté de temps d'évaluation et produit un arbre annoté d'actions. Une action décrit *comment* une construction doit être trans-

formée pendant la phase de spécialisation. On distincte quatre transformations : *evaluate*, *reduce*, *rebuild* et *identity* que l'on note respectivement *ev*, *red*, *reb* et *id*. Ainsi une construction de programme qui ne dépend que de parties statiques peut être complètement évaluée à la spécialisation et est donc annotée *ev*. Une construction est annotée *red* si elle ne peut être que partiellement réduite à la spécialisation car elle contient des parties dynamiques. Typiquement, une conditionnelle qui a un test statique et des branches qui ne peuvent pas être totalement évaluées, doit être partiellement réduite. L'action *reb* est associée aux constructions qui doivent être reconstruites mais qui contiennent cependant des calculs statiques. Enfin, toutes les constructions qui sont strictement dynamiques sont annotées *id*.

Reprenons l'exemple `dotproduct` du début du chapitre. Après l'analyse de temps d'évaluation, la fonction est annotée comme suit :

```
int dotproduct (int * u, int * v, int size) {
  int i ;
  int sum ;
  sum = 0 ;
  for(i = 0 ; i < size ; i++)
    sum = sum + u[i] * v[i] ;
  return sum ;
}
```

Ainsi, les paramètres de contrôle de la boucle `for` sont tous statiques. De plus, à chaque itération de la boucle, les données du vecteur `u` ainsi que l'indice d'itération `i` sont connues. À partir de ces informations, l'analyse d'actions détermine les transformations à effectuer à la spécialisation.

```
int dotproduct ( ⟨int * u⟩^ev, ⟨int * v⟩^id, ⟨int size⟩^ev ) {
  ⟨int i⟩^ev;
  ⟨int sum⟩^id;
  ⟨sum = 0⟩^id;
  for^red ( ⟨i = 0⟩^ev; ⟨i < size⟩^ev; ⟨i++⟩^ev )
    sum^id =^reb sum^id +^reb ⟨u[i]⟩^ev *^reb v^id[ i^ev ] ;
  ⟨return sum⟩^id;
}
```

La boucle `for` peut être réduite, c'est-à-dire déroulée. Les paramètres et expressions de contrôle de la boucle peuvent être évalués, ainsi que `u[i]` et `i`. L'affectation du corps de la boucle doit être reconstruit puisqu'il est formé d'éléments statiques et dynamiques. Le reste du programme est recopié.

2.2.1.2 Spécialisation

Une fois que le programme annoté d'actions a été produit, Tempo offre la possibilité de choisir entre effectuer une spécialisation à la compilation ou une spécialisation à l'exécution.

Spécialisation à la compilation Le processus de spécialisation à la compilation se divide en deux étapes. La première étape génère un spécialiseur dédié, parfois appelé *extension génératrice* [JGS93, GJ95]. Le spécialiseur dédié statique est obtenu en compilant les informations fournies par l'arbre d'actions. Les constructions annotées *ev* donnent lieu à la création de fonctions qui permettront d'exécuter les fragments de code correspondants lors de la spécialisation, tandis que le traitement des constructions annotées *id*, *reb* et *red* produit des fonctions qui généreront du code. La deuxième étape de la spécialisation à la compilation consiste à appeler le spécialiseur dédié avec les valeurs de spécialisation pour obtenir le programme résiduel. Le spécialiseur dédié peut être réutilisé plusieurs fois pour des valeurs de spécialisation différentes.

Spécialisation à l'exécution Le processus de spécialisation à l'exécution se divise en deux étapes ; l'une se déroule à la compilation et l'autre à l'exécution. La première étape a pour but de produire des fragments de code paramétrés par des trous dans lesquels les valeurs pourront être insérées à la spécialisation. Ces fragments de code sont compilés à l'avance. Un spécialiseur dédié dynamique est aussi créé. La deuxième étape se passe à l'exécution. Lorsque le spécialiseur dédié est invoqué avec les valeurs de spécialisation, il sélectionne et copie les fragments de code compilés, insère les valeurs de spécialisation dans les trous et met à jour les sauts entre les fragments de code de façon à obtenir la spécialisation souhaitée [CN96].

Toutes les opérations mises en œuvre sont simples et rapides. Elles permettent donc au processus de spécialisation à l'exécution d'être efficace et d'être amorti au bout de quelques utilisations du code spécialisé [NHCL98].

2.2.2 Interface utilisateur de Tempo

Nous décrivons maintenant ce que doit faire un utilisateur pour spécialiser un programme avec Tempo. Tout d'abord, il faut décrire le contexte de spécialisation souhaité et choisir le comportement de spécialisation le plus approprié. Ces paramètres sont précisés dans des fichiers de configuration que Tempo lit avant d'effectuer les analyses. Ensuite il faut lancer les différentes analyses. Enfin, il est possible de choisir entre la génération d'un spécialiseur dédié pour la spécialisation à la compilation ou à l'exécution. Dans le cas d'une spécialisation à la compilation, il faut spécifier les valeurs de spécialisation dans un fichier. Nous illustrons ce processus en décrivant la spécialisation de la fonction `dotproduct` (voir Figure 2.3).

Spécification du contexte de spécialisation Comme nous l'avons mentionné, Tempo a été développé dans l'optique de spécialiser des applications système. Ces pro-

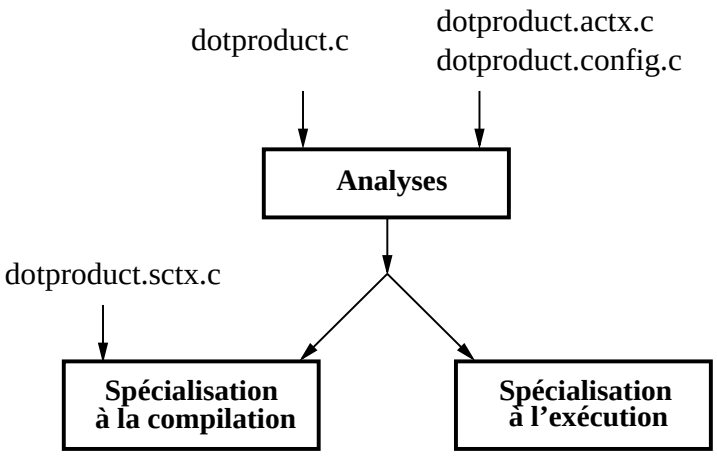


FIG. 2.3 – Processus de spécialisation de `dotproduct` avec Tempo

grammes sont souvent très grands et complexes, il n'est donc pas envisageable de les soumettre intégralement à la spécialisation. Pour contourner ce problème, Tempo offre la possibilité de spécialiser un fragment de code et permet ainsi une spécialisation modulaire [CHN+96b].

Cette spécialisation modulaire requiert de la part de l'utilisateur une description précise du contexte dans lequel le fragment de code se trouve dans le programme global. La précision des informations données a un impact prépondérant sur le degré de spécialisation qui pourra être obtenu car plus les informations sont précises, plus les analyses le seront aussi.

Il s'agit donc de spécifier comment le fragment de code et le reste du programme interagissent. Au minimum cela consiste à décrire les alias potentiels et les temps de liaison à associer à chaque paramètre d'entrée. Dans les cas plus complexes, il faut aussi décrire le comportement des fonctions appelées dans le fragment à spécialiser mais dont le code source n'est pas accessible.

Les informations de contexte doivent être décrites dans le fichier `config.sml` qui est lu par le spécialiseur au début des analyses. À travers ce fichier, il est possible d'initialiser les différents paramètres de configuration de Tempo afin de spécifier entre autre, le nom de la fonction à spécialiser, les temps de liaison des différents paramètres de la fonction et le type d'analyse souhaité (monovariant ou polyvariant par exemple). En fait Tempo possède plus de 50 paramètres de configuration grâce auxquels les utilisateurs peuvent spécifier le processus de spécialisation désiré.

Le contexte d'utilisation de la fonction à spécialiser est parfois compliqué et il faut alors le décrire précisément dans le fichier `actx.c` prévu à cet effet. Pour cela, il faut écrire un programme C qui recrée le contexte d'appel de la fonction. Il est, de la même façon, possible de décrire le comportement d'une fonction dont la définition n'est pas accessible par le processus de spécialisation mais qui effectue des effets de bord.

Reprenons l'exemple de la fonction `dotproduct`. Pour la spécialiser, il faut au mini-

mum écrire les informations suivantes dans le fichier `dotproduct.config.sml` :

```
entry_point := "dotproduct(S,D,S)";
```

Cette déclaration indique que `dotproduct` est la fonction à spécialiser. Elle doit être spécialisée pour un contexte où son premier et troisième arguments sont statiques, c'est-à-dire `u` et `size`. Mais cela est insuffisant puisque l'on veut préciser que les données pointées par `u` sont aussi connues. Pour cela il faut recréer le contexte d'appel de la fonction `dotproduct` dans le fichier `dotproduct.actx.c` :

```
int dummy_array[1];
void set_analysis_context(int ** u, int ** v, int * size) {
  *u = dummy_array;
}
```

Ce code définit un tableau d'entiers `dummy_array` et fait correspondre `u` à ce tableau. Ensuite il faut rajouter `static_locations := ["dummy_array"]` dans le fichier `dotproduct.config.sml` afin de spécifier que le tableau correspond à un emplacement mémoire statique.

Visualisation du résultat des analyses Une fois le contexte décrit, les analyses peuvent être lancées. Tempo offre la possibilité de visualiser le résultat des différentes analyses grâce à des fichiers colorés. Il est ainsi possible de visualiser le résultat de l'analyse de temps de liaison qui montre les temps de liaison calculés et les alias des pointeurs d'indirection, et le résultat de l'analyse d'action qui correspond à un programme C annoté de transformations similaire à celui montré précédemment.

Spécialisation à la compilation L'étape suivante est celle de la spécialisation. Dans le cas d'une spécialisation à la compilation il faut préciser les valeurs de spécialisation dans un fichier `dotproduct.sctx.c` :

```
int dummy_array[3] = 2,5,7;
void set_specialization_context(int ** u, int * size) {
  *u = dummy_array;
  *size = 3;
}
```

Le résultat de la spécialisation est la fonction `dotproduct_spe` montrée au début de ce chapitre.

2.3 Applications

La spécialisation de programmes a été appliquée à une large variété d'applications réelles. Ces applications présentent toutes une certaine forme de généricité de par leurs

architectures logicielles : systèmes en couches, bibliothèques génériques, *etc.* Nous en décrivons maintenant une liste non exhaustive en les organisant par domaines d'applications.

2.3.1 Systèmes d'exploitation

Les applications systèmes sont des programmes souvent écrits en C, complexes et généralement de grande taille. Ils offrent des opportunités de spécialisation à la compilation et à l'exécution. Ces opportunités se retrouvent dans des programmes qui ont été construits de façon générique et/ou modulaire dans le but d'augmenter la généralité et la flexibilité du système.

Le protocole Remote Procedure Call (RPC) [Sun90] en est un parfait exemple. Ce protocol permet d'effectuer un appel de procédure sur une machine distante de façon transparente, c'est-à-dire comme si l'appel était réalisé localement. Cela requiert d'encoder des données dont la représentation est dépendante d´une machine en une représentation indépendante d'un réseau et d'effectuer l'opération inverse, c'est-à-dire de décodage à la réception des données. Ces opérations d'encodage/décodage sont implémentées sous formes de bibliothèques génériques structurées en couche (la librairie XDR). Muller *et al.* ont spécialisé les opérations d'encodage et de décodage et ont obtenu des gains de performance proche d'un facteur 4 [MVM97, MMV+98]. Kono a utilisé la spécialisation à l'exécution sur les méthodes d'invocation distantes (*remote method invocation*) [Sof97] qui correspondent à la version objet des RPC. Les tests ont montré que le code spécialisé était 1,9 à 3 fois plus rapide que la bibliothèque Sun XDR et que le temps mis pour générer le code spécialisé à l'exécution n'était que de 0,6 msec [KM00].

McNamee *et al.* ont spécialisé les signaux Unix [MWP+01]. Les signaux Unix mettent en œuvre les communications inter-processus. Lorsqu'un processus émet répétitivement le même signal à un même processus destination, une partie des données internes des deux processus restent inchangées. Il est alors possible d'utiliser ces informations statiques pour spécialiser le code relatif à l'envoi des signaux. De manière générale, l'utilisation du code spécialisé à permis de réduire de 65% la latence d'émission d'un signal.

2.3.2 Interprètes

La spécialisation d'un interprète pour un programme donné est souvent bénéfique puisqu'elle permet d'éliminer la couche d'interprétation [JSS85].

Thibault *et al.* ont spécialisé à la compilation et à l'exécution plusieurs interprètes dont, entre autre, des interprètes de bytecode Java et l'interprète BPF. Les gains de performance obtenus sont non négligeables pour l'interprète de bytecode (32 fois plus rapide pour la spécialisation à la compilation et 21 fois pour la spécialisation à l'exécution). L'augmentation des performances de l'interprète BPF est moins importante (de l'ordre de 2) mais toujours intéressante [TCM+00].

2.3.3 Calcul scientifique

Les algorithmes scientifiques sont des programmes dont les performances d'exécution sont souvent critiques. Lawall a spécialisé avec Tempo la transformation de Fourier (FFT) pour un nombre de points en entrée donné. Cela a permis de dérouler les boucles, de propager des calculs, d'éliminer les appels aux fonctions de bibliothèque sinus et cosinus et ainsi d'obtenir des programmes résiduels allant jusqu'à 9 fois plus vite que le code original [Law98]. Baier *et al.* ont spécialisé avec un spécialiseur Fortran différents algorithmes : la FFT, l'interprétation par spline cubique et le problème de l'attraction à N-corps. Les performances des programmes résiduels sont respectivement jusqu'à 4, 1,4 et 6 fois plus grandes que celles des programmes originaux [BGZ94].

2.3.4 Graphisme

Andersen s'est intéréssé aux applications de traçage de rayons (*ray tracing*) qui doivent re-calculer en permanence les informations qui décrivent comment à partir d'un point les rayons de lumière peuvent traverser une scène. La spécialisation de l'application pour une scène donnée a permis d'augmenter d'un facteur 3 les performances du code [And96].

2.4 Bilan

Nous avons vu que la spécialisation de programmes est une technique de transformation de programmes automatique efficace qui permet d'obtenir des gains de performance non négligeables pour de nombreuses et diverses applications.

Ces gains de performance sont obtenus grâce à l'utilisation d'analyses puissantes qui permettent de réaliser une propagation de constantes agressive et inter-procédurale sous la forme de pliages de constantes, dépliages de fonctions et déroulages de boucles. Ces optimisations vont au-delà de celles offertes par les compilateurs optimisant. En effet, les compilateurs suivent des stratégies plus conservatrices et sont limités par la précision de leurs analyses. Ainsi les optimisations d'un compilateur sont souvent effectuées intra-procéduralement et consistent en une combinaison de propagations de constante relativement superficielles et d'optimisations bas niveau.

Enfin, les spécialiseurs de programmes offrent aux utilisateurs une interface fine à travers laquelle le processus de spécialisation peut être contrôlé précisément, permettant ainsi de déclencher des optimisations très pointues.

Chapitre 3

Accessibilité de la spécialisation de programmes

L'idée de pouvoir spécialiser automatiquement un programme générique et ainsi de réconcilier généricité et performance, est très attrayante. Cependant, il semble que les spécialiseurs de programmes n'ont pas encore trouvé leur place dans la batterie d'outils de programmation utilisée par les programmeurs. Dans la pratique, la plupart des utilisateurs de spécialiseurs s'avèrent être des experts du domaine. Ce constat trahit le fait que cette technique reste difficile d'accès pour des programmeurs non avertis. En effet, il ne suffit pas d'avoir identifié la généricité présente dans un programme pour obtenir son élimination par spécialisation.

Dans ce chapitre, nous présentons tout d'abord les différents comportements inappropriés que la spécialisation peut avoir et auxquels un utilisateur doit faire face. Ensuite, en se basant sur l'expérience acquise par le groupe Compose et sur la littérature existante, nous tentons d'identifier les raisons de cette situation.

3.1 Comportements inappropriés de la spécialisation

En général, avant de spécialiser un programme, le programmeur a une idée du programme spécialisé qu'il souhaite obtenir. Intuitivement, pendant la spécialisation, tous les calculs qui dépendent des paramètres d'entrée statiques doivent être évalués et les toutes occurrences de ces paramètres doivent disparaître.

Malheureusement lors de la spécialisation d'un programme, les comportements de *sous* ou de *sur-spécialisation* sont fréquents. Ainsi, en cas de sous-spécialisation, les valeurs statiques ne sont pas propagées autant que escompté et le programme résiduel n'est que partiellement, voire même pas du tout spécialisé. À l'opposé, lors d'une sur-spécialisation, les valeurs statiques sont propagées dans des contextes où elles ne permettent pas de simplifications supplémentaires. Dans certaines situations, cela mène à une explosion du code généré et le programme résiduel est alors parfois moins efficace que le programme original. Dans les situations extrêmes, des problèmes de non-terminaison peuvent arriver (chapitre 14 [JGS93]).

3.1.1 Sous-spécialisation

La sous-spécialisation arrive lorsqu'un ou plusieurs paramètres déclarés statiques deviennent dynamiques pendant l'analyse de temps de liaison. Ainsi, une fois dynamiques, ces paramètres ne peuvent plus être évalués pendant la spécialisation et le programme résiduel n'est pas ou peu spécialisé.

Les cas de figure où un paramètre déclaré statique devient dynamique sont nombreux. Le cas le plus simple correspond à la situation où une valeur dynamique est affectée au paramètre. Les situations plus complexes sont liées aux approximations effectuées par l'analyse de temps de liaison. Par exemple, toutes les variables affectées dans le corps d'une boucle possédant un test dynamique ou dans les branches d'une conditionnelle possédant un test dynamique, sont toutes considérées dynamiques après la boucle ou la conditionnelle. Si une conditionnelle a un test statique, alors toute variable affectée qui n'est pas statique dans les deux branches est considérée dynamique après la conditionnelle. Enfin, un paramètre peut devenir dynamique du fait de la granularité de l'analyse de temps de liaison. Par exemple, dans Tempo, quand un élément d'un tableau est affecté avec une valeur dynamique alors tous les autres éléments de ce tableau sont considérés par la suite comme dynamiques.

Pour mieux comprendre le genre de situation auquel les utilisateurs doivent faire face, nous étudions un exemple concret : la spécialisation de l'interprète des "Berkeley Packets Filters" (BPF) [MJ93].

Exemple de sous-spécialisation : L'interprète BPF filtre les paquets arrivants du réseau avant de les transmettre à une application. Seuls les paquets qui satisfont certains critères sont transmis. Ces critères prennent la forme d'un programme de filtrage qui est interprété.

Lorsqu'un paquet arrive du réseau, il est lu et filtré par la fonction `bpf_filter`. Cette fonction est montrée dans la Figure 3.1. Elle prend en arguments : le programme de filtrage des paquets, un paquet, la taille d'un paquet, et la quantité de données du paquet qui reste à traiter. L'interprète est implémenté itérativement. À chaque itération, l'instruction de programme pointée par `pc` est lue, décodée, exécutée et enfin `pc` est incrémenté.

Pour une session donnée, le programme de filtrage est toujours le même, il est donc interprété de nombreuses fois afin d'examiner des milliers de paquets. Spécialiser l'interprète par rapport à un programme de filtrage donné devrait permettre de compiler un programme dédié plus efficace.

Spécialisation Nous spécialisons la fonction `bpf_filter` pour un programme `pc` connu. Malheureusement, dans ce cas le programme résiduel que l'on obtient n'est pas du tout spécialisé. L'examen du fichier coloré produit par Tempo correspondant au résultat de l'analyse de temps de liaison montre que `pc` a été considéré comme dynamique par l'analyse. Ceci veut dire qu'à un moment donné `pc` est devenu dynamique et que cette valeur dynamique a été propagée dans toute la boucle du programme. Reste à

```
struct bpf_insn {
  u_short code;
  u_char jt;
  u_char jf;
  int k;
};

int bpf_filter(struct bpf_insn *pc, u_char *p,
               u_int wirelen, u_int buflen) {
  int A;
  ...
  while(1) {
    switch(pc->code) {
      case ...
      case BPF_JMP|BPF_JGT|BPF_K :
          if(A > pc->k)
            pc += pc->jt;
          else
            pc += pc->jf;
          break;
      case ...
    }
    pc = pc + 1;
  }
}
```

FIG. 3.1 – Structure de l'interprète BPF

découvrir où et pourquoi `pc` passe de statique à dynamique, ce qui sont deux problèmes distincts et souvent non triviaux à résoudre.

Explication du problème Une affectation est toujours à la base d'un changement de temps de liaison. La première chose à faire est donc de regarder les endroits où `pc` est affecté. Il s'avère que l'interprétation de certaines instructions donne lieu à une modification de `pc`, comme le montre la Figure 3.1. Dans le cas `BPF_JMP|BPF_JGT|BPF_K`, la variable `pc` est affectée dans les deux branches de la conditionnelle. Or il se trouve que le test de la conditionnelle est dynamique puisque la valeur de `A` ne peut être déterminée qu'à l'exécution. Le test étant dynamique, toutes les variables affectées dans les branches sont rendues dynamiques après la conditionnelle et ainsi `pc` devient dynamique.

```
int find(int *tab, int i, int delta, int x) {
  loop :
  if (delta == 0) {
    if (x == tab[i])
      return(i);
    else return(NOTFOUND);
  }
  if (x >= tab[i+delta]) i = i + delta;
  delta = delta/2;
  goto loop;
}

int binsearch(int *t, int delta, int x) {
  return find(t,0,delta,x);
}
```

FIG. 3.2 – Code source des fonctions `find` et `binsearch`

3.1.2 Sur-spécialisation

La sur-spécialisation survient lorsqu'une fonction est spécialisée par rapport à des valeurs statiques qui engendrent des transformations inutiles. Ainsi, par exemple, si une fonction est appelée dans une boucle qui a un test statique alors la spécialisation génère un variant de cette fonction à chaque itération. La création de tous ces variants est parfaitement inutile et est un gaspillage de place si aucun calcul dans les variants ne dépend de l'indice d'itération. De même, il faut aussi faire attention au nombre de fois qu'une boucle est déroulée ; le déroulage peut être intéressant pour un petit nombre d'itérations mais peut tout aussi être catastrophique pour les performances si le nombre d'itérations est trop important.

Pour illustrer une situation de sur-spécialisation, nous prenons un programme extrait de l'article de Neil Jones intitulé "What *Not* to do when writing an interpreter for specialization" [Jon96]. Cet article identifie un ensemble d'erreurs à ne pas commettre lors de l'utilisation de la spécialisation. Nous considérons l'exemple de la fonction `find` qui effectue la recherche d'un élément dans un tableau et dont la spécialisation avec le spécialiseur C-Mix [And94, C-M00] donne lieu à une explosion de code.

Exemple de sur-spécialisation : La fonction `find` implémente itérativement la recherche binaire d'un élément `x` dans un tableau `tab` de taille `2*delta-1`. Cette fonction est montrée dans la Figure 3.2. L'implémentation de la fonction `find` utilise un quatrième paramètre `i` qui marque la position courante dans le tableau `tab` à chaque

itération. Ainsi, un appel à la fonction `find` se fait avec `i` initialisé à zéro, comme le montre la fonction `binsearch` dans Figure 3.2.

Spécialisation Nous spécialisons la fonction `binsearch` avec le spécialiseur C-Mix pour `delta = 4` et par conséquence nous provoquons la spécialisation de la fonction `find` pour `delta = 4` et `i = 0`. Le résultat de cette spécialisation est montrée dans la Figure 3.3. Spécialiser `find` dans un contexte où `i` est statique provoque une explosion de code linéaire par rapport à la taille du tableau (c'est-à-dire `2*delta - 1`). Il n'est donc pas raisonnable d'effectuer une telle spécialisation quand le tableau devient grand. De plus, spécialiser `find` pour `i` statique n'est pas très utile puisque `i` permet seulement d'accéder aux éléments du tableau qui sont dynamiques au moment de la spécialisation. On obtient d'ailleurs un meilleur programme résiduel si on se contente de spécialiser `find` seulement par rapport à `delta` comme le montre la Figure 3.4.

Explication du problème Le spécialiseur C-Mix suit une stratégie basée sur une transformation de programmes appelée style par passage de continuation (CPS) [Bon92, CD91, Plo75]. Le principe de CPS est de propager à chaque point de programme l'ensemble des calculs restants. Dans le cas de la spécialisation de programmes, cette approche revient à spécialiser le reste du programme séparément dans chacune des branches d'une conditionnelle. Contrairement à l'approche prise par Tempo, le traitement des conditionnelles dynamiques de C-Mix ne cause aucune perte d'information.

Dans le cas de notre programme, les conditionnelles qui ont pour test `(x == tab[i])` et `(x >= tab[i+delta])` sont toutes les deux dynamiques. Dans le premier cas, les deux branches se terminent par un `return`, il n'y a donc rien à propager dans les branches. Dans l'autre cas, le reste du programme et les itérations suivantes sont propagés à la fin de chaque branche de la conditionnelle. C'est la spécialisation de chaque itération, faite séparément par rapport aux deux états résultants des branches de la conditionnelle, qui a pour effet de faire exploser la taille du programme spécialisé.

3.2 Sources des difficultés d'accessibilité

Nous donnons maintenant un aperçu des raisons qui font que spécialiser un programme n'est pas chose aisée. À cette fin, nous décrivons tout d'abord l'influence que la structure d'un programme a sur le bon déroulement de la spécialisation. Puis, nous faisons le lien entre la précision des analyses offertes par un spécialiseur donné et le degré de spécialisation du programme résiduel qui peut être obtenu. Enfin nous présentons les problèmes liés à la complexité d'utilisation d'un spécialiseur.

3.2.1 Structure des programmes

Spécialiser n'importe quel programme a peu de chance de produire un résultat intéressant. Même si un programme semble présenter des opportunités de spécialisation, une sous ou sur-spécialisation survient souvent. Il est alors plus efficace de seulement

```
int find_spe(int *tab, int x) {
  if (x >= tab[4])
    if (x >= tab[6])
      if (x >= tab[7]) {
       if (x == tab[7])
        return 7 ;
       }
      else {
       if (x == tab[6])
        return 6 ;
       }
    else
      if (x >= tab[5]) {
       if (x == tab[5])
        return 5 ;
       }
      else {
       if (x == tab[4])
        return 4 ;
       }
  else
    if (x >= tab[2])
      if (x >= tab[3]) {
       if (x == tab[3])
        return 3 ;
       }
      else {
       if (x == tab[2])
        return 2 ;
       }
    else
      if (x >= tab[1]) {
       if (x == tab[1])
        return 1 ;
       }
      else {
       if (x == tab[0])
        return 0 ;
       }
  return NOTFOUND ;
}

int binsearch_spe(int *t, int x) {
  return find_spe(t,x) ;
}
```

FIG. 3.3 – Spécialisation de `binsearch` pour `delta = 4`

```
int find_spe(int *tab, int x) {
  if (x >= tab[i + 4]) i = i + 4 ;
  if (x >= tab[i + 2]) i = i + 2 ;
  if (x >= tab[i + 1]) i = i + 1 ;
  if (x == tab[i]) return i ; else return NOTFOUND ;
}
```

FIG. 3.4 – Spécialisation de `find` pour `delta = 4`

envisager la spécialisation de petits fragments de code. Cependant la décomposition offerte par le programme source n'est pas toujours appropriée : soit le programme est trop monolithique, soit il n'est pas décomposé de façon adéquate pour la spécialisation. En fait, la spécialisabilité d'un programme est liée à la structure du programme et au style de programmation utilisé pour son implémentation. Pour rendre un programme spécialisable il est parfois possible de le réécrire en effectuant une amélioration de temps de liaison ("binding-time improvement").

3.2.1.1 Style de programmation

Le style de programmation utilisé pour le développement d'un programme a un impact certain sur la spécialisabibité de ce programme. Ainsi deux programmes sémantiquement équivalents peuvent se spécialiser de façon totalement différente. Tout dépend des structures de contrôle du programme ainsi que de l'agencement des données. Il est important lors du développement d'un programme de prendre en considération les temps de liaison des variables manipulées et, par exemple, de permettre aux tests des instructions de contrôle d'être statiques autant que possible ou regrouper les calculs arithmétiques statiques en utilisant la loi d'associativité. Ainsi lors de l'écriture d'une expression arithmétique de la forme `x + y + z` où seuls `x` et `z` sont statiques on préférera écrire `x + z + y` pour l'évaluation du calcul `x + z`.

3.2.1.2 Améliorations des temps de liaison

De manière générale, écrire un programme pour le spécialiser, demande un changement de la façon de penser et de programmer. La plupart des programmes n'ont pas été a priori développés dans le but d'être spécialisés et il faut les réécrire afin d'effectuer une amélioration de temps de liaison. Bien qu'il ne soit pas toujours facile de trouver comment modifier un programme pour le rendre spécialisable, la littérature propose des améliorations de temps de liaison pour adresser des situations bien définies [JGS93, Jon96, TCM+00].

Par exemple, une amélioration de temps de liaison bien connue permet de modifier l'interprète de BPF vu au début de ce chapitre, de façon à obtenir un meilleur pro-

```
int bpf_filter(struct bpf_insn *pc, u_char *p,
               u_int wirelen, u_int buflen) {
  int A;
  ...
  while(1) {
    switch(pc->code) {
      case ...
      case BPF_JMP|BPF_JGT|BPF_K :
        if (A >= pc->k)
         return bpf_filter(pc + pc->jt + 1,p,wirelen,buflen);
        else
         return bpf_filter(pc + pc->jf + 1,p,wirelen,buflen);
      case ...
    }
    pc = pc + 1;
  }
}
```

FIG. 3.5 – Amélioration des temps de liaison de l'interprète BPF

gramme spécialisé. Cette modification consiste à ne pas affecter une nouvelle valeur à la variable `pc` dans la conditionnelle dynamique. Cela peut se faire en appelant récursivement la fonction `bpf_filter` dans chaque branche de la conditionnelle [MWP+01]. Le résultat de cette amélioration de temps de liaison est illustré dans la Figure 3.5.

3.2.2 Précision des analyses

Spécialiser un même programme avec deux spécialiseurs différents ne produit pas forcément le même programme résiduel. Tout dépend des analyses mise en œuvre par les spécialiseurs. Il n'y a en effet pas une seule analyse de temps liaison qui permette d'adresser toutes les situations [CGL00]. Le choix des différentes analyses doit donc être fonction des applications de spécialisation visées. Cette diversité d'analyses plus ou moins précises rend le processus de spécialisation souvent imprévisible du fait de sa complexité. Il est d'autant plus difficile de comprendre ce qui s'est passé pendant une spécialisation que les spécialiseurs offrent peu d'informations quant aux transformations effectuées.

Comparons, par exemple, le comportement de Tempo et de C-Mix lors de la spécialisation de deux programmes.

Premier exemple Reprenons le cas de la fonction `find`. Nous avons vu précédemment que C-mix peut spécialiser cette fonction pour un `delta` et un `i` statiques. Il est impossible d'obtenir le même résultat avec Tempo, qui rend `i` dynamique pendant l'analyse de temps de liaison. La raison pour laquelle C-Mix a pu réaliser cette spécialisation est qu'il effectue une spécialisation basée sur le CPS tandis que Tempo est en style direct.

Deuxième exemple Considérons une fonction dans laquelle une variable de type pointeur est affectée avec l'adresse d'un tableau et est par la suite utilisée deux fois, une fois dans un contexte statique et une fois dans un contexte dynamique.

```
int f(int s, int d) {
  int a[20], *p;
  p = a;
  a[11] = 1234;
  return *((p + s) + 10) + *((p + d) + 20);
}
```

Spécialisons la fonction `f` pour `s = 1`. Les programmes résiduels que l'on obtient respectivement avec Tempo et C-Mix sont :

Tempo

```
int f_spe1(int d) {
  int a[20], *p;
  p = a;
  a[11] = 1234;
  return 1234 + *((p + d) + 20);
}
```

C-Mix

```
int f_spe2(int d) {
  int a[20], *p;
  p = a;
  a[11] = 1234;
  return *((p + 1) + 10) +
         *((p + d) + 20);
}
```

Dans le programme résiduel `f_spe1` généré par Tempo, le calcul `*((p + 1) + 10)` a été totalement évalué. De son côté, le programme `f_spe2` généré par C-Mix n'a que peu bénéficié de la spécialisation. L'explication de cette différence de trouve dans les analyses de temps de liaison des deux spécialiseurs : celle de Tempo est sensible aux contextes d'utilisation (chapitre 2) tandis que celle de C-Mix ne l'est pas.

3.2.3 Utilisation du spécialiseur

Afin de traiter des applications réalistes, les spécialiseurs sont devenus de plus en plus complexes. Ainsi il a fallu accroître la complexité du moteur de spécialisation et par la même occasion augmenter la complexité de paramétrisation de tels spécialiseurs.

Cependant, certaines fonctionnalités les plus avancées peuvent avoir un coût qui rend leur utilisation systématique non désirable. Il est donc préférable de laisser l'utilisateur décider quelles fonctionnalités utiliser. Nous avons vu dans le chapitre précédent, que Tempo disposait de plus de 50 paramètres de configuration. Le spécialiseur C-Mix n'est

pas en reste puisqu'il en présente plusieurs dizaines aussi. Manipuler ces paramètres de spécialisation n'est pas toujours évident et il est souvent nécessaire de connaître les détails de l'implémentation du spécialiseur pour comprendre les effets d'une directive donnée.

De plus, lorsque la spécialisation s'applique à un fragment de code, il faut décrire au spécialiseur comment ce bout de code interagit avec le reste du programme. Ce genre d'information est souvent complexe à décrire et les erreurs sont fréquentes si bien qu'il est fréquent que la spécialisation qui en découle ne corresponde pas au contexte de spécialisation que l'utilisateur avait en tête.

Enfin, chaque spécialiseur est différent, il a son propre ensemble de directives et son propre vocabulaire d'abstraction. Il faut donc tout réapprendre lorsque l'on change d'outil de spécialisation.

3.3 Bilan

Nous avons vu que la structure d'un programme, la précision des analyses et la paramétrisation d'un spécialiseur sont des éléments décisifs pour permettre le bon déroulement du processus de spécialisation. Si l'un des trois est inapproprié, il en résulte une spécialisation inadéquate où le programme obtenu est sous ou sur-spécialisé.

Bien qu'il existe des solutions documentées pour répondre à la majorité des problèmes qui peuvent se poser, la tâche de l'utilisateur d'un spécialiseur reste non triviale. D'après notre expérience, la difficulté ne vient pas seulement de la nécessité de trouver comment restructurer un programme, ou d'anticiper le comportement d'un spécialiseur donné, ou de décrire correctement le bon contexte de spécialisation mais aussi du fait que lorsqu'il y a un problème, l'utilisateur doit découvrir, parmi toutes les modifications potentielles, laquelle s'applique à la situation considérée. Si bien que l'utilisateur peut itérer longuement avant de trouver comment obtenir la spécialisation souhaitée.

Deuxième partie

Approche proposée

Chapitre 4

Présentation de l'approche

La spécialisation de programme est une technique d'optimisation qui permet d'éliminer la généricité présente dans un programme mais son utilisation reste réservée aux experts. Les travaux présentés dans cette thèse tentent de rendre cette technique plus accessible. Ce chapitre présente la démarche choisie pour répondre au problème de non-accessibilité et décrit l'impact de notre approche sur le processus de spécialisation hors ligne d'un programme, de l'implémentation du programme jusqu'à l'obtention du code spécialisé.

4.1 Rendre la spécialisation accessible

Comme nous l'avons vu dans le chapitre précédent, spécialiser un programme existant est une tâche difficile. Il faut étudier le code pour identifier les fragments de programme qui sont des bons candidats à la spécialisation. Il est aussi souvent nécessaire de le modifier afin que sa structure soit appropriée pour la spécialisation. Pour répondre à ce problème, nous proposons de prendre en considération la spécialisibilité d'un programme dès son développement. À ce stade, il est possible d'encoder la généricité en utilisant des motifs qui sont connus comme se spécialisant bien, et de structurer le programme de manière à bien séparer le code à spécialiser du reste du programme.

Cela n'est cependant pas suffisant pour garantir que le processus de spécialisation se déroulera bien ou du moins comme le programmeur l'avait prévu. En effet, les transformations effectuées pendant le processus de spécialisation peuvent varier d'un spécialiseur à l'autre, et quand bien même les propriétés d'analyse d'un spécialiseur donné sont connues, il est souvent difficile de prévoir le résultat qui sera obtenu. De plus le programmeur a peu de moyens pour exprimer ses intentions de spécialisation et il est alors impossible pour le spécialiseur de renseigner sur les raisons possibles d'une spécialisation inappropriée. Pour adresser ce problème, nous proposons qu'au moment du développement d'un programme, le programmeur décrive les *scénarios de spécialisation* de ce programme [LMLC02b, LMLC02a]. Ces scénarios expriment les intentions de spécialisation du programmeur, c'est-à-dire qu'ils renseignement sur les fragments de code à prendre en considération pendant la spécialisation, ainsi que sur le contexte

dans lequel ils doivent être spécialisés. Les scénarios sont organisés dans des *modules de spécialisation* distincts du code source et constituent ainsi une expertise réutilisable de la spécialisabilité du programme. À partir de ces déclarations, il est possible de vérifier que les opportunités de spécialisation spécifiées sont prisent en compte par les analyses et de détecter ainsi les problèmes potentiels, rendant par la même occasion le processus de spécialisation prévisible par rapport aux déclarations.

De façon intuitive, un scénario de spécialisation est analogue à une déclaration de types, où les types décrivent quelles informations sont connues ou inconnues pendant la spécialisation, plutôt que des ensembles de valeurs concrètes. La manière avec laquelle ces déclarations sont utilisées est similaire à l'utilisation des déclarations de types dans un langage qui offre une inférence de types tel que ML. Alors que l'inférence de types de ML est suffisant pour déterminer les types de la plupart des programmes à partir seulement des informations sur les types des composants de structure de données fournies par le programmeur, les annotations du programmeur de chaque fonction avec son type améliore la lisibilité du code et facilite la mise au point du programme. La déclaration d'un scénario de spécialisation pour chaque fonction, structure de donnée et variable globale offre des avantages similaires. Les programmes ML et les déclarations de types associées sont organisés dans des modules, ce qui facilite le compréhension globale du programme et simplifie la modification de ses fonctionnalités. Les modules de spécialisation présentent les mêmes avantages.

Enfin, pour répondre au problème lié à la complexité d'utilisation d'un spécialiseur, nous proposons d'utiliser les déclarations pour configurer automatiquement l'outil de spécialisation.

4.2 Approche

Une vue générale de notre approche est montrée dans la Figure 4.1. Une fois le développement du code et des modules de spécialisation achevé, la phase de pré-spécialisation peut avoir lieu. Cette phase consiste a compiler les modules pour extraire les informations nécessaires à la configuration du spécialiseur ainsi que les *contraintes*, c'est-à-dire les propriétés de spécialisation, qui devront être respectées. Ensuite, la vérification de ces contraintes s'effectue pendant la phase d'analyse du processus de spécialisation hors ligne. Si la vérification détermine qu'il n'est pas possible de satisfaire une contrainte alors les analyses s'arrêtent et l'erreur est pointée. Dans le cas contraire, l'étape de spécialisation avec les valeurs de spécialisation peut être effectuée.

4.2.1 Développement du code et des modules de spécialisation

Lors de l'écriture d'un programme, la manipulation de tous les paramètres qui seront directement impliqués dans le processus de spécialisation doit se faire avec précaution car ce sont les temps de liaison de ces paramètres qui détermineront le degré de spécialisation du programme. Il est donc nécessaire que le programmeur raisonne sur les propriétés de spécialisation de ces données, c'est-à-dire leurs temps de liaison. Cela n'implique cependant pas que le programmeur soit obligé d'effectuer une analyse de temps de liaison

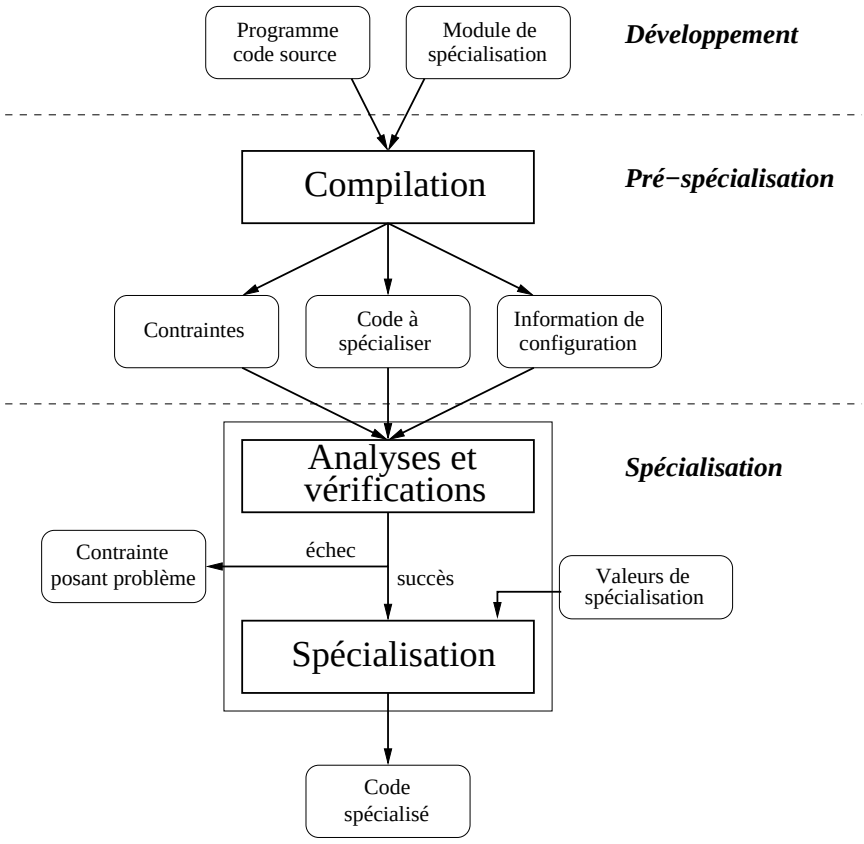


FIG. 4.1 – Vue générale de notre approche

manuellement. Tout comme il ne lui a jamais été demandé de réaliser une vérification de types d'un programme. Il doit cependant raisonner sur les types des données qu'il manipule.

Le programmeur a donc a priori prévu un ensemble de scénarios selon lesquels son programme peut être spécialisé. Notre langage de déclarations lui offre la possibilité de décrire ces scénarios. Il peut ainsi préciser les fragments de code à spécialiser en spécifiant les contextes de spécialisation, c'est-à-dire les contraintes de spécialisation des différents paramètres impliqués dans le processus de spécialisation. L'écriture des scénarios permet de documenter les caractéristiques de spécialisation d'un programme et offre au programmeur un cadre de raisonnement qui l'incite à réfléchir sur les propriétés de spécialisation de son programme.

Un module de spécialisation regroupe toutes les informations de spécialisation d'un ensemble de programmes donné et facilite de ce fait la compréhension et la réutilisation des scénarios de spécialisation. De plus, cela incite le programmeur à avoir une approche modulaire à la spécialisation. Adopter une programmation modulaire peut être bénéfique. La décomposition d'une implémentation en une collection de fonctions

et de fichiers facilite le raisonnement sur la structure et les propriétés de spécialisation du programme. Les modules de spécialisation sont spécialement conçus pour permettre de spécifier comment se combinent les propriétés de spécialisation des différentes fonctions à travers les différents fichiers. Les détails de notre langage de déclarations sont présentés dans le chapitre 5.

4.2.2 Compilation des modules de spécialisation

Une fois l'implémentation du programme et des modules de spécialisation terminée, il faut préparer la phase de spécialisation. Cela consiste à compiler les modules de spécialisation en précisant le scénario de spécialisation par rapport auquel on souhaite que la spécialisation se fasse.

Cette compilation permet de réunir toutes les contraintes de spécialisation qui doivent être respectées. De plus, les informations nécessaires à la configuration du spécialiseur sont extraites. Dans le cadre de Tempo, le spécialiseur que nous avons choisi pour notre étude, cela revient à générer automatiquement les fichiers `config.sml` et `actx.c` et à regrouper dans un même fichier tout le code qui doit être spécialisé.

L'utilisateur du spécialiseur n'a donc plus besoin de se préoccuper de la configuration de l'outil, ce qui évite les erreurs. De plus, il doit seulement se familiariser avec les modules de spécialisation, l'utilisation d'un spécialiseur donné est devenue transparente. Cela réduit le nombre de notions à acquérir et donc le temps d'apprentissage.

4.2.3 Analyses et vérification

Nous avons choisi d'effectuer les vérifications pendant l'analyse de temps de liaison. Ces vérifications peuvent être vues comme des vérifications de types. Ainsi, lorsque que l'analyse calcule les temps de liaison d'un paramètre et que ces temps de liaisons sont en contradiction avec les contraintes spécifiées par le programmeur, une erreur est levée et le processus de spécialisation est arrêté. Dans le cas contraire la spécialisation peut avoir lieu.

Les vérifications permettent donc de rendre le processus de spécialisation prévisible par rapport aux déclarations : soit il est possible d'obtenir un programme spécialisé qui satisfait les déclarations de spécialisation, soit les déclarations sont en contradiction avec le résultat des analyses et alors le processus de spécialisation est avorté. Les détails concernant les vérifications effectuées par notre approche sont présentés dans le chapitre 6.

4.2.4 Spécialisation

La phase de spécialisation n'est en rien affectée par les modifications apportées dans l'analyse de temps de liaison. La phase spécialisation se déroule donc comme décrite dans le chapitre 2. Nous présentons néanmoins à la fin du chapitre 6 quelques arguments qui permettent d'affirmer que la phase de spécialisation respecte d'une part le critère de correction standard de spécialisation et d'autre part les déclarations du programmeur.

Chapitre 5

Modules de spécialisation

Dans ce chapitre, nous motivons tout d'abord les choix de conception du langage des modules de spécialisation. Puis nous décrivons la syntaxe du langage et illustrons à l'aide d'un exemple simple les déclarations correspondant à la spécification des contraintes de temps de liaison. Nous présentons ensuite les déclarations qui permettent de préciser le contexte d'initialisation d'une spécialisation. Enfin nous expliquons comment les modules de spécialisation sont compilés.

5.1 Choix de conception

Comme nous l'avons décrit dans le chapitre 2, la spécialisation de programme est souvent implémentée en deux phases : une phase d'analyse et une phase de spécialisation. La phase d'analyses détermine comment les valeurs statiques sont propagées dans le programme. Pour obtenir un résultat satisfaisant, il nécessaire d'identifier des opportunités de spécialisation et de structurer le programme tel que ces opportunités puissent être exploitées par l'outil de spécialisation. Ces opérations requièrent une compréhension approfondie de l'implémentation et doivent donc être effectuées par le *programmeur* du code. Choisir les valeurs de spécialisation, nécessite seulement de savoir comment le code spécialisé sera utilisé. De ce fait, nous disons que la spécialisation est invoquée par un *utilisateur* qui n'est pas forcément le programmeur. De part la différence entre les niveaux d'expertise requis, notre approche fait explicitement la distinction entre la tâche du programmeur et celle de l'utilisateur.

Au minimum, la description d'un scénario de spécialisation doit préciser le point d'entrée du programme à partir duquel la spécialisation doit commencer, ainsi que les variables par rapport auxquelles le code doit être spécialisé. Cependant, l'expérience montre que ces informations ne sont pas suffisantes pour traduire les intentions de spécialisation du programmeur et ainsi permettre à un spécialiseur de programmes d'effectuer la spécialisation attendue.

Typiquement, la spécialisation ne peut être appliquée qu'à un fragment de code. Il faut donc désigner quelles sont les fonctions qui doivent être spécialisées et celles qui doivent être considérées comme externes au processus de spécialisation. Pour garantir

```
struct vec {
  int *data;
  int length;
};

int dot(struct vec *u, struct vec *v) {
  int i = 0;
  int sum = 0;
  if (u->length != v->length)
    error();
  for(i = 0; i < u->length; i++)
    sum += u->data[i] * v->data[i];
  return sum;
}
```

FIG. 5.1 – Code de la fonction `dot`

que la spécialisation satisfait les attentes du programmeur, il est nécessaire de spécifier les temps de liaison désirés dans le programme à spécialiser. Enfin, il faut préciser comment les valeurs de spécialisation seront obtenues.

Prenons comme exemple la fonction `dot` de la Figure 5.1. Cette fonction implémente la multiplication de deux vecteurs d'entiers. Elle vérifie tout d'abord que les vecteurs ont la même taille, avortant l'opération en faisant appel à la fonction `error` dans le cas contraire, et calcule ensuite le produit scalaire des vecteurs. Spécialisons cette fonction pour un vecteur `u` complètement statique. Cette spécialisation aura pour principal effet de dérouler la boucle `for` et de remplacer les références aux éléments de `u` par des constantes. L'information minimale nécessaire pour préparer cette spécialisation est composée du nom de la fonction `dot` et de l'information précisant que `u`, à l'entrée de cette fonction, est complètement statique. L'intuition du programmeur concernant le déroulement de cette spécialisation contient cependant plus d'informations. Ainsi, le programmeur peut s'attendre à ce que la fonction `error` ne soit pas spécialisée, que les valeurs des données du vecteur `u` restent statiques tout au long du produit scalaire, et que ces valeurs soient obtenues d'une manière spécifique, comme par exemple, précisées à la spécialisation ou calculées à l'aide d'une fonction externe. Le but des scénarios de spécialisation est de permettre au programmeur d'exprimer ces propriétés et de le faire d'une manière intuitive.

Plusieurs stratégies peuvent être envisagées afin de permettre au programmeur de spécifier les temps de liaisons. Une solution possible est d'annoter toutes les constructions de programmes avec un temps de liaison, et donc de réécrire le programme source à l'aide d'un *langage à deux niveaux* [EHK96, TS97]. Cependant, cette approche est

lourde et peut rendre le processus de spécialisation trop contraignant. Dans une approche différente, nous proposons que le programmeur déclare les propriétés de temps de liaison des *paramètres de spécialisation* : les variables globales, les champs de structures de données et les arguments de fonction. Les temps de liaison des paramètres de spécialisation ainsi déclarés sont alors associés à chaque référence de ces paramètres à travers tout le programme, ce qui permet de vérifier les intentions de spécialisation du programmeur. Un autre avantage pratique de cette approche est que les déclarations ne sont pas liées à la structure interne de chaque définition de fonction. La plupart des améliorations de temps de liaison décrites dans la littérature comportent des modifications de la structure du code source, et non pas de l'ensemble des paramètres de spécialisation. Ainsi, les modifications nécessaires pour la mise au point d'un processus de spécialisation ne requiert pas de changer les déclarations de temps liaison.

Pour minimiser l'apprentissage d'utilisation du langage de déclarations de spécialisation, ces déclarations doivent être le plus proche possible de la syntaxe du langage cible. Notre langage a pour but de traiter les programmes C, et donc les déclarations de propriétés de spécialisation consistent simplement à annoter des déclarations C. Pour faciliter la compréhension du processus de spécialisation, les scénarios concernant un même fragment de code sont regroupés dans un module. Plusieurs modules peuvent être déclarés pour un même fragment de code, chaque module correspondant alors à une vue différente de sa spécialisabilité. Alternativement, plusieurs scénarios peuvent être déclarés pour un même paramètre de spécialisation dans un seul module.

5.2 Contexte de spécialisation

Nous commençons par considérer les différentes situations que peut rencontrer un programmeur lors de l'écriture du code à spécialiser et des temps de liaison escomptés à travers le programme. Nous illustrons notre approche tout d'abord avec un exemple qui met en évidence plusieurs caractéristiques de notre langage de déclarations, et nous présentons ensuite les détails du langage.

5.2.1 Exemple

Considérons de nouveau la fonction `dot` de la Figure 5.1. Cette fonction présente plusieurs opportunités de spécialisation. Ainsi, si la taille des deux vecteurs est statique, le test sur la taille peut être réduit. De plus si la taille du vecteur `u` est statique, la boucle `for` peut être déroulée. Enfin, si les données d'un des deux vecteurs sont connues alors il est possible de déplier ces données dans les calculs. Le module de spécialisation `vector`, montré dans la Figure 5.2, décrit ces opportunités de spécialisation. Les opportunités associées à chaque construction de programme sont les suivantes :

La structure de données `vec` : Les scénarios `VecDS` (ligne (1)) et `VecSS` (ligne (2)) décrivent les propriétés de spécialisation de la structure `vec` qui peuvent résulter en une spécialisation intéressante. Dans les deux scénarios spécifiés, le champ `length`, représentant la taille du vecteur, est déclaré comme ayant le type `S(int)`, indiquant

```
Module vector {
 ...
 Defines {
  From dotproduct.c {
   VecDS::struct vec                                                    (1)
          {D(int *) data; S(int) length;};
   VecSS::struct vec                                                    (2)
          {S(int *) data; S(int) length;};
   Btdot1::intern dot                                                   (3)
           (VecDS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           { needs{Bterr;} };                                           (4)
   Bterr::extern error();                                               (5)
   Btdot2::intern dot                                                   (6)
           (VecSS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           { needs{Bterr;} };
 ...}...}
 Exports {VecDS; VecSS; Btdot1; Btdot2; ...}                            (7)
}
```

FIG. 5.2 – Module de spécialisation `vector`

ainsi que la taille doit être statique. Typiquement cette information permet à la spécialisation d'éliminer les tests faits sur la taille du vecteur comme par exemple les tests sur les débordements de tableaux. Dans `VecDS`, le champ `data`, qui correspond aux données du vecteur, est déclaré comme ayant le type `D(int *)`, ce qui indique que les données doivent être considérées comme dynamiques. Pour `VecSS`, les données doivent être statiques, ce qui permet aux données d'être dépliées.

La fonction `dot` : Les scénarios `Btdot1` et `Btdot2` sont deux scénarios de spécialisation possibles pour la fonction `dot`. Le scénario `Btdot1` (ligne (3)) indique que `dot` peut être spécialisée lorsque les pointeurs `u` et `v` sont tous les deux statiques et lorsque les vecteurs auxquels ils font référence satisfont le scénario `VecDS`. Le scénario `Btdot1` doit de plus décrire le comportement de spécialisation de toutes les fonctions appelées par `dot`. Dans notre cas, seule la fonction `error` est appelée. La déclaration `needs{Bterr;}` (ligne (4)) indique que le scénario `Bterr` (ligne (5)) doit être satisfait à chaque appel de la fonction `error`. Ce scénario indique de plus que `error` doit être considérée comme étant `extern`, c'est-à-dire que cette fonction ne présente pas d'intérêt à être spécialisée et ne doit donc pas être incluse dans le processus de spécialisation. Le module de spécialisation définit aussi le scénario `Btdot2` (ligne (6)) qui spécifie que `dot` peut être spécialisée si la taille des deux vecteurs est statique, et si les données du vecteur référencé par `u` le sont aussi. Un troisième scénario pourrait être défini pour le

cas où les données référencées par le pointeur `v` sont statiques.

Une fois que tous les scénarios ont été définis, nous les rendons accessibles à d'autres modules de spécialisation en rajoutant leurs noms dans la section `Exports` (ligne (7)). Le scénario `Bterr` n'est pas exporté puisque cette fonction est seulement destinée à une utilisation locale.

5.2.2 Langage de déclarations

La syntaxe complète du langage de déclarations de spécialisation est définie dans la Figure 5.3. Un module de spécialisation est introduit par le mot clé `Module` et est formé du nom du module suivi par trois sections : *imports* (qui est optionnel), *defines* et *exports*.

Imports La section *imports*, introduite par le mot clé `Imports`, permet au module courant de faire référence à des scénarios de spécialisation définis dans d'autres modules. Ceux-ci peuvent être importés de plusieurs modules à l'aide de la déclaration

`From` *file* `{`$(scen\_\ id\ ;)^+$`}`

où *file* fait référence à un fichier qui contient la définition d'un autre module. À l'intérieur de cette déclaration, plusieurs scénarios peuvent être importés.

Defines La section *defines*, introduite par le mot clé `Defines`, permet de définir une collection de scénarios de spécialisation. Chaque scénario est associé à un paramètre de spécialisation défini dans un fichier source identifié par la déclaration `From` *file*. La base d'un scénario correspond à la déclaration *bt_ info_ decl* qui prend la forme d'une déclaration C où les types ont été annotés soit par des temps de liaison simples (`S` or `D`), soit, dans le cas d'une structure, par le nom d'un scénario de spécialisation :

$$
\begin{array}{lll}
bt\_\ info\_\ decl & ::= & base\_\ type\ \ (pointer)^*\ \ id\ \ (array)^* \\
 & & \quad \mid\ \texttt{(}(pointer)^+\ \ id\ \ (array)^*\texttt{)}\ \ \texttt{()} \\
base\_\ type & ::= & base\_\ bt\texttt{(}simple\_\ type\texttt{)}\ \mid\ struct\_\ type \\
pointer & ::= & base\_\ bt\texttt{(*)} \\
array & ::= & base\_\ bt\texttt{([])} \\
base\_\ bt & ::= & \texttt{S}\ \mid\ \texttt{D}
\end{array}
$$

Ces déclarations doivent être bien formées : un pointeur dynamique ne peut pas être déclaré comme pointant sur une valeur statique. Des scénarios peuvent être déclarés pour les paramètres de spécialisation suivants :

Variables globales **:** La déclaration

$$scen\_\ id\ ::\ bt\_\ info\_\ decl$$

crée un scénario de spécialisation *scen_ id* qui associe des propriétés de spécialisation à la variable globale mentionnée dans *bt_ info_ decl*.

```
module          ::= Module module_id { (imports)? defines exports }
imports         ::= Imports { (From file { (scen_id ;)+ })* }
defines         ::= Defines { (From file { (definition ;)+ })+ }
exports         ::= Exports { (scen_id ;)+ }

definition      ::= global_def | struct_def | proc_def
global_def      ::= scen_id :: bt_info_decl
                      ({ (needs { (scen_id ;)+ })? (inits { (init_info_decl ;)+ })? })?
struct_def      ::= scen_id :: struct struct_id { (bt_info_decl ;)+ }
proc_def        ::= scen_id :: intern proc_id( (params)? )
                      ({ (needs { (scen_id ;)+ })? (inits { (init_info_decl ;)+ })? })?
                    | scen_id :: extern (bt_info)? proc_id( (params)? )
params          ::= (bt_info_decl , )* bt_info_decl

bt_info_decl    ::= base_type (pointer)* id (array)*
                    | ( (pointer)+ id (array)* ) ()
bt_info         ::= base_type (pointer)*
base_type       ::= base_bt(simple_type) | struct_type
base_bt         ::= S | D
simple_type     ::= int | long | char | ...
pointer         ::= base_bt(*)
array           ::= base_bt([])
struct_type     ::= scen_id(struct struct_id)

init_info_decl  ::= spe_param = none
                    | spe_param = expr
                    | spe_param = proc_call From file
proc_call       ::= proc_id( ((expr , )* expr)? )

module_id       ∈  Noms de module
scen_id         ∈  Noms de scénarios de spécialisation
proc_id         ∈  Noms de fonction
struct_id       ∈  Noms de structure
id              ∈  Noms d'identificateur
spe_param       ∈  Paramètres de spécialisation
```

FIG. 5.3 – Syntaxe du langage de déclarations

Structures de données : La déclaration

$$scen\_id \texttt{ :: struct } struct\_id \texttt{ \{}(bt\_info\_decl \texttt{ ;})^{+}\texttt{\}}$$

crée un scénario de spécialisation *scen_id* qui décrit les propriétés de spécialisation associées à chaque champ de la structure *struct_id*.

Fonctions : La déclaration

$$scen\_id \texttt{ :: intern } proc\_id\texttt{(}(params)^{?}\texttt{)(\{}needs\_list\texttt{\}})^{?}$$

crée un scénario de spécialisation *scen_id* qui spécifie les propriétés de spécialisation de chacun des arguments de la fonction *proc_id*. Le mot clé `intern` indique que la fonction doit être spécialisée. Dans ce cas, le scénario doit de plus préciser, dans *needs_list*, les scénarios de toutes les fonctions, variables globales, et structure de données qui sont utilisées par la fonction *proc_id*.

La déclaration

$$scen\_id \texttt{ :: extern } (bt\_info)^{?} \ proc\_id\texttt{(}(params)^{?}\texttt{)}$$

crée un scénario de spécialisation *scen_id* pour la fonction *proc_id* qui ne doit pas être spécialisée. Le scénario *scen_id* précise que les paramètres de la fonction *proc_id* doivent avoir les temps de liaison décrits par *params*. De plus, la déclaration optionnelle *bt_info* décrit le temps de liaison de la valeur de retour de *proc_id*. Les temps de liaison des paramètres et de la valeur de retour déterminent si la fonction externe est appelée à la spécialisation. Plus précisément, la fonction est appelée si tous les paramètres de la fonction sont statiques, ainsi que la valeur de retour si elle est précisée. Enfin, pour une fonction externe, la partie *needs_list* n'est pas nécessaire car la définition de la fonction n'est pas accessible par le spécialiseur de programmes.

Exports La section *exports*, introduite par le mot clé `Exports`, liste les scénarios définis dans le module courant qui sont exportés et ainsi mis à la disposition des autres modules.

La syntaxe du langage des déclarations implique que les modules de spécialisation peuvent être facilement construits en copiant et en annotant les déclarations déjà présentes dans les fichiers source C. De plus, on peut imaginer qu'il serait possible de développer un outil qui permettrait au programmeur de décorer les programmes sources avec des temps de liaison et qui générerait alors automatiquement les modules de spécialisation.

5.3 Contexte d'initialisation

En plus des contraintes de temps de liaison, le langage des déclarations de spécialisation permet au programmeur de spécifier le contexte d'initialisation des paramètres statiques.

Dans la plupart des cas, le programmeur souhaite exposer à l'utilisateur du code spécialisé, les variables globales et les arguments de fonction statiques de façon à ce qu'il puisse spécifier leurs valeurs de spécialisation. Cependant, dans certains cas, la valeur d'un paramètre d'entrée statique est directement dépendante de la valeur des autres paramètres d'entrée et il ne faut pas exposer ce paramètre à l'utilisateur. On doit alors spécifier non seulement comment ce paramètre est initialisé, mais aussi à quel moment cette initialisation à lieu, c'est-à-dire avant ou à l'intérieur du code à spécialiser.

Reprenons l'exemple de la fonction `dot` montrée dans la Figure 5.1. Supposons que `dot` soit utilisée pour calculer le produit scalaire d'un vecteur `v` de taille statique avec un autre vecteur `u` de même taille mais dont les données ont une structure prédéfinie. Dans ce cas, la façon de déterminer `u` est une opération prédéfinie du contexte de spécialisation, et ne doit donc être précisée par le programmeur dans le scénario de spécialisation. La valeur de la taille de `v` doit, quant à elle, être donnée par l'utilisateur au moment de la spécialisation.

Le comportement par défaut est de supposer que l'utilisateur fournit la valeur de spécialisation de tout paramètre déclaré comme statique. Il n'y alors dans ce cas aucune déclaration supplémentaire à rajouter. Pour les autres situations, le langage offre trois déclarations pour préciser les scénarios d'initialisation :

init_info_decl : := *spe_param* `=` `none`
| *spe_param* `=` *expr*
| *spe_param* `=` *proc_call* `From` *file*

proc_call : := *proc_id* `(` ((*expr* `,`)* *expr*)? `)`

La déclaration *spe_param* `=` `none` signifie que le paramètre de spécialisation *spe_param* est initialisé dans le corps de la fonction à spécialiser. La déclaration *spe_param* `=` *expr* initialise *spe_param* avec la valeur de *expr* au moment de la spécialisation. La déclaration *spe_param* `=` *proc_call* `From` *file* a pour effet de lancer l'exécution de *proc_call* juste avant le début de la spécialisation. Cette fonction retourne la valeur d'initialisation du paramètre statique pour la spécialisation. On suppose que cette initialisation ne crée pas d'alias parmi les paramètres d'entrée.

Nous utilisation ces déclarations pour exprimer le contexte d'initialisation de la fonction `dot` tel qu'il est décrit par le scénario `Btdot3` de la Figure 5.4. La déclaration à la ligne (1) précise que la taille du vecteur `u` obtient sa valeur au moment de la spécialisation à partir de la taille du vecteur `v`. Le champ `data` du vecteur `u` est initialisé en appelant la fonction `create_data` qui est définie dans le fichier source `init.c` et prend comme argument `v->length` (ligne (2)). Aucune information n'est précisée pour le vecteur `v`, ce qui signifie que son champ statique `length` sera initialisé par l'utilisateur.

5.4 Compilation des modules de spécialisation

Le lien entre les déclarations d'un module de spécialisation et le processus de spécialisation est fait par un compilateur. Ce compilateur comporte deux phases : une phase de collection des informations et une phase de préparation du processus de spécialisation.

```
Module vector {
 ...
 Defines {
  From dotproduct.c {
   VecDS::struct vec
          {D(int *) data; S(int) length;};
   VecSS::struct vec
          {S(int *) data; S(int) length;};
   ...
   Bterr::extern error();
   ...
   Btdot3::intern dot
           (VecSS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           { needs{Bterr;}
             inits{ u->length = v->length;                    (1)
                    u->data = create_data(v->length)
                              From init.c; }                  (2)
 ...}...}
 Exports {VecDS; VecSS; ...; Btdot3;...}
}
```

FIG. 5.4 – Module de spécialisation `vector` avec les scénarios d'initialisation

5.4.1 Collection des informations

Cette phase consiste à créer un environnent qui contient tous les scénarios de spécialisation utiles pour une spécialisation donnée.

La compilation prend en entrée un module et le scénario de spécialisation principal selon lequel on désire spécialiser une fonction. Une fois la syntaxe des modules vérifiée, tous les scénarios qui dépendent du scénario principal sont collectés. Ceci est réalisé en itérant successivement sur la liste des `needs` associée à chaque scénario. L'itération prend fin lorsqu'un point fixe a été atteint, c'est-à-dire lorsque tous les scénarios impliqués dans la spécialisation désirée ont été trouvés. Une vérification sémantique des modules de spécialisation est aussi réalisée. Ainsi, il est par exemple vérifié qu'un scénario importé d'un module est bien présent dans la section `exports` de ce module.

Une fois que l'environnement contenant tous les scénarios impliqués dans la spécialisation est construit, il est possible de passer à l'étape suivante, c'est-à-dire à la préparation du processus de spécialisation.

5.4.2 Préparation du processus de spécialisation

Contrairement à l'étape de collection des informations, les opérations effectuées lors de la préparation du processus de spécialisation sont spécifiques au spécialiseur que l'on

utilise. Nous avons implémenté notre compilateur pour qu'il transforme les déclarations de spécialisation en informations de configuration pour Tempo.

Il s'agit ici de générer automatiquement les fichiers `config.sml`, `actx.c` et `sctx.c`. Pour illustrer cette opération, nous reprenons l'exemple de `dotproduct` présenté dans le chapitre 2 :

```
int dotproduct (int *u, int *v, int size) {
  int i ;
  int sum = 0 ;
  for(i = 0 ; i < size ; i++)
    sum = sum + u[i] * v[i] ;
  return sum ;
}
```

Nous avons vu qu'il était possible de spécialiser la fonction `dotproduct` lorsque la taille et les données du vecteur `u` sont connues. Voici le module de spécialisation qu'il faut créer :

```
Module dotproduct {
  Defines {
    From dotproduct.c {
      Btdot :: intern dotproduct
             (S(int) u S([]), D(int) v D([]), S(int) size) ;
    } ;
  }
  Exports { Btdot ; }
}
```

Notons qu'au lieu d'écrire `S(int *) u`, nous avons mis `S(int) u S([])` pour préciser que `u` est en fait un tableau de données et non pas juste un pointeur. Cela permet de donner des informations plus précises aux analyses.

La compilation du module `dotproduct` avec le scénario `Btdot` comme scénario principal crée un environnent simple qui ne contient que les informations associées au scénario `Btdot`. À partir de ces informations, il est possible de générer les fichiers `config.sml` et `actx.c` :

config.sml : La fonction à spécialiser est la fonction `dotproduct`. Reste à préciser les temps de liaison des arguments. Le scénario `Btdot` indique que `u` est un pointeur statique, que `v` est dynamique et que `size` est statique. On génère donc :

```
entry_point := "dotproduct(S,D,S)" ;
```

actx.c : Les fichiers `actx.c` sont tous construits sur le même modèle : ils définissent une fonction `set_analysis_context` qui a le même nombre d'arguments que la fonction

à spécialiser et qui a pour but de recréer un contexte de spécialisation complexe. Elle a donc la forme de : `void set_analysis_context(int ** u, int ** v, int * size)` Dans notre cas, la complexité vient du fait il faut préciser que non seulement `u` est un tableau d'adresse statique mais que les données du tableau le sont aussi. En effet, `dotproduct(S,D,S)` précise que `u` est statique mais ne dit rien sur ses données. Pour exprimer cette propriété, il faut reconstruire le contexte de spécialisation de `u`. À partir de la déclaration `S(int) u S([])`, on sait que `u` est un tableau d'entiers, on crée alors une variable globale de type tableau d'entiers, auquel on fait correspondre `u` :

```
int dummy_array[1] ;
void set_analysis_context(int ** u, int ** v, int * size) {
  *u = dummy_array ;
}
```

Pour préciser que les données du tableau sont statiques la déclaration

```
static_locations := ["dummy_array"] ;
```

est rajoutée dans le fichier `config.sml`.

sctx.c : Pour permettre à l'utilisateur de préciser les valeurs de spécialisation, le compilateur génère aussi un squelette du fichier `sctx.c`. Ce squelette expose tous les paramètres qui doivent être initialisés. Si des informations d'initialisation sont décrites dans les scénarios de spécialisation, elles sont alors prises en compte dans cette génération de code. Dans le cas de notre exemple, le code qui est généré est le suivant :

```
void set_specialization_context(int ** u, int * size) {
  *u = /* to be filled */ ;
  *size = /* to be filled */ ;
}
```

Les fichiers de configuration de Tempo sont maintenant prêts. Il reste à réunir tout le code à spécialiser dans un même fichier comme le veut Tempo. Ceci est réalisé automatiquement par extraction des fragments de code qui sont impliqués dans la spécialisation des fichiers sources. Enfin pour terminer, les contraintes qui doivent être vérifiées pendant l'analyse de temps de liaison sont regroupées dans un fichier qui est lu par Tempo avant la phase d'analyse. Cela consiste à réunir les déclarations de temps de liaison associés aux variables globales, aux structures et aux arguments de fonction. La phase de pré-spécialisation une fois terminée, le processus de spécialisation peut commencer.

5.5 Bilan

Notre langage de déclarations permet au programmeur de décrire, d'une façon intuitive, les scénarios de spécialisation d'une fonction, c'est-à-dire les contextes dans lesquels la fonction doit être spécialisée.

La description d'un contexte comprend trois types d'information : l'identification des fragments de code à prendre en considération pendant la spécialisation ; la spécification des temps de liaison des paramètres impliqués dans la spécialisation ; la façon d'obtenir les valeurs de spécialisation.

Les informations les plus importantes et sur lesquelles repose toute notre approche sont les informations de temps de liaison. Les temps de liaison spécifiés pour les variables globales, les champs de structures de données et les arguments de fonction ont pour but de s'assurer que les valeurs statiques sont complètement propagées à travers l'intérieur du programme. De plus, les temps de liaison associés aux arguments d'une fonction permettent de décrire un scénario pour lequel la spécialisation de la fonction est bénéfique. L'intérêt principal est de pouvoir garantir par la suite que les blocs de base, qui sont eux-mêmes bien compris, vont bien ensemble lorsqu'ils sont combinés.

Enfin, la compilation des déclarations prépare le processus de spécialisation, rendant par la même occasion la configuration du spécialiseur transparente.

Chapitre 6

Vérification

Un module de spécialisation précise quel est le contexte dans lequel un fragment de code doit être spécialisé, et comment les fonctions spécialisées qui sont appelées dans le fragment de code interagissent. Afin de garantir que le degré de spécialisation déclaré est effectivement atteint, nous rajoutons des vérifications au processus de spécialisation de programmes standard. Le but principal de ces vérifications est de s'assurer que les contraintes déclarées sont respectées, c'est-à-dire que les informations spécifiées comme étant statiques sont effectivement bien propagées à travers le programme.

Les déclarations sont vérifiées pendant la phase d'analyse du processus de spécialisation hors ligne. Nous commençons par décrire la stratégie de vérification que nous avons choisie. Cette stratégie est appropriée pour le spécialiseur Tempo, mais devra en général être adaptée à la précision de l'analyse de temps de liaison du spécialiseur cible. Puis nous décrivons la mise en œuvre de notre stratégie pour un petit langage impératif. Enfin, nous argumentons que la spécialisation basée sur les analyses modifiées respecte à la fois le critère de correction standard d'un spécialiseur et les déclarations faites par le programmeur.

6.1 Stratégie de Vérification

Le degré de spécialisation est déterminé par les temps de liaison des références aux emplacements mémoire, car ces références correspondent aux points de programme à travers lesquels les valeurs statiques sont propagées. Ainsi, l'analyse de chaque référence de variable ou de chaque expression d'indirection vérifie que les informations de temps de liaison calculées correspondent aux contraintes des emplacements référencés. Dans d'autres cas, nous adoptons une stratégie de vérification plus paresseuse, en permettant à un temps de liaison calculé pour un emplacement mémoire d'être en conflit avec sa contrainte, mais seulement aux points de programme où la valeur de l'emplacement mémoire n'est pas directement utilisée. Cette stratégie paresseuse augmente l'ensemble des programmes qui peuvent être acceptés, contrairement à une stratégie qui effectuerait une vérification complète de tous les effets implicites, et garantit que les messages d'erreur signalant des incompatibilités de temps de liaison font seulement référence à

des constructions de programme explicites.

Dans un souci de concision, les contraintes de temps de liaison sont, dans tous les exemples de ce chapitre, représentées par des annotations directement associées aux types du programme source, plutôt que dans un module de spécialisation séparé. Les annotations possibles sont $\mathcal{S}$ pour statique, $\mathcal{D}$ pour dynamique, et $\mathcal{U}$ pour préciser qu'il n'y a pas de contrainte spécifiée. Comme dans le chapitre 2, les termes statiques sont soulignés.

Explications à travers d'exemples

Les temps de liaison d'emplacements mémoire sont modifiés ou consultés lors de l'analyse des appels de fonction, des affectations, des conditionnelles, des références de variables et des expressions d'indirection. Le traitement par analyse de temps de liaison de ces constructions interagit avec le processus de vérification de la façon suivante :

Appels de fonction : Un scénario de spécialisation associé à une fonction décrit le temps de liaison de chacun des niveaux d'indirection des différents arguments. Tous ces temps de liaison sont vérifiés aux points d'appel. Ainsi, les annotations de la fonction `f`, définie ci-dessous, précisent que les deux arguments `x` et `y` sont des pointeurs statiques et que `x` pointe sur une valeur dynamique tandis que `y` fait référence à une valeur statique :

```
void f(int^D *^S x, int^S *^S y) {
  ...
}
```

Supposons que `f` soit appelée dans le contexte suivant :

```
...
int^U a = 3;
f (&a, &a);
...
```

Dans ce cas précis, les deux arguments de `f` sont `&a` et correspondent à un pointeur statique qui fait référence à une valeur statique.[1] Le deuxième paramètre de l'appel de fonction à `f` satisfait donc la contrainte du deuxième argument de `f`. De plus, comme une valeur statique peut être contrainte à être dynamique, le temps de liaison de `&a` est déterminé comme compatible avec la contrainte de `x` et son temps de liaison est modifié en `int`$^{\mathcal{D}}$ `*`$^{\mathcal{S}}$.

Affectations : Pour garantir que la propagation des informations à travers le programme est en accord avec l'intention du programmeur, l'analyse vérifie que le temps de liaison de l'expression à droite de l'affectation est compatible avec les contraintes des emplacements mémoire qui peuvent être modifiés. Cependant, en accord avec notre stratégie paresseuse, l'analyse ne vérifie pas la compatibilité des contraintes et des temps de liaison à chaque niveau d'indirection des alias implicitement modifiés par l'affectation.

[1] L'adresse d'une variable est toujours considérée comme statique.

Nous illustrons cette vérification paresseuse avec l'affectation `y = &d` dans l'exemple suivant :

```
int^D d;

void f(int^S *^S y) {
y = &d;
...
}
```

Le temps de liaison statique de `&d` satisfait la déclaration associée à `y` qui est $\texttt{int}^{\mathcal{S}}\ *^{\mathcal{S}}$. La stratégie paresseuse utilisée lors de la vérification d'une affectation a pour conséquence que la non-concordance entre le temps de liaison dynamique de `d` et la contrainte statique de `*y` n'est pas prise en compte à ce moment, car le temps de liaison de `*y` n'a aucune incidence sur la spécialisation de l'affectation.

Conditionnelles dynamiques : Une approche couramment utilisée lors du traitement d'une affectation statique située dans une conditionnelle ou dans une boucle dynamique, consiste à considérer tous les emplacements mémoire qui sont susceptibles d'être modifiés comme étant dynamiques après la conditionnelle ou la boucle. C'est pourquoi nous avons choisi de ne pas vérifier le temps de liaison calculé par l'analyse jusqu'à ce que l'emplacement mémoire auquel il est associé soit réellement accédé.

Dans l'exemple qui suit, le paramètre `x` est affecté avec une valeur `NULL` à l'intérieur d'une conditionnelle dynamique.

```
int^D d;

void f(int^D *^S x, int^S *^S y) {
  if (d) {
    x = NULL;
    ...          /* peut contenir des utilisations de x statiques */
  }
  x = y;
}
```

L'expression `NULL` est statique ce qui est en accord avec la contrainte de `x`, cependant après la conditionnelle, `x` est considéré dynamique. Cette modification de temps de liaison n'est pas vérifiée. Pour restaurer le temps de liaison correct de `x` il est nécessaire de le faire explicitement en affectant `y` à `x`. À défaut, une erreur sera signalée à la prochaine utilisation de `x`.

Références de variable ou expressions d'indirection : Pour une référence de variable ou une expression d'indirection, nous vérifions que la contrainte sur l'emplacement mémoire est compatible avec le temps de liaison déterminé par l'analyse, mais nous ne vérifions pas qu'il y a compatibilité pour toutes les indirections de l'emplacement. Cette approche paresseuse est illustrée par la référence faite sur `y` dans l'exemple suivant :

```
int^D d;

void f(int^D *^S x, int^S *^S y) {
  y = &d;
  x = y;
}
```

Bien que l'affectation de `y = &d` ait pour effet de produire pour `*y` un temps de liaison en conflit avec sa contrainte, cette contrainte n'est pas vérifiée dans la deuxième affectation, car elle fait seulement référence à la valeur de `y`.

Une indirection qui implique un alias créé par la mise en relation entre les paramètres formels et les paramètres effectifs doit respecter non seulement les contraintes de l'emplacement mémoire référencé mais aussi les contraintes des paramètres. Considérons l'exemple qui suit pour illustrer l'impact des contraintes des alias et des paramètres de fonctions :

```
int^D d;

int f(int^S *^S y) {
  int^U *^U *^U z = &y;
  y = &d;
  return *(*z^{y})^{d};
}
```

Dans cet exemple, l'indirection `**z` présente dans l'instruction de retour respecte les contraintes de `z` et de l'emplacement référencé `d`. Ces deux contraintes autorisent une valeur dynamique mais comme la relation entre `z` et `d` est due au paramètre `y` qui a pour contrainte de faire une indirection sur une valeur statique, une erreur est signalée.

6.2 Mise en œuvre pour un petit langage impératif

Nous montrons la possibilité de mettre en œuvre notre approche en définissant un spécialiseur de programmes hors ligne pour un petit langage impératif. Ce spécialiseur est sensible au flot et traite précisément les pointeurs au moyen d'une analyse d'alias. Les boucles et la récursion ne sont pas traitées mais peuvent être rajoutées en utilisant des techniques standards [AC94], comme nous l'avons d'ailleurs fait dans notre implémentation.

Nous définissons un spécialiseur de programmes qui repose sur trois analyses : une analyse d'alias, une analyse de temps de liaison, et une analyse de temps d'évaluation. L'analyse d'alias associe chaque expression d'indirection $*E$ avec l'ensemble des *emplacements mémoire abstraits* (les variables du programme ou les indirections des variables du programme) auxquels l'expression peut faire référence. N'importe quel algorithme d'analyse standard peut être utilisé. Aucune vérification n'est effectuée pendant l'analyse d'alias, mais les informations d'alias recueillies sont cruciales afin de transmettre

les informations de temps de liaison et les contraintes à travers les pointeurs. L'analyse de temps de liaison permet de classer chaque expression comme étant soit statique, c'est-à-dire que la valeur de l'expression dépend uniquement d'informations connues au moment de la spécialisation de programme, ou soit dynamique, c'est-à-dire que la valeur de l'expression dépend d'informations non accessibles avant l'exécution du programme. En plus de calculer les temps de liaison des différentes expressions, cette phase vérifie que chaque emplacement qui possède une contrainte statique a effectivement un temps de liaison calculé qui est statique. De plus, l'analyse contraint chaque emplacement déclaré comme dynamique a être dynamique, même s'il est possible de le considérer comme statique. Cette stratégie est primordiale et permet au processus de spécialisation d'être prévisible par rapport aux contraintes déclarées dans les modules de spécialisation. L'analyse de temps d'évaluation, quant à elle, s'assure de la cohérence du programme spécialisé. Elle adresse le problème lié à la spécialisation d'une expression statique pour laquelle la valeur au moment de la spécialisation n'est pas significative à l'exécution.

Nous présentons maintenant en détail la syntaxe du langage traité, ainsi que les analyses qui ont un impact sur les annotations de temps de liaison, c'est-à-dire les analyses de temps de liaison et de temps d'évaluation.

6.2.1 Syntaxe du langage

Le langage traité est un langage impératif simplifié non récursif qui inclut des fonctions à un seul argument, des affectations, des conditionnelles et des expressions d'indirection. Un programme est composé d'une collection de variables globales et de fonctions définies comme suit :

$$
\begin{array}{rrrl}
G & \in & \text{Globale} & ::= T\ \texttt{x} \\
F & \in & \text{Fonction} & ::= \texttt{f(}T\ \texttt{x)}I \\
T & \in & \text{Type} & ::= \texttt{int} \mid T\ \texttt{*} \\
I & \in & \text{Instruction} & ::= L\texttt{ = }E\texttt{;} \mid \texttt{\{}T\ \texttt{x;}\ I\texttt{\}} \mid \texttt{\{}I_1\ I_2\texttt{\}} \\
 & & & \quad \mid \texttt{if (}E\texttt{) then }I_1\texttt{ else }I_2 \mid \texttt{f(}E\texttt{)} \\
E & \in & \text{Expression} & ::= \texttt{x} \mid \texttt{\&x} \mid \texttt{*}E \\
L & \in & \text{L-expression} & ::= \texttt{x} \mid \texttt{*}E
\end{array}
$$

Le point d'entrée d'un programme est une instruction `I` qui invoque une des fonctions définies. La sémantique du langage est standard.

6.2.2 Analyse de temps de liaison

L'analyse de temps de liaison détermine les temps de liaison de chaque construction de programme à partir des temps de liaison calculés pour chaque emplacement mémoire abstrait et des contraintes déclarées par le programmeur dans les modules de spécialisation. Un temps de liaison est dénoté comme étant soit S (statique), soit D (dynamique), tel que $\mathsf{S} \sqsubseteq \mathsf{D}$. L'opération de majorant $b \sqcup b'$ est commutative et calcule un nouveau temps de liaison tel que :

$$\mathsf{S} \sqcup \mathsf{S} = \mathsf{S} \qquad \mathsf{D} \sqcup b = \mathsf{D}$$

Une contrainte déclarée par le programmeur est dénotée comme étant soit $\mathcal{S}$, indiquant ainsi que l'emplacement mémoire doit être considéré statique lorsqu'il est référencé, soit $\mathcal{D}$, indiquant alors que l'emplacement doit être considéré dynamique lorsqu'il est référencé, soit $\mathcal{U}$, indiquant enfin qu'il n'y a aucune contrainte associée à cet emplacement mémoire. Les contraintes de temps de liaison sont ordonnées telles que $\mathcal{U} \sqsubseteq \mathcal{S} \sqsubseteq \mathcal{D}$. Nous définissions un opérateur de majorant stricte $\uplus$ tel que pour toute contrainte c :

$$c \uplus c = c \qquad \mathcal{U} \uplus \mathcal{S} = \mathcal{S} \qquad \mathcal{U} \uplus \mathcal{D} = \mathcal{D}$$

.

Nous définissons de plus une opération $c \oplus b$ qui produit un temps de liaison compatible avec la contrainte c et le temps de liaison b. Cette opération est définie comme suit :

$$\mathcal{S} \oplus \mathsf{S} = \mathsf{S} \qquad \mathcal{D} \oplus b = \mathsf{D} \qquad \mathcal{U} \oplus b = b$$

Nous remarquons que $\uplus$ et $\oplus$ ne sont pas définis pour toutes les paires, car certaines paires telles que $\mathcal{S} \oplus \mathsf{D}$ et $\mathcal{S} \oplus \mathsf{D}$ ne se sont pas compatibles. Ces opérations peuvent donc être toutes les deux utilisées pour calculer un résultat mais aussi pour vérifier la compatibilité.

Pour effectuer nos vérifications nous définissons trois environnements : Σ fait correspondre chaque emplacement mémoire à sa contrainte, Γ fait correspondre chaque emplacement mémoire à son temps de liaison et enfin Φ fait correspondre chaque nom de fonction à sa définition qui est de la forme $(\mathtt{x}, [c, c_1, \ldots, c_m], I)$ où $\mathtt{x}$ est l'argument de la fonction, $[c, c_1, \ldots, c_m]$ sont les contraintes de temps de liaison de chaque niveau d'indirection de l'argument $\mathtt{x}$ et I est le corps de la fonction.

L'annotation correcte d'une instruction I est spécifiée par le jugement $\Sigma, \Gamma, \Phi \vdash_{\mathrm{i}} I : \hat{I}, \Gamma'$, où $\hat{I}$ est l'instruction I annotée de temps de liaison, et Γ' est un nouvel environnement de temps de liaison qui reflète les affectations traitées pendant l'analyse de I. L'annotation correcte d'une expression E est spécifiée par le jugement $\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b$, où b est un temps de liaison et $\hat{E}^b$ est une expression annotée de temps de liaison. Enfin, l'annotation correcte d'une L-expression L est spécifiée par le jugement $\Sigma, \Gamma \vdash_{\mathrm{l}} L : \hat{L}^b$, où b est un temps de liaison et $\hat{L}^b$ est une L-expression annotée de temps de liaison.

La plupart des règles bien formées correspondent aux règles usuelles d'une analyse de temps de liaison sensible au flot appliquée à un langage impératif [HN00]. Nous ne présentons donc ici que les règles dans lesquelles les contraintes spécifiées dans les modules de spécialisation sont impliquées.

Variable de référence : Le temps de liaison d'une référence de variable $\mathtt{x}$ est calculée à partir de la contrainte $\Sigma(\mathtt{x})$ et du temps de liaison courant $\Gamma(\mathtt{x})$ tel que :

$$\Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{x} : \mathtt{x}^{\Sigma(\mathtt{x}) \oplus \Gamma(\mathtt{x})}$$

```
int^S *^S c;
int^S a;
void h() {
  a = 4;
  f(&a);
}
```

```
void f(int^S *^S x) {
  int^U b = 3;
   c = x;
  ... g(&b) ...;
}
```

```
int g(int^D *^D z) {
  int^U *^U d;
  if(...) d = z;
  else d = c;
  return *d;
}
```

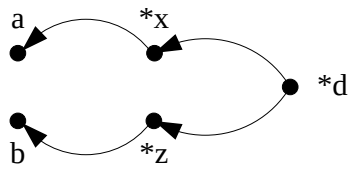


FIG. 6.1 – Alias initiaux et intermédiaires

Expression d'indirection : Le temps de liaison d'une expression d'indirection $*E$ prend en compte le temps de liaison b de l'expression E, ainsi que les temps de liaison et les contraintes associés aux alias potentiels de $*E$, c'est-à-dire des emplacements mémoire qui peuvent être référencés par $*E$. L'analyse de temps de liaison doit de plus prendre en considération les contraintes des paramètres de fonction à travers lesquels ces emplacements mémoire sont passés par référence.

L'analyse distingue différents types d'alias que nous présentons avec l'exemple de la Figure 6.1. Dans la définition de `h`, l'appel `f(&a)` lie le paramètre `x` à l'adresse de la variable globale `a`. Nous disons donc que `a` est un alias de `*x`. Comme `a` est l'emplacement mémoire qui contient la valeur, nous qualifions `a` d'*alias initial*. Le temps de liaison de `*x` doit respecter les contraintes de l'alias initial `a`. Cependant un principe de notre approche est que toutes les contraintes sont vérifiées aux appels de fonction, afin de garantir que les informations sont propagées à travers le programme comme précisé par le programmeur. Pour ce faire, nous tenons compte de la contrainte de `*x` en plus. Comme aucun nouveau emplacement mémoire n'est alloué pour l'indirection d'un paramètre, l'alias `*x` est qualifié d'*alias intermédiaire*. Dans le corps de la fonction `f`, il y a une affectation `c = x`. Après cette affectation, `*c` possède deux alias : l'alias initial `a` et l'alias intermédiaire `*x`.

L'exemple de la Figure 6.1 montre de plus d'une expression d'indirection peut avoir plusieurs alias initiaux et intermédiaires. Considérons la fonction `g`. Cette fonction affecte au pointeur sur entier `d` avec soit le paramètre `z`, soit la variable globale `c`. L'appel à la fonction `g` dans la définition de `f` montre que `*z` a pour alias initial la variable locale `b` de la fonction `f`. En conséquence, l'affectation `d = z` implique que `*d` a pour alias initial `b` et pour alias intermédiaire `*z`. Suite à l'analyse de la valeur de `c`, l'affection `d = c` implique que `*d` a pour alias initial `a` et pour alias intermédiaire `*x`. Comme l'analyse d'alias ne peut pas déterminer quelle branche de la conditionnelle est choisie, la référence à `*d` à la fin de `g` possède deux alias initiaux `a` et `b` et deux alias intermédiaires `*x` et `*z`.

Maintenant que nous avons déterminé les différents types d'alias qui sont associés

à une expression d'indirection, nous présentons comment ces informations sont prises en compte pour déterminer le temps de liaison de l'expression. Les alias initiaux correspondent à des emplacements mémoires concrets contenant les valeurs possibles auxquelles on accède. En conséquence, nous imposons que toutes les contraintes des alias initiaux soient identiques ou $\mathcal{U}$, et nous combinons ces contraintes avec l'opérateur stricte $\uplus$. Les alias intermédiaires reflètent simplement le passage d'un paramètre. Nous les combinons avec l'opération de majorant $\sqcup$, ce qui implique qu'une contrainte dynamique est obtenue si une seule des contraintes est dynamique. Les contraintes des alias initiaux sont combinées avec les contraintes des alias intermédiaires à l'aide de l'opérateur stricte $\uplus$, afin de garantir que les contraintes des alias initiaux sont toujours respectées. De plus, ces contraintes doivent être compatible avec le temps de liaison calculé pour l'expression d'indirection qui prend en considération le temps de liaison des alias initiaux. La règle pour une expression d'indirection est la suivante où l'opérateur $\mathcal{L}(*E)$ permet d'accéder aux alias initiaux κ et intermédiaires δ de l'expression $*E$:

$$\frac{\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^{b} \qquad \mathcal{L}(*E) = (\kappa, \delta) \qquad b' = \bigsqcup\{\Gamma(l) \mid l \in \kappa\} \qquad c_{\mathrm{I}} = \biguplus\{\Sigma(l) \mid l \in \kappa\} \qquad c_{\mathrm{i}} = \bigsqcup\{\Sigma(p) \mid p \in \delta\}}{\Sigma, \Gamma \vdash_{\mathrm{e}} *E : (*\hat{E})^{(c_{\mathrm{I}} \uplus c_{\mathrm{i}}) \oplus (b \sqcup b')}}$$

Instruction d'affection : Le temps de liaison b d'une instruction d'affection est le majorant des temps de liaison b' et b'' déterminés pour les sous-expressions. La compatibilité de ce temps de liaison avec les contraintes des alias initiaux κ et intermédiaires δ est vérifiée. S'il n'y a qu'un seul alias initial l, le nouveau temps de liaison de cet emplacement est le temps de liaison de l'affectation, comme le montre la règle suivante :

$$\frac{\Sigma, \Gamma \vdash_{\mathrm{l}} L : \hat{L}^{b'} \qquad \Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^{b''} \qquad b = b' \sqcup b'' \qquad \mathcal{L}(L) = (\kappa, \delta) \qquad \kappa = \{l\} \qquad \Sigma(l) \oplus b \qquad c_{\mathrm{i}} = \biguplus\{\Sigma(p) \mid p \in \delta\} \qquad c_{\mathrm{i}} \oplus b}{\Sigma, \Gamma, \Phi \vdash_{\mathrm{i}} L = E\,; : \hat{L}^{b'} =^{b} \hat{E}^{b''}\,;, \Gamma[l \mapsto b]}$$

S'il y a plusieurs alias initiaux, alors le nouveau temps de liaison de chaque emplacement potentiel est le majorant du temps de liaison de l'affectation et de l'ancien temps de liaison de l'emplacement.

Appel de fonction : Un scénario de spécialisation associé à une fonction décrit la contrainte de temps de liaison de l'argument, ainsi que les contraintes des différents niveaux d'indirection de l'argument selon son type. La règle de bonne annotation pour

un appel de fonction est la suivante :

$$\frac{\begin{array}{c}\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b \\ \mathtt{f} \mapsto (\mathtt{x}, [c, c_1, \ldots, c_m], I) \in \Phi \\ \mathcal{L}^*(E) = [(\kappa_1, \delta_1), \ldots, (\kappa_m, \delta_m)] \\ ((c_1 \uplus (\biguplus\{\Sigma(l) \mid l \in \kappa_1\})) \sqcup (\bigsqcup\{\Sigma(p) \mid p \in \delta_1\})) \oplus (\bigsqcup\{\Gamma(l) \mid l \in \kappa_1\}) \\ \vdots \\ ((c_m \uplus (\biguplus\{\Sigma(l) \mid l \in \kappa_m\})) \sqcup (\bigsqcup\{\Sigma(p) \mid p \in \delta_m\})) \oplus (\bigsqcup\{\Gamma(l) \mid l \in \kappa_m\}) \\ \Sigma' = \Sigma[\mathtt{x} \mapsto c, *^1\mathtt{x} \mapsto c_1, \ldots, *^m\mathtt{x} \mapsto c_m] \\ \Sigma', \Gamma[\mathtt{x} \mapsto c \oplus b], \Phi \vdash_{\mathrm{i}} I : I', \Gamma'[\mathtt{x} \mapsto b'_{\mathtt{x}}] \\ \mathsf{update\_fn}(\mathtt{f}, I')\end{array}}{\Sigma, \Gamma, \Phi \vdash_{\mathrm{i}} \mathtt{f}(E) : \mathtt{f}(\hat{E}^b), \Gamma'}$$

L'analyse d'un appel de fonction inclut trois étapes : l'analyse de l'argument (le jugement $\Sigma, \Gamma \vdash_{\mathrm{e}} E : \hat{E}^b$), la vérification que le temps de liaison de l'argument et les contraintes de chaque niveau d'indirection de l'argument correspondant (le groupe d'hypothèses du milieu), et finalement l'analyse du corps de la fonction appelée (les trois dernières lignes d'hypothèses).

La déclaration de la contrainte d'un paramètre spécifie un temps de liaison pour chacun des niveaux d'indirection m possibles, à partir de la liste $[c, c_1, \ldots, c_m]$ enregistrée dans l'environnement Φ. L'opération $\mathcal{L}^*(E)$ accède aux alias de E à chaque niveau d'indirection possible. À chaque niveau, la contrainte du paramètre est vérifiée d'être compatible avec les contraintes des alias initiaux et intermédiaires, et avec les temps de liaison des alias initiaux

Le corps I de la fonction appelée est analysé par rapport à l'environnement de contraintes courant Σ étendu avec les contraintes des indirections potentielles du paramètre ($*^i$ fait référence à i indirections de $\mathtt{x}$), et l'environnement de temps liaison courant Γ étendu avec le paramètre associé avec son temps de liaison. L'opérateur `update_fn` est alors utilisé pour mettre à jour une mémoire implicite de définitions de fonction annotée, avec le corps de fonction annoté $\hat{I}$. L'environnement de temps de liaison résultat Γ' reflète les effets de bord effectués par le corps de la fonction appelée sur des variables non locales.

6.2.3 Analyse de temps d'évaluation

Lorsque l'analyse de temps d'évaluation détermine qu'une affectation statique doit apparaître dans le programme spécialisé, l'instruction est ré-annotée comme étant à la fois statique et dynamique [HN00]. Cette ré-annotation n'interfère pas avec la propagation des informations statiques à travers le programme, et donc les contraintes de temps de liaison restent satisfaites.

La ré-annotation en dynamique d'une valeur statique non-réifiable qui apparaît dans

un contexte dynamique, peut restreindre la propagation des informations statiques pendant la spécialisation. Considérons l'exemple suivant :

```
int^D *^S x ;

if (x != NULL)
... *(x+1) ...
```

La contrainte sur `x` indique que le résultat de l'expression `*(x+1)` est dynamique. Donc, l'expression `(x+1)` qui correspond à un pointeur statique, apparaît dans un contexte dynamique et doit aussi être considéré comme dynamique. Cette ré-annotation implique à son tour que `x` soit considéré comme dynamique. Cela a pour effet d'empêcher l'addition d'être réalisée pendant la spécialisation et de créer une contradiction avec la contrainte $*^{\mathcal{S}}$ sur le type de `x`. Cependant, provoquer une erreur à cet endroit et donc obliger le programmeur de déclarer `x` comme étant complètement dynamique, empêcherait la simplification du test de la conditionnelle qui ne dépend pourtant que d'informations statiques. Donc une contrainte sur une variable qui possède une valeur non-réifiable est seulement garantie d'être satisfaite lorsque la variable apparaît dans un contexte statique. Cette propriété est garantie par les vérifications effectuées par l'analyse de temps de liaison et par le fait que l'analyse de temps d'évaluation ne ré-annote pas ce genre de référence.

6.3 Spécialisation

La phase de spécialisation simplifie les constructions statiques et reconstruit les constructions dynamiques pour former le programme spécialisé. Comme les déclarations de temps de liaison sont entièrement encodées dans le résultat des analyses, les modules de spécialisation ne sont pas directement référencés pendant la phase de spécialisation. Un spécialiseur standard peut donc être utilisé.

La spécialisation basée sur notre analyse de temps de liaison modifiée respecte à la fois la sémantique du programme source (le critère de correction standard d'un spécialiseur) et les déclarations faites dans les modules de spécialisation par le programmeur. Nous donnons ci-dessous quelques arguments qui indiquent comment ces règles de correction peuvent être montrées. De plus amples détails sont disponibles dans [LMLC02a].

6.3.1 Correction par rapport à la sémantique

Le critère standard de correction pour un spécialiseur est le suivant : si la spécialisation par rapport aux paramètres statiques réussit, l'exécution du programme spécialisé par rapport aux paramètres dynamiques doit produire le même résultat que l'exécution du programme original par rapport aux paramètres statiques et dynamiques.

Cette propriété est respectée par notre approche. En effet, l'ajout de vérifications de contraintes de temps de liaison à une analyse de temps de liaison résulte en une analyse plus conservatrice. Ainsi, les expressions qui pourraient être considérées statiques

sont considérées dynamiques lorsque les paramètres dont elles dépendent sont déclarés dynamiques. En combinant l'analyse de temps de liaison que nous avons proposée avec l'analyse de temps d'évaluation que nous avons présentée ci-dessus [HN00], nous pouvons montrer, en utilisant des techniques standards [JGS93], que notre approche respecte le critère de correction standard d'un spécialiseur.

6.3.2 Correction par rapport aux déclarations du programmeur

Intuitivement, une spécialisation qui respecte les déclarations du programmeur doit produire un programme spécialisé dans lequel toutes les variables déclarées statiques ont été éliminées. Cependant, comme il est précisé dans la section précédente, l'existence de valeurs non-réifiables (telles que les adresses qui n'ont aucune représentation possible dans le programme spécialisé) a pour conséquence qu'il n'est pas possible d'obtenir ce résultat. Nous proposons donc que notre stratégie de vérification permette que les variables déclarées statiques qui possèdent des valeurs non-réifiables apparaissent dans certains contextes dans le programme spécialisé.

Pour caractériser la relation entre les déclarations de temps de liaison et le traitement des valeurs non-réifiables, nous annotons la sémantique du langage présenté précédemment tel que chaque étape du traitement d'une expression est annotée avec le type de contexte dans lequel l'expression apparaît : contexte statique ou contexte dynamique. Un contexte statique est un contexte où une simplification doit être effectuée si le contexte correspond entièrement à une expression statique. Un contexte dynamique est quant à lui un contexte où aucune simplification ne peut être faite même si le contexte correspond à une expression statique. Pour respecter les déclarations de temps de liaison, la spécialisation doit produire un programme dans lequel un emplacement déclaré statique n'est jamais référencé dans un contexte statique. De plus, si un tel emplacement mémoire possède un type réifiable, il ne devrait jamais être référencé dans un contexte dynamique.

La sémantique se construit comme suit. Nous supposons une sémantique standard dans laquelle la sémantique d'une instruction est définie par le jugement $\rho, \Phi \models_s I : \rho'$, où ρ est une mémoire qui associe chaque emplacement mémoire avec sa valeur, Φ est un environnement qui fait correspondre chaque nom de fonction à sa définition, I est une instruction et ρ' est une nouvelle mémoire produite par l'exécution de l'instruction. La sémantique d'une expression est définie par le jugement $\rho \models_e E : v$, où ρ est une mémoire qui associe chaque emplacement mémoire avec sa valeur, E est une expression, et v est la valeur de l'expression. Dans la sémantique annotée, l'évaluation de chaque expression est annotée avec le contexte dans lequel elle apparaît. Plus précisément, la sémantique des expressions est donnée par un jugement de la forme $b, \rho \models_e E : v$, où b est soit S soit D. La règle pour une expression d'indirection est la suivante :

$$\frac{\mathsf{D}, \rho \models_{\mathrm{e}} E : l \qquad b, l \mapsto v \in \rho}{b, \rho \models_{\mathrm{e}} *E : v}$$

Indépendamment du contexte dans lequel une expression d'indirection se trouve, une expression d'indirection apparaît dans un contexte dynamique. De plus, les informations

de contexte sont propagées à la référence de l'emplacement mémoire dans la mémoire ρ ; ici le contexte de l'emplacement mémoire référencé l est celui correspondant à l'expression d'indirection toute entière. Ces annotations n'ont aucun effet sur la valeur d'une expression ; elles nous permettent simplement de se souvenir du contexte dans lequel chaque expression apparaît.

Le traitement des valeurs non-réifiables dans Tempo ré-annote seulement une expression statique possédant une valeur non-réifiable lorsque cette expression est dans un contexte dynamique [HN00]. En conséquence, le spécialiseur Tempo produit un programme résiduel dans lequel un emplacement mémoire statique n'est jamais référencé dans un contexte statique, et un emplacement mémoire statique ayant une valeur réifiable n'est jamais référencé dans un contexte dynamique.

6.4 Bilan

L'intérêt des vérifications est de rendre le processus de spécialisation prévisible par rapport aux déclarations. Notre stratégie a pour but de s'assurer de la complète propagation des valeurs déclarées statiques et du bon contexte de spécialisation des fonctions appelées.

Notre stratégie consiste à effectuer une comparaison stricte, c'est-à-dire à tous les niveaux d'indirection, des contraintes associées aux arguments d'une fonction avec les temps de liaison des paramètres utilisés à chaque appel de cette fonction. Nous avons adopté, dans les autres cas, une stratégie plus paresseuse qui consiste à ne vérifier que les contraintes des emplacements directement référencés. Cela permet aux vérifications d'accepter plus de programmes tout en garantissant une bonne propagation des valeurs statiques.

Les contraintes sont non seulement utilisées pour effectuer les vérifications de compatibilité, mais aussi pour calculer les nouveaux temps de liaison. Cela a pour effet, lorsqu'aucune incompatibilité n'a été détectée, de forcer un contexte de spécialisation à respecter les contraintes déclarées, typiquement de forcer un temps de liaison calculé statique à être dynamique.

Notre stratégie permet donc d'obtenir exactement la spécialisation désirée lorsque c'est possible, et de signaler une erreur dans le cas contraire.

Chapitre 7

Évaluation du langage de déclarations

Notre langage de déclarations a pour but de permettre à un plus grand nombre d'accéder aux fonctionnalités principales de la spécialisation de programmes et non pas de traiter des opportunités de spécialisations arbitrairement complexes. Ainsi notre approche permet au programmeur d'exprimer son intention de spécialisation et d'obtenir de la part de l'outil de spécialisation un retour sur la faisabilité de la spécialisation déclarée.

Dans ce chapitre, nous reprenons tout d'abord les exemples de sous et sur-spécialisation décrits dans le chapitre 3 et décrivons les bénéfices apportés par notre approche dans de telles situations. Puis nous faisons le point sur l'expressivité des déclarations. Enfin, nous présentons d'autres travaux existants qui permettent au programmeur de contrôler le processus de spécialisation.

7.1 Apports des déclarations

Les scénarios de spécialisation permettent d'exprimer des contraintes très fortes sur les paramètres. Ainsi un paramètre déclaré statique doit être statique quand il est utilisé dans le programme, et un paramètre déclaré dynamique doit être considéré comme dynamique par les analyses même s'il s'avère être statique à certains points du programme. Cela permet au programmeur d'obtenir exactement ce qu'il veut, si c'est possible, et rien d'autre, sinon. De plus, dans le cas d'un échec, les vérifications renseignent le programmeur en lui précisant quel paramètre est en conflit avec les déclarations. Nous étudions maintenant l'impact de l'utilisation de notre langage dans des situations de sous et sur-spécialisation.

7.1.1 Contrôle de la spécialisation

Grâce aux déclarations, il est possible de contrôler la spécialisation afin d'éviter une sur-spécialisation. Pour illustrer cette situation, reprenons l'exemple des fonctions

```
Module search {
  Defines {
    From search.c {
      Btfind::intern find(D(int *) tab,
                          D(int)   i,
                          S(int)   delta,
                          D(int)   x);
      Btsearch::intern binsearch(D(int *) t,
                                 S(int)   delta,
                                 D(int)   x)
                  { needs { Btfind; } };
    }
  }
  Exports { Btsearch; }
}
```

FIG. 7.1 – Module de spécialisation pour les fonctions `binsearch` et `find`

`find` et `binsearch` du chapitre 3. Nous avions vu que lorsque l'on spécialise `binsearch` pour `delta = 4` cela entraîne la spécialisation de la fonction `find` pour `delta = 4` et `i = 1` ce qui provoque une explosion de code. Pour éviter cette situation, nous avons besoin d'exprimer que seul l'argument `delta` de la fonction `find` doit être propagé comme statique. Avec les scénarios de spécialisation il est possible de déclarer une telle spécialisation comme le montre la Figure 7.1. Le scénario `Btfind` stipule que la fonction `find` se spécialise bien dans un contexte où `delta` est statique et les autres arguments dynamiques. Le scénario `Btsearch` associé à la fonction `binsearch` utilise le scénario `Btfind`, forçant ainsi l'appel à la fonction `find` à être spécialisé pour un contexte où seul `delta` est considéré statique, ce qui permet d'avoir un bon comportement de spécialisation.

7.1.2 Détection des incohérences

Les vérifications réalisées pendant l'analyse de temps de liaison permettent la détection de toute incohérence entre les contraintes déclarées et les temps de liaison calculés. Lorsqu'une incohérence est détectée cela signifie qu'un certain nombre de variables déclarées comme statiques ne peuvent pas être propagées comme statiques. Cette situation caractérise exactement un cas de sous-spécialisation. Même si notre approche ne permet pas de remédier directement à cette situation, elle apporte néanmoins deux choses : elle renseigne sur la région du code où le problème arrive et elle donne le nom du paramètre dont le temps de liaison est *le premier* en conflit.

Identification de la région de code du problème L'utilisation des modules de spécialisation permet d'isoler le problème à la granularité d'une fonction. Prenons le cas d'une fonction `f` qui fait appel à deux fonctions `g` et `h` :

```
int f(...) {
  ...
  g(...) ;
  h(...) ;
  ... ;
}
```

```
Module exemple {
  ...
  Btg1 :: intern g(...);
  Btg2 :: intern g(...);
  Bth  :: intern h(...);
  Btf  :: intern f(...)
            { needs { Btg1; Btgh;}};
...
}
```

Dans le scénario de `Btf`, les contrats établis par la déclaration `needs` stipulent que si les fonctions `g` et `h` sont appelées dans des contextes satisfaisant leurs scénarios alors toute la généricité présente dans `g` et `h` qui dépend de leurs paramètres statiques respectifs sera éliminée correctement. En conséquence, si un problème survient pendant la spécialisation de `f` alors le problème ne pourra se situer que dans le corps de la fonction `f` puisque le bon comportement de spécialisation des fonctions `g` et `h` a normalement été vérifié a priori.

Il est possible de restreindre davantage la zone de recherche d'erreur grâce aux contraintes mises sur les différents paramètres. Ainsi les utilisations de tous ces paramètres forment des points de vérifications, c'est-à-dire des *gardes*, qui peuvent s'avérer très utiles pour cerner l'endroit où il y a eu un changement de temps de liaison de statique à dynamique.

Identification de la première occurrence d'un paramètre en conflit Les fichiers colorés de Tempo permettent de visualiser les temps de liaison que l'analyse de temps de liaison a déterminés pour chaque construction (variable, expression, *etc.*), ce qui est d'une aide précieuse. Cependant, lorsque le résultat n'est pas celui escompté, il est souvent difficile de détecter l'origine du problème. En effet, lorsqu'un paramètre devient dynamique, la propagation de sa valeur peut provoquer le changement de temps de liaison des autres paramètres. Identifier le paramètre qui a tout déclencher est difficile, surtout lorsque ce paramètre se situe dans un cycle du graphe de contrôle comme par exemple dans une boucle. En effet, lors du traitement d'une boucle, l'analyse de temps de liaison propage itérativement les temps de liaison jusqu'à obtenir un point fixe, c'est-à-dire jusqu'à ce que les temps de liaison des différentes constructions contenues dans la boucle ne changent plus. Dans cette situation, déterminer a posteriori le contexte de temps de liaison dans lequel étaient les différentes constructions lorsque qu'un paramètre donné est devenu dynamique est délicat puisque seul le contexte de la dernière itération est visible.

Les vérifications rajoutées à l'analyse de temps de liaison par notre approche af-

```
struct bpf_insn {
  u_short code;
  u_char jt;
  u_char jf;
  int k;
};

int bpf_filter(struct bpf_insn *pc, u_char *p,
               u_int wirelen, u_int buflen) {
  int A;
  ...
  while(1) {
    switch(pc->code) {
      case ...
      case BPF_JMP|BPF_JGT|BPF_K :
          if(A > pc->k)
            pc += pc->jt;
          else
            pc += pc->jf;
          break;
      case ...
    }
    pc = pc + 1;
  }
}
```

FIG. 7.2 – Structure de l'interprète BPF

fichent, dans l'ordre que les constructions sont analysées[1], les expressions dans lesquelles un ou plusieurs paramètres utilisés sont en conflit avec leurs contraintes. Une fois le premier paramètre en conflit identifié, il faut comprendre pourquoi son temps de liaison a changé, et effectuer l'amélioration de temps de liaison appropriée pour résoudre le problème. Il faut alors relancer les analyses pour voir si les autres conflits étaient des conflits réels ou des seulement des conséquences du conflit en amont.

Comprendre le changement de temps de liaison Reprenons l'exemple de sous-spécialisation du chapitre 3, c'est-à-dire la spécialisation avec Tempo de l'interprète BPF (Figure 7.2). Suivant notre approche, il nous faut définir le module de spécialisation qui

[1] L'analyse de temps de liaison est faite en avant.

```
Module bpf {
  Defines {
    From bpf_filter.c {
      Btstruct::struct bpf_insn {
                     S(u_short) code;
                     S(u_char)  jt;
                     S(u_char)  jf;
                     S(int)     k;
                   };
      Btbpf::intern bpf_filter
                (Btstruct(struct bpf_insn) S(*) pc,
                 D(u_char *)                    p,
                 D(u_int)                       wirelen,
                 D(u_int)                       buflen);
    }
  }
  Exports{Btbpf;}
}
```

FIG. 7.3 – Module de spécialisation de l'interprète BPF

décrit les scénarios de spécialisation souhaités, un pour la structure `bpf_insn` et un autre pour la fonction `bpf_filter`. Le résultat de la spécification du module de spécialisation `bpf` est montré dans la Figure 7.3. Ce module spécifie que la fonction `bpf_filter` se spécialise lorsque `pc` est connu, c'est-à-dire que `pc` est un tableau d'adresse connue contenant des structures de type `bpf_insn` dont tous les champs sont connus.

Après la compilation des modules pour le scénario `Btbpf`, les analyses peuvent commencer. Dans notre cas, elles signalent que le temps de liaison de l'utilisation de `pc` est dynamique dans l'expression `pc = pc + 1` et qu'il est incompatible avec sa contrainte. Cependant l'expression ne renseigne pas directement sur les raisons du changement de temps de liaison de `pc`, elle correspond seulement à la première expression dans laquelle `pc` est utilisé avec un temps de liaison inapproprié. Reste à découvrir la raison en amont de cette modification de temps de liaison.

Nous savons que ce n'est pas au niveau du point de programme `pc = pc +1` que `pc` est devenu dynamique. Il n'a donc pu être modifié qu'au travers d'un de ses alias ou directement par une approximation faite par l'analyse de temps de liaison lors du traitement d'une structure de contrôle. Or `pc` n'a pas d'alias dans le programme, il faut donc chercher au niveau des structures de contrôle.

Pour mieux cerner le problème, nous utilisons le fait que l'analyse détecte les incompatibilités lors de l'utilisation d'une variable et nous ajoutons des utilisations factices

de `pc` après chaque conditionnelles. Cette mise en place de gardes de temps de liaison va permettre de détecter quelle conditionnelle provoque le changement de temps de liaison de `pc`. En relançant les analyses, une nouvelle erreur est signalée indiquant le point de programme juste après la conditionnelle :

```
if (A > pc->k) pc = pc + pc->jt; else pc = pc + pc->jf;
```

Cette conditionnelle est donc la source du problème. Le test dépend de `A` dont la valeur est inconnue. Si bien que même si les affectations de `pc` sont statiques dans les branches, l'analyse de temps de liaison considère `pc` dynamique après la conditionnelle. Il ne reste plus qu'à consulter les améliorations de temps de liaison existantes qui adressent cette situation particulière.

Dans notre exemple, les gardes qui permettent de détecter l'origine du problème sont rajoutées par le programmeur. Il serait cependant tout à fait envisageable d'offrir au programmeur le moyen de spécifier les paramètres à observer et d'effectuer alors une mise en place automatique des gardes appropriées. Une autre alternative serait d'étendre les analyses de façon à associer à chaque emplacement mémoire dynamique, la dernière expression qui a provoqué le changement de temps de liaison de statique à dynamique. Enfin, un environnement de mise au point, similaire à celui de `gdb` pourrait être construit permettant ainsi au programmeur de rajouter et d'enlever dynamiquement des gardes et par la même occasion d'observer le processus d'analyse de temps de liaison.

7.2 Expressivité du langage

Dans un souci de rendre la spécialisation de programmes accessible aux non-experts, les déclarations ont été conçues de façon à être les plus simples et les plus intuitives possibles. Le nombre d'abstractions qu'il est nécessaire de connaître est limité, ce qui permet de réduire le temps d'apprentissage du langage et incite les programmeurs à rester concentrés en priorité sur ce qui agit sur la spécialisation, c'est-à-dire sur les temps de liaison des paramètres manipulés. Ce qu'un programmeur peut exprimer avec notre langage est limité comparé par rapport à ce qu'il possible d'exprimer avec, par exemple, les directives de Tempo. Cependant nous avons trouvé que le niveau expressivité qu'offre notre langage est suffisant pour traiter de nombreux exemples de spécialisation standards.

Interface unique à la spécialisation de programmes C L'idée sous-jacente de notre approche est que notre langage devrait pourvoir être utilisé comme pré-processeur de n'importe quel spécialiseur pour le langage C. En effet, c'est le compilateur qui se charge de traduire les déclarations dans les abstractions propres à un spécialiseur donné. Cette traduction est cependant limitée par les abstractions offertes par le spécialiseur utilisé. Par exemple, il n'est pas possible de spécifier dans C-Mix qu'une structure de données est partiellement statique et donc tout scénario de spécialisation associé à une structure de données qui n'est pas complètement statique ou dynamique ne pourra être satisfait.

Le processus de spécialisation reste donc fortement lié aux fonctionnalités offertes

par le spécialiseur. Cependant, les déclarations permettent au programmeur de préciser d'une manière unique ce qu'il veut, sans se soucier des abstractions offertes par le spécialiseur. C'est ensuite le compilateur des déclarations et le spécialiseur qui renseignent le programmeur sur la faisabilité de satisfaire les déclarations.

Paramétrisation automatique du spécialiseur Dans notre approche, le compilateur des modules de spécialisation génère automatiquement les informations de configuration requises par le spécialiseur utilisé à partir des déclarations. Les déclarations renseignent principalement sur les contraintes de temps de liaison des différentes paramètres et ont un impact direct sur l'analyse de temps de liaison. Cependant, un spécialiseur possède d'autres analyses ou transformations automatiques qu'il faut contrôler, telles que les stratégies de déroulage de fonctions. Comme aucune information de ce type n'apparaît dans les déclarations de spécialisation, le compilateur génère des valeurs par défaut pour configurer ces paramètres de configuration.

Polyvariance des fonctions et des structures Les contraintes mises sur les paramètres et les appels de fonctions sont valides dans tout le corps de la fonction spécialisée. Ainsi, si une fonction `g` est appelée plusieurs fois, chaque appel est soumis à la même contrainte. Il peut cependant arriver que l'on ait besoin de déclarer une contrainte différente pour chaque appel présent dans le corps d'une fonction donnée, c'est-à-dire d'avoir une polyvariance des contextes d'appel d'une fonction.

Notre approche ne permet pas de traiter cette situation : les déclarations sont distinctes du code source et il est donc difficile d'associer une contrainte à un appel donné. Il est cependant possible de contourner ce problème en réalisant des copies de la fonction. Cette technique est d'ailleurs couramment appliquée lorsque l'analyse de programmes utilisée est monovariante par rapport aux appels de fonctions. Ainsi, il faut introduire dans le programme source autant de copies de la fonction qu'il y a de contextes d'appel que l'on souhaite distinguer. Les nouvelles fonctions sont ensuite invoquées à la place des appels de la fonction originale. Enfin il faut déclarer un scénario de spécialisation pour spécifier le contexte d'utilisation de chaque copie. Le problème est similaire avec les structures de données, c'est-à-dire que le langage ne permet qu'une utilisation monovariante d'une structure de données dans le corps d'une fonction. Ici aussi, créer des copies de la structure de donnée permet d'obtenir la polyvariance.

Faire de la copie de fonctions et de structures de données n'est envisageable que si le nombre de copies reste petit. Dans tous les exemples d'applications système que nous avons traiter jusqu'à présent, pas plus de 2 ou 3 copies étaient nécessaires.

Exemples traités Nous avons utilisé notre langage pour traiter des applications réalistes. Nous avons spécialisé, entre autres le RPC, l'interprète BPF, le `printf` de la bibliothèque C, la FFT, *etc.* Le degré d'expressivité du langage s'avère jusqu'à présent suffisant pour effectuer un bon nombre d'exemples.

Nous avons aussi testé notre langage lors de travaux pratiques dans le cadre d'un cours sur la spécialisation de programmes au niveau DEA. Les étudiants ont pu spécia-

liser en moins de deux heures plusieurs petits exemples standards alors que les étudiants des années précédentes arrivaient à peine à traiter tous les exemples en une scéance de quatre heures.

Notre langage est aussi en cours d'évaluation par Philips qui utilise notre prototype pour spécialiser le code de leurs décodeurs numériques (*set top boxes*). La prise en main du langage et du compilateur a donné lieu à un tutorial d'une heure à la fin duquel ils ont été capables d'écrire par eux-mêmes les modules de spécialisation du RPC et d'effectuer avec succès la spécialisation du code.

7.3 Travaux connexes

La difficulté à obtenir le programme spécialisé désiré à l'aide de techniques automatiques a donné lieu à elle seule à un ensemble d'approches qui permettent au programmeur de contrôler le processus de spécialisation. La plupart des travaux existants requiert d'annoter directement le programme source, en violant dans certains cas la syntaxe du langage de programmation. Certaines stratégies permettent une seule annotation possible par définition de fonction. Aucune des stratégies existantes, à l'exception des classes de spécialisation [VCMC97], n'offre la possibilité de structurer les déclarations de spécialisation.

DyC Le système de spécialisation à l'exécution DyC propose au programmeur des annotations qui permettent de contrôler les différents aspects du processus de spécialisation, comme par exemple la propagation des valeurs de spécialisation [GMP+00]. À partir des annotations qui identifient les variables statiques, DyC infère automatiquement la région de code à spécialiser. Bien que le langage des annotations permettent au programmeur de contrôler plusieurs aspects de la stratégie utilisée par ce processus d'inférence, aucun renseignement n'est fourni quant à la région de code qui est sélectionnée. En dehors du manque de précision inhérent à tous les moteurs d'analyse statique, les inférences faites sont rendues encore plus complexes par l'utilisation d'un compilateur optimisant (le Multiflow compiler) qui obscurcit la relation entre les annotations du programmeur et les transformations [GMP+00].

C-Mix L'évaluateur partiel de programmes C, C-Mix, offre des annotations qui contrôlent les temps de liaison des variables et des appels de fonctions [C-M00]. Les annotations peuvent être spécifiées dans le fichier source, un fichier de configuration ou directement en ligne de commande. Les variables peuvent être annotées comme étant statiques ou dynamiques. Si un temps de liaison dynamique est inféré pour une variable déclarée statique, une erreur est signalée. Si un temps de liaison statique est inféré pour une variable déclarée dynamique alors le temps inféré est rendu dynamique. Bien que nous utilisons la même stratégie de vérification, le fait que l'analyse de C-Mix soit insensible au flot fait que quelques programmes qui sont acceptés par notre vérification, sont rejetés par C-Mix. D'une manière générale, les annotations de C-Mix traitent globalement les variables, tandis que nos annotations sont plus fines, permettant de préciser les temps de liaison tous les types d'objet auxquels une variable peut faire référence. Par exemple, les annotations de C-Mix ne permettent pas d'exprimer que le

temps de liaison de l'indirection d'une variable et différent de celui de la variable.

Schism L'évaluateur partiel Schism traite un sous-ensemble du langage de programmation Scheme et offre un mécanisme de `filtre` qui permet au programmeur de spécifier quels arguments statiques doivent être propagés dans le corps de la fonction à spécialiser [Con93]. Comme les filtres utilisent la syntaxe de Scheme, ils peuvent être éliminés grâce à des macros afin de permettre l'exécution normale du programme source. Un filtre peut inclure des calculs sur les temps de liaison, permettant ainsi au programmeur de déclarer plusieurs stratégies pour une même fonction. Cependant, les informations de spécialisation sont distribuées à travers le code du programme.

Fabius Le système de génération de code à l'exécution Fabius traite le langage de programmation ML et requiert que le programmeur sépare les arguments statiques et dynamiques de chaque fonction par le mécanisme de curryfication, tel que le premier argument contienne les informations statiques et le deuxième les informations dynamiques [LL96]. Typiquement cette stratégie nécessite la modification du programme et ne permet la spécification que d'une seule stratégie par fonction. Cependant, comme les déclarations de temps de liaison n'utilisent aucune syntaxe particulière, le programme peut être directement normalement exécuté ou spécialisé.

Langages à plusieurs niveaux Les langages à plusieurs niveaux, comme MetaML [TS97] et ‘C [EHK96], offrent des notations haut-niveau pour décrire la construction de fragments de code. Bien que que de tels langages permettent la construction de code spécialisé qui satisfait les déclarations précises du programmeur, les annotations sont plus fastidieuses et difficiles à effectuer que les annotations des paramètres de fonctions proposées par notre approche. De plus, le programme source n'est pas écrit avec un langage standard et ne peut donc être traiter que par un outil spécifique.

Classes de spécialisation Les classes de spécialisation forment une approche déclarative qui permet de spécifier les opportunités de spécialisation présentes dans des programmes Java [VCMC97]. Les déclarations expriment l'intégration du code spécialisé dans le programme original. Une classe de spécialisation indique les noms des méthodes à partir desquelles la spécialisation doit commencer, identifie les arguments connus au moment de la spécialisation, et sélectionne une spécialisation à la compilation ou à l'exécution. Ces informations sont compilées en une description du contexte de spécialisation du point d'entrée et la définition originale de la fonction d'entrée est modifiée pour permettre le choix entre la version d'origine ou spécialisée de la fonction. Cependant, le programmeur ne peut pas guider la propagation des paramètres statiques et donc ne peut pas contrôler comment le processus de spécialisation est effectué.

Notre approche est complémentaire avec celle des classes de spécialisation dans la mesure où notre but est de permettre au programmeur de contrôler le processus de spécialisation. Comme une classe de spécialisation, un module de spécialisation permet au programmeur de décrire les propriétés de temps de liaison du point d'entrée. Nos déclarations concernant les temps de liaison des paramètres de fonction, des variables globales, et des structures de données permettent au programmeur de décrire plus en détail comment les informations dérivées des arguments statiques de la fonction d'entrée doivent se propager dans le programme.

Annotations dans d'autres types d'outils automatiques L'importance de

permettre au programmeur de diriger les techniques automatiques a été reconnue depuis longtemps dans des domaines tels que les systèmes d'aide à la démonstration [CAB+86], et commence aussi à être reconnue dans le domaine des analyses de programmes [Gro01, Ryd97].

7.4 Bilan

Notre langage offre un compromis entre accessibilité et précision de contrôle du processus de spécialisation. Il facilite l'accès à la technique de spécialisation mais ne permet pas l'utilisation de toute la puissance du moteur de spécialisation offert par l'outil. Cependant l'expressivité de notre langage s'est avérée satisfaisante pour traiter un bon nombre d'exemples de spécialisation. De plus, notre approche permet de détecter les problèmes responsables des comportements inappropriés du processus de spécialisation, ce qui est un avantage non négligeable comparé aux approches précédentes. Dans les conditions où l'utilisation de la technique de spécialisation de programmes est plus facile et que le processus de spécialisation est prévisible par rapport aux déclarations, il est envisageable d'utiliser la spécialisation de programmes d'une manière plus systématique pour la génération de code optimisé.

Troisième partie

Dimension génie logiciel

Chapitre 8

Dimension génie logiciel

Les deux premières parties de cette thèse étaient principalement axées sur le fonctionnement de la spécialisation de programmes. Nous avons tout d'abord décrit comment cette technique est mise en œuvre et ce qu'elle permet de faire. Puis nous avons présenté notre approche déclarative qui propose de prendre la spécialisibilité d'un programme en considération dès son implémentation et qui rend le processus de spécialisation prévisible par rapport aux déclarations. Nous regardons maintenant comment la spécialisation de programmes peut s'intégrer dans un contexte de génie logiciel.

La spécialisation de programmes fait partie d'un paradigme de programmation appelé programmation génératrice et qui est défini comme suit [CE00] :

> *La programmation génératrice est un paradigme de génie logiciel fondée sur la modélisation de famille de systèmes logiciels tel que, étant donnée une spécification de besoins, un produit intermédiaire ou final hautement personnalisé et optimisé peut être automatiquement produit à la demande, à partir de composants élémentaires et réutilisables, au moyen d'informations de configuration.*

Le principe de cette approche est donc de créer une famille de systèmes logiciels et de générer automatiquement chaque membre de cette famille à partir des informations de configuration qui décrivent les particularités du membre que l'on veut obtenir (voir Figure 8.1). Dans le contexte de la spécialisation de programmes, c'est le spécialiseur qui sert de générateur. Chaque *variant*, c'est-à-dire chaque membre de la famille, est donc obtenu par spécialisation du ou des programmes génériques correspondants à la famille de systèmes.

Dans ce chapitre, nous décrivons tout d'abord les étapes de création d'une famille de systèmes. Puis nous donnons un aperçu des différentes techniques qui peuvent être envisagées pour générer automatiquement les variants. Enfin nous expliquons comment notre approche déclarative s'intègre dans ce processus.

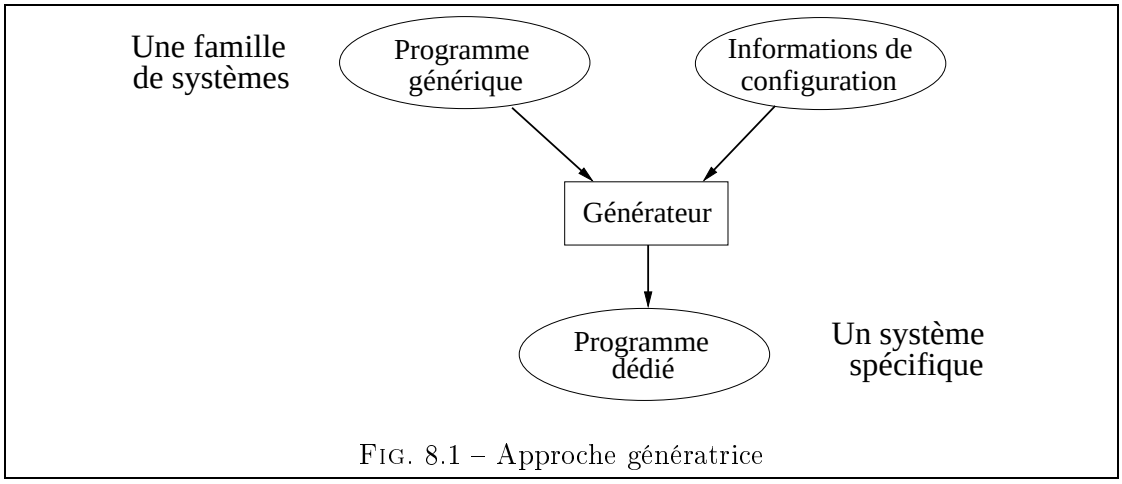


FIG. 8.1 – Approche génératrice

8.1 Familles de systèmes

Une famille de systèmes est un ensemble de systèmes qui possèdent des propriétés et fonctionnalités communes. L'idée des familles de systèmes n'est pas récente puisque Parnas introduisait déjà la notion de *familles de programmes* en 1976 [Par76]. Il définit alors une famille comme un ensemble de programmes qu'il est utile d'étudier en identifiant d'abord les propriétés communes des différents programmes puis en déterminant les particularités respectives de chaque programme. Récemment l'approche famille de systèmes a pris un nouvel essor sous l'appellation de *lignes de produits* [Bat98, Bos98, DS99, WL99]. Là aussi il s'agit d'identifier les points communs des différents produits et de construire une ligne de produits à partir d'une architecture logicielle qui identifie tous les composants nécessaires et permet la génération de chaque produit par configuration et assemblage les différents composants. Cette approche est motivée par son aspect pratique et économique par opposition à l'approche qui consiste à développer séparément tous les membres de la ligne de produits.

Une famille de systèmes est généralement créée selon un processus appelé *ingénierie de composants réutilisables*[1] ([SEIa], [CE00, chapitre 2]). Ce processus se décompose en trois phases. Tout d'abord, une *analyse de domaine* identifie les propriétés communes et les particularités distinctes de chaque membre de la famille. Puis la *conception de domaine* définit les composants de base nécessaires ainsi que leur organisation. Enfin, l'*implémentation de domaine* consiste à développer les composants. De nombreuses approches à la création de famille de systèmes ont été développées comme par exemples Draco [Nei89], FAST [Wei96], PuLSE [BFK+99], *etc.*

8.1.1 Analyse de domaine

Le but de l'analyse de domaine est de réunir toutes les informations qui sont pertinentes pour le domaine considéré et de les intégrer dans une modélisation. Elle consiste

[1] Le terme anglais est *domain engineering*.

donc à définir et à étudier le domaine et à en déterminer les éléments d'informations réutilisables. Les sources d'informations sont variées : systèmes existants, experts du domaines, prototypes, documentations, *etc.*

Il existe de nombreuses techniques d'analyse de domaine comme par exemples ODM (Organization Domain Modeling) [SCK+96] et FODA (Feature-Oriented Domain Analysis) [KCH+90]. Toutes ces techniques ont des approches différentes pour représenter un domaine. Elles peuvent décrire un domaine sous la forme d'objets, de caractéristiques ("features"), *etc.* De manière générale, chaque technique cherche à accroître la compréhension d'un domaine en capturant les informations dans des modèles formels.

La modélisation d'un domaine consiste à créer une représentation explicite des propriétés communes et variables des systèmes du domaine considéré, de la sémantique des propriétés et des concepts du domaine, et des dépendances entre les points variations. Les types de représentations sont variés : diagrammes objets, diagrammes d'interactions et d'états-transitions, diagrammes d'entités-relations et de flot de données, *etc.*

8.1.2 Conception de domaine

Le but de la conception de domaine est de développer une architecture logicielle pour la famille de systèmes. Il n'a pas de consensus pour définir une architecture logicielle, si bien qu'il en existe de nombreuses définitions [SEIb]. Un exemple de définition est donné par Garlan et Shaw pour qui une architecture logicielle décrit les éléments de bases des systèmes, les interactions entre ces éléments, les styles d'architecture qui guident leur organisation ainsi que les contraintes de ces styles [GS94]. En général, un système particulier est défini sous la forme d'une collection de composants et d'interactions entre ces composants. Une fois construit, un tel système peut alors être considéré à son tour comme un élément composite d'un système plus grand.

L'architecture logicielle d'un système peut être organisée selon différents styles. Par exemple, elle peut être en couches, c'est-à-dire sous forme hiérarchique, où chaque couche effectue une tâche particulière et correspond à un niveau d'abstraction donné. Une architecture peut aussi être formée de pipes et de filtres. Chaque filtre effectue un travail donné sur le flot de données à traiter et les pipes servent de connecteurs. Différentes combinaisons de ces filtres permettent de modifier le comportement du système global. D'autres styles existent : client-serveur, orienté objet, gestionnaire d'évènements, *etc* [GS94]. Il n'est pas rare qu'un système soit composé de sous-systèmes construits sur des styles différents.

8.1.3 Implémentation de domaine

Cette phase consiste à implémenter l'architecture et les différents composants de manière générique afin qu'ils couvrent l'ensemble des propriétés et particularités des systèmes cibles. Ce sont précisément les points de variations identifiés pendant la phase d'analyse qui définissent le degré de généricité de chaque composant. Il est aussi parfois possible de réutiliser des infrastructures ou des composants existants comme par exemple des composants sur étagère.

8.2 Génération des variants

Une fois la famille de systèmes construite, il est nécessaire de pouvoir produire à la demande un système de la famille donné. Cela consiste à assembler et à configurer les composants génériques. Cette étape peut se faire manuellement mais dans ce cas l'approche est fastidieuse, longue, ad hoc et propice aux erreurs pour les implémentations de grande échelle. Il est donc préférable d'automatiser le processus de configuration. Nous présentons maintenant quelques techniques de configuration et nous distinguons deux types de configuration : configuration fonctionnelle et configuration de code.

8.2.1 Configuration fonctionnelle

La configuration fonctionnelle a pour effet de restreindre le comportement d'un programme en figeant la valeur de certains paramètres. Pour illustrer cette situation, prenons par exemple le cas de la fonction `power` dont l'implémentation est montrée par la Figure 8.2. La fonction `power` possède deux paramètres `base` et `power` et élève `base` à la puissance `expon`. Effectuer une configuration fonctionnelle de `power` consiste, par exemple, à associer définitivement la valeur 3 au paramètre `expon`, restreignant la fonction `power` a ne calculer que des puissances de 3 de `base`. Intuitivement cela correspond à avoir la fonction `power` toujours appelée avec 3 comme deuxième paramètre :

```
int power(int base, int expon) {
  return power_bis(base, 3) ;
}

int power_bis(int base, int expon) {
  int accum = 1 ;
  while (expon > 0) {
    accum *= base ;
    expon- ; }
  return(accum) ;
}
```

Ce type de configuration est courant dans le cadre de la programmation orientée objet. En effet, si l'on considère un composant implémenté sous la forme d'une classe, sa configuration se fait à travers le mécanisme d'héritage et de liaison dynamique. C'est le cas notamment pour les JavaBeans [Sun].

8.2.2 Configuration de code

Contrairement à la configuration fonctionnelle qui n'est que superficielle, la configuration de code va plus loin puisqu'elle peut permettre, par exemple, de sélectionner des fragments de code en fonction des valeurs des paramètres de configuration ou d'effectuer des calculs qui dépendent de ces paramètres. Nous présentons maintenant quelques techniques qui rendent possible une configuration de code.

```
int power(int base, int expon) {
  int accum = 1;
  while (expon > 0) {
    accum *= base;
    expon-; }
  return(accum);
}
```

FIG. 8.2 – Implémentation de la fonction `power`

Directives #ifdef Les directives de pré-compilation #ifdef permettent la sélection de fragments de code. Les programmes, parsemés de ces directives, sont transformés par un pré-processeur avant la compilation. Cette technique est couramment utilisée dans les systèmes d'exploitation comme par exemple dans le système d'exploitation configurable eCos [Hat00].

Aspects Une autre approche consiste à utiliser les mécanismes offerts par la programmation par aspects [KLM+97] qui permettent d'insérer automatiquement des fragments de code à des points d'ancrage prédéfinis. Ainsi, par exemple, Coady *et al.* utilisent la programmation par aspect pour choisir entre différentes stratégies d'anticipation de lecture de données ("data prefetching") [CKFS01]. Notons que la programmation par aspects offre un avantage non-négligeable par rapport aux directives #ifdef puisqu'elle permet de développer séparément les différents fragments de code et de les intégrer dans le programme final automatiquement alors que dans le cas de l'utilisation des directives de pré-compilation, les différents fragments de code doivent être "entremêlés" par le programmeur.

Patrons C++ Une méthode bien connue pour permettre la configuration de code est d'utiliser les mécanismes des patrons du langage C++. À l'origine, les patrons C++ ont été créés pour faciliter la programmation générique. Toutefois, il s'avère qu'ils permettent aussi d'effectuer des calculs complexes lors de la compilation. Cette particularité a été mise à profit pour créer des bibliothèques scientifiques [SL98, Vel98] efficaces.

Pour illustrer cette technique, reprenons le cas de la fonction `power`. Supposons que la valeur de `expon` soit connue au moment de la configuration. Si cette valeur n'est pas trop grande, il est intéressant de la propager dans le code, ce qui aura pour effet de dérouler la boucle. Ainsi, si la valeur de `expon` est 3, on aimerait obtenir un code similaire à :

```
int power(int base) {
  return base*base*base;
}
```

En C++, si le programmeur décide qu'il est intéressant de dérouler la boucle dans certains cas, il doit écrire, en plus du code générique (semblable au code C de la Figure 8.2), une autre implémentation de la fonction `power`. Cette autre implémentation guide le compilateur afin que les transformations nécessaires soient effectuées. Dans notre cas, la solution est d'utiliser le compilateur pour qu'il déroule récursivement la fonction `power`. Voici l'implémentation qui permet d'obtenir ce comportement :

```
template<int expon>
inline int power(const int& base)
{ return power<expon-1>(base) * base; }

template<>
inline int power<1>(const int& base)
{ return base; }

template<>
inline int power<0>(const int& base)
{ return 1;}
```

Ainsi, à la compilation, l'appel `power<3>(base)` est successivement transformé par le compilateur comme suit :

```
power<3>(base)
power<2>(base) * base
power<1>(base) * base * base
base * base * base
```

Ce résultat correspond exactement à ce que l'on souhaitait. En fait, on peut obtenir grâce aux templates un résultat proche de celui qui peut être obtenu par spécialisation de programmes [Vel99].

Spécialisation de programmes Contrairement aux patrons C++, la spécialisation de programmes n'est pas utilisée dans le développement de logiciels. Cependant, nous avons vu qu'elle permet d'éliminer la généricité qui dépend des paramètres connus à la spécialisation. Cette technique est donc tout à fait appropriée pour configurer le code d'un composant logiciel. Elle offre en plus un avantage non négligeable par rapport aux techniques présentées précédemment : les transformations de code sont déterminées et effectuées par le spécialiseur tandis que pour les autres techniques le programmeur est impliqué dans la mise en œuvre des transformations.

Ainsi, comparons par exemple l'approche qui consiste à utiliser les patrons C++ avec celle qui repose sur la spécialisation de programme. Pour mettre en œuvre une stratégie

de configuration, le programmeur doit implémenter non seulement les fonctionnalités du composant mais aussi le code qui spécifie au compilateur *comment* le composant doit être configuré. Cette configuration est donc limitée par la volonté du programmeur de coder explicitement de telles transformations. Ainsi le programmeur doit non seulement implémenter la version générique d'un programme mais aussi toutes les versions qui peuvent être référencées à la compilation, comme nous l'avons montré précédemment.

Pour ne pas à avoir à implémenter dans les composants les transformations qui permettront la configuration, il serait souhaitable d'utiliser un générateur qui détermine et effectue automatiquement les transformations nécessaires à partir d'une spécification précisant les informations de configuration. Dans ces conditions, notre approche déclarative à la spécialisation de programmes apparaît tout à fait appropriée pour effectuer la configuration de composants génériques. Ainsi, si on reprend l'exemple de `power`, il suffit d'écrire dans un module de spécialisation le scénario suivant :

```
Btpower :: power(D(int) base, S(int) expon);
```

Les transformations nécessaires sont ensuite automatiquement déterminées et effectuées par le spécialiseur.

8.3 Création de composants configurables

Nous proposons d'utiliser notre approche déclarative à la spécialisation de programmes pour configurer les composants réutilisables d'une famille de systèmes [LMC00, LMC01, LMCE02]. Nous présentons tout d'abord les raisons qui font que notre approche à la spécialisation de programmes s'adapte tout particulièrement bien au développement de composants réutilisables. Puis nous décrivons comment notre approche s'intègre dans le processus de création de familles de systèmes.

8.3.1 Spécialisation et composants réutilisables

Notre approche propose de prendre en considération la spécialisabilité d'un programme dès son implémentation. Ceci est parfaitement en accord avec la démarche génie logicielle de création de famille de systèmes. En effet, l'identification des points de variations de la famille de systèmes, c'est-à-dire des paramètres qu'il faut faire varier pour obtenir les différents systèmes cibles, est faite très tôt dans le processus d'ingénierie de composants réutilisables. Les paramètres de configuration sont donc connus au moment de l'implémentation et peuvent être immédiatement déclarés comme étant des paramètres devant rester statiques pendant le processus de spécialisation.

À partir du moment où les paramètres statiques sont connus, le programmeur peut implémenter le code des composants génériques en faisant particulièrement attention à l'utilisation de ces paramètres. Notamment il doit bien séparer les calculs statiques des calculs dynamiques et s'assurer que les structures de contrôle qui permettent de choisir le comportement du composant dépendent d'informations complètement statiques. L'implémentation du composant doit s'accompagner de la création de modules de spécialisation dans lesquels le programmeur décrit les scénarios de spécialisation appropriés du composant. Cette action est primordiale pour pouvoir vérifier que les informations

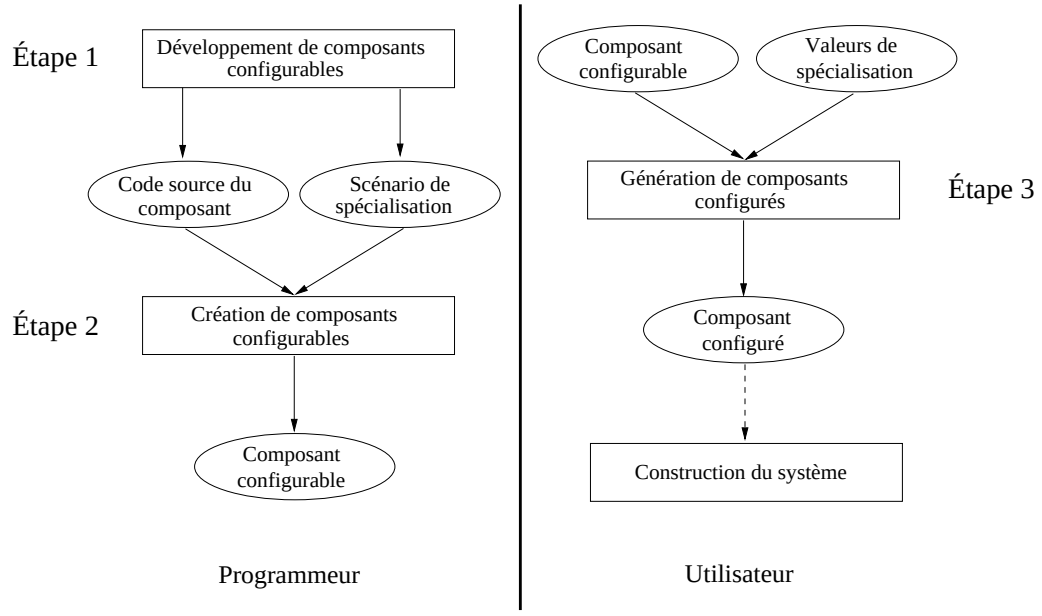


FIG. 8.3 – Utilisation de notre approche pour configurer un composant

de configuration sont correctement propagées dans le composant et qu'ainsi toute la généricité qui dépend de ces informations est bien éliminée.

Le code du composant et des modules associés peuvent être réutilisés pour le développement de composants plus complexes. Les modules de spécialisation offrent alors une documentation claire des `contrats` de configuration du composant sans avoir à étudier le code. Ces contrats impliquent que si le composant est utilisé dans un contexte qui satisfait les scénarios de spécialisation alors la configuration du composant par rapport aux paramètres de configuration identifiés est garantie.

8.3.2 Intégration de notre approche dans le processus de création de famille de systèmes

L'approche famille de systèmes comprend plusieurs phases : une phase de création de composants génériques réutilisables et une phase de configuration où les composants sont transformés au moment de leur déploiement dans le système cible. La phase de création est elle-même divisée en trois étapes : analyse, conception et implémentation. Notre approche a un impact sur l'étape d'implémentation et sur la phase de configuration.

Nous avons découpé notre processus de configuration en deux parties comme le montre la Figure 8.3. La première partie est réalisée par le programmeur du composant générique et la deuxième partie par l'utilisateur de ce composant.

La tâche du programmeur consiste tout d'abord à développer des composants configurables (étape 1), c'est-à-dire à implémenter le code des composants génériques et leurs modules de spécialisation associés, puis à les rendre configurables (étape 2). Cette

étape de création de composants configurables fait appel à l'outil de spécialisation. Il s'agit ici d'effectuer les analyses et de générer un spécialiseur dédié pour le composant considéré. Si les analyses ne produisent aucune erreur signalant une incompatibilité entre les contraintes déclarées et le résultat des analyses, alors on est garanti que les paramètres statiques seront bien propagés dans le composant et donc que le spécialiseur dédié permettra d'éliminer la généricité qui a été identifiée.

La tâche de l'utilisateur se résume à préciser les valeurs des paramètres de configuration qui satisfont les besoins du système cible et à exécuter le spécialiseur dédié pour effectuer la configuration (étape 3). Une fois configuré, le composant peut être inséré dans le système cible. L'utilisateur peut bien sûr générer autant de variants qu'il y a de contextes d'utilisation pour le composant.

Pour faciliter la tâche du programmeur et de l'utilisateur, nous avons développé un environnement graphique. Cet environnement graphique offre deux interfaces, une pour le programmeur et une pour l'utilisateur. La première interface assiste le programmeur dans la création de composants configurables et la deuxième permet à l'utilisateur de spécifier les valeurs des paramètres de configuration d'un composant configurable donné. Cet environnement est présenté dans le chapitre 10.

Nous illustrons dans le chapitre 9, la création d'un composant configurable pour le domaine des codes correcteurs d'erreurs.

8.4 Bilan

Dans un contexte de génie logiciel, la spécialisation de programmes peut être utilisée pour configurer les composants d'une famille de systèmes. Notre approche déclarative permet au programmeur d'identifier les paramètres de configuration en les déclarant statiques dans les modules de spécialisation associés aux différents composants. La phase de vérification de notre approche garantit que toute la généricité qui dépend des paramètres de configuration sera éliminée pendant le processus de spécialisation. Les modules de spécialisation forment une documentation claire des propriétés de configuration des composants et s'intègrent naturellement dans l'approche de création de composants réutilisables.

Chapitre 9

Exemple de création d'un composant logiciel configurable

9.1 Codes correcteurs d'erreurs

Nous illustrons notre approche par le développement d'encodeurs de données pour les Codes Correcteurs d'Erreurs (CCE) [LC83]. Les CCE permettent d'éviter la perte de données lors de communications numériques (par exemple par modems ou sans fil) en transmettant des informations redondantes. Comme les erreurs de transmission sont inévitables, la mise en œuvre des CCE constitue une opération critique : elle autorise un transfert de données maximal tout en maintenant une qualité de transmission acceptable.

Un encodeur CCE présente de nombreuses caractéristiques configurables. Tout d'abord, il existe de nombreux algorithmes d'encodage tels que le bit de parité, les codes cycliques, les codes en blocs linéaires, *etc.* De plus, la longueur d'une donnée à coder et la quantité d'information redondante à rajouter sont variables.

Du fait de la taille de l'espace de configuration que présente ce domaine d'applications, la génération d'encodeurs CCE nécessitent un procédé automatique. Nous avons utilisé notre approche pour développer une ligne de produits pour CCE. Les modules de spécialisation permettent de décrire la configurabilité durant le développement des composants. Les encodeurs CCE configurés sont automatiquement générés par spécialisation pour des spécifications de configuration données [LMCE02].

9.2 Analyse de domaine et de conception

Nous donnons un aperçu succinct de l'analyse de domaine et de conception des CCE et nous décrivons comment les informations de configuration seront propagées à travers l'architecture des CCE.

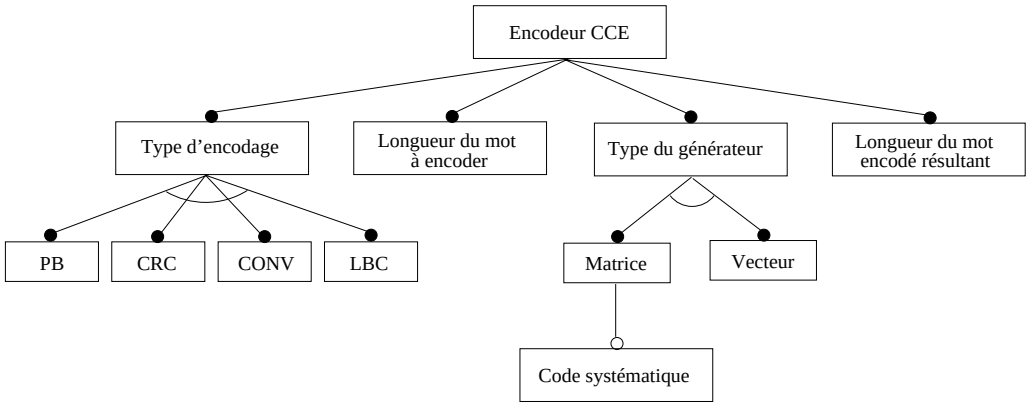


FIG. 9.1 – Diagramme des caractéristiques d'un encodeur

9.2.1 Analyse du domaine des CCE

Comme nous l'avons mentionné dans le chapitre précédent, une analyse de domaine a pour but d'identifier les propriétés communes et variables des applications du domaine étudié, ainsi que les dépendances entre ces points de variations. Pour faciliter la visualisation de ces informations nous utilisons l'approche FODA [KCH+90]. La modélisation des caractéristiques ("features") d'un domaine dans FODA offre un moyen simple de représenter l'étendue du domaine considéré et les différents points de variations de façon indépendante de l'implémentation.

Dans le modèle à caractéristiques, les propriétés importantes d'un concept sont représentées sous la forme d'une hiérarchie de noeuds appelée *diagramme des caractéristiques* ("feature diagrams"). Les noeuds correspondent aux points de variation et permettent donc de représenter clairement des propriétés réutilisables et configurables.

Le diagramme des caractéristiques correspondant à un encodeur CCE est montré dans la Figure 9.1. Un encodeur est caractérisé par le type de codage (par exemple : bit de parité, code convolutionnel), la taille des données à encoder, la taille des données une fois encodées et le type du générateur utilisé pour générer les informations redondantes (par exemple : un vecteur ou une matrice). Dans un diagramme FODA, les caractéristiques obligatoires sont indiquées par des arcs terminant par un cercle plein, tandis que les caractéristiques optionnelles sont représentées par des arcs terminant par un cercle vide (c'est notamment le cas pour les propriétés d'une matrice). Les caractéristiques alternatives sont indiquées par un arc de cercle placé en travers des arcs du diagramme. Par exemples, les algorithmes d'encodage alternatifs que nous avons considérés pour notre composant sont : le bit de parité (PB), les codes convolutionnels (CONV), les codes cycliques (CRC) et les codes en bloc linéaires (LBC).

Les contraintes peuvent être exprimées à coté du diagramme des caractéristiques pour préciser, par exemple, des combinaisons de caractéristiques illégales ou obligatoires. Dans le cas des CCE, une combinaison de caractéristiques obligatoire est illustrée par le fait qu'un encodeur de type LBC utilise une matrice comme type de générateur.

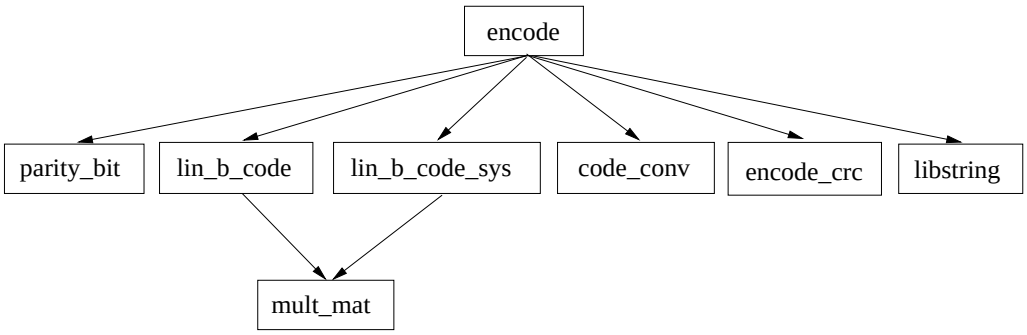


FIG. 9.2 – Architecture d'un encodeur

Les noeuds dans un diagramme des caractéristiques correspondent aux points de variations du système. La Figure 9.1 montre qu'un encodeur CCE possède quatre points de variations : le type d'encodage, la longueur du mot à encoder, le type du générateur de données utilisé et la taille du mot résultant une fois encodé.

9.2.2 Conception

Cette phase définit l'architecture de la ligne de produit. Il s'agit principalement d'identifier quels composants sont nécessaires et comment ces composants sont connectés. Nous avons choisi de développer notre ligne de produit sous la forme d'une architecture en couches.

Chaque composant de cette architecture est lui-même construit à partir de sous-composants. La Figure 9.2 montre l'ensemble des sous-composants du composant `encode`. Le composant `encode`, sommet de la hiérarchie, doit sélectionner l'algorithme d'encodage à utiliser en fonction du type d'encodage désiré et des propriétés du générateur. Les composants au deuxième niveau de la hiérarchie comportent les différents algorithmes d'encodage, à l'exception du composant `libstring` qui offre des fonctions de manipulation de chaînes de caractères. Chaque algorithme d'encodage calcule le mot encodé en fonction du mot à coder, de la longueur du mot à coder, de la longueur du mot encodé qui l'on veut obtenir et du générateur. Les composants `lin_b_code` et `lin_b_code_sys` font tous deux appel au composant `mult_mat`.

Le composant `encode` pourrait ensuite incorporé dans un système plus grand comme par exemple un système de traitement de flot de données où il ferait office de filtre.

9.3 Implémentation

En général, pour créer un composant générique, le programmeur introduit des paramètres et du code pour sélectionner les comportements appropriés. Cette paramétrisation se traduit aussi par la définition de structures de données et/ou de variables globales.

La généricité introduite sera plus tard éliminée au moment de la configuration selon des scénarios bien définis. À l'opposé des stratégies courantes qui consistent à exprimer en détail *comment* le composant doit être configuré, le programmeur peut utiliser notre langage déclaratif pour spécifier *quoi* configurer. La mise en œuvre de la configuration est gérée par le spécialiseur de programmes.

9.3.1 Modules de spécialisation

Parallèlement au développement d'un composant, le programmeur doit spécifier les scénarios de spécialisation de ce composant dans un module de spécialisation. Comme ces déclarations sont ensuite vérifiées afin de s'assurer de la bonne propagation des informations de configuration, le programmeur peut se permettre d'écrire du code générique sans se soucier du surcoût de la généricité de son implémentation.

Cependant, de la même façon qu'un programmeur doit raisonner sur les types des données contenues dans le code, il doit aussi raisonner sur les propriétés de spécialisation de ces données. Les points de variation qui ont été identifiés pendant la phase d'analyse de domaine sont des paramètres de configuration qui doivent permettre de sélectionner le comportement de l'encodeur. Le programmeur doit donc prêter une attention toute particulière à l'utilisation des ces paramètres et s'assurer notamment que les structures de contrôle qui guident le comportement de l'encodeur, dépendent bien seulement de données connues à la configuration.

Un scénario de spécialisation définit le contexte dans lequel la configuration du code du composant est garantie, c'est-à-dire que toute la généricité dépendante des paramètres de configuration peut être éliminée. De plus, comme les aspects de spécialisation du composant sont définis dans un fichier différent des fichiers sources, ils ne perturbent pas le code et peuvent être facilement modifiés. Un avantage évident d'un module de spécialisation est qu'il permet à un programmeur utilisateur d'un composant d'avoir une documentation claire des possibilités de configuration du composant sans avoir à étudier le code.

Nous présentons maintenant l'implémentation de deux sous-composants de l'encodeur CCE et décrivons comment les scénarios de spécialisation qui sont définis pour chaque composant, se combinent pour former un scénario de spécialisation global pour l'encodeur.

9.3.2 Déclarations intra-module

Les déclarations intra-module d'un module de spécialisation décrivent les scénarios de spécialisation associés aux paramètres de spécialisation (variables globales, paramètres de fonction et structures de données) à travers lesquels la généricité a été introduite dans un composant.

Considérons le composant `mult_mat` de la ligne de produits CCE. Ce composant inclut une procédure qui implémente la multiplication d'un vecteur de booléens `data` de taille `k` par une matrice `matrix` de taille `n`×`k` et stocke le résultat dans un vecteur `result` à partir de l'indice `ind`. L'implémentation de ce composant est la suivante :

```
void multmat(int *data, int k, int n, int **matrix, int *result, int ind) {
  int tmp, i, j;
  for(i = 0; i < n; i++) {
    tmp = 0;
    for(j = 0; j < k; j++)
      tmp = tmp ^ (data[j] & matrix[j][i]);
    result[ind + i] = tmp; }
}
```

Cette procédure présente plusieurs degrés de généricité. Tout d'abord, la taille du vecteur et celle de la matrice peuvent changer. De plus, les valeurs de la matrice ne sont pas figées. Enfin, le résultat de la multiplication peut être inséré à partir de n'importe quel indice du vecteur `result`. Les tailles du vecteur et de la matrice ainsi que les données de la matrice sont des paramètres de configuration du composant `encode` et seront donc connues au moment de la configuration. Il est possible de configurer d'avantage le composant `mult_mat` en précisant la valeur de `ind`. Spécialiser le composant pour un tel contexte et pour un ensemble de valeurs choisies arbitrairement, produit le résultat suivant :

```
void multmat(int *data, int *result) {
  result[0] = 0 ^ (data[0] & 1) ^ (data[1] & 0);
  result[1] = 0 ^ (data[0] & 0) ^ (data[1] & 1);
  result[2] = 0 ^ (data[0] & 0) ^ (data[1] & 1); }
```

Toute la généricité qui dépendait de `k`, `n`, `matrix` et `ind` a bien été éliminée. Un tel scénario de spécialisation est donc approprié pour permettre une bonne configuration du composant `mult_mat`. Pour spécifier ce scénario de spécialisation, le développeur du composant doit écrire le module de spécialisation suivant :

```
Module mult_mat {
   Defines {
     From multmat.c {
        Btmultmat :: intern multmat(D(int *) data, S(int) k,
                                    S(int) n, S(int) matrix S([][]),
                                    D(int *) result, S(int) ind);
     }}
   Exports {Btmultmat;}
}
```

Le module de spécialisation, `mult_mat`, définit un scénario de spécialisation, `Btmultmat`, qui spécifie que la fonction `multmat`, définie dans le fichier `multmat.c`, peut être spécialisée si tous ses arguments, sauf `data` et `result`, sont statiques. De plus, la déclaration inclut le mot clé `intern` qui indique que le code source de cette fonction est disponible et peut donc être transformé.

9.3.3 Déclarations inter-module

Comme nous l'avons décrit dans le chapitre 5, notre langage possède les mécanismes d'*import/export* comme la plupart des langages à modules. Lors de l'écriture

d'un module, il est possible d'importer des scénarios appartenant à d'autres modules et d'exporter des scénarios nouvellement définis pour les rendre accessibles aux autres modules. Ainsi les mécanismes d'import/export permettent de combiner les scénarios de spécialisation définis dans différents modules et de créer un scénario de spécialisation global à tout un système.

Nous illustrons ces mécanismes avec la création d'un composant qui implémente le code en blocs linéaires systématique. Ce composant est le suivant :

```
#include "multmat.h"
void syst_LBC(int *data, int k, int n, int **matrix, int *result) {
  int i ;
  for(i = 0 ; i < k ; i++)
    result[i] = data[i] ;
  multmat(data, k, n - k, matrix, result, k) ;
}
```

La fonction `syst_LBC` copie le vecteur d'entrée dans le vecteur de sortie et appelle la fonction `multmat` pour calculer les données redondantes et les ajouter à la fin du vecteur de sortie. Le scénario de spécialisation de la fonction `syst_LBC` est très similaire à celui déclaré pour la procédure `multmat`. La copie de vecteur effectuée par la fonction `syst_LBC` peut être déroulée. De plus, la fonction `multmat` pourra être spécialisée puisque son contexte d'appel est compatible avec le contexte décrit par son scénario. Le module de spécialisation pour l'encodage LBC est défini comme suit :

```
Module systLBC {
  Imports {
     From multmat.mdl {Btmultmat;}}
  Defines {
     From systLBC.c {
       BtsystLBC :: intern systLBC(D(int *) data, S(int) k, S(int) n,
                                   S(int) matrix S([][]), D(int *) result)
             {needs{Btmultmat;}};
     }}
  Exports {BtsystLBC;}
}
```

Dans tous les cas, notre processus de spécialisation vérifie que le contrat de spécialisation de la fonction `multmat` est respecté, c'est-à-dire que l'utilisation de la fonction `multmat` concorde bien avec son scénario de spécialisation. Cette vérification est primordiale pour garantir que chaque composant du système est configuré correctement.

9.4 Génération des variants

Une fois que le programmeur a terminé l'implémentation du code des différents composants et de leurs modules associés, il doit rendre l'encodeur configurable, c'est-à-dire qu'il doit créer un spécialiseur dédié pour ce composant. Cette étape est précédée de la phase d'analyse qui détermine comment spécialiser le code et qui vérifie que

les contraintes de spécialisation déclarées dans les modules peuvent être respectées. Si aucune erreur n'est détectée, alors toute la généricité qui dépend des paramètres de configuration est garantie de disparaître du composant au moment de la configuration. Pour assister le programmeur dans la génération de composants configurables, nous avons développé un environnement graphique que nous présentons dans le chapitre suivant.

Une fois le composant rendu configurable, il peut être mis à disposition de ses utilisateurs potentiels. Ce sont ces utilisateurs qui vont préciser les valeurs des paramètres de configurations afin d'obtenir l'encodeur qui répond à leur besoin. Pour notre exemple CCE, l'ensemble des valeurs de configuration d'un encodeur inclut le type d'encodeur, le générateur, la longueur du mot à coder et la longueur du mot codé souhaités.

9.5 Étude expérimentale

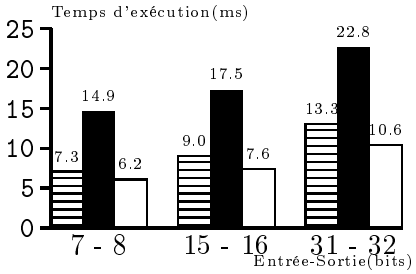
Nous avons créé un composant configurable pour la ligne de produits CCE à partir duquel des encodeurs spécifiques peuvent être générés. Notre implémentation couvre les codes suivants : bit de parité, codes convolutionnels, codes cycliques, et enfin codes en blocs systématiques et non systématiques. Nous avons comparé les performances des encodeurs obtenus par spécialisation avec celles de l'encodeur générique et celles d'encodeurs dédiés directement écrits à la main à partir d'algorithmes classiques.

L'expérience a consisté à mesurer le temps mis pour coder mille segments de données. Les tailles de données utilisées traduisent l'éventail d'utilisation de chaque encodeur et sont de ce fait très variées. L'expérience a été réalisée sous SunOS version 5.7, sur un Sun UltraSPARC 1 qui possédait une mémoire principale de 128 mb et un cache de données et d'instructions de 16 kb. Les résultats de ces mesures sont présentés dans la Figure 9.3

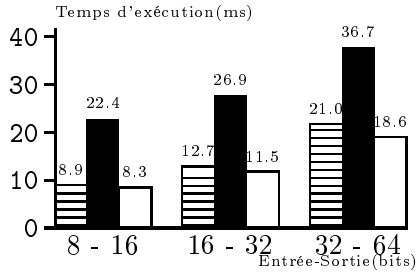
De façon générale, les encodeurs générés par spécialisation sont au moins aussi efficaces que ceux écrits à la main. L'encodeur spécialisé `LBC` est même 4 fois plus rapide que l'encodeur `LBC` développé manuellement (voir le graphe 9.3-d). Ce gain d'efficacité est souvent dû au dépliage de boucle et au pliage de constante qui sont longs et souvent fastidieux à effectuer à la main. Comme prévu, l'encodeur générique est le moins performant. Cependant la différence est négligeable entre l'encodeur générique et l'encodeur écrit à la main dans le cas de l'encodeur à code cyclique (graphe 9.3-f). Cela s'explique par le fait que l'inefficacité introduite par la généricité de l'encodeur générique est annulée par le coût des autres calculs qui sont effectués.

9.6 Bilan

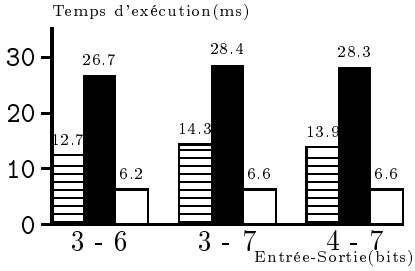
Outre les performances satisfaisantes des encodeurs spécialisés comparé à celles de l'encodeur générique, notre approche présente plusieurs avantages. Tout d'abord, les encodeurs peuvent être produits très rapidement une fois que l'encodeur générique est mis au point. De plus, les encodeurs spécialisés ne peuvent pas comporter d'erreur car la spécialisation conserve la sémantique des programmes. Donc, si l'encodeur générique est



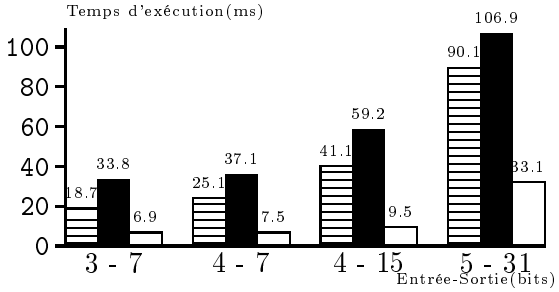
9.3-a) Bit de parité



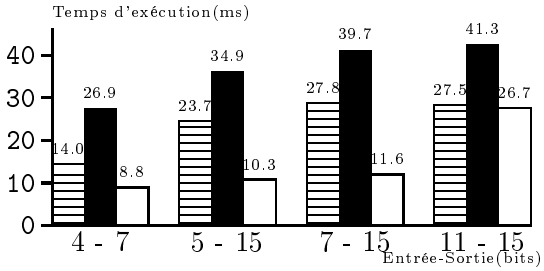
9.3-b) Code convolutionnel

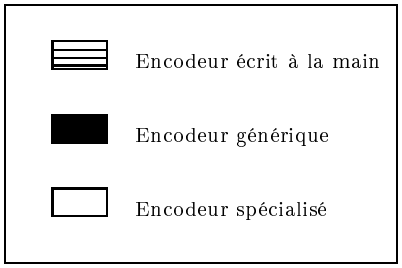


9.3-c) Code en bloc linéaires systématiques

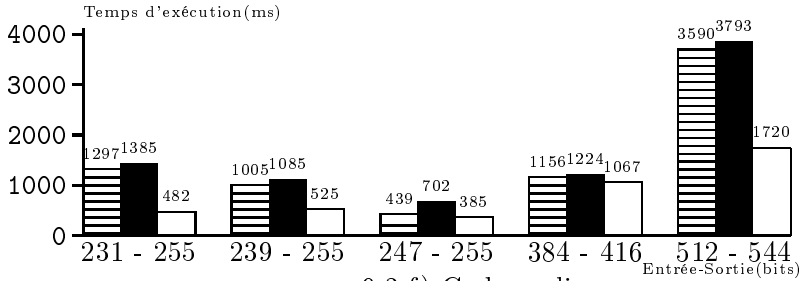


9.3-d) Code en bloc linéaires





9.3-e) Code cyclique



9.3-f) Code cyclique

FIG. 9.3 – Jeux de tests pour les encodeurs CCE

correct, tous ses variants le seront aussi. L'approche manuelle qui consiste à implémenter chaque encodeur spécifique est quant à elle beaucoup plus lente et sujette aux erreurs de programmation.

L'observation du code des encodeurs spécialisés a montré que toute la généricité qui dépendaient des paramètres de configuration a disparu. La propagation des informations de configuration s'est donc bien faite en accord avec les déclarations de spécialisation.

Chapitre 10

Environnement de conception de composants configurables

Pour assister le développement de composants configurables, nous avons construit un environnement graphique qui offre deux interfaces : une interface programmeur pour préparer un composant à être configurable et une interface utilisateur pour personnaliser un composant configurable en fonction de son utilisation finale.

10.1 Interface programmeur

L'interface programmeur (Figure 10.1) permet au programmeur de préparer le processus de configuration. L'interface affiche la liste de tous les modules de spécialisation présents dans le répertoire courant. La sélection de l'un de ces modules permet de visualiser la liste des scénarios selon lesquels le composant sélectionné peut être spécialisé. Choisir un de ces scénarios revient à spécifier le scénario principal de la spécialisation désirée. Les commandes offertes par l'interface permettent au programmeur de visualiser et de modifier les déclarations faites dans un module et de lancer le processus de création du composant spécialisable.

Les commandes "Module Editor" et "Customization Sheet Editor" peuvent être utilisées une fois qu'un module a été choisi. La commande "Module Editor" démarre un éditeur de texte contenant le code source du module considéré ; de nouveaux scénarios peuvent être ajoutés et les scénarios existants peuvent être effacés ou modifiés. La commande "Customization Sheet Editor" créée une nouvelle fenêtre qui correspond à l'éditeur de composants (Figure 10.2). Cet éditeur permet au programmeur de spécifier une description textuelle haut niveau des champs de structures, des variables globales et des fonctions qui sont mentionnées dans le module de spécialisation. Ces descriptions sont utilisées plus tard dans l'interface utilisateur lors de la génération du formulaire à travers lequel les valeurs de spécialisation peuvent être données. En plus des descriptions textuelles, le programmeur précise comment les valeurs de spécialisation doivent être spécifiées par l'utilisateur. Deux choix sont possibles, soit la valeur est directement entrée dans le formulaire (c'est typiquement le cas lorsqu'il s'agit d'un entier ou d'une

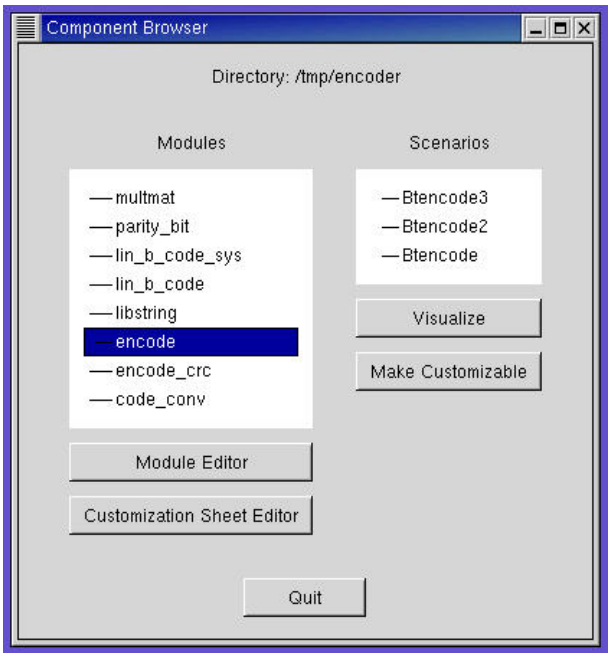


FIG. 10.1 – Interface programmeur

chaîne de caractères), soit le nom d'une fonction est mentionné et devra être exécutée pour initialiser le paramètre (par exemple dans le cas d'une matrice).

La commande "Visualize" créée deux nouvelles fenêtres : la fenêtre "Component Dependencies" et la fenêtre "Scenario Dependencies Window" (Figure 10.3) qui permettent de contrôler visuellement les relations entre le scénario de point d'entrée et les autres scénarios impliqués dans le processus de spécialisation. La fenêtre "Component Dependencies" présente un graphe où les noeuds correspondent aux différents modules impliqués dans la spécialisation et les arcs représentent la relation de dépendance `needs` entre les scénarios définis dans les modules. La fenêtre "Scenario Dependencies Window" permet une visualisation plus fine des informations de dépendance. Elle présente un arbre dont la racine est le scénario principal de la spécialisation et qui organise ensuite les scénarios dont il dépend selon qu'ils correspondent à des données, c'est-à-dire des structures de données ou des variables globales, ou à du code, c'est-à-dire des fonctions appelées. Pour chaque scénario affiché, les types sont colorés de façon à indiquer les contraintes de temps de liaison. Cela permet notamment de contrôler comment les informations de temps de liaison sont propagées à travers les scénarios.

Enfin la commande "Make Customizable" effectue la compilation des modules de spécialisation, en regroupant tout d'abord dans un fichier les fragments de programmes impliqués dans le processus de spécialisation et en générant les fichiers de configuration du spécialiseur Tempo. Puis elle lance la phase d'analyse sur le fichier contenant le code à spécialiser. Si aucune erreur n'est détectée alors un spécialiseur dédié est généré. Le composant ainsi traité est maintenant configurable et peut être distribué aux utilisateurs

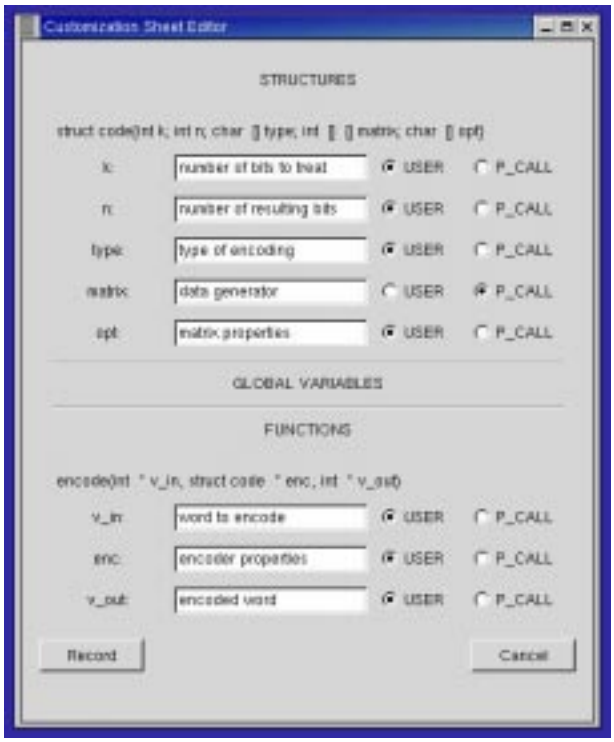


FIG. 10.2 – Éditeur de composants

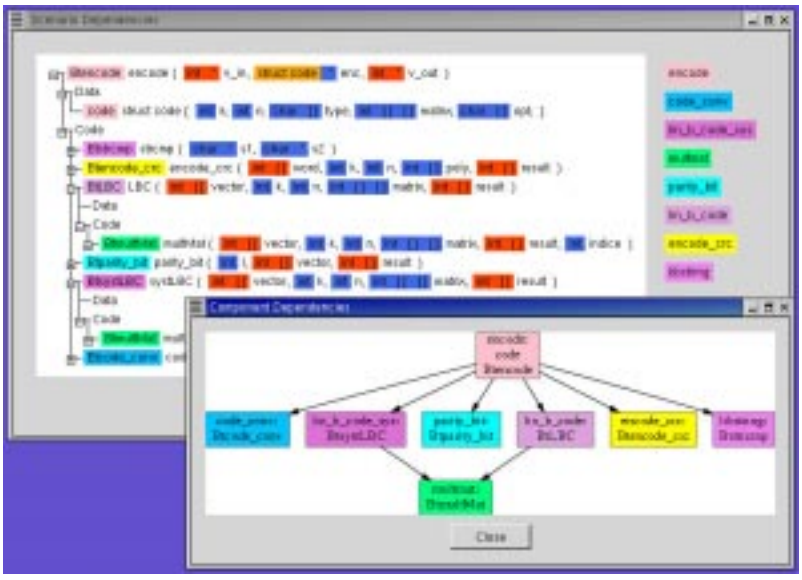


FIG. 10.3 – Outils de visualisation

potentiels.

10.2 Interface utilisateur

L'interface utilisateur (Figure 10.4) permet de créer des variants du composant configurable. L'interface affiche à l'utilisateur la liste des composants configurables présents dans le répertoire courant. La sélection de l'un de ces composants provoque la création d'une nouvelle fenêtre : Component Customization Interface (Figure 10.5). Cette fenêtre correspond au formulaire à travers lequel l'utilisateur doit donner les valeurs des paramètres de configuration. Ce sont les descriptions textuelles entrées par le programmeur qui sont affichées. Une fois les valeurs entrées, la commande "Customize" lance la spécialisation et génère le composant configuré. Cette opération peut être réitérée autant de fois qu'il y a de variants à produire.

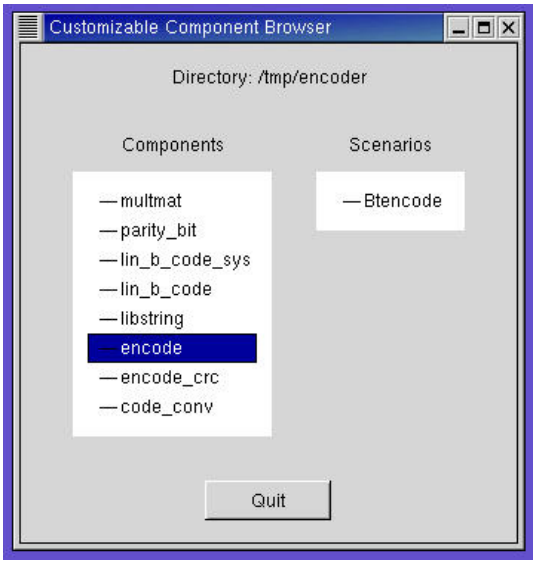


FIG. 10.4 – Interface utilisateur

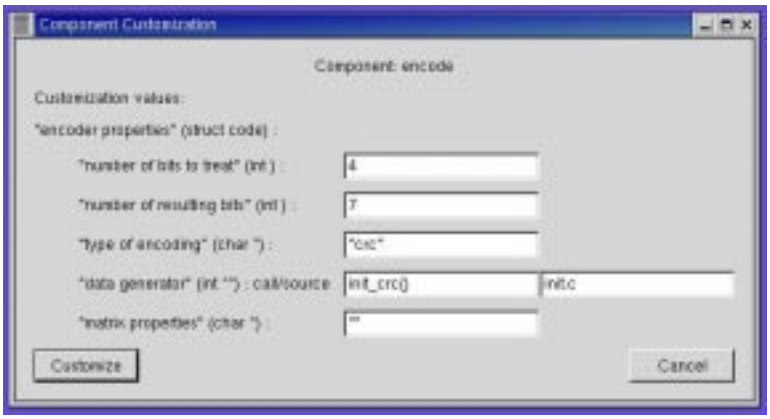


FIG. 10.5 – Interface de customisation d'un composant

10.3 Bilan

L'environnement graphique que nous avons développé rend l'utilisation d'un spécialiseur de programme complètement transparente. Les outils de visualisation sont très utiles pour avoir une vision globale des différents composants et scénarios qui sont impliqués dans le processus de spécialisation.

Chapitre 11

Conclusion

Il y a quelques années, Barbara Ryder faisait remarquer qu'un des défis posé par les analyses de programmes à la compilation était d'explorer comment ces analyses pouvaient être utilisées de façon efficace dans des outils de programmation [Ryd97]. La spécialisation de programmes repose sur de nombreuses analyses et n'échappe pas à ce défi. Bien que cette technique ait été étudiée de façon intensive ces 20 dernières années et que de nombreuses avancées pratiques aient été réalisées, elle ne fait toujours pas partie des utilitaires de programmation standards et de ce fait son intégration dans un processus de développement logiciel n'a pas eu lieu.

Nous avons présenté dans cette thèse une approche déclarative à la spécialisation de programmes C qui a pour but de faciliter l'utilisation de cette technique. Notre langage masque les concepts complexes d'évaluation partielle et permet de déclarer intuitivement des propriétés de spécialisation. Les déclarations sont organisées dans des modules de spécialisation, offrant une approche structurée et modulaire à la spécialisation. Les propriétés déclarées sont vérifiées pendant le processus de spécialisation ce qui rend la spécialisation d'un programme prévisible par rapport aux déclarations et offre ainsi la possibilité d'envisager la spécialisation automatique et systématique de composants logiciels.

Nous récapitulons dans ce chapitre les contributions de cette thèse et présentons quelques perspectives de recherche.

11.1 Contributions

Les contributions de cette thèse sont doubles. Dans un premier temps, nous présentons une approche déclarative à la spécialisation de programmes et dans un deuxième temps nous proposons une intégration de la spécialisation de programmes dans le développement de composants logiciels réutilisables.

Approche déclarative Le but principal de notre approche est de faciliter l'utilisation de la technique de spécialisation de programmes. Notre approche repose sur un langage déclaratif qui permet au programmeur de préciser son intention de spéciali-

sation et d'obtenir de la part du spécialiseur un retour quant à la faisabilité de la spécialisation décrite. À cette fin, nous avons conçu un langage de déclarations qui est indépendant des caractéristiques particulières d'un spécialiseur donné. La vérification de ces déclarations pendant la phase d'analyses du spécialiseur améliore la prédictabilité du processus de spécialisation. Les messages d'erreur signalant l'impossibilité de satisfaire les déclarations aident à identifier les problèmes de spécialisation. Enfin, la séparation des déclarations de spécialisation du programme source et l'organisation des déclarations de spécialisation dans des modules facilitent la réutilisation de l'expertise de spécialisation acquise.

Nous avons développé notre approche pour la spécialisation de programmes C. Pour ce faire, nous avons conçu et implémenté un compilateur pour traiter les déclarations de spécialisation. Nous avons utilisé le spécialiseur Tempo comme moteur de spécialisation et augmenté son analyse de temps de liaison par la vérification des déclarations.

Nous avons utilisé avec succès notre approche sur de nombreux exemples d'évaluation partiel existants tels que la FFT [Law98], la bibliothèque XDR de RPC [MVM97, MMV+98], le BPF [TCM+00] et les signaux Unix [MWP+01]. De plus, nous avons testé notre langage dans le cadre d'un cours de DEA. Comparé aux années précédentes, les étudiants qui ont utilisés les modules de spécialisation ont plus rapidement et plus facilement obtenu les programmes spécialisés désirés. Notre langage a aussi subi une première évaluation dans un contexte industriel et les résultats obtenus sont encourageants.

Composants logiciels configurables Nous proposons dans cette thèse d'utiliser la spécialisation de programmes pour configurer des composants logiciels génériques implémentés dans le contexte de familles de systèmes. Notre approche déclarative s'intègre dans la phase d'implémentation du domaine du processus d'ingénierie de composants réutilisables. S'appuyant sur les phases d'analyse et de conception du domaine qui ont mis en évidence les paramètres de configuration, le développeur d'un composant peut implémenter le code de façon générique et décrire dans des modules de spécialisation les propriétés de configuration de ce composant. Contrairement à d'autres approches existantes où le programmeur doit mettre en œuvre la configuration du code, c'est le spécialiseur qui détermine automatiquement les transformations à effectuer. De plus, l'utilisation des modules de spécialisation permet de garantir que la généricité qui dépend des paramètres de configuration sera bien éliminée pendant le processus de spécialisation.

Nous avons divisé le processus de configuration en deux phases. La première phase est effectuée par le programmeur : elle consiste à concevoir un composant configurable, c'est-à-dire à s'assurer que les descriptions de spécialisation du composant peuvent être satisfaites. La deuxième étape permet aux utilisateurs du composant de le configurer en fonction de leurs besoins. Pour assister le programmeur et les utilisateurs dans leurs tâches respectives, nous avons développé un environnement graphique d'aide à la spécialisation de composants. Enfin, pour illustrer notre approche nous avons créé un composant configurable pour le domaine des code correcteurs d'erreurs.

11.2 Perspectives et futures directions

Notre approche déclarative à la spécialisation de programmes se devait de remplir deux conditions primordiales : permettre au programmeur de décrire intuitivement les propriétés de spécialisation d'un programme et garantir que les déclarations de spécialisation peuvent être satisfaites par un spécialiseur donné. C'est sous ces conditions qu'une utilisation plus systématique de la spécialisation de programmes peut être rendue possible.

Les travaux présentés dans cette thèse ouvrent la voie dans cette direction mais de nombreuses investigations et travaux restent à effectuer, que ce soit au niveau du langage de déclarations, des outils d'aide à la spécialisation ou des applications pouvant bénéficier de la spécialisation.

Extensions du langage de déclarations Le langage des déclaration présenté dans cette thèse se concentre principalement sur la déclaration des temps de liaison des paramètres impliqués dans une spécialisation. Ces informations sont importantes puisque ce sont elles qui déterminent la spécialisation. Cependant pour spécifier correctement un contexte de spécialisation, il faudrait aussi pouvoir décrire les alias qui existent au début de la spécialisation, ainsi que les effets de bord des fonctions externes qui seront exécutées à la spécialisation. Concevoir des déclarations pour décrire de telles informations reste un problème ouvert qui n'a pas été adressé. Pour preuve, les solutions offertes par les spécialiseurs pour C existants sont ad hoc. Ainsi Tempo demande au programmeur d'écrire des programmes C décrivant les calculs d'alias et les effets bord effectués par les fonctions externes. Cette solution permet d'adresser n'importe quelle situation mais est sujette aux erreurs. Le spécialiseur C-Mix n'offre pas la possibilité de décrire les alias mais permet de caractériser le comportement d'une fonction externe selon quatre catégories [C-M00], ce qui n'offre pas la possibilité de donner une description précise.

En plus de la spécification du contexte de spécialisation, il serait intéressant de permettre au programmeur de contrôler en fonction des valeurs de spécialisation les transformations effectuées par le spécialiseur. Par exemple, empêcher le déroulage d'une boucle si la valeur de la variable de contrôle est supérieure à une constante donnée ou à la valeur d'un autre paramètre de spécialisation peut s'avérer utile pour limiter la taille du code résiduel.

Outils d'aide à la spécialisation La mise au point de programmes spécialisables est souvent non triviale. Même avec le support des modules de spécialisation, plusieurs itérations sont souvent nécessaires. Les informations d'erreur produites par le spécialiseur pendant l'analyse de temps de liaison doivent être les plus précises possibles pour renseigner au mieux le programmeur. Ainsi, par exemple, connaître la raison exacte d'un changement de temps de liaison est très utile au programmeur pour mieux identifier les problèmes de spécialisation. De plus, pour assister d'avantage le programmeur, il semble intéressant de développer un environnement de mise au point à travers lequel on pourrait, par exemple, suivre l'évolution de certains paramètres pendant l'analyse de temps liaison ou insérer des gardes de contrôle.

Dans notre environnement graphique d'aide à la conception de composants configurables, la visualisation des dépendances entre les scénarios offre un bon moyen de vérifier quels fragments de code sont impliqués et dans quels contextes ils vont être spécialisés. Il serait de plus intéressant d'obtenir des informations sur le résultat de spécialisation escompté, sous la forme par exemple d'une estimation du pourcentage de code qui peut être éliminé, ou le nombre de conditionnelles pouvant être être réduites, ou de la taille du code qui peut être généré. Dans le cadre de Tempo, une analyse de l'arbre d'actions qui est produit à la fin de la phase d'analyses pourrait permettre d'extraire et de déduire de telles informations.

Utilisation de la spécialisation de programmes Dans la mesure où la spécialisation de programmes peut être appliquée de façon automatique et systématique, il est possible d'envisager son utilisation dans de nombreux contextes. Nous présentons deux utilisations possibles.

Nous avons proposé dans cette thèse de configurer des composants logiciels par spécialisation à la compilation. Une direction future de nos travaux est de permettre la reconfiguration dynamique d'un composant. Prenons, par exemple, une application audio qui traite des flux de données et qui dans certaines situations doit adapter son comportement en fonction des ressources disponibles sur le système. Cette adaptation pourrait être mise en œuvre par une reconfiguration de certaines parties de son code. Cette reconfiguration dynamique reposera évidemment sur le spécialiseur à l'exécution de Tempo.

Les générateurs de code sont d'autres applications qui pourrait bénéficier de la spécialisation. Tous les types de générateurs ne sont pas concernés, seuls ceux qui génèrent du code qui fait appel à des fonctions de bibliothèques sont des bons candidats. Prenons cette fois l'exemple des appels de procédure à distance (RPC). À partir d'une description des données qui sont à transmettre, le générateur du RPC, *rcpgen*, produit le code à exécuter sur le client et le serveur pour effectuer les opérations d'encodage et de décodage des données. Ces opérations sont mise en œuvre par les fonctions de la bibliothèque XDR. Comme l'ont montré Muller *et al.*, cette bibliothèque présente de nombreuses opportunités de spécialisation [MVM97, MMV+98]. En associant des modules de spécialisation à cette bibliothèque et en modifiant *rpcgen* pour qu'il fasse appel à un spécialiseur, il serait possible de produire de manière automatique et systématique des fonctions d'encodage et de décodage spécialisées. Cette caractéristique n'est pas propre à ce générateur, il est en effet envisageable d'appliquer la même approche dans le cadre des générateurs d'interfaces graphiques, par exemple.

Bibliographie

[AC94] J. M. Ashley et 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.

[And94] L.O. Andersen. « *Program Analysis and Specialization for the C Programming Language* ». PhD thesis, Computer Science Department, University of Copenhagen, mai 1994. DIKU Technical Report 94/19.

[And96] P.H. Andersen. « Partial evaluation applied to ray tracing ». Dans W. Mackens et S.M. Rump, éditeurs, *Software Engineering in Scientific Computing*, pages 78–85. Vieweg, 1996. DIKU Technical Report D-289.

[Bat98] Don S. Batory. « Product-Line Architectures ». Dans *Invited presentation, Smalltalk und Java in Industrie and Ausbildung*, Erfurt, Germany, octobre 1998.

[BFK+99] J. Bayer, P. Flege, P. Knauber, R. Laqua, D Muthih, K. Schmid, T. Widen, et J.-M. DeBaud. « PuLSE : A methodology to develop software product lines ». Dans *Symposium on Software Reuse*, pages 122–131, mai 1999.

[BGZ94] R. Baier, R. Glück, et R. Zöchling. « Partial Evaluation of Numerical Programs in Fortran ». Dans PEPM'94 [PEP94], pages 119–132.

[Bon90] A. Bondorf. « *Self-Applicable Partial Evaluation* ». PhD thesis, DIKU, University of Copenhagen, Denmark, 1990.

[Bon92] A. Bondorf. « Improving Binding Times Without Explicit CPS-Conversion ». Dans *ACM Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, CA, USA, juin 1992. ACM Press.

[Bos98] J. Bosch. « Evolution and Composition of Reusable Assets in Product-Line Architectures : A Case Study ». Dans *First Working IFIP Conference on Software Architecture*, octobre 1998.

[BS94] A.A. Berlin et R.J. Surati. « Partial Evaluation for Scientific Computing : The Supercomputer Toolkit Experience ». Dans PEPM'94 [PEP94], pages 133–141.

[C-M00] « C-Mix$_{\text{II}}$ User and Reference Manual ». http ://www.diku.dk/research-groups/topps/activities/cmix/download/, 2000.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, et S.F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[CD91] C. Consel et O. Danvy. « For a Better Support of Static Data Flow ». Dans Hughes [Hug91], pages 496–519.

[CE00] K. Czarnecki et U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CGL00] N. H. Christensen, R. Glück, et S Laursen. « Binding-Time Analysis in Partial Evaluation : One Size Does Not Fit All ». Dans *Perspectives of System Informatics*, volume 1755 de *lncs*, pages 80–92. D. Bjørner, M. Broy, A. V. (Eds.), 2000.

[CHL+98] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, et N. Volanschi. « Tempo : Specializing Systems Applications and Beyond ». *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[CHN+96a] C. Consel, L. Hornof, F. Noël, J. Noyé, et E.N. Volanschi. « A Uniform Approach for Compile-Time and Run-Time Specialization ». Dans Danvy et al. [DGT96], pages 54–72.

[CHN+96b] C. Consel, L. Hornof, F. Noël, J. Noyé, et E.N. Volanschi. « A Uniform Approach for Compile-Time and Run-Time Specialization ». Dans Danvy et al. [DGT96], pages 54–72.

[CKFS01] Y. Coady, G. Kiczales, M. Feeley, et G. Smolyn. « Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code ». Dans *ACM SIGSOFT Symposium on Foundations of Software Engineering*, août 2001.

[CN96] C. Consel et F. Noël. « A General Approach for Run-Time Specialization and its Application to C ». Dans *Conference Record of the* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, janvier 1996.

[Con93] C. Consel. « A Tour of Schism : a partial evaluation system for higher-order applicative languages ». Dans *Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, juin 1993. ACM Press.

[DGT96] O. Danvy, R. Glück, et P. Thiemann, éditeurs. *Partial Evaluation, International Seminar, Dagstuhl Castle*, numéro 1110 dans Lecture Notes in Computer Science, février 1996.

[DS99] JN. DeBaud et K. Schmid. « A Systematic Approach to Derive the Scope of Software Product Lines ». Dans *Proceedings of 21st International Conference on Software Engineering*, 1999.

[EGH94] M. Emani, R. Ghiya, et L. J. Hendren. « Context-sensitive interprocedural points-to analysis in the presence of function pointers ». Dans *ACM SIGPLAN Conference on Programming Lanugage Design and Implementation*, pages 242–256. ACM Press, juin 1994.

[EHK96] D.R. Engler, W.C. Hsieh, et M.F. Kaashoek. « ‘C : A Language for High-level, Efficient, and Machine-Independent Dynamic Code Generation ». Dans *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 131–144, St. Petersburg Beach, FL, USA, janvier 1996. ACM Press.

[GJ95] R. Glück et J. Jørgensen. « Efficient Multi-level Generating Extensions for Program Specialization ». Dans M. Hermenegildo et S. Doaitse Swierstra, éditeurs, *Proceedings of the* 7^{th} *International Symposium on Programming Language Implementation and Logic Programming*, volume 982 de *Lecture Notes in Computer Science*, pages 259–278, Utrecht, The Netherlands, septembre 1995.

[GMP+00] B. Grant, M. Mock, M. Philipose, C. Chambers, et S.J. Eggers. « DyC : An Expressive Annotation-Directed Dynamic Compiler for C ». *Theoretical Computer Science*, 248(1–2) :147–199, 2000.

[Gro01] B. Grobauer. « Cost Recurrences for DML Programs ». Dans *ICFP 2001 : International Conference on Functional Programming*, pages 253–264, Florence, Italy, septembre 2001. ACM Press.

[GS94] D. Garlan et M. Shaw. « An Introduction to Software Architecture ». Rapport Technique CMU-CS-94-166, Carnegie Mellon University, janvier 1994.

[Hat00] Red Hat. « eCos : Embedded Configurable Operating System », 2000. http ://sources.redhat.com/ecos.

[HN00] L. Hornof et J. Noyé. « Accurate Binding-Time Analysis for Imperative Languages : Flow, Context, and Return Sensitivity ». *Theoretical Computer Science*, 248(1–2) :3–27, 2000.

[Hug91] J. Hughes, éditeur. *Functional Programming Languages and Computer Architecture*, volume 523 de *Lecture Notes in Computer Science*, Cambridge, MA, USA, août 1991. Springer-Verlag.

[JGS93] N.D. Jones, C. Gomard, et P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, juin 1993.

[Jon96] N.D. Jones. « What *Not* to Do When Writing an Interpreter for Specialisation ». Dans Danvy et al. [DGT96], pages 216–237.

[JSS85] N.D. Jones, P. Sestoft, et H. Søndergaard. « An experiment in partial evaluation : the generation of a compiler generator ». Dans J.-P. Jouannaud, éditeur, *Rewriting Techniques and Applications*, volume 202 de *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, et S. Peterson. « Feature-Oriented Domain Analysis (FODA) Feasibility Study ». Rapport Technique CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, novembre 1990.

[KKZG94] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, et R. Glück. « Fortran Program Specialization ». Dans U. Meyer et G. Snelting, éditeurs, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, et J. Irwin. « Aspect-Oriented Programming ». Dans M. Aksit et S. Matsuoka, éditeurs, *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, volume 1241 de *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, juin 1997. Springer.

[KM00] J. Kono et T. Masuda. « Efficient RMI : Dynamic Specialization of Object Serialization ». Dans *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 308–315, Taipei, Taiwan, avril 2000. IEEE Computer Society Press.

[Law98] J.L. Lawall. « Faster Fourier Transforms via Automatic Program Specialization ». Dans John Hatcliff, Torben Æ. Mogensen, et Peter Thiemann, éditeurs, *Partial Evaluation – Practice and Theory ; Proceedings of the 1998 DIKU Summer School*, volume 1706 de *Lecture Notes in Computer Science*, pages 338–355, Copenhagen, Denmark, juillet 1998. Springer-Verlag.

[LC83] S. Lin et D. J. Costello. *Error Control Coding : Fundamentals and Applications*. Prentice Hall : Englewood Cliffs, NJ, 1983.

[LL96] P. Lee et M. Leone. « Optimizing ML with Run-Time Code Generation ». Dans *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, PA, USA, mai 1996. ACM SIGPLAN Notices, 31(5).

[LMC00] A.-F. Le Meur et C. Consel. « Generic Software Component Configuration Via Partial Evaluation ». Dans *SPLC'2000 Workshop – Product Line Architecture*, Denver, Colorado, août 2000.

[LMC01] A.-F. Le Meur et C. Consel. « Configurabilité garantie des composants par les modules de spécialisation ». Dans *Journées composants : flexibilité du système au langage*, Besançon, France, octobre 2001.

[LMCE02] A.-F. Le Meur, C. Consel, et B. Escrig. « An Environment for Building Customizable Software Components ». Dans *IFIP/ACM Conference on Component Deployment*, Berlin, Germany, juin 2002.

[LMLC02a] A.-F. Le Meur, J. Lawall, et C. Consel. « A Pragmatic Approach to Declaring Specialization Scenarios ». Rapport Technique 1285-02, LaBRI, Bordeaux, France, décembre 2002.

[LMLC02b] A.-F. Le Meur, J. Lawall, et C. Consel. « Towards Bridging the Gap Between Programming Language and Partial Evaluation ». Dans *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–18, Portland, OR, USA, janvier 2002. ACM Press.

[LS91] J.W. Lloyd et J.C. Shepherdson. « Partial evaluation in logic proramming ». *Journal of Logic Programming*, 11 :217–242, 1991.

[Mey91] U. Meyer. « Techniques for Partial Evaluation of Imperative Languages ». Dans *Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105, New Haven, CT, USA, septembre 1991. ACM SIGPLAN Notices, 26(9).

[MJ93] S. McCanne et V. Jacobson. « The BSD Packet Filter : A New Architecture for User-level Packet Capture ». Dans *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, San Diego, CA, USA, janvier 1993. USENIX.

[MMV+98] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, et A. Goel. « Fast, Optimized Sun RPC Using Automatic Program Specialization ». Dans *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, mai 1998. IEEE Computer Society Press.

[MVM97] G. Muller, E.N. Volanschi, et R. Marlet. « Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol ». Dans *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125. ACM Press, juin 1997.

[MWP+01] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, C. Goel, C. Consel, G. Muller, et R. Marlet. « Specialization Tools and Techniques for Systematic Optimization of System Software ». *ACM Transactions on Computer Systems*, 19 :217–251, mai 2001.

[Nei89] J. M. Neighbors. « Draco : A Method for Engineering Reusable Software Systems ». Dans T. Biggerstaff et ACM Press Frontier Serie Addison Wesley Perlis, A. eds., éditeurs, *Software Reusability*, volume 1, pages 295–319, 1989.

[NHCL98] F. Noël, L. Hornof, C. Consel, et J. Lawall. « Automatic, Template-Based Run-time Specialization : Implementation and Experimental Study ». Dans *International Conference on Computer Languages*, pages 132–142, Chicago, IL, mai 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[Par76] D.L. Parnas. « On the Design and Development of Program Families ». *IEEE Transactions on Software Engineering*, 2 :1–9, mars 1976.

[PEP94] *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, FL, USA, juin 1994. Technical Report 94/9, University of Melbourne, Australia.

[Plo75] G. D. Plotkin. « Call-by-name, call-by-value and the lambda-calculus ». Dans *Theoretical Computer Science*, pages 125–159, 1975.

[Ruf95] E. Ruf. « Context-Insensitive Alias Analysis Reconsidered ». Dans *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22. ACM SIGPLAN Notices, 30(6), juin 1995.

[Ryd97] B.G. Ryder. « A Position Paper on Compile-time Program Analysis ». *ACM SIGPLAN Notices*, 32(1) :110–114, janvier 1997.

[SCK+96] M. Simos, D. Creps, C. Klinger, L. Levine, et D. Allemang. « Organisation Domain Modeling (ODM) Guidebook ». Rapport Technique Version 2.0. STARS-VC-A025/001/00, juin 1996.

[SEIa] Software Engineering Institute : SEI. « Domain Engineering : A Model-Based Approach ». http ://www.sei.cmu.edu/domain-engineering/domain_engineering.html.

[SEIb] Software Engineering Institute : SEI. « How Do You Define Software Architecture ? ». http ://www.sei.cmu.edu/architecture/definitions.html.

[SL98] J. G. Siek et A. Lumsdaine. « The Matrix Template Library : A generic programming approach to high performance numerical linear algebra ». Dans *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[SLCM99] U. Schultz, J. Lawall, C. Consel, et G. Muller. « Towards Automatic Specialization of Java Programs ». Dans *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 de *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, juin 1999.

[Sof97] Java Soft. « *Java Remote Method Invocation Specification* ». http ://www.javasoft.com, 1997.

[Sun] Java Sun.
http ://java.sun.com/products/javabeans.

[Sun90] Sun Microsystems. « *Network Programming Guide* ». Sun Microsystems, mars 1990.

[TCM+00] S. Thibault, C. Consel, R. Marlet, G. Muller, et J. Lawall. « Static and Dynamic Program Compilation by Interpreter Specialization ». *Higher-Order and Symbolic Computation*, 13(3) :161–178, 2000.

[TS97] W. Taha et T. Sheard. « Multi-Stage Programming with Explicit Annotations ». Dans *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, juin 1997. ACM Press.

[VCMC97] E.N. Volanschi, C. Consel, G. Muller, et C. Cowan. « Declarative Specialization of Object-Oriented Programs ». Dans *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, GA, USA, octobre 1997. ACM Press.

[Vel98] T.L. Veldhuizen. « Arrays in Blitz++ ». Dans *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, volume 1505 de *Lecture Notes in Computer Science*, Santa Fe, NM, USA, décembre 1998. Springer-Verlag.

[Vel99] T. L. Veldhuizen. « C++ Templates as Partial Evaluation ». Dans *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, TX, USA, janvier 1999. ACM Press.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, et S. Seligman. « Automatic Online Partial Evaluation ». Dans Hughes [Hug91], pages 165–191.

[Wei96] D.M. Weiss. « Family-oriented Abstraction Specification and Translation : the FAST Process ». Dans *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland*, pages 14–22. IEEE Press, Piscataway, NJ, 1996.

[WL99] D.M. Weiss et C.T.R. Lai. *Software Product-Line Engineering : A Family-Based Software Development Process.* Addison Wesley, 1999.

Résumé

L'évaluation partielle est une transformation de programmes qui permet de spécialiser automatiquement un programme pour un contexte d'utilisation donné. Cette technique de spécialisation a suscité beaucoup d'attention ces vingt dernières années et a donné lieu à de nombreuses avancées aussi bien théoriques que pratiques. Ainsi, la spécialisation de programme a été étudiée pour les langages fonctionnels, impératifs, logiques et à objets. Plusieurs spécialiseurs ont été développés pour des langages à taille réelle tels que C ou Java et ont été utilisés avec succès pour de nombreuses applications dans des domaines aussi variés que les systèmes d'exploitation, le calcul scientifique et le graphisme.

En dépit des nombreux succès de la spécialisation de programmes, cette technique n'est pas encore accessible à des programmeurs non-experts. Une raison de cette non-accessibilité est qu'il est difficile de décrire les opportunités de spécialisation. Il est de ce fait courant qu'un programme soit trop ou insuffisamment spécialisé. Le programmeur doit alors faire face au délicat problème de comprendre pourquoi le programme spécialisé ne correspond pas à ce qu'il attendait.

Nous avons développé un langage de déclarations haut niveau qui permet au programmeur de préciser quels sont les fragments de code et les invariants qui doivent être pris en considération par le processus de spécialisation. Ces déclarations sont distinctes du code et sont écrites par le programmeur lors développement du programme. La syntaxe des déclarations est similaire à la syntaxe du langage traité, dans notre cas C. Les déclarations sont vérifiées avant la phase de spécialisation afin de renseigner le programmeur quant à la faisabilité de la spécialisation désirée.

Cette approche permet de rendre le processus de spécialisation prévisible par rapport aux déclarations et offre ainsi la possibilité d'envisager la spécialisation automatique et systématique de composants logiciels. Nous avons utilisé notre approche dans le cadre de nombreux exemples et notamment lors du développement d'un composant spécialisable dans le domaine des codes correcteurs d'erreurs.

Nous avons conçu et implémenté un compilateur pour le langage de déclarations, ainsi qu'un environnement graphique. Cet environnement offre aux programmeurs des outils d'aide au développement de composants spécialisables et permet aux utilisateurs de ces composants de les spécialiser en fonction de leur besoin.

Mots clés

Spécialisation de programmes, génie logiciel.