N. d'ordre: 1810

THÈSE

présentée devant

l'Université de Rennes 1

pour obtenir

le grade de Docteur de l'Université de Rennes 1
Mention : Informatique
École Doctorale : Sciences Pour l'Ingénieur
Institut de Formation Supérieure en Informatique et Communication

par

Luke HORNOF

Titre de la thèse

ANALYSES STATIQUES POUR LA SPÉCIALISATION EFFECTIVE DE PROGRAMMES RÉALISTES

Soutenue le 27 juin 1997 devant la commission d'examen

MM.	Jean-Pierre Banâtre	Président
	Charles Consel	Directeur de thèse
	Pierre Jouvelot	Rapporteur
	Torben Mogensen	Rapporteur
	Thomas Jensen	Examinateur

Remerciements

Je tiens à remercier tous les membres de l'avant-projet COMPOSE de m'avoir aidé à réaliser ce travail. Je remercie mon encadreur de thèse, Charles Consel, qui a créé l'environnement et a fourni le cadre pour ce travail, ces deux aspects étaient nécessaires afin d'espérer l'accomplir. De nombreuses discussions avec Jacques Noyé ont abouti à des idées critiques. J'ai aussi beaucoup appris lors de conversations avec Bárbara Moura où nous avons comparé nos différentes approches. D'autres personnes ont aidé à la conception et la mise en œuvre de Tempo, je voudrais citer François Noël et Nic Volanski. L'expertise de programmes systèmes a été largement fournie par Gilles Muller.

La communauté de recherche en général m'a aidé de différentes manières. Les chercheurs de Oregon Graduate Institute m'ont transmis une expérience riche. L'un d'entre eux, Calton Pu, m'a offert généreusement ses conseils à plusieurs reprises. Des échanges avec des professeurs et étudiants de DIKU, Université de Copenhague, et Olivier Danvy, de l'Université de Aarhus, ont donné lieu à des discussions fructueuses. Peter Lee, de Carnegie Mellon University, a été un point de repère pendant plusieurs années.

Je voudrais aussi remercier d'autres personnes. Julia Lawall m'a beaucoup aidé lors de la rédaction des premières versions du document. Jean-Pierre Banâtre a accepté d'être président de mon jury. Pierre Jouvelot, Torben Mogensen, et Thomas Jensen ont non seulement donné des commentaires sur les dernières versions du texte, mais également accepté de participer à mon jury de thèse. Gilles Lesventes a traduit seul tout le document en français. Remi Douence, Ronan Gaugne et Valérie Gouranton ont apporté des commentaires sur la traduction française.

Finalement, je voudrais remercier ma famille pour son soutien affectif et sa disponibilité pour les problèmes pratiques. Je voudrais surtout remercier mon père qui m'a aidé à comprendre l'importance de l'éducation et à qui je voudrais dédier cette thèse.

Table des matières

Table des figures

Chapitre 1

Introduction

Cette thèse montre comment la spécialisation de programmes peut optimiser automatiquement et efficacement des applications existantes et réalistes. Les études précédentes ont démontré que la spécialisation d'applications de grande taille, comme les programmes systèmes, introduit des accélérations (ou speedups) impressionnantes mais ces spécialisations étaient faites à la main [Mas92, MP89, PMI88]. Une telle approche s'est avérée trop consommatrice en temps, fastidieuse et encline aux erreurs pour être systématiquement et méthodologiquement appliquée. La clef est de réaliser ces spécialisations *automatiquement*. Les techniques de spécialisation automatique ont été étudiées depuis quelque temps et de nombreuses avancées théoriques comme pratiques ont été faites [And94a, BW93a, BM90, Con93a, DRVH95, JGS93a, Mog88]. Malheureusement, l'état d'avancement actuel ne permet pas à la technologie de la spécialisation de traiter efficacement la plupart des applications réalistes existantes. Plutôt, les applications pour lesquelles la spécialisation automatique s'est montrée efficace sont typiquement relativement petites et simples, écrites à la main pour favoriser la spécialisation ou écrites dans des langages de programmation spécifiques au domaine étudié ou peu utilisés.

Notre approche pour résoudre ce problème débute par une étude approfondie d'une classe d'applications existantes, les applications des systèmes d'exploitation [CPW93, PAB+95, VMC96a, VMC+96b]. De tels programmes sont compliqués. Ils sont écrits dans un langage compliqué comme C et leur comportement est complexe parce que la modularité comme l'efficacité sont importantes. Nous explorons les limites de la technologie existante de la spécialisation et identifions les aspects spécifiques faisant défaut et qui permettraient une spécialisation automatique et efficace de tels programmes. Nous abordons ces fonctionnalités manquantes et introduisons une nouvelle analyse statique qui traite précisément les motifs de programmes communs à toutes ces applications existantes. Ces fonctionnalités spécifiques incluent :

La sensibilité au flot. Une description d'analyse différente est faite pour chaque point de programme.

La sensibilité au contexte. Une fonction est analysée selon le contexte d'analyse spécifique à chaque point d'appel.

La sensibilité au retour. Une description d'analyse différente est faite pour les effets de bord et la valeur retournée d'une fonction.

La sensibilité à l'utilisation Les différentes utilisations d'une variable peuvent avoir des descriptions d'analyse différentes .

La sensibilité au flot a été exploitée dans d'autres analyses statiques [MRB95] mais nous l'incorporons pour la première fois dans les analyses d'un spécialiseur. Des analyses sensibles au contexte existent déjà dans le cadre de l'évaluation partielle

de langages plus simples, tels que les langages fonctionnels [Con93b], mais nous développons des aspects nouveaux de façon à traiter des langages impératifs. De plus, traiter des systèmes existants et réalistes met à jour le besoin de la sensibilité au retour et à l'utilisation, qui sont des concepts totalement nouveaux.

Pour développer une analyse incluant toutes ces fonctionnalités, nous présentons une approche qui diffère fondamentalement des travaux existants, ceci sur plusieurs points de vue, notamment :

- Une analyse en deux phases.
- De nouvelles transformations de programmes.

Notre nouvelle approche sépare les différentes tâches réalisées par une analyse de temps de liaison en deux phases distinctes, une phase d'analyse de temps de liaison suivie d'une phase de transformation. Cette séparation simplifie et clarifie la conception de l'analyse. Cette approche nous permet de plus d'exprimer de nouvelles transformations de programmes, telles que l'évaluation et la résidualisation d'une même construction de programme.

Nos analyses sont utilisables à la fois pour la spécialisation à la compilation et la spécialisation à l'exécution. Nous avons intégré nos analyses dans un évaluateur partiel et l'avons appliqué à un ensemble d'applications existantes, incluant un système d'exploitation commercial. Nous avons obtenu des accélérations significatives tant lors de la spécialisation à la compilation que lors de la spécialisation à l'exécution.

Dans la suite de ce chapitre, nous faisons une présentation générale de l'évaluation partielle, une technique de spécialisation de programmes qui transforme un programme généraliste en un programme pour un usage particulier. Nous débutons par un petit exemple et montrons les différentes façons de spécialiser. Puis, nous résumons sommairement, évaluons les évaluateurs partiels existants et examinons les applications qu'ils traitent. Une discussion sur les programmes systèmes existants vient ensuite, suivie d'un aperçu des fonctionnalités d'analyses particulières nécessaires pour spécialiser efficacement ces programmes.

1.1 Évaluation partielle

L'évaluation partielle est une transformation de programme qui prend un programme et une partie de ses données et génère un nouveau programme qui exploite ces données. Cela donne un cas particulier du programme initial qui est par conséquent considéré comme *spécialisé*. Diminuer la généralité d'un programme permet que des calculs qui ne dépendent que des valeurs d'entrées fournies soient exécutés lors de la transformation. Le programme spécialisé résultant ne contient que le reste des calculs qui dépendent des valeurs d'entrée indisponibles. Le but est que le programme spécialisé contienne moins de calculs et ainsi soit plus efficace que le programme original.

1.1.1 Approche en ligne contre approche hors-ligne

Il y a deux types d'évaluateurs partiels : les *en ligne* et les *hors-ligne*. Un évaluateur partiel en ligne produit un programme résiduel en une seule étape [JGS93a, Mey91, WCRS91]. On peut le voir comme un interpréteur non-standard où l'état associe une valeur à une variable connue et une représentation textuelle de la variable à une variable inconnue. Ces représentations textuelles combinées produisent ultimement le programme résiduel.

À l'opposé, l'approche hors-ligne divise la spécialisation en deux étapes: une étape d'analyse qui détermine les transformations suivie d'une étape qui les réalise [JGS93a, Con93d, And94b]. On ne fournit pas à l'étape d'analyse les valeurs

d'entrées connues mais seulement leur *description abstraite*. Se basant sur cette information fragmentaire, l'analyse calcule une transformation pour chaque construction dans le programme. Cette information est ensuite transmise à l'étape de transformation, à qui on fournit aussi les valeurs réelles d'entrée. La transformation correspondant à chaque construction est alors réalisée, produisant le programme résiduel.

Comme l'approche en ligne prend en compte les valeurs réelles d'entrée pour déterminer une transformation, elle peut produire de meilleurs résultats que l'approche hors-ligne. Par exemple, considérons une instruction conditionnelle dont on connaît la valeur de test, dont la branche positive affecte une valeur connue à une variable et dont la branche négative affecte une valeur inconnue à la même variable. Si le test est positif, l'évaluateur partiel en ligne peut utiliser cette valeur pour déduire que la valeur connue sera affectée à la variable et utiliser cette information après la conditionnelle. Une approche hors-ligne n'a pas les valeurs réelles d'entrée lors de la phase d'analyse (i.e. au moment où l'on détermine les transformations) et par conséquent ne peut pas déterminer laquelle des deux branches sera prise. Une approximation est faite dans laquelle on suppose que la variable se voit affecter une valeur inconnue.

D'un autre coté, l'approche hors-ligne a de nombreux avantages. L'étape de transformation peut exploiter l'information globale sur le programme généré pendant l'analyse de temps de liaison, qui s'avére essentielle quand on traite des programmes réalistes et existants. Par exemple, traiter un langage impératif nécessite des informations globales comme la relation entre la définition d'une variable et ses utilisations. De plus, on peut exiger des analyses qu'elles traitent précisément et correctement des constructions de langage complexes tels que les pointeurs, les tableaux et les structures de données. Un autre avantage est que, lorsque les transformations sont déterminées à l'avance, l'étape de transformation hors-ligne est plus simple et plus rapide. Par conséquent, l'approche hors-ligne est cruciale pour une spécialisation efficace.

1.1.2 À la compilation contre à l'exécution

Le but de l'évaluation partielle est de minimiser le temps requis pour exécuter un programme. La phase avant l'exécution du programme s'appelle le *temps de compilation*, ceci inclut toutes les étapes pour passer d'un code source à un code objet. La phase où un programme s'exécute s'appelle le temps d'exécution ou *runtime* en anglais. La *spécialisation à la compilation* a trait à la spécialisation d'un programme en fonction des valeurs d'entrée qui deviennent disponibles avant que le programme ne soit exécuté. Le programme résiduel est compilé et le code objet résultant peut alors être exécuté. Le temps entre le moment où les valeurs d'entrée connues deviennent disponibles et le moment où le programme spécialisé peut être exécuté est substantiel puisqu'il inclut à la fois le temps de spécialisation et le temps de compilation. C'est acceptable pour une spécialisation à la compilation puisqu'aucun coût supplémentaire d'exécution n'est introduit.

Pour certaines applications, toutefois, les valeurs d'entrées connues ne deviennent disponibles qu'à l'exécution. Spécialiser un programme en fonction de ces valeurs lors de l'exécution du programme est connu sous le nom de *spécialisation à l'exécution*[CN96, NHCL96]. La spécialisation à l'exécution n'est avantageuse que si le temps économisé par l'exécution du programme résiduel le plus efficace est supérieur au temps nécessaire à générer le programme résiduel. Donc, il est crucial de minimiser le coût de la spécialisation de programmes à l'exécution.

1.2 Un exemple concret

De façon à clarifier les idées, nous prenons un petit exemple et montrons comment il est spécialisé. Premièrement nous donnons un exemple de spécialisation en ligne. Puis nous montrons l'approche hors-ligne, consistant à l'analyse de temps de liaison suivie de la spécialisation. Enfin, nous montrons comment l'approche hors-ligne conduit à des spécialiseurs à l'exécution efficaces.

1.2.1 Spécialisation en ligne

Considérons la fonction `dotproduct` qui calcule le produit scalaire de deux vecteurs de longueur `size`.

```
int dotproduct(int size, int u[], int v[])
{
  int i;
  int res;

  res = 0;
  for (i = 0; i < size; i++)
    res = res + u[i] * v[i];
  return res;
}
```

Un appel normal à cette fonction fournit une valeur pour chacune des trois entrées telles que `size = 5`, `u[] = {4,8,3,2,9}`, et `v[]= {3,3,4,5,5}`. Avec ces arguments, une évaluation complète de la fonction rend le produit scalaire approprié, `103`. D'un autre coté, si on ne fournit que certaines de ces valeurs d'entrée, la fonction ne peut pas être totalement évaluée. Elle peut, toutefois, être partiellement évaluée en fonction des valeurs d'entrée disponibles. En supposant que la valeur `size` et le vecteur `u[]` soient passés avec les mêmes valeurs que ci-dessus, l'évaluation partielle en ligne produit le programme résiduel suivant :

```
int dotproduct_1(int v[])
{
  int res;

  res = 4 * v[0];
  res = res + 8 * v[1];
  res = res + 3 * v[2];
  res = res + 2 * v[3];
  res = res + 9 * v[4];
  return res;
}
```

Le résultat de l'évaluation partielle n'est pas un produit scalaire, mais un programme spécialisé qui calcule un produit scalaire (pour des vecteurs de longueur 5 dont le premier vecteur contient les valeurs `{4,8,3,2,9}`). Nous nous attendons à ce que la version spécialisée du produit scalaire soit plus efficace que la version générale et originale et qu'elle fournisse le même résultat. Tous les calculs qui dépendent de `size` et du vecteur `u[]` sont exécutés lors de la transformation et ne sont donc plus présents dans le programme résiduel. Par exemple, la boucle est déroulée puisqu'elle ne dépend que de `size` et chaque occurrence de `u[i]` est exemplarisée

avec sa valeur. De fait, évaluer la fonction spécialisée `dotproduct_1` avec l'entrée `v[] = {3,3,4,5,5}` produit le même résultat que précédemment c'est-à-dire 103.

De par la production, en fonction de ces valeurs d'entrée, d'un programme spécialisé plus efficace, les différents appels à `dotproduct_1` sont plus efficaces. Dans la pratique, réaliser une telle spécialisation peut être utile si, par exemple, la fonction originale est appelée de nombreuses fois avec les mêmes deux premières valeurs d'entrée, comme c'est le cas dans la multiplication de matrices.

1.2.2 Spécialisation hors-ligne

Maintenant regardons comment la spécialisation hors-ligne spécialise le même programme. La spécialisation hors-ligne consiste en deux étapes: l'étape d'analyse suivie de l'étape de transformation. Une fois le programme analysé, il y a de nombreuses façons de le spécialiser. Tout d'abord, nous présentons la spécialisation traditionnelle, puis une spécialisation plus rapide appelée extension génératrice. Enfin, nous montrons comment l'extension génératrice peut être adaptée pour réaliser efficacement la spécialisation à l'exécution.

Analyse de temps de liaison

La première analyse de l'approche hors-ligne est l'*analyse de temps de liaison*, qui détermine une transformation pour chaque construction. L'analyse de temps de liaison est une interprétation abstraite sur deux valeurs, ou *temps de liaison*, *static* et *dynamic*. *Static* est l'abstraction d'une valeur connue et *dynamic* est l'abstraction d'une valeur inconnue. Une description abstraite des valeurs connues est fournie à l'analyse pour créer un état abstrait initial. L'analyse de temps de liaison utilise cet état pour traiter le programme, annotant chaque construction du programme avec un temps de liaison. Les constructions qui ne dépendent que de valeurs statiques sont considérées comme statiques, tandis que celles qui dépendent d'une valeur dynamique sont considérées comme dynamiques.

Considérons à nouveau l'exemple du produit scalaire afin de voir comment une analyse de temps de liaison l'annote. La description abstraite des paramètres d'entrée spécifie que la variable `size` et le vecteur `u[]` sont statiques et que le vecteur `v[]` est dynamique. Le programme annoté par des temps de liaison est :

```
int dotproduct(int size, int u[], int v[])
{
  int i;
  int res;

  res = 0;
  for (i = 0; i < size; i++)
    res = res + u[i] * v[i];
  return res;
}
```

Le texte en *italique* indique que la construction est statique et le texte en **gras** indique qu'elle est dynamique. Puisque des constructions statiques comme l'opération sur l'indice de boucle *i = 0* et l'indexation *u[i]*, ne dépendent que de valeurs statiques, elle peuvent être réalisées durant la spécialisation. Les constructions dynamiques, telle que la boucle **for** ou les affectations à la variable **res**, ne peuvent pas être évaluées au moment de la spécialisation et doivent par conséquent être présentes dans le programme résiduel. Dans cet exemple, la première affectation à la

variable **res** est résidualisée afin de permettre l'utilisation de **res** dans la deuxième affectation.

Spécialisation

En partant du programme annoté par les temps de liaison ci-dessus, un spécialiseur hors-ligne effectue des transformations basées sur les annotations de temps de liaison. Les constructions qui sont complètement statiques, telles que *u[i]*, sont évaluées. Les constructions complètement dynamiques sont résidualisées, comme l'est **return res**. Les constructions dynamiques qui contiennent des composants statiques sont transformées par l'évaluation des composants statiques et la résidualisation des composants dynamiques. Par exemple, la boucle **loop** est déroulée puisque son test est statique et son corps est résidualisé. L'indexation $\mathbf{v}[i]$ est résidualisée, son indice étant lui exemplarisé par une valeur constante. Ce procédé produit le programme résiduel suivant :

```
int dotproduct_1(int v[])
{
  int res;

  res = 0;
  res = res + 4 * v[0];
  res = res + 8 * v[1];
  res = res + 3 * v[2];
  res = res + 2 * v[3];
  res = res + 9 * v[4];
  return res;
}
```

Le programme spécialisé est semblable à la version en ligne vue à la section 1.2.1. Ici, toutefois, la première affectation à la variable `res` est résidualisée. Comme mentionné plus tôt, l'approche hors-ligne ne produit pas toujours des résultats aussi précis que l'approche en ligne. Dans cet exemple, la variable `res` contient la valeur statique `0` pendant la première itération de la boucle et des valeurs dynamiques lors des suivantes. L'analyse de temps de liaison fait une approximation, en supposant que la variable `res` est dynamique durant toutes les itérations de la boucle. Ceci implique que toutes les utilisations de la variable `res` soient résidualisées, ce qui par la suite oblige l'affectation initiale à `res` à être elle aussi résidualisée. Bien que le programme résiduel soit moins optimal que la version en ligne, la différence n'est pas significative. En fait, compiler les deux programmes avec un compilateur optimisant générerait des programmes exécutables identiques.

Malgré ce désavantage, un évaluateur partiel hors-ligne a un avantage important. Comme les transformations sont calculées à l'avance, l'étape de transformation est plus efficace. Le spécialiseur hors-ligne interprète simplement un programme annoté par des temps de liaison et déclenche des transformations basées sur les annotations.

Extension génératrice

Un spécialiseur encore plus efficace peut être produit en éliminant le sur-coût de l'interprétation du programme annoté. Ceci est obtenu en codant explicitement le flot de contrôle de spécialisation dans le spécialiseur, créant un spécialiseur dédié connu sous le nom d'*extension génératrice* [JGS93a, GJ95]. Alors qu'un spécialiseur hors-ligne prend un programme annoté et des valeurs d'entrée statiques et produit un programme spécialisé, une extension génératrice ne prend que les entrées statiques pour produire le programme spécialisé.

Une extension génératrice peut être automatiquement produite à partir d'un programme annoté de temps de liaison. Par exemple, l'extension génératrice produite à partir du programme du produit scalaire annoté est :

```
int gen_dotproduct_SSD(int size, int u[])
{
  int i = 0;

  printf("int dotproduct_1(int v[])\n");
  printf("{\n");
  printf("  int res;\n");
  printf("  \n");
  printf("  res = 0;\n");

  for(i = 0; i < size; i++)
    printf("  res = res + %d * v[%d];\n", u[i], i);

  printf("  return res;\n");
  printf("}\n");
}
```

Ce programme prend les valeurs d'entrée statiques `size` et `u[]`, et produit directement la fonction spécialisée `dotproduct_1`. Le premier groupe d'instructions `printf` écrit l'en-tête de fonction, ses paramètres formels, le bloc de début puis la déclaration et la définition de la variable `res`. Le dernier groupe d'instructions `printf` écrit l'instruction de retour et le bloc de fin. Ces groupes d'instructions écrivent *toujours* les mêmes lignes de code pour chaque spécialisation, indépendamment des valeurs d'entrée statiques. Par conséquent, les premières et dernières lignes de chaque spécialisation seront nécessairement exactement les mêmes.

La ligne écrite par l'instruction `printf` au milieu, toutefois, change en fonction des valeurs d'entrée statiques `size` et `u[]`. Tout d'abord, puisqu'elle est dans une boucle, la ligne de code est écrite `size` fois. De plus, le contenu de la ligne de code écrite diffère à chaque fois. L'instruction d'affectation générée par l'instruction `printf` contient les valeurs statiques calculées par `u[i]` et `i`, qui sont insérées à leur place adéquate comme indiqué par les deux caractères d'échappement `%d` dans la chaîne de contrôle.

Extension génératrice cible

Tous les spécialiseurs considérés jusqu'à présent produisent des programmes résiduels qui sont écrits dans le même langage que le programme source. Une extension génératrice qui génère du code source, tel que celui défini ci-dessus, s'appelle une *extension génératrice source*. Une *extension génératrice cible*, au contraire, produit des programmes résiduels qui sont écrits dans un langage cible, qui peut être différent de celui du programme source [CN96].

Une extension génératrice cible est nécessaire pour obtenir une spécialisation à l'exécution efficace. Nous avons déjà mentionné que pour la spécialisation à l'exécution, il était crucial de minimiser le temps entre le moment où les valeurs d'entrées connues deviennent disponibles et le temps où le programme spécialisé est prêt à être exécuté. Nous avons déjà vu qu'une extension génératrice minimise le coût de spécialisation. Ici, nous montrons comment utiliser une extension génératrice cible pour minimiser le coût de compilation.

L'aspect clef est de choisir du code objet exécutable comme langage cible de l'extension génératrice cible. Ceci élimine complètement le temps de compilation à l'exécution, puisque le programme résiduel est déjà sous forme exécutable. Une autre

façon de voir ceci est de considérer que l'ordre de spécialisation et de compilation est inversé. Plutôt que de spécialiser et de compiler ensuite, cette approche compile et ensuite spécialise.

Créer automatiquement une extension génératrice cible qui produit des programmes résiduels exécutables implique, à la compilation, de produire les blocs de construction de base à partir desquels les programmes spécialisés sont composés, en même temps que l'extension génératrice cible qui les utilise. Chaque bloc de construction de base, appelé *patron*, est un fragment de code paramétrés par des *trous*. Un trou est un garde-place introduit pour représenter un morceau de code manquant. Ces patrons doivent être écrits dans le langage cible. On peut créer automatiquement des patrons dans le langage cible en construisant les patrons correspondants dans le langage source et en les traduisant en langage cible en utilisant un compilateur.

Pour l'exemple du produit scalaire, il y a trois *patrons*, un pour chacun des trois "groupes d'instructions `printf`" dans l'extension génératrice du temps de compilation. Les patrons source sont :

t_1
```
int dotproduct_1(int u[])
{
  int res;

  res = 0;
```
t_2
```
  res = res + h0 * v[h1];
```
t_3
```
  return res;
}
```

Chaque patron source est simplement la concaténation de chaînes de contrôle `printf` contiguës dans l'extension génératrice source correspondante. Le premier patron source, t_1, représente les premières lignes du programme résidualisé, le patron source t_2 représente la ligne du milieu et le patron source t_3 représente les dernières lignes. Le patron source t_2 contient deux trous, représentant les valeurs statiques qui ne peuvent pas être déterminées avant la spécialisation. Ces trous, représentés ici par h_1 et h_2, correspondent aux caractères d'échappement `%d` dans les chaînes de contrôle dans l'extension génératrice source.

Nous passons maintenant à l'extension génératrice cible elle-même, qui manipule les versions exécutables des ces patrons pour produire un programme résiduel exécutable. L'extension génératrice cible a la même structure que l'extension génératrice source—la différence est que le code cible est généré à la place du code source. À nouveau, les valeurs de trous sont des valeurs statiques calculées pendant la spécialisation. La commande `output_template` prend le nom du patron et les valeurs de trous avec lesquelles le patron doit être exemplarisé et rend le patron cible exemplarisé. L'extension génératrice cible pour le produit scalaire est comme suit :

```
int target_gen_dotproduct_SSD(int size, int u[])
{
  int i;

  output_template(t1);
  for (i = 0; i <size; i++) {
    h0 = u[i];
    h1 = i;
    output_template(t2, h0, h1);
  }
  output_template(t3);
}
```

Dans cet exemple, le patron t_1 est écrit, suivi par une boucle qui écrit et exemplarise le patron t_2 autant de fois que `size` l'indique, suivi par le patron t_3. Les premières et dernières lignes, représentées par les patrons t_1 et t_3 sont les mêmes pour chaque spécialisation. Seule la ligne du milieu, représentée par le patron t_2 change en fonction des valeurs statiques.

Cette approche produit un programme spécialisé qui est déjà exécutable ce qui veut dire qu'il peut immédiatement être appelé. Cette approache est cruciale pour la spécialisation à l'exécution. D'un autre coté, comme les patrons sont compilés avant qu'ils ne soient exemplarisés et composés, ils ne peuvent pas être optimisés en fonction des valeurs statiques réelles qui ne deviendront disponibles qu'au cours de la spécialisation. Par conséquent, les programmes spécialisés résultants peuvent ne pas être aussi optimisés que le même programme spécialisé avant d'être compilé.

1.3 Évaluation partielle et applications

Nous avons maintenant l'idée de base de ce qu'est l'évaluation partielle et de comment elle fonctionne. Notre but est d'appliquer l'évaluation partielle à des programmes réalistes et existants. Dans cette section, nous considérons des évaluateurs partiels existants et expliquons pourquoi ils ne sont pas capables de traiter efficacement de tels programmes. Nous montrons les défauts spécifiques de ces systèmes et expliquons comment les nouvelles analyses que nous introduisons remédient à ces carences.

Avant de détailler ces sujets, décrivons ce que nous entendons par programme *réaliste*. Tout d'abord, le programme doit faire quelque chose d'utile, comme résoudre un problème réel ou rendre un service qui intéresse réellement les gens. Ceci exclut les exemples "jouets". De plus, nous excluons les programmes qui ne sont d'intérêt que pour la communauté de l'évaluation partielle. Beaucoup de résultats prometteurs ont été réalisés dans ce domaine. Le but est de voir si ces résultats sont applicables dans d'autres domaines. Par conséquent, nous considérons des programmes issus d'autres domaines.

Expliquons aussi pourquoi nous nous intéressons à des applications *existantes*. Si des systèmes existants n'ont pas à être totalement réécrits, les techniques de spécialisation peuvent être plus largement utilisées et les chercheurs d'autres domaines peuvent bénéficier de la technologie de la spécialisation. Par exemple, même si des opportunités de spécialisation sont identifiées, il est improbable que des programmeurs vont les exploiter si cela requière de re-mettre en œuvre leur système dans un autre langage. Deuxièmement, comparer un programme existant et non-spécialisé avec sa version spécialisée donne une comparaison équitable pour établir l'intérêt réel de la spécialisation. Si la performance d'un programme existant est de toute

première importance, le programme sera déjà de quelque manière optimisé. Par conséquent, les accélérations dues à la spécialisation ne viendront pas du simple remplacement de quelques calculs évidents et triviaux.

Avec ceci à l'esprit, regardons maintenant les évaluateurs partiels existants.

1.3.1 Évaluateurs partiels existants

Un centre d'intérêt précoce et prédominant de la recherche en évaluation partielle était l'auto-application et la génération de compilateurs [AK82, CK91, GJ89, JS80, JGS93a, Jør92]. Ces systèmes incluent les approches en ligne et hors-ligne, et traitent des langages de programmation fonctionnels, impératifs mais aussi logiques. En général, ces systèmes étaient plus de nature exploratoire ou pédagogique et n'avaient pas l'intention de traiter des applications réalistes. Plutôt, les applications consistaient typiquement en des programmes comme des interprètes, des analyseurs ou des programmes de pattern matching.

D'autres évaluateurs partiels ont été conçus pour traiter des applications plus réalistes. FUSE [AWS91, BW90, WCRS91] et Blitzkrieg [BS94, Sur95], par exemple, sont des systèmes en ligne qui présentent des accélérations significatives pour des programmes numériques tels que le problème aux n corps, l'interaction de particules et la simulation de circuit. Ces études révèlent que les opportunités de spécialisation existent pour ces types de programmes. Beaucoup de ces programmes sont indépendants des données, ce qui veut dire que le flot de contrôle du programme ne dépend pas des données en entrée sur lesquelles il opère. Par conséquent, ces programmes peuvent être spécialisés en fonction du flot de contrôle, ce qui n'élimine pas que des constructions comme les boucles mais déclenche aussi des optimisations additionnelles.

Mais ces deux évaluateurs partiels traitent un sous-ensemble de Scheme. Les programmes numériques existants ne sont pas écrits en Scheme. Ils étaient plutôt réécrits en Scheme en vue d'être spécialisés. Les accélérations étaient déterminées en comparant les programmes Scheme non-spécialisés avec des programmes Scheme spécialisés. Dans certains cas, les non-spécialisés n'étaient pas conçus dans le but d'être rapides. Par exemple, un programme numérique de calcul intensif testé était écrit originalement dans un but de compréhension. Par conséquent, le programme, intentionnellement, ne tirait pas avantage de cas particuliers comme l'élévation à la puissance zéro ou un [BW90].

Beaucoup d'applications réalistes sont mises en œuvre dans des langages orientés objet. L'évaluateur partiel Simili a été conçu pour traiter un petit sous-ensemble d'un langage orienté objet, dans le but de développer des techniques que l'on pourrait propager jusqu'à des langages orientés objet existants [MS92]. Les applications visées incluaient les systèmes de fenêtrage et les gestionnaires de fichiers.

Simili est un évaluateur partiel en ligne. Bien que le langage traité soit un sous-ensemble réduit d'un langage orienté objet, la complexité de l'approche en ligne a rendu le spécialiseur inapplicable sur les exemples de grande taille. L'évaluateur partiel en ligne nécessitait de maintenir un trop grand nombre d'informations pendant la spécialisation. Ces observations ont confirmé notre intuition que la spécialisation hors-ligne, dans laquelle cette information supplémentaire est confinée dans l'étape d'analyse, est mieux adaptée aux langages complexes. Simili obtenait, néanmoins, des accélérations raisonnables pour la fonction d'Ackerman et pour un petit interprète de machine de Turing.

Ces exemples montrent que les techniques en ligne existantes qui peuvent traiter des applications réalistes ne savent traiter qu'un langage simple, tandis que les techniques qui traitent un langage plus complexe ne peuvent gérer que des problèmes réduits. Une fois encore, ceci laisse sous-entendre que les techniques hors-ligne sont mieux adaptées pour traiter des applications réalistes écrites dans un langage réa-

liste. Un certain nombre de travaux ont été choisis pour poursuivre dans cette direction.

Les programmes numériques sont généralement écrits en Fortran. Deux évaluateurs partiels hors-ligne traitent Fortran: SFAC [BF93, BF96] et FSpec [BGZ94, GNZ95, KKZG94]. SFAC a été appliqué à des programmes existants et de puissance industrielle, tels que des programmes qui sont utilisés dans des centrales nucléaires ou des satellites de télécommunications. Cependant, le but de SFAC était de créer un outil qui aiderait les utilisateurs à comprendre un système complexe et de grande taille. Pour voir quelles parties d'un programme dépendaient d'une certaine variable d'entrée, l'utilisateur indiquait que la variable d'entrée était *dynamique*. Spécialiser le programme avec cette information résidualiserait toutes les parties du programme dépendant de cette variable d'entrée. Comme ces transformations n'avaient pas d'intention de performance, des bancs de test n'ont jamais été mis en œuvre.

FSpec, d'un autre coté, a été spécifiquement prévu pour améliorer les performances de programmes Fortran. Les applications numériques traitaient des exemples tels que la transformation de Fourier, l'interpolation par spline cubique et l'intégration de fonction, obtenant des accélérations significatives. FSpec, toutefois, ne traite qu'un sous-ensemble de Fortran et les applications traitées étaient relativement petites. Notre but est d'améliorer cette approche pour traiter des exemples plus grands et plus compliqués tels que les programmes de systèmes d'exploitation.

C-Mix [And94a] est un évaluateur partiel hors-ligne avec une approche similaire à celle de FSpec. Toutefois, il traite le langage de programmation C dans son intégralité, incluant les structures de données et les pointeurs. De plus, il utilise des analyses basées sur des contraintes, efficaces, et une extension génératrice pour traiter efficacement des programmes de grande taille et réalistes. Des accélérations ont été obtenues pour une analyse lexicale, un programme numérique, un programme graphique et un programme de modélisation écologique. Ces applications étaient, toutefois, écrites à la main. Comme nos études montrent que les analyses statiques dans C-Mix ne sont pas capables de traiter efficacement du code système, il n'est pas clair dans quelle mesure ces améliorations ont été introduites ou non lors de l'écriture de ces application de façon à permettre au spécialiseur d'atteindre les accélérations énoncées. Néanmoins, ce système apparaît comme le plus prometteur de tous les évaluateurs partiels existants.

1.3.2 Applications systèmes

Bien que la spécialisation automatique de programmes ait obtenu des accélérations significatives pour une grande variété d'applications, aucune de ces applications n'était un programme réaliste et existant. Nous montrons que ces types de programmes peuvent, en fait, être spécialisés avec succès. En particulier, nous appliquons l'évaluation partielle à des programmes existants de systèmes d'exploitation. Ces programmes sont des programmes de grande taille et complexes écrits en C. De plus, ils contiennent des opportunités de spécialisation à la fois au temps de compilation et au temps d'exécution. Par conséquent, nous choisissons une approche hors-ligne, puisqu'elle est mieux adaptée au traitement d'applications complexes et qu'elle permet une spécialisation à l'exécution efficace.

Des études empiriques indiquent que la technologie existante de l'évaluation partielle hors-ligne, telle que FSpec ou C-Mix, n'est pas suffisamment avancée pour spécialiser efficacement des logiciels systèmes. Ceci est du au manque de précision des analyses de temps de liaison quand on gère certaines caractéristiques de base des programmes C, comme les pointeurs ou les structures de données, et aussi quand on traite des exemples de programmes compliqués comme on en trouve dans les programmes systèmes. En particulier, nous avons trouvé que la *sensibilité* au *flot*, au *contexte*, au *retour* et à l'*utilisation* sont nécessaires à l'analyse de temps

de liaison [HN97, HNC96]. Essayons de caractériser ces programmes basés sur nos études, et montrons alors comment ces fonctionnalités d'analyse sont nécessaires pour traiter efficacement des applications systèmes.

Applications systèmes

Les applications systèmes sont des applications de grande taille fournissant une très grande variété de services. Nous avons identifié deux aspects communs à la plupart des applications systèmes : la généralité et la modularité. La généralité permet à un système de fournir de nombreux services et la modularité est cruciale pour structurer et gérer la complexité d'un système.

La *généralité* est importante car elle permet la réutilisation de code. Dans le contexte des systèmes d'exploitation, les appels à des librairies sont des services génériques qui fournissent un ensemble de fonctionnalités, permettant que nombre de services liés puissent être mis en œuvre par la même fonction. Beaucoup de composants de systèmes différents peuvent ainsi utiliser les mêmes fonctions de librairies. Par conséquent, ces fonctions sont hautement paramétrées, fournissant des fonctionnalités génériques qui peuvent être exemplarisées selon les besoins de chaque composant individuel.

La généralité est aussi le résultat d'interfaces strictes et contraignantes. Par exemple, à la fois pour des raisons de sécurité et de simplicité de conception, il est important de minimiser l'interface entre les processus des usagers et le système d'exploitation. Ceci est fait en minimisant le nombre d'appels systèmes disponibles à l'usager. De façon à fournir tous les services nécessités avec un nombre minimum de fonctions, quelques appels systèmes généraux sont fournis.

Créer un système *modulaire* a un certain nombre d'avantages logiciels. Un module fournit un service indépendant et bien défini. La capacité à composer des modules permet de créer facilement des services plus complexes. Par exemple, un système monolithique peut être organisé en des sous-systèmes séparés. Ces sous-systèmes sont des modules qui peuvent être composés horizontalement pour former un système complet. Les modules peuvent aussi être composés verticalement, comme dans le cas de couches logicielles mettant en œuvre des protocoles.

Les différents composants d'un système ont besoin de communiquer avec les autres composants systèmes. Cette communication est typiquement réalisée en encapsulant de l'information système utile en une certaine forme d'état système. L'exécution de chaque composant est guidée par l'état. Cet état est passé de composant en composant—chaque composant accède et modifie les valeurs d'état pertinentes à ce composant. Enfin, le contenu de l'état guide l'exécution globale du système.

Une autre façon de communiquer pour un module est de fournir une sorte de feed-back à son appelant. Une technique courante pour fournir ce feed-back est de retourner une sorte de statut d'erreur, indiquant si la fonctionnalité désirée s'est terminée avec succès ou si une erreur est intervenue et a causé l'arrêt prématuré de l'exécution. Ces valeurs de statut d'erreur sont rétro-propagées à travers de multiples fonctions et sont éventuellement utilisées par la fonction qui a initié l'appel pour décider de la marche à suivre.

Opportunités de spécialisation

Les programmes systèmes contiennent de nombreuses opportunités de spécialisation. Ces opportunités proviennent des nécessaires généralité et modularité du système, qui introduisent des valeurs connues de plusieurs façons.

Nous avons vu qu'il y a beaucoup de fonctions génériques paramétrées, telles que les appels aux librairies et les appels au système, qui sont utilisés dans une large variété de contextes. Chacun de ces contextes contient typiquement un nombre

de valeurs connues. La spécialisation permet que ces valeurs connues et toutes les communications qui en dépendent soient éliminées. Par exemple, une fonction de librairie qui copie de la mémoire prend un vecteur de chaînes de caractères mais est souvent appelé avec un vecteur de taille un [VMC96a].

La composition de modules crée aussi des opportunités de spécialisation. Une fois que l'ensemble des modules est composé, soit verticalement soit horizontalement, l'usage spécifique d'un module général devient connu. Ceci introduit des valeurs connues qui peuvent être exploitées par un évaluateur partiel. En éliminant le surcoût introduit par l'interface, la composition de modules peut être rationalisée pour ne plus contenir que les calculs utiles. Dans un appel de procédure à distance (Remote Procedure Call en anglais), par exemple, la couche Internet Protocol peut supporter à la fois UDP et TCP. Dans les cas où UDP est utilisé, les tampons de retransmission peuvent être éliminés [VMC96a].

Pendant l'exécution typique d'un programme système, l'information communiquée entre les modules contient aussi beaucoup de valeurs connues. L'état, qui pilote le comportement du système, contient typiquement à la fois des valeurs connues qui permettent aux modules d'être spécialisés. Un état système de fichier, par exemple, peut contenir de l'information spécifiant que les accès au fichier sont exclusifs et séquentiels. Savoir ces valeurs permet d'optimiser la détection de fin de fichier ainsi que le nombre de tâches à éliminer [PAB+95]. Le feed-back par lequel les modules remontent des valeurs d'erreur statiques est aussi connu pendant la spécialisation dans certains cas.

1.3.3 Analyses hors-ligne appliquées aux applications systèmes

Maintenant que nous avons une idée des types de programmes que nous voulons traiter, regardons les analyses utilisées pour traiter ces programmes. Les analyses de temps de liaison existantes, telles que celles utilisées dans FSpec ou C-Mix, ne sont pas assez précises pour exploiter toutes les opportunités de spécialisation qui existent dans les programmes systèmes. Ceci est du au fait que ces analyses ne sont pas sensibles au flot, au contexte, au retour ou à l'utilisation. Pour chacune de ces fonctionnalités, nous expliquons d'abord comment une analyse ne possédant pas cette fonctionnalité perd de la précision et donc comment une analyse la possédant n'en perd pas. Nous donnons alors des exemples spécifiques de la nécessité de ces fonctionnalités pour traiter du code de système d'exploitation.

Sensibilité au flot

Une analyse de temps de liaison insensible au flot n'a qu'un état de temps de liaison global pour tout le programme, connu sous le nom de division de temps de liaison. Si une variable se voit affecter une valeur dynamique quelque part dans le programme, elle est alors considérée comme dynamique partout.

La sensibilité au flot, cependant, calcule une description d'analyse différente pour chaque instruction de mise à jour [HN97]. Cela permet à une variable à qui on a affecté une variable statique d'être considérée comme statique, même si la variable est auparavant ou ultérieurement affectée avec une valeur dynamique. Le besoin de structurer correctement le programme pour obtenir une bonne division de temps de liaison, comme c'était le cas auparavant, est donc éliminé.

Le problème majeur du traitement des programmes systèmes avec une analyse insensible au flot est que la division de temps de liaison est une division globale. Une division de temps de liaison globale est suffisante pour de petites fonctions, qui peuvent ne contenir que quelques variables ou ne définir chaque variable qu'une fois. Mais cela ne suffit pas pour des programmes de grandes taille ou compliqués.

Inévitablement des valeurs dynamiques se répandent dans la division et causent une perte globale. Avec une analyse insensible au flot, la seule façon d'éviter cette perte est d'écrire précautionneusement l'application en gardant la division de temps de liaison à l'esprit. Une analyse sensible au flot retire cette restriction et permet de traiter des programmes existants et réalistes.

Sensibilité au contexte

Dans une analyse insensible au contexte, les fonctions sont analysées en fonction d'un unique contexte d'appel de temps de liaison, qui est une approximation de tous les contextes d'appels. Si un contexte d'appel contient beaucoup de valeurs statiques et un autre seulement peu, les deux appels utiliseront une fonction analysée en fonction du contexte qui a peu de valeurs statiques. Les analyses qui analysent des fonctions selon un seul contexte d'appel sont aussi dénommées *monovariantes* [Hen91, JGS93b].

D'un autre coté, la sensibilité au contexte permet que des fonctions soient analysées en fonction de l'état spécifique de chaque appel, permettant aux différentes valeurs statiques dans chaque d'être exploitées par chaque appel [HN97]. Même si les contextes d'appels contiennent un différent nombre de valeurs statiques, chaque appel peut exploiter les valeurs statiques spécifiques de son contexte spécifique. Les analyses qui analysent des fonctions selon plusieurs contextes d'appels sont aussi dénommées *polyvariantes* [Con93c, JGS93b].

Nous avons vu que le code système contient beaucoup de fonctions génériques paramétrées, appelées par une variété de composants systèmes dans un grand intervalle de contextes d'appels. Ces contextes contiennent des valeurs statiques qui peuvent être exploitées par un évaluateur partiel. Mais une analyse insensible au contexte ne peut pas exploiter pleinement toutes les valeurs statiques. Au contraire, une valeur n'est considérée statique que si elle l'est dans tous les contextes d'appels. Une analyse sensible au contexte, à l'opposé, est capable d'exploiter toutes les valeurs statiques.

Sensibilité au retour

Une analyse insensible au retour calcule la même valeur de temps de liaison pour la valeur retournée et les effets de bords d'une fonction. Si la fonction a une valeur de retour statique mais contient des effets de bords dynamiques, la valeur de retour est considérée comme dynamique.

La sensibilité au retour permet que la valeur de retour d'une fonction ait une description d'analyse différente de celle des effets de bord de la fonction [HN97]. Si une fonction a une valeur de retour statique mais des effets de bords dynamiques, la valeur de retour est toujours considérée comme statique.

Les fonctions communiquent et fournissent du feed-back en rapportant des valeurs de statut d'erreur. Durant la spécialisation, quelques unes de ces valeurs de statut deviennent statiques. Puisque ces valeurs sont souvent retournées par une fonction qui effectue des effets de bord, une analyse insensible au retour considère aussi cette valeur de retour comme dynamique. Une analyse sensible au retour permet que cette valeur de retour soit considérée statique.

Sensibilité à l'utilisation

De façon à voir pourquoi la sensibilité à l'utilisation est nécessaire, il faut d'abord comprendre comment le contexte d'utilisation d'une variable joue sur son temps de liaison. Une valeur statique dans un contexte dynamique est évaluée pendant la spécialisation et la valeur résultante est convertie, ou *réifiée*, en sa représentation

textuelle. Les valeurs pour lesquelles il existe une représentation textuelle correspondante, comme les entiers, peuvent être réifiées, mais les valeurs qui n'ont pas de représentation textuelle, telles que les pointeurs, les structures et les tableaux, ne peuvent pas être réifiées. Par conséquent, l'utilisation d'une variable statique non-réifiable dans un contexte dynamique doit être dynamique.

Une analyse insensible à l'utilisation force toutes les utilisations de la variable à avoir le même temps de liaison. Par exemple, supposons qu'une variable se voit affecter une valeur statique et qu'elle soit ensuite utilisée plusieurs fois. Si *un quelconque* des contextes est dynamique, alors *toutes* les utilisations deviennent dynamiques. Ceci force inutilement les utilisations qui apparaissent dans des contextes statiques à être considérées dynamiques.

La sensibilité à l'utilisation empêche une utilisation de variable qui *doit* être résidualisée d'interférer avec les autres utilisations qui *pourraient* être évaluées [HNC96]. Par conséquent, même si une utilisation de variable devient dynamique de par son contexte dynamique, les autres utilisations de la variable dans les contextes statiques restent statiques.

La sensibilité à l'utilisation est plus précise que la sensibilité au flot. La sensibilité au flot associe une valeur de temps de liaison différente à une variable à chaque fois qu'elle est affectée. Pour chaque affectation, toutefois, une variable peut être utilisée plusieurs fois. La sensibilité à l'utilisation associe une valeur de temps de liaison différente à chacune de ces utilisation.

Comme nous l'avons indiqué plus tôt, la communication à l'intérieur des modules et entre les modules fournit aussi des opportunités de spécialisation. L'état système est habituellement mis en œuvre par des structures de données complexes qui combinent des valeurs telles que des pointeurs, des structures et des tableaux. Comme un état système est habituellement partiellement statique, il est utilisé dans des calculs statiques comme dans des calculs dynamiques. Dans des analyses insensibles à l'utilisation, si une valeur non-réifiable est utilisée dans un contexte dynamique, toutes ses utilisations deviennent dynamiques, indépendamment de leurs contextes. Par conséquent, l'état global est considéré comme dynamique et aucune valeur statique n'est exploitée. Une analyse sensible à l'utilisation permet d'accéder avec succès aux parties statiques de l'état système partiellement statique.

1.4 Résumé

Ce travail démontre que l'évaluation partielle est une technique utile pour l'optimisation de programmes du monde réel. Nous choisissons une approche hors-ligne, car elle est mieux adaptée pour exploiter ces opportunités. Les analyses statiques utilisent des informations globales pour pré-calculer les transformations de programmes appliquées durant la spécialisation, ce qui est nécessaire pour aborder les complexités des programmes existants et écrits dans un langage réaliste. De plus, l'approche hors-ligne permet une spécialisation à l'exécution rapide, car l'information de temps de liaison permet de générer une extension génératrice cible automatiquement.

Nous avons décrit les opportunités de spécialisation qui existent dans les programmes systèmes et avons montré pourquoi les analyses existantes ne sont pas capables d'exploiter pleinement ces opportunités. Dans le chapitre 2 nous expliquons en détail la notion de flot, de contexte, de retour et de sensibilité à l'utilisation. Nous introduisons aussi notre approche de l'analyse de temps de liaison en deux phases. La première phase de notre approche comporte la phase de temps de liaison, qui détermine si une construction est statique au moment de la spécialisation. Cette phase est présentée dans le chapitre 3. La deuxième phase, dite phase de transformation, détermine si une construction doit être évaluée ou résidualisée. Elle est présentée

dans le chapitre 4. Ces analyses ont été mises en œuvre et intégrées à Tempo, notre évaluateur partiel pour C, comme le montre le chapitre 5. Le Chapitre 6 présente les résultats de l'application de Tempo à un ensemble d'applications, incluant un système d'exploitation commercial. Nous terminons par des remarques concernant ce travail dans le chapitre 8.

Chapitre 2

Caractéristiques des analyses

Dans le chapitre 1 nous avons exploré les techniques de spécialisation existantes et montré pourquoi elles étaient inadéquates pour traiter des programmes existants et réalistes. Notre étude empirique de programmes systèmes a démontré le besoin d'incorporer de nouvelles fonctionnalités telles que la sensibilité au flot, au contexte, au retour et à l'utilisation dans les analyses de spécialisation. Une brève description de chacune de ces fonctionnalités a déjà été donnée.

Dans ce chapitre, nous clarifions plus en avant chaque fonctionnalité en donnant des exemples spécifiques dans lesquels le comportement des programmes systèmes est représenté. Pour chaque exemple, nous présentons d'abord le programme source initial, les résultats de chaque phase de notre analyse et le programme spécialisé résultant. Dans certains cas, nous montrons aussi comment la spécialisation avec des analyses moins précises produit un programme pauvrement spécialisé. Comme la sensibilité à l'utilisation est une nouvelle fonctionnalité, nous fournissons aussi une explication sur la façon de calculer les annotations de sensibilité à l'utilisation.

2.1 Analyse de temps de liaison en deux phases

Faisons tout d'abord un bref survol de nos analyses de façon à comprendre comment elles traitent les exemples spécifiques. C'est important puisque notre approche coupe l'analyse de temps de liaison traditionnelle en deux phases.

L'analyse de temps de liaison détermine un temps de liaison pour chaque construction. Le temps de liaison détermine directement la transformation utilisée pour la construction durant la spécialisation. Les constructions statiques sont évaluées alors que les constructions dynamiques sont résidualisées. Calculer ce temps de liaison nécessite que l'analyse de temps de liaison réalise deux tâches différentes. Tout d'abord, les constructions sont annotées comme statiques si elles ne dépendent que de valeurs statiques. Cette tâche peut être vue comme la détermination de la *disponibilité* des valeurs dont dépend une construction. Dans certains cas, tels que le traitement d'un sous-ensemble réduit d'un langage fonctionnel, réaliser cette tâche est suffisant. Le spécialiseur peut évaluer les constructions statiques et résidualiser les constructions dynamiques.

Quand on s'intéresse à des programmes ou des langages de programmation plus compliqués, réaliser cette première tâche n'est pas suffisant. Ceci parce que simplement évaluer les constructions statiques et résidualiser les constructions dynamiques durant la spécialisation produit un programme résiduel qui est ou sous-optimal ou même incorrect. Il en découle qu'il est préférable, et même nécessaire dans certains cas, que des constructions qui ne dépendent que d'information statique et par conséquent *pourrait* être calculées au temps de spécialisation soient résidualisées.

C'est la deuxième tâche de l'analyse de temps de liaison : identifier les constructions statiques qui devraient être résidualisées et les obliger à être dynamiques. Cela veut dire que certaines constructions seront annotées comme dynamiques même si elles ne dépendent que de valeurs statiques.

Notre approche de l'analyse de temps de liaison sépare explicitement ces deux tâches en deux phases distinctes. La première phase, que nous appellerons *phase de temps de liaison*, détermine simplement si une construction dépend ou non de valeurs statiques. À ce point, toutes les constructions qui ne dépendent que de valeurs statiques sont annotées comme statiques. La deuxième phase prend alors en compte comment le programme devrait être spécialisé et annote en conséquence le programme. Pour distinguer cet aspect de l'analyse de temps de liaison, nous appelons cette deuxième étape la *phase de transformation* ; cela décrit sa fonctionnalité. De plus, la phase de transformation annote le programme avec des annotations différentes, une fois encore pour distinguer ces annotations de celles de temps de liaison. Les constructions qui sont évaluées sont annotées *evaluate* et celles résidualisées sont annotées *residualize*.

Regardons maintenant des exemples qui illustre la sensibilité au flot, au contexte, au retour et à l'utilisation. Pour chaque exemple, nous présentons d'abord le programme source initial. Puis nous donnons le programme annoté par la phase de temps de liaison, où les constructions statiques sont en *italique* et les constructions dynamiques en **gras**. Souvenons-nous, cette information reflète uniquement la disponibilité de l'information au temps de spécialisation. Puis, nous donnons le programme annoté par les transformations, où les constructions en *italique* doivent être évaluées et celles en **gras** doivent être résidualisées. Ces annotations indiquent directement comment chaque construction est transformée au temps de spécialisation. Enfin, nous montrons le programme spécialisé résultant de l'exécution de ces transformations.

2.2 Sensibilité au flot

Pour les langages impératifs, les instructions d'affectation mettent à jour la mémoire. Nous appelons la mémoire abstraite manipulée par une analyse statique un *état*. L'évolution de cet état est déterminée par les opérations de mise à jour réalisées à chaque affectation. Une analyse sensible au flot associe un état d'analyse différent à chacune de ces affectations. Cela permet à des variables à différents points de programme d'avoir différentes descriptions d'analyse. À la fois la phase de temps de liaison et la phase de transformation sont sensibles au flot.

Dans l'exemple de la Fig. 2.1, la fonction `f()` contient une séquence d'affectations dans laquelle la variable `x` est lue et écrite plusieurs fois. Cette fonction est analysée avec un état initial de temps de liaison spécifiant que les paramètres `x`, `y` et `p` sont tous statiques et que la variable globale `d` est dynamique.

Les annotations de temps de liaison montrent comment les affectations mettent à jour l'état de temps de liaison et comment les variables y accèdent. La première affectation est statique, ce qui oblige `x` à rester statique dans l'état. La deuxième affectation est dynamique, puisque le membre droit contient une utilisation de la variable `d` qui est dynamique. Cela rend `x` dynamique dans l'état, ce qui à son tour entraîne le caractère dynamique de la prochaine utilisation de `x`. En associant un état à chaque point de programme et en le mettant à jour selon les temps de liaison des affectations de cette façon, une analyse de temps de liaison sensible au flot est capable de considérer `x` comme statique à certains points de programme et dynamique à d'autres.

Les occurrences de `x` en membre gauche d'une affectation sont toutes considérées comme statiques, même si `x` est dynamique. Ceci est dû au fait que l'état de temps

Code source

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = x + d;
  x = x + y;
  ⋮                /* p = &x or &y */
  *p = d;
  x = x + y;
}
```

Code annoté de temps de liaison

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = x + d;
  x = x + y;
  ⋮
  *p = d;          /* alias: p → { x, y } */
  x = x + y;
}
```

Code annoté de transformation

```
int d;
void f(int x, int y, int *p)
{
  x = x + y;
  x = x + d;
  x = x + y;
  ⋮
  *p = d;          /* alias: p → {x, y} */
  x = x + y;
}
```

Code spécialisé (avec x = 2, y = 3)

```
int d;
void f_1(int x, int y, int *p)
{
  x = 5 + d;
  x = x + 3;
  ⋮
  *p = d;
  x = x + y;
}
```

FIG. 2.1 – *Sensibilité au flot.*

de liaison contient un temps de liaison pour le *contenu* de la variable `x`, pas pour son *adresse*. L'adresse de la variable est toujours connu au temps de spécialisation et par conséquent est toujours considéré comme statique, quelle que soit sa description de temps de liaison dans l'état.

La phase de transformation utilise alors ces annotations de temps de liaison pour déterminer la transformation pour chaque construction. La plupart des constructions statiques deviennent des constructions annotées evaluate et des constructions dynamiques deviennent des constructions annotées residualize—mais il y a quelques exceptions. Tout d'abord, les membres gauches statiques d'affectations dynamiques sont annotés residualize. Une fois encore, c'est parce que calculer l'adresse de la variable au temps de spécialisation et réifier la valeur résultante crée un programme résiduel invalide. Par conséquent, la phase de transformation annote ces affectations par residualize, ordonnant au spécialiseur de résidualiser les identificateurs de variables. Deuxièmement, même si tous les paramètres ont un temps de liaison statique, ils sont tous annotés par une opération residualize. Ceci est dû au fait que ces paramètres doivent apparaître dans le programme résiduel puisque les trois variables, `x`, `y` et `p`, ont des utilisations qui sont résidualisées.

Cet exemple illustre aussi quelques autres aspects des analyses. En général, les pointeurs et la synonymie (aliasing en anglais) peuvent créer des définitions *ambiguës*: affectations pour lesquelles l'analyse ne peut déterminer statiquement quel emplacement mémoire sera modifié à l'exécution. Dans l'exemple, nous supposons que les lignes omises (représentées par ⋮) contient du code à l'issue duquel le pointeur `p` peut pointer soit sur la variable `x` soit sur la variable `y`. Cela implique que la définition suivante de `*p` est une affectation ambiguë (les alias annotés de temps de liaison ou de transformation apparaissent en commentaires à coté de l'expression d'indirection de pointeur). Comme cette affectation est dynamique, les deux emplacements mémoire deviennent dynamiques.

La phase ultérieure de spécialisation est guidée par les annotations de transformation. Les constructions annotées evaluate sont évaluées et constructions annotées residualize sont résidualisées. Les instructions annotées evaluate disparaissent complètement. Les expressions annotées evaluate sont évaluées et la valeur résultante est réifiée dans le code résiduel. Les expressions et les instructions annotées residualize sont résidualisées.

2.3 Sensibilité au contexte

La sensibilité au contexte permet qu'une fonction soit analysée en fonction des différentes descriptions d'analyse, ou *contextes*, produisant un exemplaire annoté de la fonction pour chaque contexte. Un contexte contient l'information pertinente pour analyser la fonction, c'est-à-dire les temps de liaison des paramètres formels mais aussi les temps de liaison des variables non-locales (*e.g.*, une variable globale) utilisées par la fonction. Comme les exemplaires annotées sont traités séparément, l'analyse est capable d'exploiter les valeurs statiques de chaque contexte spécifique.

Le deuxième exemple montre une fonction `f()` qui contient une séquence d'appels à une fonction `g()`, comme on le voit sur la Fig. 2.2. La fonction `f()` est analysée avec une description initiale de temps de liaison spécifiant que la variable globale `x` est statique, alors que les variables globales `y` et `d` sont dynamiques. Le contexte de temps de liaison du premier appel consiste en un paramètre effectif statique, une variable non-locale statique `x` et une variable non-locale dynamique `y` (les contextes de temps de liaison et de transformation apparaissent en commentaires à coté des appels de fonctions). Le premier exemplaire de la fonction est alors analysé en fonction de ce contexte. Noter que `x` est dynamique après que le premier

Code source

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5);
  g(5);
  g(5);
}
```

```
g(int z)
{
  x = (x + z) + y;
}
```

Code annoté de temps de liaison

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
}
```

```
void g(int z) /* ctx: z, x, y */
{
    x = (x + z) + y;
}
void g(int z) /* ctx: z, x, y */
{
  x = (x + z) + y;
}
```

Code annoté de transformation

```
int x, y, d;
void f()
{
  x = 1;
  y = d;
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
  g(5); /* ctx: z, x, y */
}
```

```
void g(int z) /* ctx: z, x, y */
{
    x = (x + z) + y;
}
void g(int z) /* ctx: z, x, y */
{
    x = (x + z) + y;
}
```

Code spécialisé

```
int y, d;
void f_1()
{
  y = d;
  g_1();
  g_2();
  g_2();
}
```

```
void g_1()
{
  x = 6 + y;
}
void g_2()
{
x = (x + 5) + y;
}
```

FIG. 2.2 – *Sensibilité au contexte.*

exemplaire de `g()` soit analysé. Ceci crée un contexte de temps de liaison différent pour le deuxième appel à `g()`, dans lequel `x` est dynamique. Par conséquent, un second exemplaire de la fonction est créé et annoté selon ce nouveau contexte. Le troisième appel à `g()` a le même contexte de temps de liaison que le second appel, donc aucun nouvel exemplaire n'est créé.

Le programme, y compris les deux exemplaires annotés de temps de liaison de la fonction `g()`, est alors annoté avec les transformations. La phase de transformation est aussi sensible au contexte, ce qui veut dire qu'un exemplaire supplémentaire de chaque fonction est créé pour chaque contexte de transformation additionnel généré pendant l'analyse. Nous ne voyons pas cette situation sur cet exemple, puisque tous les contextes de transformations sont identiques.

Enfin, le programme est spécialisé. Dans le programme résiduel, chaque exemplaire de la fonction `g()` produit une définition de fonction résiduelle différente. Puisque le spécialiseur est aussi sensible au contexte (on parle aussi de *spécialiseur polyvariant*) il est possible que des contextes de spécialisation différents produisent des fonctions résiduelles différentes. Cela n'arrive pas dans cet exemple, puisque les contextes de spécialisation du second appel et du troisième appel à `g()` sont les mêmes (z = 5). Par conséquent, les deux appels partagent la même définition de fonction résiduelle.

2.4 Sensibilité au retour

La sensibilité au retour permet à une fonction de retourner une valeur statique même si la fonction contient des effets de bord dynamiques et est par conséquent résidualisée. Dans le troisième exemple, que l'on voit en Fig. 2.3, la fonction `f()` contient un appel à la fonction `g()`, dont la valeur de retour est utilisée dans des calculs ultérieurs. Puisque la sensibilité au retour est un nouveau concept, nous montrons d'abord comment une analyse de temps de liaison insensible au retour aurait annoté ce même exemple de la Fig. 2.3. La fonction `f()` est analysée avec un état initial de temps de liaison spécifiant que les variables globales `x`, `y` et `d` sont toutes dynamiques. Rappelons qu'une analyse insensible au retour ne calcule qu'un temps de liaison et qu'une transformation pour la fonction entière. Par conséquent, les parties dynamiques de `g()` qui sont résidualisées interdisent que la valeur de retour statique soit évaluée. L'instruction return est résidualisée et l'opération de multiplication n'est pas évaluée.

Une analyse sensible au retour calcule deux temps de liaison différents pour chaque fonction, un pour les effets de bord et un autre pour la valeur de retour, comme le montre la Fig. 2.4. Pour visualiser ces deux temps de liaison dans la définition de la fonction, nous indiquons que la fonction contient des effets de bord dynamiques en annotant l'identificateur **g** comme dynamique et la valeur de retour statique en annotant son type de retour *int* comme statique. Pour visualiser les deux temps de liaison dans le point d'appel de la fonction, l'identificateur ***g*** est annoté comme à la fois statique et dynamique (indiqué par ***italique_et_gras***). La sensibilité au retour permet que la valeur statique retournée par `g()` puisse être utilisée par le point d'appel, ce qui à son tour rend possible l'opération de multiplication qui dépend de cette valeur statique pour être elle-même considérée comme statique.

Les annotations de transformation sont similaires. Dans la définition de la fonction, l'identificateur **g** est annoté par residualize et son type de retour *int* par evaluate. Dans le point d'appel, l'identificateur ***g*** est annoté à la fois par evaluate et residualize.

Le spécialiseur exploite la valeur de retour retournée par `g()` pour évaluer l'opération de multiplication et résidualiser l'appel de manière à résidualiser ses effets de bords. La définition spécialisée de `g()` est une fonction vide—ne retournant plus

Code source

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Code annoté de temps de liaison

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Code annoté de transformation

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Code spécialisé

```
int x, y, d;
void f_1()
{
  x = (g_1() * 2) + d;
}
```

```
int g_1()
{
  y = 1 + d;
  return 4;
}
```

FIG. 2.3 – *Insensibilité au retour.*

Code source

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Code annoté de temps de liaison

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Code annoté de transformation

```
int x, y, d;
void f()
{
  x = (g(1) * 2) + d;
}
```

```
int g(int z)
{
  y = z + d;
  return (z + 3);
}
```

Code spécialisé

```
int x, y, d;
void f_1()
{
  g_1();
  x = 8 + d;
}
```

```
void g_1()
{
  y = 1 + d;
}
```

FIG. 2.4 – *Sensibilité au retour.*

une valeur.

2.5 Sensibilité à l'utilisation

La sensibilité à l'utilisation aborde la manière complexe avec laquelle les programmes systèmes utilisent les pointeurs et les structures de données. Cette situation apparaît dans beaucoup d'autres programmes réalistes. Le but est d'empécher les utilisations d'une variable qui *doivent* être résidualisées d'interférer avec d'autres utilisations de la variable qui *pourraient* être évaluées.

Ceci n'est pas un problème dans la phase de temps de liaison, puisque toutes les utilisations d'une variable ont le même temps de liaison en un même point de programme. Cela pose un problème, néanmoins, pour la phase de transformation, puisqu'une variable peut être annotée avec différentes transformations, y compris au même point de programme. Ceci parce qu'une transformation de variable dépend de son contexte.

Ceci est un nouveau concept encore jamais exploré: c'est un point subtil mais important. Par conséquent, en plus de fournir des exemples qui démontrent le besoin de la sensibilité à l'utilisation, nous montrons aussi comment sont calculées les annotations de sensibilité à l'utilisation. Ceci devrait aussi clarifier les différences entre la phase de temps de liaison et la phase de transformation de notre analyse, mais aussi motiver l'introduction de nouvelles transformations.

2.5.1 Exemples

Considérons deux exemples différents qui illustrent le besoin de sensibilité à l'utilisation, un qui traite de pointeurs, l'autre de structures. Ces exemples sont basés sur des programmes systèmes existants que nous avons étudiés. Pour chaque exemple, nous montrerons les annotations produites par une analyse insensible à l'utilisation suivies par le même programme annoté par une analyse sensible à l'utilisation.

Dans la Fig. 2.5, une variable pointeur `p` se voit affecter l'adresse d'un tableau et est ultérieurement utilisée deux fois dans la même instruction. Si la fonction `f()` est analysée avec un contexte de temps de liaison qui spécifie que toutes les variables globales sont statiques exceptée la variable `d`, la phase de temps de liaison annote toutes les constructions comme statiques, exceptées celles qui dépendent de `d`. Particulièrement, l'affectation du pointeur `p` ainsi que ses deux utilisations sont annotées comme statiques.

Toutefois, une phase de transformation insensible à l'utilisation annote l'affectation mais aussi les deux utilisations comme devant être résidualisées. Durant la spécialisation, une construction statique qui apparaît dans un contexte dynamique est évaluée et la valeur résultante est réifiée en sa représentation textuelle correspondante qui est alors résidualisée. Quelques valeurs ne peuvent pas être réifiées, toutefois. Par exemple, réifier une valeur de pointeur residualiserait une adresse dans un programme résiduel, ce qui est une transformation illégale. Les valeurs de pointeurs, de structures et de tableaux sont toutes *non-réifiables*. Par conséquent, la phase de transformation oblige les constructions non-réifiables statiques dans des contextes dynamiques à être résidualisées, au lieu d'évaluer la construction et de réifier la valeur résultante.

Dans l'exemple de la Fig. 2.5, nous voyons que la deuxième utilisation du pointeur `p` est statique et apparaît dans un contexte dynamique. Par conséquent, elle est annotée par une transformation residualize. Comme une analyse insensible à l'utilisation requière que toutes les utilisations d'une variable aient un unique temps de liaison, toutes les utilisations du pointeur `p` sont obligées de devenir dynamiques. Résidualiser la première utilisation du pointeur `p` entraîne que l'expression entière

Code source

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Code annoté de temps de liaison

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Code annoté de transformation

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Code spécialisé (avec s = 3)

```
int a[10], *p, d, x;
void f_1()
{
  a[4] = 1234;
  p = a;

  x = *((p + 3) + 1) + *((p + d) + 2);
}
```

FIG. 2.5 – *Insensibilité à l'utilisation (programme avec pointeur).*

Code source

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Code annoté de temps de liaison

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Code annoté de transformation

```
int a[10], *p, s, d, x;
void f()
{
  a[4] = 1234;
  p = a;

  x = *((p + s) + 1) + *((p + d) + 2);
}
```

Code spécialisé (avec s = 3)

```
int a[10], *p, d, x;
void f_1()
{
  a[4] = 1234;
  p = a;

  x = 1234 + *((p + d) + 2);
}
```

FIG. 2.6 – *Sensibilité à l'utilisation (programme avec pointeur).*

Code source

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Code annoté de temps de liaison

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Code annoté de transformation

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Code spécialisé (avec `s1.s` = 10)

```
struct {int s; int d;} s1, s2;
int x;
void f_1()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

FIG. 2.7 – *Insensibilité à l'utilisation (programme avec structure).*

Code source

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Code annoté de temps de liaison

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Code annoté de transformation

```
struct {int s; int d;} s1, s2;
int x;
void f()
{
  s1 = s2;

  x = (s1.s + 3) + (s1.d + 4);
}
```

Code spécialisé (avec `s1.s` = 10)

```
struct {int s; int d;} s1, s2;
int x;
void f_1()
{
  s1 = s2;

  x = 13 + (s1.d + 4);
}
```

FIG. 2.8 – *Sensibilité à l'utilisation (programme avec structure).*

dans laquelle il apparaît soit résidualisée. De plus, la définition du pointeur `p` doit être résidualisée.

Le programme est ultérieurement spécialisé en fonction de ces annotations. À cause de la phase de transformation insensible à l'utilisation, la résidualisation de l'utilisation de `p` dans le contexte dynamique empêche l'utilisation de `p` dans le contexte statique d'être évaluée. Cette situation empêche aussi les expressions d'additions statiques d'être évaluées.

La sensibilité à l'utilisation évite ce problème. Toutes les utilisations d'une variable ne sont plus obligées d'avoir la même transformation en un même point de programme et par conséquent, les utilisations résidualisées n'interfèrent pas avec les utilisations évaluées. Une phase de transformation sensible à l'utilisation par conséquent annote le fragment de programme comme le montre la Fig. 2.6. L'utilisation de `p` dans le contexte statique est évaluée pendant la spécialisation, ce qui à son tour permet que les additions ultérieures soient évaluées.

Un autre exemple qui motive le besoin de la sensibilité à l'utilisation est montré en Fig. 2.7, il contient un exemple typique de comment des structures de données sont utilisées dans les programmes systèmes. Une valeur est affectée à la structure de données `s1` et chacun de ses champs est utilisé dans des calculs.

L'analyse de temps de liaison de cet exemple se fait avec un état de temps de liaison qui spécifie que la variable `x` est dynamique et que les structures de données `s1` et `s2` sont partiellement statiques. En particulier, le champ `s` des structures de données est statique alors que le champ `d` est dynamique. La phase de temps de liaison annote la définition de `s1` comme dynamique, puisqu'un de ses champs contient une donnée dynamique. Il y a deux utilisations ultérieures de `s1`. L'accès au champ statique `s` est considéré comme statique tandis que l'accès au champ `d` est dynamique. Une fois encore, nous trouvons que la structure de données `s1`, qui est considérée comme statique puisqu'elle apparaît en membre gauche d'une affectation, est utilisée à la fois dans un contexte statique et un contexte dynamique.

La phase de transformation doit contraindre l'utilisation statique de `s1` dans le contexte dynamique à être résidualisée, puisque les valeurs de structures ne sont pas réifiables. Une analyse insensible à l'utilisation obligerait par conséquent toutes les utilisations de `s1` à être résidualisées. Les annotations de transformation résultantes et le programme spécialisé sont montrés en Fig. 2.7.

Une analyse sensible à l'utilisation évite de perdre cette information. L'utilisation statique de `s1` dans un contexte dynamique est toujours résidualisée (puisqu'elle n'est pas réifiable), mais l'utilisation statique de `s1` dans le contexte statique est évaluée. Les annotations de transformation et la spécialisation correspondante sont données en Fig. 2.8.

2.5.2 Annotations de calcul

De façon à clarifier l'idée de sensibilité à l'utilisation, il est important de comprendre comment les temps de liaison et les transformations de variables et de leurs définitions sont calculées.

Tout d'abord, le temps de liaison d'une variable est déterminé par l'affectation qui définit la variable, puisque cette affectation détermine les valeurs dont la variable dépend. La variable est statique si toutes les valeurs dont elle dépend sont statiques. Dans ce cas, toutes les utilisations de la variable sont considérées comme statiques, puisqu'elles peuvent être calculées au temps de spécialisation. Si la variable se voit affecter une valeur dynamique, toutes les utilisations définies par cette affectation sont considérées comme dynamiques.

Pour déterminer si les valeurs dont dépend une variable sont statiques ou non, nous utilisons la fonction *values*(), que montre la Fig. 2.9, pour exprimer les valeurs dont une variable dépend et la fonction *values-bt*() détermine si toutes ces valeurs

$$
\begin{array}{rcl}
\mathit{values}(id) & = & \textit{valeur dont dépend } id \\
\mathit{context}^e(id) & = & \textit{valeurs dont dépend le contexte de } id \textit{ (au point de programme } e\textit{)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{values\text{-}bt}(id) & = & \displaystyle\bigsqcup_{bt \,\in\, (\{bt(c) \;\mid\; c \in \mathit{values}(id)\} \,\cup\, \{S\})} bt \\
\mathit{context\text{-}bt}^e(id) & = & \displaystyle\bigsqcup_{bt \,\in\, \{bt(c) \;\mid\; c \in \mathit{context}^e(id)\}} bt
\end{array}
$$

FIG. 2.9 – *Temps de liaison des valeurs et des contextes.*

	temps de liaison	transformations
domaine	$Bt = \{S, D\}$	$Tr = \{\{\}, \{E\}, \{R\}, \{E, R\}\}$
treillis	$\begin{array}{c} D \\ \mid \\ S \end{array}$	$\begin{array}{c} \{E,R\} \\ / \quad \backslash \\ \{E\} \quad \{R\} \\ \backslash \quad / \\ \{\} \end{array}$
majorant	$\bigsqcup$	$\bigcup$
$\overline{\ }: Bt \to Tr$	$\overline{D} = \{R\}$ $\overline{S} = \{E\}$	

FIG. 2.10 – *Temps de liaison et transformations.*

sont statiques. La Fig. 2.10 contient le domaine de temps de liaison $\{S, D\}$, représentant statique et dynamique respectivement, le treillis qui ordonne ces éléments et l'opérateur de majorant $\sqcup$ utilisé pour calculer *values-bt*(). Si une variable ne dépend d'aucune valeur (*e.g.,* une variable qui apparaît comme expression en membre gauche, *values-bt*() retourne statique.

Deuxièmement, les temps de liaison des contextes dans lesquels une variable apparaît sont calculés en utilisant la fonction $\mathit{context}^e()$ pour représenter les valeurs dans un contexte, où l'exposant e indique le point de programme d'une utilisation de variable spécifique et la fonction $\mathit{context\text{-}bt}^e()$ pour exprimer le temps de liaison du contexte. Le contexte est considéré comme statique si toutes les valeurs dans le contexte sont statiques.

Rappelons que les annotations de temps de liaison sont les mêmes pour les exemples de programmes avec pointeur (Fig. 2.5 et 2.6). Le pointeur `p` est défini par une affectation statique et utilisé dans un contexte statique et un contexte dynamique. Cette information est calculée comme suit :

$$\begin{aligned} \mathit{values}(p) &= \{a\} \\ \mathit{context}^1(p) &= \{s, 1\} \\ \mathit{context}^2(p) &= \{d, 2\} \end{aligned}$$

$$\begin{array}{rlll} \mathit{values\text{-}bt}(p) & = bt(a) & = S & = S \\ \mathit{context\text{-}bt}^1(p) & = bt(s) \sqcup bt(1) & = S \sqcup S & = S \\ \mathit{context\text{-}bt}^2(p) & = bt(d) \sqcup bt(2) & = D \sqcup S & = D \end{array}$$

Étant donné l'information de temps de liaison, la phase de transformation détermine comment chaque construction sera transformée. Les quatre transformations possibles sont montrées en Fig. 2.10. Ce nouveau domaine exprime l'ensemble des transformations possibles dont on peut avoir besoin. Les transformations $\{E\}$ et $\{R\}$ sont utilisées pour indiquer que la construction sera évaluée ou résidualisée, respectivement. Dans certains cas, une construction sera à la fois évaluée *et* résidualisée, ce qui est représenté par la transformation $\{E, R\}$. La transformation $\{\}$ correspond à une construction qui n'est pas utilisée, comme du code mort, et par conséquent n'a pas besoin d'être ni évaluée ni résidualisée.

Dans beaucoup de cas, comme vu dans les exemples présentés plus tôt, toutes les constructions dynamiques sont résidualisées et la plupart des constructions statiques sont évaluées. Dans ces cas, la fonction $\overline{}$ (définie en Fig. 2.10) est simplement appliquée au temps de liaison d'une construction pour déterminer sa transformation correspondante.

Toutefois, calculer la transformation d'une construction n'est pas toujours aussi facile. Comme mentionné auparavant, des complications surviennent quand une valeur statique apparaît dans un contexte dynamique. Si la valeur est réifiable, alors elle peut être évaluée et le résultat réifié. Si, cependant, la valeur est non-réifiable, elle ne doit pas être évaluée puisque le résultat ne peut pas être réifié dans le programme résiduel. Par conséquent, même si la construction est statique, elle doit être résidualisée. Déterminer cette transformation prend en compte non seulement le temps de liaison d'une variable mais aussi le temps de liaison de son contexte.

La Fig. 2.11 définit la fonction $\mathit{use\text{-}tr}^e(id)$ qui calcule la transformation pour chaque utilisation d'une variable et la fonction $\mathit{def\text{-}tr}(id)$ pour sa définition. Pour une variable dont la valeur peut être réifiée, ces transformations ne dépendent que du temps de liaison donné par *values-bt()*. Une variable statique sera évaluée alors qu'une variable dynamique sera résidualisée. Une variable statique dans un contexte dynamique sera évaluée et la valeur résultante sera réifiée. Pour les valeurs non-réifiables, cependant, la transformation est calculée en prenant le majorant des temps de liaison calculés par *values-bt()* et le majorant de tous les temps de liaison calculés par *context-bt()*. Ce calcul est fait de façon à résidualiser les variables statiques dans les contextes dynamiques.

Cependant, l'analyse insensible à l'utilisation calcule une transformation pour toutes les utilisations. Tous les temps de liaison des contextes sont fusionnés, donc un contexte dynamique oblige toutes les utilisations à être considérées comme dynamiques. De même, cette même transformation est alors utilisée pour la définition de la variable.

En regardant les annotations de transformation du premier exemple (Fig. 2.5), on voit que la définition du pointeur `p` et toutes se utilisations sont annotées par residualize. Ces annotations sont calculées comme suit :

		valeur réifiable	valeur non-réifiable
use-insensitive	$use\text{-}tr^{e}(id)$	$\overline{values\text{-}bt(id)}$	$\overline{values\text{-}bt(id) \sqcup \bigsqcup_{e1 \in uses(id)} context\text{-}bt^{e1}(id)}$
	$def\text{-}tr(id)$	$use\text{-}tr^{e \in uses(id)}(id)$	$use\text{-}tr^{e \in uses(id)}(id)$
use-sensitive	$use\text{-}tr^{e}(id)$	$\overline{values\text{-}bt(id)}$	$\overline{values\text{-}bt(id) \sqcup context\text{-}bt^{e}(id)}$
	$def\text{-}tr(id)$	$use\text{-}tr^{e \in uses(id)}(id)$	$\bigcup_{e1 \in uses(id)} use\text{-}tr^{e1}(id)$

FIG. 2.11 – *Calculer les transformations des utilisations et des définitions.*

$$use\text{-}tr^{1}(p) = \overline{values\text{-}bt(p) \sqcup \bigsqcup_{e1 \in \{1,2\}} context\text{-}bt^{1}(p)} = \overline{S \sqcup (D \sqcup S)} = \overline{D} = \{R\}$$

$$use\text{-}tr^{2}(p) = \overline{values\text{-}bt(p) \sqcup \bigsqcup_{e1 \in \{1,2\}} context\text{-}bt^{1}(p)} = \overline{S \sqcup (D \sqcup S)} = \overline{D} = \{R\}$$

$$def\text{-}tr(p) = use\text{-}tr^{e \in \{1,\ 2\}}(p) = \{R\}$$

Le pointeur p dépend d'une seule valeur statique et apparaît dans un contexte statique et un contexte dynamique. La transformation pour cette première utilisation, calculée par $use\text{-}tr^{1}(p)$, est le majorant de ces temps de liaison. Puisque le majorant est dynamique, la transformation correspondante est $\{R\}$. La transformation pour sa deuxième utilisation est calculée de façon similaire par $use\text{-}tr^{2}(p)$. La définition de p est annotée avec la transformation calculée par $def\text{-}tr(p)$. Puisque les deux utilisations ont nécessairement la même transformation, $\{R\}$, l'une ou l'autre des transformations peut être utilisée.

Le but d'une analyse sensible à l'utilisation est de permettre que des utilisations différentes aient des transformations différentes, ce qui est réalisé par les fonctions sensibles à l'utilisation $use\text{-}tr(id)$ et $def\text{-}tr(id)$ définies en Fig. 2.11. Les transformations des utilisations sont combinées d'une nouvelle manière pour calculer la transformation de la définition, utilisant le nouveau treillis de transformations et l'opération de majorant définie en Fig. 2.10. Comme les transformations sont calcu-

lées séparément pour chaque utilisation, nous distinguons les différentes utilisations en indiçant *use-tr()* avec le point de programme (e) correspondant à l'utilisation. Pour une variable dont la valeur peut être réifiée, les transformations des utilisations et de définition sont calculées comme dans une analyse insensible à l'utilisation. Pour les valeurs non-réifiables, cependant, la transformation de chaque utilisation ne prend en compte que le contexte spécifique dans lequel la variable est utilisée. C'est ici que la sensibilité à l'utilisation est obtenue. Comme des utilisations différentes de la même variable peuvent désormais avoir des transformations différentes, la transformation pour la définition correspondante doit prendre ceci en compte. C'est le dessein de la transformation $\{E, R\}$.

Revenons sur le premier exemple (Fig. 2.6), nous voyons que la phase de transformation sensible à l'utilisation annote effectivement l'utilisation statique du pointeur `p` par evaluate et l'utilisation statique de `p` dans le contexte dynamique par residualize. De plus, sa définition est annotée par evaluate et residualize, (indiqué par ***italique_et_gras***). Ceci est calculé comme suit :

$$\begin{array}{rlllll}
\textit{use-tr}^1(p) & = \overline{\textit{values-bt}(p) \sqcup \textit{context-bt}^1(p)} & = \overline{S \sqcup D} & = \overline{D} & = \{R\} \\
\textit{use-tr}^2(p) & = \overline{\textit{values-bt}(p) \sqcup \textit{context-bt}^2(p)} & = \overline{S \sqcup S} & = \overline{S} & = \{E\} \\
\textit{def-tr}(p) & = \bigcup_{e1 \in \{1,2\}} \textit{use-tr}^1(p) & = \overline{D} \cup \overline{S} & = \{R\} \cup \{E\} & = \{E, R\}
\end{array}$$

Comme avant, le pointeur `p` ne dépend que d'une seule valeur statique et apparaît dans un contexte statique et un contexte dynamique. Cependant, seul le temps de liaison de son premier contexte est utilisé pour calculer la transformation pour sa première utilisation, $\textit{use-tr}^1(p)$. Le premier contexte est dynamique, ce qui mène à la transformation $\{R\}$. De la même façon, seul le temps de liaison de son deuxième contexte est utilisé pour calculer la transformation de sa deuxième utilisation, $\textit{use-tr}^2(p)$. Comme le contexte est statique, la transformation est $\{E\}$. le majorant de ces deux transformations est alors calculé par $\textit{def-tr}(p)$, produisant la transformation $\{E, R\}$ pour sa définition.

Spécialiser le programme selon ces transformations sensibles à l'utilisation produit de meilleurs résultats. Au temps de spécialisation, la définition de `p` est à la fois évaluée et résidualisée. Évaluer cette définition permet que l'utilisation de `p` dans un contexte statique puisse être évaluée. Le fait que l'utilisation résiduelle de `p` apparaisse dans le programme résiduel veut dire que cette définition de `p` doit aussi être résidualisée.

Dans le second exemple (Fig. 2.7), nous calculons les transformations suivantes pour la structure `s1`:

$$\begin{array}{rcl}
\textit{values}(s1) & = & \{\} \\
\textit{context}^1(s1) & = & \{s, 3\} \\
\textit{context}^2(s1) & = & \{d, 4\}
\end{array}$$

$$\begin{array}{rlll}
\textit{values-bt}(s1) & = S & & \\
\textit{context-bt}^1(s1) & = bt(s) \sqcup bt(3) & = S \sqcup S & = S \\
\textit{context-bt}^2(s1) & = bt(d) \sqcup bt(4) & = D \sqcup S & = D
\end{array}$$

La phase de transformation insensible à l'utilisation produit les annotations vues dans la Fig. 2.7 comme suit :

$$\begin{array}{rlll}
\textit{use-tr}^1(s1) & = \overline{\textit{values-bt}(s1) \sqcup \bigsqcup_{e1 \in \{1,2\}} \textit{context-bt}^1(s1)} & = \overline{S \sqcup (S \sqcup D)} & = \overline{D} \quad = \{R\} \\
\textit{use-tr}^2(s1) & = \overline{\textit{values-bt}(s1) \sqcup \bigsqcup_{e1 \in \{1,2\}} \textit{context-bt}^1(s1)} & = \overline{S \sqcup (S \sqcup D)} & = \overline{D} \quad = \{R\} \\
\textit{def-tr}(s1) & = \textit{use-tr}^{e \in \{1,\ 2\}}(s1) & = \textit{use-tr}^1(s1) & = \{R\}
\end{array}$$

La phase de transformation sensible à l'utilisation, d'un autre coté, calcule l'information suivante et produit les annotations vues en Fig. 2.8 .

$$\begin{array}{rlll}
\textit{use-tr}^1(s1) & = \overline{\textit{values-bt}(s1) \sqcup \textit{context-bt}^1(s1)} & = \overline{S \sqcup S} \quad = \overline{S} & = \{E\} \\
\textit{use-tr}^2(s1) & = \overline{\textit{values-bt}(s1) \sqcup \textit{context-bt}^2(s1)} & = \overline{S \sqcup D} \quad = \overline{D} & = \{R\} \\
\textit{def-tr}(s1) & = \bigcup_{e1 \in \{1,2\}} \textit{use-tr}^1(s1) & = \overline{S} \cup \overline{D} \quad = \{E\} \cup \{R\} & = \{E, R\}
\end{array}$$

Comme dans l'exemple du pointeur, nous voyons ici que l'analyse insensible à l'utilisation contraint toutes les utilisations de `s1` à avoir la même transformation, en l'occurrence $\{R\}$, puisqu'un de ses contextes est dynamique. L'analyse sensible à l'utilisation permet que les différentes utilisations aient des transformations différentes. Dans cet exemple, une utilisation est évaluée alors que l'autre est résidualisée. Supporter ces différentes transformations d'utilisations, de même, requière que la définition de `s1` soit évaluée et résidualisée au temps de spécialisation.

2.6 Résumé

Nous avons vu plusieurs exemples qui démontrent comment des analyses qui sont sensibles au flot, au contexte, au retour et à l'utilisation réalisent une spécialisation plus efficace. Notre approche implique deux analyses séparées. La phase de temps de liaison détermine si une construction ne dépend que d'information statique pendant la spécialisation. Elle est suivie par l'analyse de transformation, qui détermine la transformation de chaque construction. Nous avons vu comment calculer ces temps de liaison et les transformations. Le calcul des temps de liaison est similaire à celui des analyses précédentes puisqu'il utilise le même domaine abstrait et le même treillis, bien que notre analyse de temps de liaison ne calcule que l'information de disponibilité. Le calcul des transformations est plus compliqué, impliquant l'introduction d'un nouveau domaine abstrait et d'un nouveau treillis. Ceci mène à une nouvelle transformation qui à la fois évalue et résidualise une construction.

Chapitre 3

Analyse de temps de liaison

Dans ce chapitre, nous présentons la phase de temps de liaison en utilisant un cadre d'analyse de flot de données (cf, par exemple, [ASU86, MR90]). Premièrement, nous présentons l'information préliminaire nécessaire pour comprendre l'analyse comme le langage traité et la façon de modéliser les emplacements mémoire. Nous supposons que, précédant la phase de temps de liaison, un certain nombre d'analyses de pré-traitement ont été exécutées. Ces analyses incluent une analyse d'alias, une analyse de définitions et une analyse inter-procédurale de chaînes utilisation-définitions. Nous décrivons ces analyses et ensuite présentons la phase de temps de liaison de notre analyse de temps de liaison.

Langage Nous considérons un sous-ensemble de C décrit en Fig. 3.1. Les constructions telles que les appels de fonction, les retours multiples, les structures et les pointeurs sont inclus dans ce sous-ensemble de manière à montrer les aspects importants de nos analyses. Beaucoup d'autres constructions, telles que les expressions conditionnelles, l'opérateur virgule et les branchements sont traduits dans ce sous-ensemble. Par exemple, les boucles `for` et `while` sont converties en boucles `do-while`. D'autres simplifications sont faites pour clarifier et simplifier la présentation de l'analyse. Les appels de fonctions non-vides, par exemple, sont supposés affecter directement un identificateur avec leur valeur de retour, identificateur qui peut être alors utilisé dans des calculs ultérieurs. Cette stratégie simplifie l'analyse sans restreindre son applicabilité. Nous supposons que tous les programmes sont transformés avant l'analyse, si besoin est, pour qu'ils satisfassent cette contrainte. Les constructions qui ne sont pas récrites, telles que break/continue et l'allocation dynamique de mémoire, sont traitées directement par l'analyse bien qu'elles ne soient pas présentées ici. La seule exception à ceci sont les fonctions récursives qui ne sont pas traitées par l'analyse.

Locations Il y a trois types d'emplacements mémoire : emplacements de variables, emplacements de retour et emplacements de champ de structure. Un emplacement pour une variable, si cette variable n'est pas une structure, est représenté par un identificateur simple. La valeur de retour d'une fonction est dénotée par $RETURN(f)$, où f est un point de programme de la fonction où l'instruction `return` apparaît. Un emplacement pour un type structure *type* et un composant *champ* est dénoté *type.champ*. Nous supposerons que, étant donné un type structure, la fonction *location()* rend les emplacements des composants de structure associés à ce type. Noter qu'associer des emplacements aux composants de structure sur une base de type structure et de champ permet à différents composants d'une structure donnée d'avoir des temps de liaison différents mais contraint les mêmes composants de structures différentes d'un même type donné à avoir le même temps de liaison.

Domaines:

$$
\begin{array}{rcl}
const & \in & \mathcal{C}onstant \\
id & \in & \mathcal{I}dentifier \\
bop & \in & \mathcal{B}inaryOperator
\end{array}
$$

Syntaxe abstraite:

program	::=	type-def* decl* fun-def$^+$	programme
type-def	::=	`struct` *id* { decl$^+$ }	définition de type
decl	::=	type-spec *id*	déclaration
type-spec	::=	`void` | `char` | `int` | ...	types de base
	|	`*` type-spec	type pointeur
	|	`struct` *id*	type structure
fun-def	::=	type-spec *id* `(` decl* `)` stmt	définition de fonction
stmt	::=	lexp `=` exp	affectation
	|	`if (` exp `)` stmt `else` stmt	conditionnelle
	|	`do` stmt `while (` exp `)`	boucle
	|	{ stmt* }	bloc
	|	*id* `(` exp* `)`	appel de fonction vide
	|	*id* `=` *id* `(` exp* `)`	appel de fonction non-vide
	|	`return`	retour de fonction vide
	|	`return` exp	retour de fonction non-vide
lexp	::=	*id*	variable
	|	lexp`.`*id*	sélecteur de structure
	|	`*` exp	indirection
exp	::=	*const*	constante
	|	*id*	variable
	|	lexp`.`*id*	sélecteur de structure
	|	`&` lexp	adresse
	|	`*` exp	indirection
	|	exp *bop* exp	expression binaire

FIG. 3.1 – *Syntaxe d'un sous-ensemble de C.*

Ce traitement approximatif a jusqu'à présent été suffisant pour traiter les applications existantes que nous avons rencontrées et peut être étendu pour obtenir plus de précision si besoin.

Analyse d'alias Notre analyse d'alias (ou de synonymes) est très similaire à celles existantes [EGH94, Ruf95]. Elle est basée sur le modèle "points-to" de synonymie. L'analyse d'alias donne, pour chaque expression d'indirection $*^e exp$ au point de programme e (les exposants sont utilisés pour dénoter des points de programme dans la syntaxe d'un programme), l'ensemble $aliases(e)$ des alias correspondant, c'est-à-dire l'ensemble des emplacements que l'expression peut représenter.

Analyse de définitions L'analyse de définitions calcule, pour chaque instruction au point de programme s, l'ensemble des emplacements *defs-stmt*(s), qui *peuvent* être définis par l'instruction. Un emplacement est défini s'il apparaît en membre gauche d'une affectation, donné par *defs-lexp*(s). Si le membre gauche est un identificateur, alors l'emplacement de l'identificateur est défini. Une expression d'indirection de pointeur définit tous les emplacements possibles sur lesquels le pointeur peut pointer. Définir un composant de structure ne définit que le simple composant spécifié, tandis que définir la structure entière définit tous les composants de la structure.

Pour une affectation donnée, un ensemble contenant plusieurs emplacements est calculé quand l'emplacement défini à l'exécution ne peut pas être déterminé statiquement. Ces définitions *ambiguës* sont dues soit à la synonymie soit, avec notre représentation des structures, à l'affectation de composant de structure. Si, d'autre part, une analyse statique peut déduire l'emplacement qui sera défini à l'exécution par une instruction, alors la définition est considérée *non-ambiguë*. Dans ce cas, la fonction *unambiguous-defs*() retourne l'emplacement non-ambigu défini, qui peut être utilisé par les analyses ultérieures pour produire des résultats plus précis. Par exemple, dans la phase de temps de liaison, une définition non-ambiguë permet qu'une variable dynamique devienne statique. Dans l'analyse de transformation, une définition non-ambiguë est traitée comme une suppression de définition, indiquant que l'emplacement est définitivement défini.

Comme il est important de comprendre les fonctions *defs-stmt*(s) et *unambiguous-defs*() pour comprendre comment les temps de liaison sont calculés, nous les définissons en Fig. 3.2.

Analyse inter-procédurale de chaînes utilisation-définitions Un résumé des emplacements non-locaux utilisés et définis est calculé pour chaque fonction, semblable à l'information d'analyse inter-procédurale [Bar78]. Nous considérons que *used-non-locals*() et *def-non-locals*() représentent respectivement les emplacements non-locaux utilisés et définis d'une fonction. Noter que cette phase doit suivre (ou être combinée avec) une analyse d'alias. Quand, comme expression en membre droit (membre gauche), un pointeur pointe potentiellement sur plusieurs emplacements, tous ces emplacements doivent être considérés comme utilisés (définis). Aussi, la notion d'utilisation et de définition est liée en réalité à l'analyse plutôt qu'aux exécutions réelles. En particulier, dans le cas d'une définition ambiguë d'indirection de pointeur, tous les emplacements sur lesquels pointe (potentiellement) un pointeur doivent aussi être considérés comme utilisés puisque leurs temps de liaison sont utilisés pour calculer les temps de liaison des emplacements (potentiellement) définis.

définitions de fonction :

id^f `(` decls `)` stmt^s :

$\mathit{defs\text{-}fun}(f) = \mathit{defs\text{-}stmt}(s)$

instructions :

lexp^{e_1} `=`^{s} exp^{e_2} :

$\mathit{defs\text{-}stmt}(s) = \mathit{defs\text{-}lexp}(e_1)$

$\mathit{unambiguous\text{-}defs}(s) = (\mid \mathit{defs}(s) \mid = 1) \rightarrow \mathit{defs\text{-}stmt}(s); \{\}$

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$\mathit{defs\text{-}stmt}(s) = \mathit{defs\text{-}stmt}(s_1) \cup \mathit{defs\text{-}stmt}(s_1)$

`do`^{s} stmt^{s_0} `while` `(` exp^e `)` :

$\mathit{defs\text{-}stmt}(s) = \mathit{defs\text{-}stmt}(s_0)$

`{` $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ `}`^{s} :

$\mathit{defs\text{-}stmt}(s) = \cup_{1 \leq i \leq n} \mathit{defs\text{-}stmt}(s_i)$

id^s `(` $\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$ `)` :

$\mathit{defs\text{-}stmt}(s) = \mathit{defs\text{-}fun}(f_{id})$

id_1 `=`^{s} id_2 `(` $\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$ `)` :

$\mathit{defs\text{-}stmt}(s) = \mathit{defs\text{-}fun}(f_{id_2}) \cup \{id_1\}$

$\mathit{unambiguous\text{-}defs}(s) = \{id_1\}$

`return`^{s} :

$\mathit{defs\text{-}stmt}(s) = \{\}$

`return`^{s} exp^e :

$\mathit{defs\text{-}stmt}(s) = \{RETURN(f)\}$

expressions en membre gauche :

$\mathit{is\text{-}a\text{-}struct}(e)$:

$\mathit{defs\text{-}lexp}(e) = \mathit{locations}(\mathit{type}(e))$

id^e :

$\mathit{defs\text{-}lexp}(e) = \{id\}$

`*`^{e} exp^{e_0} :

$\mathit{defs\text{-}lexp}(e) = \mathit{aliases}(e)$

lexp^{e_0} `.`$^e id$:

$\mathit{defs\text{-}lexp}(e) = \{\mathit{type}(e_0).id\}$

FIG. 3.2 – *Analyse de définition.*

Phase de temps de liaison

Considérons maintenant une analyse qui calcule des temps de liaison. Nous donnons d'abord une description du modèle abstrait de mémoire utilisé par l'analyse. Puis nous donnons les équations de flot de données qui calculent une description de temps de liaison pour chaque instruction dans le programme. Enfin, nous expliquons comment cette information est utilisée pour annoter un programme.

États Nous appellerons les ensembles de valeurs propagés par la phase de temps de liaison, des états. Les états de temps de liaison sont des éléments de $BtState = Location \to Bt$, où Bt est le domaine (en forme de treillis) de temps de liaison défini auparavant. Nous noterons $\setminus$ l'opérateur binaire de $BtState \times Locations \to BtState$ initialisant à bottom un ensemble d'emplacements.

L'opérateur d'union $\underline{\sqcup}$ sur les états de temps de liaison est défini comme une application point à point de l'opérateur de majorant $\sqcup$ sur l'espace de la fonction *BtState*.

Nous utiliserons un graphe pour représenter les états. On y accède via une opération de consultation qui prend un graphe (un ensemble de paires emplacement/temps de liaison) et un emplacement et rend le temps de liaison correspondant. Tous les emplacements n'ont pas besoin d'apparaître dans le graphe. Un emplacement qui n'apparaît pas dans le graphe est considéré comme indéfini—dans ce cas la fonction de consultation retourne simplement l'élément bottom.

Il est facile de montrer que $(BtState, \underline{\sqcup})$ est un demi-treillis supérieur fini et que l'ensemble des fonctions de transfert générées par clôture par union et composition est un espace d'opérations monotones [NN91]. Cela veut dire que l'ensemble des équations correspondant à un programme donné peut être résolu en utilisant un des algorithmes itératifs standards [ASU86].

3.1 Équations de flot de données

Pour chaque instruction, la phase de temps de liaison calcule deux états différents. L'état de temps de liaison qui décrit les variables dans l'instruction en question est appelé l'*état d'instruction*.

Un *état de retour* supplémentaire est utilisé pour collecter et propager les états d'instruction inter-procéduralement. L'état de retour est initialisé au début d'une fonction. Chaque fois qu'on rencontre une instruction `return`, l'état d'instruction de l'instruction `return` est ajouté à l'état de retour jusqu'ici calculé. À la fin de la définition de fonction, l'état de retour retourné par la fonction est une combinaison de tous les états d'instruction de chaque `return`. Le temps de liaison de la valeur retournée par la fonction est propagée aux sites appelants via l'emplacement mémoire de retour $RETURN(f)$. Chaque instruction `return` non-vide initialise le temps de liaison de cet emplacement au temps de liaison de l'expression retournée.

Les équations de flot de données liant l'état d'instruction $in(s)$ et l'état de retour *ret-in(s)* à l'entrée d'une instruction (en un point de programme s) avec l'état d'instruction $out(s)$ et l'état de retour *ret-out(s)* à la sortie de l'instruction sont données en Fig. 3.3.

Définition de fonction

Une définition de fonction admet un état initial $in(f)$, qui est utilisé comme état d'instruction d'entrée $in(s)$ pour le corps de la fonction. L'état d'instruction initial pour un programme contient S pour les paramètres déclarés statiques et D pour ceux déclarés dynamiques. L'état de retour du corps *ret-in(s)* est initialisé au début

définitions de fonction:

id^f (decls) $body^s$:

$in(s) = in(f)$

$ret\text{-}in(s) = \{\}$

$ret\text{-}out(f) = \{(loc, lookup(ret\text{-}out(s), loc)) \mid loc \in def\text{-}non\text{-}locals(f)\}$

instructions :

lexp^{e_1} `=`^{s} exp^{e_2}:

$out(s) = t_s(in(s))$ $\quad$ $ret\text{-}out(s) = ret\text{-}in(s)$

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$in(s_1) = in(s)$ $\quad$ $ret\text{-}in(s_1) = ret\text{-}in(s)$

$in(s_2) = in(s)$ $\quad$ $ret\text{-}in(s_2) = ret\text{-}in(s)$

$out(s) = t_{e,s_1}(out(s_1)) \sqcup t_{e,s_2}(out(s_2))$ $\quad$ $ret\text{-}out(s) = t_{e,s_1}(ret\text{-}out(s_1)) \sqcup t_{e,s_2}(ret\text{-}out(s_2))$

`do`^{s} stmt^{s_0} `while` `(` exp^e `)`:

$in(s_0) = in(s) \sqcup t_{e,s_0}(out(s_0))$ $\quad$ $ret\text{-}in(s_0) = in(s) \sqcup t_{e,s_0}(ret\text{-}out(s_0))$

$out(s) = out(s_0)$ $\quad$ $ret\text{-}out(s) = ret\text{-}out(s_0)$

{ $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ }^{s}:

$in(s_1) = in(s)$ $\quad$ $ret\text{-}in(s_1) = ret\text{-}in(s)$

$in(s_{i+1}) = out(s_i), 1 \leq i < n$ $\quad$ $ret\text{-}in(s_{i+1}) = ret\text{-}out(s_i), 1 \leq i < n$

$out(s) = out(s_n)$ $\quad$ $ret\text{-}out(s) = ret\text{-}out(s_n)$

id^s $(\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n})$:

$ctx = \{(formal\text{-}loc(i, id), exp\text{-}bt(e_i, in(s))) \mid 1 \leq i \leq n\} \cup$
$\{(loc, lookup(in(s), loc)) \mid loc \in used\text{-}non\text{-}locals(id)\}$

$in(f_{id,ctx}) = ctx$

$out(s) = (in(s) \setminus def\text{-}non\text{-}locals(id)) \sqcup ret\text{-}out(f_{id,ctx})$

$ret\text{-}out(s) = ret\text{-}in(s)$

id_1 `=`^{s} id_2 `(` $\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$ `)`:

$ctx = \{(formal\text{-}loc(i, id_2), exp\text{-}bt(e_i, in(s))) \mid 1 \leq i \leq n\} \cup$
$\{(loc, lookup(in(s), loc)) \mid loc \in used\text{-}non\text{-}locals(id_2)\}$

$in(f_{id_2,ctx}) = ctx$

$out(s) = (in(s) \setminus (def\text{-}non\text{-}locals(id_2) \cup \{id_1\}))$
$\sqcup\ ret\text{-}out(f_{id_2,ctx})$
$\sqcup \{(id_1, lookup(ret\text{-}out(f_{id_2,ctx}), RETURN(f_{id_2,ctx})))\}$

$ret\text{-}out(s) = ret\text{-}in(s)$

`return`^{s}:

$out(s) = \{\}$

$ret\text{-}out(s) = in(s) \sqcup ret\text{-}in(s)$

`return`^{s} exp^e:

$out(s) = \{\}$

$ret\text{-}out(s) = ret\text{-}in(s) \sqcup in(s) \sqcup \{(RETURN(f), exp\text{-}bt(e, in(s)))\}$

FIG. 3.3 – *Phase de temps de liaison—équations de flot de donnés.*

$$t_s(state) = \{(loc, \textit{stmt-bt}(s)) \mid loc \in \textit{defs-stmt}(s)\} \underline{\sqcup} (state \setminus \textit{unambiguous-defs}(s))$$

$$t_{e,s}(state) = \{(loc, \textit{exp-bt}(e, state)) \mid loc \in \textit{defs-stmt}(s)\} \underline{\sqcup} state$$

FIG. 3.4 – *Phase de temps de liaison—fonctions de transfert.*

de toute fonction pour indiquer qu'aucune instruction `return` n'a été rencontrée. Après que le corps ait été analysé, l'état de retour du corps, *ret-out(s)*, est utilisé pour définir l'état de retour de la fonction, *ret-out(f)*. En particulier, cet état de retour contient les variables non-locales définies et leurs temps de liaison associés. Noter que, comme l'état de retour de la fonction est défini en termes d'état de retour du corps, l'état d'instruction de sortie du corps, *out(s)*, n'est pas utilisé.

Voyons maintenant comment les états d'instructions sont calculés à l'intérieur du corps de chaque fonction. Cet aspect intra-procédural de l'analyse est exprimé en liant l'état d'instruction $in(s)$ à l'entrée d'une instruction avec l'état d'instruction $out(s)$ à la sortie de l'instruction. Puis nous montrerons comment les aspects inter-procéduraux de l'analyse sont exprimés en reliant *ret-in(s)* avec *ret-out(s)*.

Intra-procédural

Une affectation à un point de programme s modifie l'état d'instruction comme le décrit la fonction de transfert $t_s()$ (que l'on peut voir en Fig. 3.4). Elle met à jour chaque emplacement dans l'ensemble des définitions possibles pour l'affectation avec le temps de liaison de l'affectation fourni par *stmt-use-bt*(s) (voir Fig. 3.5). Noter que le temps de liaison de l'affectation dépend de l'état d'entrée de l'affectation. Si l'affectation est ambiguë, il faut prendre une approximation sûre : le nouveau temps de liaison de chaque emplacement défini est le majorant de son précédent temps de liaison et du temps de liaison de l'affectation. Ceci est réalisé en associant à tous les emplacements définis le temps de liaison de l'affectation et en unissant cet état à l'état où toutes les définitions non-ambiguës ont été mises à bottom. Mettre à jour l'état pour chaque instruction est la façon dont une analyse sensible au flot permet à une variable liée à une valeur dynamique de devenir statique.

Pour calculer la bonne approximation de l'état à un point de jonction, le temps de liaison de chaque emplacement potentiellement défini à l'intérieur de l'instruction conditionnelle ou de la boucle est égal au majorant de son précédent temps de liaison et du temps de liaison du test de la conditionnelle ou de la boucle—celui-ci est fourni par *exp-bt*$(e, state)$ (cf Fig. 3.5). Une seconde fonction de transfert, $t_{e,s}()$ (aussi en Fig. 3.4), prend en compte les dépendances de contrôle pour calculer l'état aux points de jonction. Si le spécialiseur ne duplique pas les continuations, comme c'est le cas pour Tempo, les points de jonction existent à l'issue des instructions conditionnelles et des boucles. Si un test est dynamique, tous les emplacements possiblement définis dans la portée du test sont considérés comme dynamiques. Dans le cas d'un test statique, l'opération d'union n'a aucun effet ; la fonction de transfert est la fonction identité.

Par exemple, considérons le cas d'une variable à qui est affectée une valeur statique dans une branche d'une conditionnelle dont le test est dynamique. Au temps de spécialisation, la valeur de la variable restera inconnue après le point de jonction. Au temps d'exécution, si la branche est prise, la variable se verra affecter une nouvelle valeur; sinon, elle gardera la valeur qu'elle avait avant l'entrée dans la conditionnelle. Une telle variable doit être considérée comme dynamique à l'issue de la branche. En général, prendre pour chaque emplacement potentiellement défini dans une branche le majorant du temps de liaison de son utilisation et du temps de

liaison du test donne la bonne approximation.

Inter-procédural

La sensibilité au contexte est obtenue en dupliquant les définitions de fonction et les équations de flot de données correspondantes en fonction des différents contextes d'appels rencontrés dans le programme de la fonction considérée. Les différents exemplaires de la même fonction sont distinguées en indiçant le point de programme de fonction f avec son identificateur id et le contexte spécifique d'appel ctx, ce qui construit un point de programme de fonction sensible au contexte $f_{id,ctx}$.

Un appel de fonction vide utilise l'état d'instruction *in(s)* pour créer le contexte d'appel, ctx. Ce contexte d'appel combine les parties de l'état concernant les entrées de l'appel en calculant le temps de liaison de chaque paramètre effectif et en l'associant à chaque paramètre formel correspondant et aussi les temps de liaison des emplacements mémoire non-locaux qui peuvent être *utilisés* par la fonction, en prenant en compte les autres appels imbriqués.

L'appel *formal-loc*(i, id) retourne simplement l'emplacement associé au $i^{ème}$ paramètre formel de la fonction id. Comme le nombre d'emplacements définis par un programme donné est fini, ces contextes sont finis.

L'état de sortie *out(s)* d'un appel est obtenu en mettant à jour les temps de liaison des emplacement non-locaux définis qui peuvent être *définis* par l'appel, là encore en prenant en compte tous les autres appels imbriqués. Dans le cas d'un appel de fonction non-vide, le temps de liaison de l'emplacement en membre gauche prend de plus la valeur du temps de liaison de l'emplacement de retour.

Considérons, par exemple, une conditionnelle avec des instructions `return` dans chaque branche ainsi que des chemins qui ne retournent rien. Au point de jonction, seuls les chemins qui ne retournent rien seront fusionnés et propagés vers l'instruction suivante en tant qu'état de retour de la conditionnelle. D'un autre coté, tous les chemins qui retournent une valeur doivent aussi être fusionnés, prenant en compte la possibilité d'instructions `return` sous contrôle dynamique.

La Fig. 3.3 montre que les équations restantes pour les états de retour *ret-in(s)* et *ret-out(s)* sont très similaires à celles des états d'instructions associés *in(s)* et *out(s)*. Pour l'instruction d'affectation et les appels de fonction, *ret-out(s)* est simplement égal à *ret-in(s)*. Pour les conditionnelles, les boucles et les blocs, *ret-in(s)* et *ret-out(s)* sont liés aux mêmes équations que $in(s)$ et $out(s)$. En particulier, la fonction de transfert $t_{e,s}()$ gère les emplacements de retour sous contrôle conditionnel.

3.2 Annotations de temps de liaison

Résoudre ces équations de flot de données produit un état d'instruction pour chaque instruction, avec lequel le programme est annoté, comme on le voit en Fig. 3.5. Le temps de liaison pour la majeure partie des constructions est le majorant de ses sous-composants. Par exemple, le temps de liaison d'une affectation est le majorant des temps de liaison de ses expressions en membre gauche et en membre droit, et le temps de liaison d'une conditionnelle est le majorant des temps de liaison des expressions de test et de chacune des branches.

Puisque notre analyse est sensible au retour, deux temps de liaison sont calculés pour un appel de fonction non-vide. Ces deux temps de liaison sont calculés à la définition de la fonction mais utilisés pour annoter le point d'appel de la fonction. Le premier, calculé par *fun-body-bt()*, détermine si le corps de la fonction est complètement statique. Un corps complètement statique est simplement évalué au temps de spécialisation. Le second temps de liaison, produit par *fun-return-bt()*, détermine si

définition de fonction:

id^f `(` decls `)` stmt^s:

$\mathit{fun\text{-}body\text{-}bt}(f) = \mathit{stmt\text{-}bt}(s)$

$\mathit{fun\text{-}return\text{-}bt}(f) = \bigsqcup_{s1 \in returns(s)} \mathit{stmt\text{-}bt}(s1)$

instructions:

lexp^{e_1} `=`^{s} exp^{e_2}:

$\mathit{stmt\text{-}bt}(s) = \mathit{lexp\text{-}bt}(e_1, in(s)) \sqcup \mathit{exp\text{-}bt}(e_2, in(s))$

`if`^{s} `(` exp^e `)` $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$\mathit{stmt\text{-}bt}(s) = \mathit{exp\text{-}bt}(e, in(s)) \sqcup \mathit{stmt\text{-}bt}(s_1) \sqcup \mathit{stmt\text{-}bt}(s_2)$

`do`^{s} stmt^{s_0} `while` `(` exp^e `)`:

$\mathit{stmt\text{-}bt}(s) = \mathit{exp\text{-}bt}(e, out(s_0)) \sqcup \mathit{stmt\text{-}bt}(s_0)$

`{` $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ `}`^{s}:

$\mathit{stmt\text{-}bt}(s) = \bigsqcup_{1 \le i \le n} \mathit{stmt\text{-}bt}(s_i)$

id^s `(`$\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$`)`:

$\mathit{stmt\text{-}bt}(s) = \mathit{fun\text{-}body\text{-}bt}(f_{id})$

id_1`=`$^{s_1} id_2^{s_2}$ `(`$\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$`)`:

$\mathit{stmt\text{-}bt}(s_2) = \mathit{fun\text{-}body\text{-}bt}(f_{id_2})$

$\mathit{stmt\text{-}bt}(s_1) = \mathit{fun\text{-}return\text{-}bt}(f_{id_2})$

`return`^{s}:

$\mathit{stmt\text{-}bt}(s) = S$

`return`^{s} exp^e:

$\mathit{stmt\text{-}bt}(s) = \mathit{exp\text{-}bt}(e, in(s))$

expression en membre gauche:

id^e:

$\mathit{lexp\text{-}bt}(e, state) = S$

$lexp^{e_0}$`.`$^e id$:

$\mathit{lexp\text{-}bt}(e, state) = \mathit{lexp\text{-}bt}(e_0, state)$

`*`^{e} exp^{e_0}:

$\mathit{lexp\text{-}bt}(e, state) = \mathit{exp\text{-}bt}(e_0, state)$

expression en membre droit:

$\mathit{loc\text{-}bt}(loc, state) = \mathit{struct\text{-}loc}(loc) \rightarrow \bigsqcup_{l \in locations(type(loc))} state(l); state(loc)$

$const^e$:

$\mathit{exp\text{-}bt}(e, \_) = S$

id^e:

$\mathit{exp\text{-}bt}(e, state) = \mathit{loc\text{-}bt}(id)$

$lexp^{e_0}$`.`$^e id$:

$\mathit{exp\text{-}bt}(e, state) = \mathit{lexp\text{-}bt}(e_0, state) \sqcup \mathit{loc\text{-}bt}(type(e_0)\,.\,id)$

`&`^{e} lexp^{e_0}:

$\mathit{exp\text{-}bt}(e, state) = \mathit{lexp\text{-}bt}(e_0, state)$

`*`^{e} exp^{e_0}:

$\mathit{exp\text{-}bt}(e, state) = \mathit{exp\text{-}bt}(e_0, state) \sqcup (\bigsqcup_{loc \in aliases(e)} state(loc))$

$\text{exp}_1^{e_1}$ bop^e $\text{exp}_2^{e_2}$:

$\mathit{exp\text{-}bt}(e, state) = \mathit{exp\text{-}bt}(e_1, state) \sqcup \mathit{exp\text{-}bt}(e_2, state)$

FIG. 3.5 – *Phase de temps de liaison—annotations de temps de liaison.*

tous les instructions `return` sont statiques. Dans ce cas, la valeur retournée par la fonction est statique et elle peut être utilisée par le spécialiseur, même si le corps de la fonction est dynamique et par conséquent résidualisé.

De plus, certaines constructions n'ont pas de sous-composants, comme l'instruction vide `return`, une variable en membre gauche et une constante en membre droit. Ces constructions sont toujours considérées comme statiques, indépendamment de l'état de temps de liaison, puisqu'elles sont toujours calculées au temps de spécialisation. Rappelons que pour une variable en membre gauche, seule l'adresse de la variable—pas son contenu—est nécessaire pour qu'elle soit évaluée.

3.3 Résumé

Le code résultant annoté de temps de liaison indique si les constructions dépendent de valeurs connues ou inconnues. Une construction est annotée statique si elle ne dépend que de valeurs statiques—dans ce cas elle peut être évaluée au temps de spécialisation. Nous avons déjà mentionné, toutefois, que les constructions qui *peuvent* être évaluées ne *doivent* pas toutes être évaluées. Par exemple, les valeurs statiques non-réifiables dans des contextes dynamiques doivent être résidualisées. Par conséquent, le code annoté de temps de liaison doit être analysé dans la phase de transformation pour déterminer la transformation correcte pour chaque construction. Le code annoté de transformations résultant est alors utilisé pour diriger la spécialisation.

Chapitre 4

Analyse au temps d'évaluation

Les annotations de temps de liaison indiquent si les constructions dépendent de valeurs connues ou inconnues au moment de la spécialisation. La phase de transformation utilise cette information pour calculer une transformation pour chaque construction. Premièrement, les transformations sont calculées pour les utilisations de variable. Ces transformations sont alors utilisées pour calculer les transformations des définitions.

Alors que la phase de temps de liaison propage de l'information en avant d'une définition de variable vers ses utilisations, la phase de transformation propage de l'information *en arrière*, des utilisations de variables vers leur définition. Par conséquent, l'analyse pour calculer les transformations est une analyse arrière.

Dans ce chapitre, nous présentons cette analyse arrière. Nous décrivons d'abord les états utilisés pour calculer cette information et donnons ensuite les équations de flot de données décrivant l'analyse. Le programme résultant est annoté de transformations.

États Comme dans la phase de temps de liaison, les ensembles des valeurs propagées par la phase de transformation sont aussi des *états*. Les états de transformation, cependant, sont $TrState = Location \rightarrow Tr$, où Tr est le domaine de transformations—en forme de treillis— défini auparavant. On notera $\setminus$ l'opération binaire de $TrState \times Locations \rightarrow TrState$ initialisant un ensemble d'emplacements mémoire à la valeur bottom.

Une fois encore, nous utilisons un graphe comme représentation des états. La fonction de consultation (lookup) prend un graphe (un ensemble de couples emplacement/transformation) et un emplacement et retourne la transformation correspondante. Un emplacement qui n'apparaît pas dans le graphe est considéré comme indéfini, dans ce cas la fonction de consultation retourne simplement l'élément bottom.

L'opération d'union $\underline{\cup}$ sur les états de transformation est défini comme une application point à point de l'opération de majorant $\cup$ sur le domaine de la fonction $TrState$. ($TrState, \underline{\cup}$) est un demi-treillis supérieur fini et l'ensemble des fonctions de transfert générées par clôture par union et composition est un espace d'opérations monotones. L'ensemble des équations correspondant à un programme donné peut être résolu en utilisant un des algorithmes itératifs standards.

4.1 Équations de flot de données

L'analyse de transformation utilise aussi deux états. L'*état d'instruction* est propagé en arrière à l'intérieur d'une procédure. Chaque fois qu'une variable est utilisée, l'état d'instruction est mis à jour pour inclure la transformation associée à l'utilisation de la variable. La transformation calculée pour chaque utilisation d'une variable prend en compte son contexte comme expliqué en Figure 2.11. Toutes les variables dynamiques sont résidualisées pendant la spécialisation et sont par conséquent annotées par une transformation $\{R\}$. Une variable statique est annotée par $\{E\}$ si elle est utilisée dans un contexte statique et $\{R\}$ si elle est utilisée dans un contexte dynamique. Ces annotations sont combinées ainsi que d'autres informations comme le fait que toutes les constantes sont considérées $\{E\}$, pour déterminer les transformations pour les autres expressions. Une définition de variable est alors annotée avec une transformation donnée en consultant la variable dans l'état. De plus, cette information de transformation est utilisée pour déterminer la transformation de définitions. Précisément, la transformation pour la définition d'une variable est calculée en prenant le majorant des transformations de la variable, selon le treillis de transformation vu en Figure 2.10.

L'*état de retour* est utilisé pour propager les états de transformation inter-procéduralement. L'état de retour initial pour une fonction est créé à partir de l'état d'instruction au point d'appel de la fonction. Puisque la phase de transformation est une analyse arrière, une fonction est analysée en partant de la fin de la fonction. Comme la fin d'une fonction est représentée par les instructions `return` dans le corps de la fonction, l'état d'instruction de chaque instruction `return` est égal à l'état de retour de la fonction. L'analyse propage alors les états d'instruction intra-procéduralement jusqu'à ce que le début de la fonction soit atteint. L'état de retour initial de la dernière fonction d'un programme est initialisé par l'ensemble vide, puisqu'aucune variable n'est utilisée après la fin du programme.

Les équations de flot de données de la phase de transformation sont données en Figure 4.1, les fonctions de transfert de base sont données en figure 4.2.

Définitions de fonction

Chaque fonction est analysée en fonction d'un état de retour initial *ret-out*(f), constitué des transformations pour les variables non-locales définies dans la fonction. Cet état de retour est utilisé comme l'état de retour *ret-out*(s) du corps de la fonction.

À l'issue de l'analyse du corps, l'état d'instruction du corps, *in(s)*, est utilisé pour définir l'état d'instruction de la fonction, *in(f)*. Cet état d'instruction est constitué des variables non-locales utilisées dans la fonction et de leurs transformations associées. Notons que pour cette analyse, il est inutile de calculer un état de retour *ret-in*(f) à l'entrée de la fonction.

À l'intérieur du corps de la fonction, seuls les états d'instruction sont calculés et propagés. Cet aspect de l'analyse est exprimé en reliant l'état d'instruction $out(s)$ à la sortie de l'instruction avec l'état d'instruction $in(s)$ à l'entrée de l'instruction. Comme l'état de retour *ret-out*(s) ne varie pas dans le corps de la fonction, il n'est pas recalculé à chaque instruction. Il est utilisé, cependant, pour définir l'état d'instruction à chaque instruction `return`.

Analyse intra-procédurale

La fonction de transfert $t_s()$ est utilisée pour calculer l'état d'entrée d'une affectation $in(s)$ à partir de l'état de sortie $out(s)$ comme suit. Dans le cas d'une affectation non-ambiguë, l'emplacement mémoire défini est intialisé à $\{\}$; il n'y a

définition de fonction:

id^f (decls) $body^s$:

$in(f) = \{(loc, lookup(in(s), loc)) \mid loc \in used\text{-}non\text{-}locals(f)\}$

$out(s) = out(f)$

instructions:

lexp^{e_1} `=`^{s} exp^{e_2}:

$in(s) = t_s(out(s))$

`if`^{s} (exp^e) $\text{stmt}_1^{s_1}$ `else` $\text{stmt}_2^{s_2}$:

$in(s) = t_e(in(s_1) \underline{\sqcup} in(s_2))$

$out(s_1) = out(s)$

$out(s_2) = out(s)$

`do`^{s} stmt^{s_0} `while` (exp^e):

$in(s) = in(s_0)$

$out(s_0) = t_e(out(s)) \underline{\sqcup} t_e(in(s_0))$

{ $\text{stmt}_1^{s_1} \ldots \text{stmt}_n^{s_n}$ }^{s}:

$in(s) = in(s_1)$

$out(s_i) = in(s_{i+1}), 1 \leq i < n$

$out(s_n) = out(s)$

id^s ($\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$):

$in(s) = in(e_1)$

$out(e_i) = in(e_{i+1}), 1 \leq i < n$

$out(e_n) = (out(s) \setminus used\text{-}non\text{-}locals(id)) \underline{\sqcup} in(f_{id,ctx})$

$out(f_{id_2,ctx}) = ctx$

$ctx = \{(loc, lookup(out(s), loc)) \mid loc \in def\text{-}non\text{-}locals(f)\}$

id_1 `=`^{s} id_2 ($\text{exp}_1^{e_1} \ldots \text{exp}_n^{e_n}$):

$in(s) = t_s(in(e_1))$

$out(e_i) = in(e_{i+1}), 1 \leq i < n$

$out(e_n) = (out(s) \setminus used\text{-}non\text{-}locals(id_2)) \underline{\sqcup} in(f_{id_2,ctx})$

$out(f_{id_2,ctx}) = ctx$

$ctx = \{(loc, lookup(out(s), loc)) \mid loc \in def\text{-}non\text{-}locals(f)\}$

`return`^{s}:

$in(s) = out(s)$

`return`^{s} exp^e:

$in(s) = t_e(out(s))$

FIG. 4.1 – *Phase de transformation—équations de flot de données*

$$t_s(state) = (\textit{def-state}(s, \textit{def-tr}(s, state)) \underline{\cup}\ state) \setminus \textit{unambiguous-defs}(s)$$

$$t_e(state) = \textit{exp-state}(e, \textit{exp-tr}(e)) \underline{\cup}\ state$$

FIG. 4.2 – *Phase de transformation—fonctions de transfert*

pas de transformation à propager. La fonction *def-bt*() calcule la transformation d'une affectation en prenant le majorant des transformations de tous les emplacements potentiellement définis par l'affectation. La fonction *stmt-use*() met à jour l'état de transformation, basé sur la transformation de l'affectation et les résultats de la phase de temps de liaison, que l'on récupère via la fonction *exp-use-bt*().

Les états de sortie $out(s1)$ et $out(s2)$ des branches d'une instruction conditionnelle sont définis en fonction de l'état de sortie $out(s)$ de l'instruction conditionnelle. Une deuxième fonction de transfert, $t_e()$ (aussi en Figure 4.2), est utilisée pour unir les états d'entrée $in(s1)$ et $in(s2)$ des branches pour définir l'état d'entrée $in(s)$ de la conditionnelle. L'instruction de boucle utilise de la même façon cette fonction de transfert pour définir son état d'entrée.

Pour une séquence d'instructions, l'état de sortie est passé à la dernière instruction. L'état d'entrée de chaque instruction est utilisé comme état de sortie de l'instruction précédente. À la fin, l'état de sortie de la première instruction est l'état de sortie de la séquence.

Analyse inter-procédurale

Dans la phase de transformation, le contexte d'appel dépend de l'information à chaque point d'appel, information qui affecte l'analyse de la fonction. Comme les transformations sont propagées en arrière vers une définition de variable, l'information nécessaire est constituée de l'ensemble des utilisations de chaque variable non-locale définie dans la fonction. Ce contexte d'appel ctx est ensuite utilisé comme état de retour $\textit{ret-out}(f_{id,ctx})$ de la fonction.

L'état de sortie $out(s)$ de l'appel de fonction est mis à jour avec les variables non-locales utilisées de l'état d'entrée de la fonction $in(f_{id,ctx})$. L'état résultant est utilisé pour établir la mise en relation entre les paramètres formels et les paramètres effectifs. Les définitions de variables peuvent ne pas venir des affectations seules (comme on l'a vu dans l'analyse intra-procédurale) mais aussi de la relation formel/effectif. À cause de ceci, la transformation d'une définition de paramètre, comme celle d'une affectation, peut être $\{E\}$, $\{R\}$, $\{E, R\}$ ou $\{\}$. Là encore, comme pour l'affectation, cette information regroupe les utilisations du paramètre formel dans le corps de la fonction. De même, comme pour l'affectation, le spécialiseur doit être capable d'évaluer et de résidualiser la même relation formel/effectif.

Après qu'une transformation soit calculée pour chaque paramètre, l'état d'entrée $in(s)$ pour l'instruction d'appel est défini. Pour une fonction non-vide, la fonction de transfert $t_s()$ est utilisée pour prendre en compte l'affectation.

À chaque instruction *return*, l'état d'instruction est égal à l'état de retour de sortie $\textit{ret-out}(s)$. Cet état est défini, une fois par fonction, par l'état de retour de fonction $\textit{ret-out}(f)$ et par conséquent l'état d'instruction à chaque instruction `return` à l'intérieur d'une fonction est toujours le même. Pour une instruction de retour non-vide, la fonction de transfert $t_e()$ est utilisée pour prendre en compte l'expression.

instructions:

$\textit{def-tr}(s, state) = \bigcup_{loc \in defs(s)} lookup(state, loc)$

$\textit{def-state}(s, tr) = \textit{lexp-state}(lexp(s), tr) \underline{\bigcup} \textit{exp-state}(exp(s), tr)$

expressions:

$\textit{loc-state}(loc, tr) = \textit{is-a-struct-loc}(loc) \rightarrow \underline{\bigcup}_{loc' \in locations(type(loc))} \{(loc', tr)\}; \{(loc, tr)\}$

expressions en membre gauche:

$\textit{lexp-state}(id, \_) = \{\}$

$\textit{lexp-state}(lexp^{e_0}.id, tr) = \textit{lexp-state}(lexp^{e_0}, tr)$

$\textit{lexp-state}(*exp^{e_0}, tr) = \textit{exp-state}(exp^{e_0}, tr)$

expressions en membre droit:

$\textit{exp-state}(const^{e}, \_) = \{\}$

$\textit{exp-state}(id^{e}, \{\mathrm{E}\}) = \textit{loc-state}(id, \{\mathrm{E}\})$

$\textit{exp-state}(id^{e}, tr)$

$= (\textit{exp-bt}(e) = S \wedge \textit{is-liftable}(type(e))) \rightarrow$

$\{(id, \{E\})\}$

;

$\textit{loc-state}(loc, tr)$

$\textit{exp-state}(lexp^{e_0}.^{e}id, tr) = \textit{lexp-state}(lexp^{e_0}, tr) \underline{\bigcup} \textit{loc-state}(type(e_0).id, \textit{exp-tr}(e))$

$\textit{exp-state}(\&lexp^{e_0}, tr) = \textit{lexp-state}(lexp^{e_0}, tr)$

$\textit{exp-state}(*exp^{e_0}, \{\mathrm{E}\}) = \textit{exp-state}(exp^{e_0}, \{\mathrm{E}\})$

$\textit{exp-state}(*^{e}exp^{e0}, tr)$

$= (\textit{exp-bt}(e_0) = S \wedge \textit{is-liftable}(type(e))) \rightarrow$

$\underline{\bigcup}_{loc \in aliases(e)} \{(loc, \{\mathrm{E}\})\}$

;

$\underline{\bigcup}_{loc \in aliases(e)} \{(loc, tr)\}$

$\textit{exp-state}(exp_1^{e_1} bop^{e} exp_2^{e_2}, tr) = \textit{exp-state}(exp_1^{e_1}, tr) \underline{\bigcup} \textit{exp-state}(exp_2^{e_2}, tr)$

FIG. 4.3 – *Phase de transformation—fonctions auxiliaires*

4.2 Annotations de transformation

Résoudre ces équations produit des états de transformation pour chaque instruction en combinant les temps de liaison des valeurs et les temps de liaison des contextes. Bien que ce ne soit pas explicitement montré dans les équations de flot de données, chaque construction est annotée en fonction de son état de transformation associé. La fonction *def-bt*() calcule le temps de liaison d'une affectation en prenant le majorant des temps de liaison de tous les emplacements potentiellement définis par l'affectation. La fonction *stmt-use*() annote les définitions avec le majorant des transformations des emplacements définis. La fonction *lexp-use*() prend un point de programme et un temps de liaison du contexte en ce point de programme. Elle rend un couple emplacement/transformation avec lequel l'état de transformation sera mis à jour. Un identificateur qui apparaît en membre gauche n'ajoute aucune transformation à l'état puisque seule l'adresse de la variable est utilisée et pas son contenu. La transformation pour un accès à un champ de structure est calculée en

appelant *lexp-use*() avec la structure. Une indirection de pointeur est calculée de la même façon en appelant *exp-use*() avec le pointeur.

La fonction *exp-use*() fait la même chose avec les expressions en membre droit. Les expressions constantes n'utilisent pas de variable et n'ajoute donc aucun couple emplacement/transformation à l'état. Notons que le temps de liaison du contexte n'est pas pris en compte puisque toutes les valeurs constantes peuvent être réifiées.

Déterminer la transformation pour un identificateur de variable utilise la fonction *loc-use*(). Cette fonction prend un emplacement et un temps de liaison de contexte et rend le couple emplacement/transformation approprié. Si l'emplacement est celui d'une structure, il retourne la transformation correspondant au majorant de tous les champs de la structure. Sinon, l'emplacement est celui d'une variable, la transformation corresponadnt au temps de liaison du contexte de la variable est retournée.

La transformation pour un identificateur de variable qui apparaît dans un contexte statique est déterminée en appelant la fonction avec ce contexte statique. Si l'emplacement est une structure et que tous les champs sont dynamiques, la transformation $\{R\}$ est retournée. Sinon, la transformation $\{E\}$ est retournée. Un identificateur de variable qui apparaît dans un contexte dynamique n'est considéré $\{E\}$ que si la variable est statique et que sa valeur peut être réifiée. Sinon, la transformation $\{R\}$ est retournée. C'est ainsi que les utilisations statiques non-réifiables dans des contextes dynamiques sont résidualisées. Une fois encore, comme l'analyse est sensible à l'utilisation, seules les instances nécessaires de la variable sont résidualisées.

Les expressions restantes sont traitées de façon similaire, en utilisant la fonction *loc-use*() pour l'information de structure et la fonction *aliases*() pour l'information d'alias.

4.3 Résumé

La phase de transformation détermine une transformation pour chaque construction. Ceci est obtenu en prenant un programme annoté de temps de liaison et en effectuant une analyse arrière qui réalise deux tâches. Premièrement, la transformation pour chaque utilisation de variable est déterminée en prenant en compte le temps de liaison de la variable et du contexte où elle est utilisée. Deuxièmement, les transformations pour les définitions de variable sont calculées. Le programme annoté résultant est alors utilisé pour la spécialisation. Les constructions annotées $\{E\}$ seront évaluées. Les constructions annotées $\{R\}$ seront résidualisées. Les affectations annotées $\{E, R\}$ seront à la fois évaluées et résidualisées. Les affectations annotées $\{\}$ correspondent à du code mort et sont simplement ignorées.

Chapitre 5

Tempo

Les analyses décrites dans cette thèse ont été mises en œuvre et intégrées dans Tempo, notre évaluateur partiel pour C. Ce chapitre donne une vue d'ensemble de l'intégralité du système Tempo. Premièrement, nous donnons un résumé des motivations principales du développement de cet évaluateur partiel, puis nous faisons une description du système. Nous expliquons comment l'analyse en deux phases s'intègre bien avec les autres phases de l'étape d'analyse et montrons comment les résultats de ces analyses pilotent différentes étapes de transformation.

5.1 Motivation

L'évaluation partielle atteint un niveau de maturité qui rend cette technique de transformation de programmes capable de s'attaquer à des langages réalistes et à des programmes d'application de taille réelle. Un certain nombre de points cruciaux doivent être résolus afin que cette approche devienne réellement utilisable.

Applications réalistes. Jusqu'à présent, la plupart des évaluateurs partiels ont été développés sans viser des applications réalistes. Maintenant que l'évaluation partielle aborde des programmes plus réalistes, elle fait face aussi à de nouveaux défis quand elle s'attaque à des applications de taille réelle. Cette nouvelle situation montre que l'ensemble usuel et passe-partout d'analyses et de transformations disponible dans les évaluateurs partiels traditionnels est de peu d'utilité devant certains besoins cruciaux des programmes réalistes. En conséquence, non seulement un évaluateur partiel a besoin d'offrir un ensemble extensible de transformations mais les transformations elles-mêmes devraient être développées en se basant sur des motifs de programmes trouvés dans des applications typiques d'un domaine donné.

Un besoin pour des principes de conception. Les programmes écrits dans des langages réalistes comme C exhibent une très large variété de situations où l'évaluation partielle peut être appliquée. Comme les évaluateurs partiels traitent des langages de plus en plus riches, leur taille et leur complexité augmentent de façon drastique. En résultat, il est désormais évident qu'il faut proposer des principes de conception pour structurer la complexité croissante des nouveaux évaluateurs partiels.

La spécialisation au temps de compilation est restrictive. Quand on étudie les composants de systèmes logiciels réels, il devient évident que spécialiser exclusivement les programmes au temps de compilation est restrictif. En fait, il existe de

nombreux invariants qui ne sont connus qu'au moment de l'exécution et peuvent alors être utilisés pour une spécialisation extensive. Cette situation apparaît, par exemple, quand un ensemble de fonctions met en œuvre des transactions sur des sessions. Quand une session est ouverte, beaucoup de fragments d'informations sont connus, mais seulement à l'exécution. Ils peuvent être utilisés pour spécialiser les fonctions qui réalisent les transactions réelles. Puis, quand la session est close, les invariants deviennent invalides et les fonctions spécialisées peuvent être éliminées. Bien que la spécialisation au temps de compilation semble impliquer des techniques différentes de la spécialisation au temps d'exécution, les deux formes de spécialisation sont conceptuellement identiques. Elles doivent donc être modélisées dans une approche uniforme plutôt qu'étudiées séparément. De plus, quand on considère des langages réalistes, le niveau d'effort requis pour développer un spécialiseur est tel que rechercher une certaine uniformité pour appréhender les deux cas de spécialisation est crucial.

La conception de Tempo résout tous les points cités. Ses analyses et ses transformations sont capables de traiter efficacement des programmes réalistes.

Une approche uniforme. Cette approche est hors-ligne en cela qu'elle sépare le processus d'évaluation partielle en deux parties : une étape d'analyse suivie d'une étape de transformation[JSS89, CD93]. La première partie comprend une phase de temps de liaison dont le but est de déterminer les calculs statiques et dynamiques pour un programme donné et une division (statique/dynamique) de ses entrées ; elle est suivie d'une phase de transformation qui détermine les transformations de chaque calcul. Ces transformations sont alors utilisées pour décider d'une *action de spécialisation* pour chaque construction du programme [CD90].

L'étape de transformation de l'évaluateur partiel effectue les transformations correspondant au programme annoté d'actions selon les valeurs de spécialisation fournies. La spécialisation est alors simplement guidée par l'information produite par l'étape d'analyse. À de nombreux points de vue, cette conception est très semblable à celle utilisée pour mettre en œuvre les langages de programmation. Pour un programme donné, de la même manière qu'un compilateur produit des instructions machine, l'étape d'analyse produit des transformations de programme. De la même manière qu'un système d'exécution exécute du code compilé, la phase de transformation (c'est-à-dire le spécialiseur) exécute les transformations de programme produites par l'étape d'analyse. De la même façon qu'un code compilé est exécuté de nombreuses fois avec des valeurs d'entrée différentes, un programme annoté d'actions peut être spécialisé de nombreuses fois selon des valeurs de spécialisation différentes. Puisque la spécialisation a été *compilée*, il est exécuté de façon très efficace. À cause de la séparation entre l'analyse et la transformation, cette dernière peut être mise en œuvre dans un langage différent de la première et de ce fait la mise en œuvre est facilitée. De façon plus importante, cette conception permet de traiter des programmes annotés d'actions de différentes manières. La dernière forme de spécialisation correspond à la production d'une extension génératrice [And94a, Ers77].

Spécialisation au temps de compilation et spécialisation au temps d'exécution. De manière plus intéressante, les actions peuvent être utilisées comme une base pour réaliser la spécialisation au temps d'exécution. En effet, les actions modélisent directement la forme des programmes résiduels puisqu'elles expriment comment chaque construction d'un programme doit être transformée. De fait, nous avons développé une stratégie pour réaliser une spécialisation au temps d'exécution basée sur les actions [CN96]. Essentiellement, un programme annoté d'actions est utilisé pour générer automatiquement des patrons de code source au temps de la

compilation. Puis, à l'exécution, les patrons compilés sont sélectionnés et complétés par des valeurs d'exécution avant d'être évalués. Cette nouvelle approche a de nombreux avantages : elle est générale puisqu'elle est basée sur une approche générale de développement d'évaluateurs partiels ; elle est portable puisque la plupart du processus de spécialisation est fait au niveau du source ; elle est efficace dans le sens où la spécialisation est amortie après quelques exécutions du code spécialisé.

Un aspect important de notre approche est que l'analyse d'un programme est identique qu'il soit spécialisé au temps de compilation ou au temps d'exécution. C'est une conséquence directe du type d'information calculée par l'étape d'analyse du système.

Applications systèmes. Tempo a été dirigé vers un domaine particulier à savoir les applications systèmes. Plus précisément, les fonctionnalités de Tempo ont été guidées par des motifs de programmes basés sur des études concrètes de nombreux programmes systèmes et en étroite collaboration avec des chercheurs en système. En résultat, la précision des analyses cruciales comme l'analyse d'alias et l'analyse de temps de liaison est adaptée à des programmes systèmes typiques. De plus, les transformations de programmes abordent les cas importants de tels programmes.

Plus généralement, cette nouvelle approche de conception d'un évaluateur partiel a eu une conséquence importante ; elle a clairement montré la nécessité d'une évaluation partielle *orientée module*. Les évaluateurs partiels traditionnels supposent qu'ils traitent un programme complet. Toutefois, les programmes systèmes sont de taille assez grande pour atteindre les limites de ce qu'une analyse de l'état de l'art comme l'analyse d'alias peut traiter [WL95]. Nous avons par conséquent fait en sorte que que l'on puisse spécialiser des morceaux d'un grand système.

Résumé. Tempo fournit une série de contributions concernant la conception et l'architecture d'un évaluateur partiel pour un langage réel. Nous présentons une architecture pour évaluateur partiel suffisamment puissante pour être utilisée pour des langages réalistes et des applications de taille réelle. Cette architecture a été utilisée pour développer un évaluateur partiel de programmes C. Contrairement à la plupart de systèmes existants, notre évaluateur partiel a été attentivement conçu pour traiter des opportunités de spécialisation spécifiques présentes dans un domaine particulier, à savoir les applications systèmes. Cette stratégie nous permet de mieux assurer l'applicabilité de l'évaluation partielle. Bien que dédiée à un domaine d'application particulier, l'architecture est néanmoins ouverte : des nouvelles transformations de programme peuvent être aisément introduites au niveau de l'analyse d'actions. Aussi, notre architecture a montré sa généralité dans le fait qu'elle permet à la fois la spécialisation au temps de compilation et la spécialisation au temps d'exécution. En conséquence, cette nouvelle forme généralisée de spécialisation élargit grandement la portée d'application de l'évaluation partielle.

Dans la prochaine section , nous présentons l'étape d'analyse. La section suivante décrit les différentes étapes de transformation.

5.2 Étape d'analyse

Comme le montre la Figure 5.1, l'étape d'analyse est constituée de quatre phases principales : le frontal, les analyses d'alias, de définition et de chaînes utilisation/définition, l'analyse de temps de liaison et l'analyse d'actions. L'analyse de temps de liaison est divisée en deux parties : la phase de temps de liaison et la phase de transformation. Le résultat de l'étape d'analyse est un programme annoté qui et utilisé pour spécialiser le programme soumis.

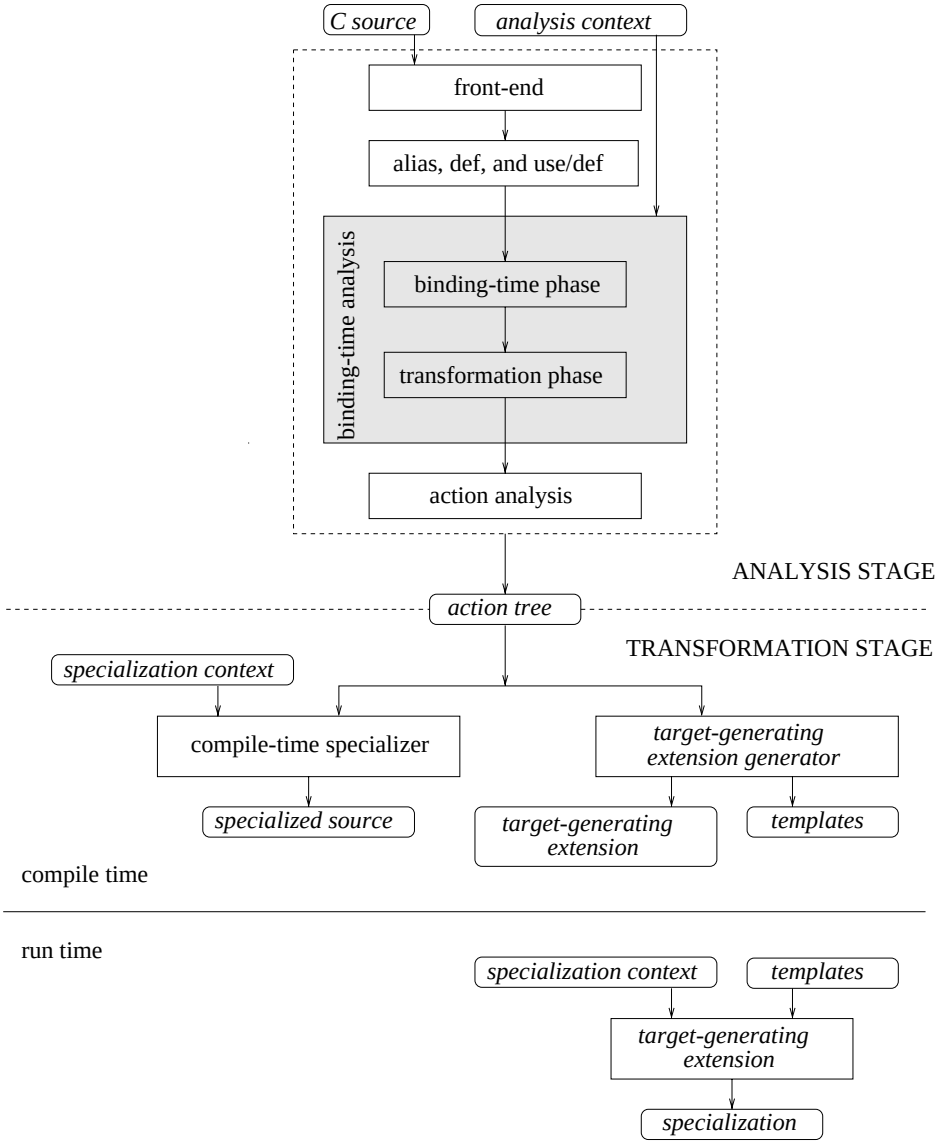

FIG. 5.1 – *Une vue générale de Tempo*

5.2.1 Frontal

Cette phase transforme un programme C en un arbre de syntaxe abstraite C (abrégé en AST pour *abstract syntax tree*). Nous n'avons pas écrit de nouvel analyseur mais avons plutôt réutilisé les composants de SUIF [WFW+94]. SUIF est un système pour l'expérimentation d'optimisation de programmes dans le contexte du code scientifique et des machines parallèles. Notre syntaxe abstraite n'est pas basée sur le format intermédiaire SUIF, qui est de trop bas niveau en ce qui nous concerne, mais sur une représentation intermédiaire utilisée par un programme SUIF qui retransforme ce format en code C. Avec quelques modifications mineures à SUIF, nous sommes capables de passer de programmes C à des AST de haut niveau et simples. En particulier, notre syntaxe abstraite comprend une construction unique de boucle et une unique conditionnelle. Elle ne comprend pas l'opérateur virgule ni les branchements.

Deux transformations supplémentaires importantes sont appliquées aux AST obtenus de SUIF. Tout d'abord, pour permettre les analyses compositionnelles, les instructions `goto` sont éliminées comme le suggèrent Erosa et Hendren [EH94]. Deuxièmement, le renommage rend chaque identificateur unique de façon à aider à la construction d'une mémoire statique aplatie (cf section 5.3). Ce renommage facilite aussi le dépliage des fonctions au temps de post-traitement.

5.2.2 Analyses d'alias, de définitions et de chaînes utilisation/définition

Comme le langage C offre l'utilisation de pointeurs, une analyse d'alias est cruciale pour déterminer l'ensemble d'alias de chaque variable dans un programme. Cette information permet au calcul des propriétés de temps de liaison des variables de prendre en compte les effets de bord. Notre analyse est très semblable à celles déjà existantes [EGH94, Ruf95]. Elle est basée sur le modèle "points-to" de synonymie. Elle est inter-procédurale, insensible au contexte et sensible au flot. Contrairement à d'autres contextes d'utilisation d'une analyse d'alias, l'évaluation partielle ne requière pas une information très précise. Cette situation est essentiellement due au type de calculs statiques que peut avoir à effectuer l'évaluation partielle. En effet, nous basant sur notre expérience, les calculs statiques dépendent d'invariants dont la validité suit typiquement un schéma très clair. Notre choix d'une analyse insensible au contexte est confirmé par l'étude de Ruf [Ruf95] qui montre que les bénéfices empiriques d'une analyse sensible au contexte restent encore à mesurer.

L'analyse prend en arguments un ensemble de fonctions, une fonction but, une description des paires initiales "points-to" et une description de l'effet des fonctions externes sur la relation "points-to". Les deux derniers arguments rendent possible une analyse d'alias orientée module. Cette caractéristique est cruciale pour pouvoir spécialiser des parties d'un système de grande taille. En supposant que les programmes soumis à spécialisation sont corrects, l'analyse ne traite pas séparément les paires "points-to" *possibles* et *définies* (cf [CWKZ90, Ruf95]).

L'analyse d'alias est complétée par une analyse de définitions qui calcule l'ensemble des emplacements mémoire qui *peuvent* être définis par l'instruction. De plus, une analyse de chaînes utilisation/définition calcule la mémoire non-locale affectée par chaque appel de fonction.

5.2.3 Analyse de temps de liaison

Nous avons développé une analyse de temps de liaison qui annote des programmes C avec leurs temps de liaison, étant donné un ensemble de fonctions,

des informations sur leurs alias et leurs effets de bord, une fonction but et une description de temps de liaison du contexte. Ce contexte comprend les paramètres de la fonction but, un état global et, comme pour l'analyse d'alias, les fonctions externes.

À l'instar de l'analyse d'alias, la conception de l'analyse de temps de liaison est menée selon les opportunités typiques de spécialisation qui apparaissent dans les programmes systèmes. Notre analyse de temps de liaison sépare les différentes tâches effectuées lors de cette analyse en deux phases distinctes. Premièrement, la *phase de temps de liaison* détermine si oui ou non une construction dépend de valeurs statiques. À ce stade, toutes les constructions qui ne dépendent que de valeurs statiques sont annotées comme statiques. Deuxièmement, la *phase de transformation* prend en compte comment le programme doit être spécialisé et annote le programme en conséquence. Les constructions qui doivent être évaluées sont annotées $\{E\}$ et celles qui doivent être résidualisées sont annotées $\{R\}$. Certaines constructions sont à la fois évaluées et résidualisées, dans ce cas elles sont annotées $\{E, R\}$.

L'analyse de temps de liaison est sensible au flot, au contexte, au retour et à l'utilisation. Cette précision permet de spécialiser efficacement des programmes systèmes existants.

5.2.4 Analyse d'actions

La phase de transformation de l'analyse de temps de liaison annote chaque construction avec une transformation. Ces annotations décrivent *quelles* transformations sont appliquées aux constructions au temps de spécialisation. L'analyse d'actions prend un programme annoté de transformations et affecte une *action* de spécialisation à chaque construction. Une action de spécialisation décrit *comment* une construction est transformée au temps de spécialisation.

Il y a quatre actions générales de spécialisation : *evaluate*, *reduce*, *rebuild* et *identity*. Toutes les constructions annotée avec la transformation $\{E\}$ auront une action *evaluate* (abrégé en *ev*). Ces constructions ne dépendent que de sous-composants annotés $\{E\}$ et sont par conséquent complètement évalués au temps de spécialisation. Les constructions avec une transformation $\{R\}$ seront annotées avec une action *reduce*, *rebuild* ou *identity*, selon la situation. L'action *reduce* (abrégé en *red*) est affectée à une occurrence de construction de langage qui peut être réduit au temps de spécialisation. Par exemple, une instruction conditionnelle est réduite quand son expression de test est $\{E\}$. L'action *rebuild* (abrégé en *reb*) annote une construction qui nécessite d'être reconstruite mais comporte encore des calculs $\{E\}$. L'action *identity* (abrégé en *id*) annote des fragments de programme purement $\{R\}$. Au temps de spécialisation, les constructions annotées *id* sont simplement recopiées tel quel dans le programme résiduel.

Les constructions annotées avec une transformation $\{E, R\}$ sont associées à deux actions de spécialisation, une pour chaque partie de la transformation. La partie E est translatée en une action *ev*. La partie R devient *red*, *reb* ou *id* selon le contexte où la construction est utilisée.

L'analyse d'actions n'applique aucune transformation associée à une construction ; elle décrit simplement comment la transformation sera effectuée. Par conséquent, spécialiser selon un programme annoté de transformations et selon un programme annoté d'actions produit le même programme résiduel. Cependant, les actions de spécialisations offrent un certain nombre d'avantages.

Une compilation plus avancée du processus de spécialisation. Traditionnellement, le spécialiseur est directement dirigé par l'information de temps de liaison. La complexité de l'interprétation de l'information de temps de liaison dépend de la complexité de l'information de temps de liaison elle-même. Pour des langages

réalistes comme le langage C, cette spécialisation implique l'interprétation d'une information de temps de liaison détaillée d'objets tels que des structures de données. Ce processus peut ralentir notablement la phase de spécialisation. Cette situation est améliorée en compilant l'information de temps de liaison en des actions avant la spécialisation.

Une définition plus précise des transformations de programmes. Définir les actions de spécialisation oblige explicitement le concepteur de l'évaluateur partiel à définir précisément l'ensemble des transformations de programmes nécessaires pour le langage donné. Non seulement cela fournit une meilleure documentation à l'utilisateur, mais cela définit aussi en détail la sémantique de la phase de spécialisation.

Une meilleure séparation entre le pré-traitement et la spécialisation. Comme les programmes annotés d'actions capturent l'essence du processus de spécialisation, ils peuvent être exploités de différentes manières. De fait, dans notre système d'évaluation partielle, les actions peuvent à la fois être interprétées et compilées, et utilisées dans la spécialisation au temps de compilation comme à celle au temps d'exécution. Cet espace d'applications montre la généralité des informations exprimées par les actions.

5.3 Étape de transformation

Un programme annoté a différents *back-ends*. Par back-ends, nous entendons étapes de transformations. Elles peuvent être divisées en deux : les actions peuvent être exploitées lors d'une spécialisation au temps de compilation ou lors d'une spécialisation au temps d'exécution.

5.3.1 Spécialisation au temps de compilation

Habituellement, le résultat du pré-traitement est utilisé pour la spécialisation au temps de compilation. De la même manière que les instructions machines produites par un compilateur peuvent être interprétées par un simulateur ou directement exécutées sur une machine, les actions sont soit interprétées ou compilées. Décrivons ces deux stratégies.

5.3.1.1 Interprétation des actions

Un interpréteur d'actions correspond au spécialiseur usuel. Il consiste à répartir différentes opérations sur chaque action d'un programme et à les exécuter en réalisant ainsi cette action. Comme l'information disponible avant la spécialisation a été extensivement exploitée, le spécialiseur est simple, a une structure claire et est efficace.

Conceptuellement, un spécialiseur combine un interpréteur standard pour effectuer les calculs statiques et un interpréteur non-standard pour reconstruire les fragments de programme correspondant aux calculs dynamiques. Contrairement à l'interprétation standard, les programmes ont parfois besoin d'être évalués de façon spéculative. Typiquement, pour les expressions conditionnelles avec une expression de test dynamique, les deux branches doivent être évaluées partiellement puisque la valeur du test est inconnue à la spécialisation. Un mécanisme est donc nécessaire pour traiter les branches indépendamment l'une de l'autre : elles doivent être traitées avec la même mémoire initiale disponible avant la prise en compte des branches.

cette situation demande de faire une copie de la mémoire et de la restaurer à une étape ultérieure.

Une approche pour résoudre ce problème est d'écrire un spécialiseur qui inclut un interprète de programmes C complet. Un inconvénient de cette approche est l'effort de développement majeur qu'elle implique de par la richesse syntaxique du langage C et aussi de par la grande variété de types de base et d'opérations de conversion entre eux (qui sont parfois dépendantes de la machine cible).

Une autre option est d'interfacer un spécialiseur avec un interprète C existant. Toutefois, les interprètes C existants n'offrent pas un modèle de mémoire qui supporte l'évaluation spéculative. La mise en œuvre de cette copie de mémoire nécessiterait que la mémoire de l'interprète soit en quelque sorte marquée et impliquerait un parcours coûteux de la mémoire.

Nous avons développé une troisième option qui nous permet d'utiliser un compilateur standard pour effectuer les calculs statiques, préservant ainsi la sémantique de ces calculs. La première partie de cette approche consiste à aplanir la portée des variables statiques d'un programme. En effet, seules les variables statiques peuvent être impliquées dans les calculs statiques. À cet effet, l'idée est de renommer toutes les variables du programme de telle façon qu'elles puissent être toutes globales. Bien sûr, cette transformation exclut d'avance la spécialisation de fonctions récursives qui seraient partiellement statiques. Cependant, les programmes systèmes n'exploitent habituellement pas cette facilité de langage. Une conséquence de cette conception est que toutes les structures de données statiques peuvent être facilement copiées pour mettre en œuvre l'évaluation spéculative. De plus, l'appel à des fonctions externes (c'est-à-dire qui ne sont pas traitées par le spécialiseur) peut être fait facilement puisque la disposition de la mémoire est compatible.

L'autre partie de notre approche consiste à encapsuler les fragments purement statiques (*ev*) dans des fonctions C. Ces fonctions C peuvent être directement compilées par un compilateur standard C et liées au spécialiseur. Le spécialiseur se concentre ainsi uniquement sur les opérations chargées de reconstruire des fragments de programmes. Pour traiter un fragment *ev*, le spécialiseur appelle simplement la fonction C qui effectue les calculs correspondants.

L'idée d'utiliser un compilateur standard pour effectuer une partie ou tout le processus de spécialisation n'est pas nouvelle. Andersen [And94a] utilise une approche similaire dans son évaluateur partiel (C-Mix) en produisant des extensions génératrices. Cependant, cette gestion de la mémoire est plus complexe en ce que chaque objet en entrée est indexé par un numéro de version et possède une "fonction objet" qui peut sauvegarder ou restaurer sa valeur et la comparer à une autre copie. Ces opérations ont pour but de partager des copies d'un même objet (c'est-à-dire une *matrice*) entre plusieurs mémoires statiques. Toutefois, dans notre cas, la spécialisation orientée module réduit grandement le besoin de telles spécialisations. En effet, contrairement à C-Mix qui requière qu'un programme soit spécialisé tout d'un coup, Tempo peut spécialiser séparément des fragments de programme. De plus, C-Mix contient une mémoire symbolique, utilisée, par exemple, quand des fonctions sont dépliées ou quand on manipule des pointeurs sur des objets dynamiques. À l'opposé, le spécialiseur de Tempo ne contient que la mémoire C statique utilisée par les fonctions *ev*. Le dépliage est vu comme une opération de post-traitement de la spécialisation au temps de compilation. Cette approche simplifie grandement le spécialiseur et ne dégrade pas la qualité des programmes spécialisés.

5.3.1.2 Compilation des actions

L'alternative naturelle à l'interprétation des actions est leur compilation. Le même système d'"exécution" que celui décrit précédemment pour la spécialisation, est réutilisé. En effet, les fragments *ev* peuvent toujours être empaquetés

dans des fonctions comme pour l'interprétation des actions, et ainsi être traités directement par le compilateur C. Le problème principal dans la compilation des actions est de compiler les calculs partiellement statiques, c'est-à-dire de reconstruire le code résiduel. Un simple compilateur d'actions est une tâche presque triviale à réaliser, comme le montrent Consel et Danvy [CD90]. Un processus de compilation similaire est aussi connu sous le nom d'extension génératrice [And92, And94a, BW93b, Ers77]. Une différence est que cette dernière approche est basée directement sur l'information de temps de liaison et donc beaucoup plus compliquée qu'un compilateur d'actions.

5.3.2 Spécialisation au temps d'exécution

Les actions peuvent être utilisées non seulement pour spécialiser des programmes au temps de compilation mais aussi pour effectuer la spécialisation au temps d'exécution. En fait, la spécialisation au temps d'exécution basée sur des actions n'est qu'une autre manière d'exploiter cette information. En effet, une action décrit comment transformer une construction et par conséquent un arbre d'actions décrit précisément l'ensemble des programmes spécialisés possibles. Cet ensemble peut être formellement défini par une grammaire d'arbres.

Nous avons développé un interprète abstrait qui produit une grammaire d'arbres pour un programme annoté d'actions donné. Comme simple exemple de ce que cette analyse produit pour un tel programme, considérons le fragment d'arbre d'actions suivant :

reb(`PLUS`*(id(*`VAR("x")`*), ev(...)))*

Supposons un fragment *ev* de type entier, notre analyse produit alors la règle de dérivation ci-dessous :

`L` → `PLUS(VAR("x"), HOLE(INT))`

Une fois produite, la grammaire d'arbre est utilisée pour générer des patrons [KEH93], c'est-à-dire des fragments de code source paramétrés par des "trous" pour les valeurs à l'exécution. Pour la règle de grammaire ci-dessus, nous obtenons le patron suivant :

`x +` *int_hole*

La grammaire d'arbres permet aussi de compiler les patrons dans leur contexte, c'est-à-dire en perdant l'information de flot de contrôle, en utilisant un compilateur standard. En conséquence, la qualité des patrons compilés est aussi bonne que le compilateur utilisé. Précisément, dans notre mise en œuvre d'un spécialiseur au temps d'exécution, nous avons utilisé le compilateur GNU C.

Une caractéristique clef de notre approche est que la spécialisation au temps d'exécution revient seulement à exécuter les calculs statiques, à sélectionner les patrons, à remplir les trous des patrons avec des valeurs d'exécution et à effectuer les branchements entre patrons. La simplicité de ces opérations fait que la spécialisation au temps d'exécution est un processus très efficace qui ne nécessite en moyenne, selon les expérimentations préliminaires, que très peu d'exécutions du code spécialisé pour amortir le coût de spécialisation.

Une description complète de notre approche de la spécialisation au temps d'exécution est présentée ailleurs [CN96, NHCL96].

5.4 Résumé

Nous avons présenté une approche de conception d'évaluateurs partiels pour des langages réalistes. Tempo est basé sur une approche générale qui consiste à séparer le processus en deux parties : l'étape d'analyse compile, après un certain nombre d'analyses statiques, des programmes en des actions et l'étape de transformation exécute alors ces actions pour effectuer la spécialisation proprement dite. Comme nous l'avons montré, la séparation a bon nombre d'avantages et, en particulier, rend possible l'intégration à la fois de la spécialisation au temps de compilation et de la spécialisation au temps d'exécution dans le même système.

Une deuxième caractéristique clef qui différencie Tempo des évaluateurs partiels standards est que cet évaluateur partiel a été développé avec un domaine d'applications particulier à l'esprit, à savoir le code système. Cette décision vient de la conviction qu'un évaluateur partiel général ne serait pas capable d'effectuer les optimisations spécifiques désirées. Nous avons en conséquence opté pour une approche ascendante qui consiste à étudier les opportunités de spécialisations d'applications spécifiques et ensuite à prendre des décisions de conception basées sur ces études.

Chapitre 6

Résultats expérimentaux

L'évaluateur partiel Tempo est utilisé pour spécialiser une grande variété d'applications existantes, complexes et du monde réel. Dans cette section, nous résumons les applications qui ont déjà été spécialisées par Tempo. Nous donnons aussi deux exemples issus de ces applications qui montrent comment les fonctionnalités de l'analyse de temps de liaison, décrite plus tôt, permettent d'exploiter les données statiques.

6.1 Applications spécialisées par Tempo

Nous appliquons actuellement Tempo à une large variété de programmes afin de faire un bilan de son efficacité. Tout d'abord, nous présentons les résultats de l'application de Tempo aux programmes systèmes. De plus, nous obtenons des accélérations significatives pour plusieurs algorithmes numériques et procédures de traitement d'images. Comme l'étape d'analyse est utilisée à la fois pour la spécialisation au temps de compilation et pour celle au temps d'exécution, nous donnons des exemples pour ces deux types de spécialisation. Enfin, nous expliquons comment Tempo est utilisé pour générer des applications à partir de spécifications.

6.1.1 Systèmes d'exploitation

Ces dernières années, les projets de recherches majeurs ont concentré leurs efforts sur la conception et la mise en œuvre de systèmes d'exploitation à la fois hautement paramétrés et efficaces. Ces buts apparemment antinomiques ont conduit les chercheurs à élargir le spectre des techniques utilisées dans le développement de systèmes. Plus précisément, des techniques de langages de programmation ont été introduites pour effectuer une tâche aussi critique que l'adaptation ou personnalisation de composants de systèmes en fonction de paramètres donnés. Dans ce contexte, l'évaluation partielle est devenue une technique clef pour développer des systèmes d'exploitation adaptatifs. Parmi les exemples de tels projets, citons Spin [BSP+95], ExoKernel [EKO95], Scout [MMO+94] et Synthetix [CPW93, CPW94].

Les travaux antérieurs ont montré que la spécialisation de composants de systèmes d'exploitation en fonction de valeurs d'états systèmes apparaissant fréquemment peut produire des accélérations significatives [Mas92, MP89, PMI88, VMC96a]. Prenons par exemple, le système de fichiers d'Unix. Une opération telle que `read` sur un fichier est écrite dans un style générique afin de s'adapter à une large variété d'utilisations. À chaque lecture, l'opération `read` interprète l'état système pour déterminer les opérations de bas niveau spécifiques qu'elle doit exécuter. De nombreuses parties de l'état système deviennent connues du système d'exploita-

tion à l'ouverture d'un fichier : quel fichier est lu, quel processus effectue la lecture, le type du fichier, la taille de ses blocs, si l'*inode* est en mémoire, etc. L'opération `read` peut être spécialisée à l'ouverture pour éliminer le sur-coût d'interprétation dépendant de cette information.

Un autre exemple est le protocole Remote Procedure Call (RPC) [Sun90, RAA+92]. Ce protocole de système d'exploitation fournit une interface entre un client sur une machine locale et un serveur sur une machine distante pour faire en sorte qu'une procédure distante semble être locale. Les opérations principales effectuées par RPC sont le codage et décodage de données d'une représentation dépendante d'une machine en une représentation indépendante d'un réseau et de gérer les échanges de messages sur le réseau. RPC est écrit de façon générique, inclut un choix de protocole de transport (TCP ou UDP), différentes mises en œuvre de codage/décodage et un ensemble de fonctions pour coder/décoder de l'information de chaque type de données possible. Avant que tout appel de procédure à distance se fasse, il faut l'initialiser, fournissant alors des valeurs d'état système comme les emplacements mémoire et la taille des buffers, de même que toutes les structures de protocoles et des informations de codage/décodage. Spécialiser RPC à l'initialisation en fonction de ces valeurs élimine les opérations qui dépendent de ces valeurs.

Le système d'exploitation Synthesis combine ce genre de spécialisation avec un certain nombre d'autres points et obtient des gains significatifs d'efficacité, allant d'un facteur 3 à 56 [Mas92, MP89, PMI88]. Nous avons effectué nos propres études pour déterminer quelle accélération pouvait être obtenue par la simple spécialisation. Nous voulions de plus déterminer si ces accélérations pouvaient être obtenues en spécialisant un système d'exploitation existant plutôt qu'un système comme Synthesis qui a été spécialement écrit pour bien se spécialiser. Enfin, l'appel système de lecture fichier de Hewlett-Packard HP-UX a été spécialisé à la main [PAB+95]. Des accélérations, allant de 1.1 à 3.6 selon la taille des données lues, ont été obtenues. De façon similaire, le RPC de Chorus a été aussi spécialisé à la main [VMC96a]. Spécialiser le client a donné une accélération de 3.7, avec une accélération globale de 1.5.

Ces études démontrent l'intérêt de la spécialisation, mais elles ont été faites à la main. Un des buts premiers de Tempo est d'obtenir *automatiquement* de telles accélérations. Comme déjà mentionné, Tempo a été spécialement conçu pour traiter des programmes systèmes. Tempo a été appliqué au code client comme au code serveur du SUN RPC, version 1984 [MMV+97, MVM97]. Le code non spécialisé représente environ 3200 lignes de code.

Pour établir l'efficacité de la spécialisation automatique de tels composants, nous avons comparé les performances du code original du SUN RPC avec le code spécialisé par Tempo. Le programme test, représentatif d'une application réseau, utilisait des appels de procédures à distance pour échanger de grandes quantités de données. Ce banc de test a été effectué sur deux plates-formes différentes afin de vérifier que les résultats obtenus n'étaient pas particuliers à une plate-forme donnée. La première plate-forme était composée de deux stations de travail Sun IPX 4/50 sous SunOS 4.1.4 avec 32 MB de cartes mémoire Fore Systems ESA-200 ATM et un lien ATM à 100 Mbits/sec. La deuxième plate-forme était constituée de deux Pentium PC 166MHz sous Linux avec 96M de mémoire et une connexion réseau Fast-Ethernet à 100Mbits/s. Tous les programmes ont été compilés par `gcc` version 2.7.2 avec l'option d'optimisation `-O2`.

Une boucle d'appel de procédure à distance contient le codage et décodage des données client, la gestion des messages et le temps de transmission le long du câble physique. Les temps d'exécution des fonctions client non-spécialisées et spécialisées ont été enregistrés. L'accélération des fonctions spécialisées est montrée en Tableau 6.1. Les temps ont été de plus enregistrés pour la boucle complète du RPC. Cela donne l'accélération globale, montrée en Tableau 6.2. Ces temps représentent

Taille du tableau	IPX/SunOs			PC/Linux		
	Original	Spécialisé	Speedup	Original	Spécialisé	Speedup
20	0.047	0.017	**2.7**	0.071	0.063	**1.1**
100	0.20	0.057	**3.5**	0.11	0.069	**1.5**
250	0.49	0.13	**3.7**	0.17	0.08	**2.1**
500	0.99	0.30	**3.3**	0.29	0.11	**2.6**
1000	1.96	0.62	**3.1**	0.51	0.17	**3**
2000	3.93	1.38	**2.8**	0.97	0.29	**3.3**

TAB. 6.1 – *Performances du marshaling client en ms*

Taille du tableau	IPX/SunOs			PC/Linux		
	Original	Spécialisé	Speedup	Original	Spécialisé	Speedup
20	2.32	2.13	**1.08**	0.69	0.66	**1.00**
100	3.32	2.74	**1.20**	0.99	0.87	**1.10**
250	5.02	3.60	**1.39**	1.58	1.25	**1.20**
500	7.86	5.23	**1.50**	2.62	2.01	**1.30**
1000	13.58	8.82	**1.53**	4.26	3.17	**1.34**
2000	25.24	16.35	**1.54**	7.61	5.68	**1.34**

TAB. 6.2 – *Performance du cycle complet en ms*

une moyenne de 10000 exécutions.

Nous voyons, Tableau 6.1, que le code souche client spécialisé est de 2.75 à 3.75 fois plus rapide sur le IPX/SunOS et entre 1.2 et 3.65 fois plus rapide sur le PC/Linux. Comme prévu, l'élimination des calculs dépendant des valeurs disponibles à l'initialisation améliore de façon significative les fonctions de marshaling. Les temps de boucles du Tableau 6.2 indiquent que l'accélération obtenue de la spécialisation du marshaling client a un impact significatif sur l'accélération globale. Ces accélérations ne sont pas fortes, toutefois, puisque ces temps tiennent compte des portions de code non spécialisées et du temps de transmission.

RPC dépend lourdement des structures de données intermédiaires dues à la modularité et à la portabilité, qui ajoutent des opportunités où l'évaluation partielle peut améliorer les performances. Au lieu d'éliminer les calculs qui dépendent de valeurs connues à l'initialisation, Tempo a été utilisé pour éliminer ces structures de données intermédiaires, une technique connue sous le nom de *élimination des copies* [VMC+96c]. L'approche consiste, premièrement, à équiper le programme pour maintenir un *historique* des copies de structures de données, puis à le transformer automatiquement en code augmenté. Le résultat final est qu'au lieu d'être propagée de nombreuses fois à travers chaque couche du RPC, la donnée n'est finalement copiée qu'une seule fois. Cette recherche étant en cours, les mesures quantitatives de l'efficacité de cette approche ne sont à ce jour pas disponibles.

6.1.2 Algorithmes numériques et routines de traitement d'images

Tempo a été aussi appliqué à une variété d'algorithmes numériques et de routines de traitement d'images. L'intégration de Romberg approxime l'intégrale d'une fonction sur un intervalle en utilisant des estimations trapézoïdales. L'interprétation par spline cubique approxime une fonction en utilisant un polynôme de degré 3. Les polynômes de Chebyshev approximent les fonctions continues sur un intervalle

Application	Invariant	Temps Original	Spécialisation CT Temps	**Speedup**
romberg	n=2	7.00	5.00	**1.40**
	n=4	20.00	14.00	**1.43**
	n=6	60.00	40.00	**1.50**
	n=8	196.00	138.00	**1.42**
spline cubique	n=10	11.20	6.20	**1.81**
	n=20	23.30	14.30	**1.63**
	n=30	35.50	20.90	**1.70**
chebyshev	n=5	68.00	12.00	**5.67**
	n=10	246.00	28.00	**8.79**
	n=15	520.00	49.00	**10.61**
	n=20	913.00	75.00	**12.17**
FFT	n=16	43.06	7.97	**5.40**
	n=32	101.81	19.42	**5.24**
	n=64	225.81	45.55	**4.96**
	n=128	483.34	134.61	**3.59**
dither	4,5,9,5	2.06	0.39	**5.28**
in ppm	4,5,9,5	3.03e^6	1.72e^6	**1.76**

TAB. 6.3 – *Algorithmes numériques et routines de traitement d'images spécialisés par Tempo*

donné. On trouve ces trois programmes dans le travaux de Glück, Nakashige et Zöchling sur l'application de l'évaluation partielle aux algorithmes mathématiques [GNZ95]. Ils sont basés sur des algorithmes donnés par Kincaid and Cheney [KC91] et Press *et al.* [PTVF93]. Ces programmes calculent leurs approximations en itérant jusqu'à convergence leur algorithme correspondant.

La transformation de Fourier traduit une donnée d'un domaine de temps vers un domaine de fréquences. L'entrée est une séquence de points et le nombre de points, la sortie est le coefficient de Fourier de chaque point. La transformation rapide de Fourier (FFT en abrégé) est l'algorithme le plus rapide connu pour calculer la transformation de Fourier. Le programme que nous avons utilisé a été écrit par Dave Edelblute. [1]

Le *dithering* transforme une image digitale en réduisant le nombre de couleurs utilisées dans l'image. Le programme *ppmdither* que nous avons utilisé est extrait de la librairie portable pixmap très répandue. [2]

En comparaison avec les exemples des programmes systèmes, ces programmes sont relativement petits, contenant moins de 100 lignes de code. Ces programmes sont compilés par `gcc` version 2.7.2 avec l'option d'optimisation `-O2` et exécutés sur Sun SparcStation 20 Model 70 sous SunOS 4.1.4 avec 96 MB de mémoire.

La spécialisation de l'intégration de Romberg, de l'interpolation par spline cubique et de l'approximation de Chebyshev ont été effectuées en considérant que le nombre d'itérations, n, était connu et en spécialisant alors le programme selon cette valeur. Quand le nombre d'itérations est connu, le flot de contrôle devient connu, ce qui autorise la spécialisation à dérouler des boucles et propager des constantes.

[1] Disponible par ftp sur `ftp.usc.edu/pub/C-numanal/fft-stuff.tar.gz` dans le fichier `asum.2`.

[2] Copyright (C) 1991 by Christos Zoulas. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Available by ftp from *e.g.* `ftp.stanford.edu/class/cs248/netpbm/ppm/ppmdither.c`.

Cela génère de plus du code en ligne directe contenant plus de constantes. Celui-ci peut être compilé ultérieurement en du code efficace. Dans chaque cas, différentes valeurs de n ont été testées pour observer l'impact correspondant.

Les résultats, que l'on peut voir dans le Tableau 6.3, montrent que la spécialisation de l'intégration de Romberg et de l'interpolation par spline cubique produisent de faibles accélérations, allant de 1.4 à 1.8 . Au delà des instructions de test et de branchement éliminées par le déroulement de la boucle, les calculs éliminés comprennent l'exponentiation entière, la multiplication flottante et la division flottante. Les accélérations pour l'approximation de Chebyshev sont plus significatives ; le programme spécialisé s'exécute jusqu'à 15 fois plus vite que l'original. L'origine principale de cette accélération vient du fait que l'approximation contient des appels à la fonction de librairie cosinus très coûteuse à l'exécution. Des dépliages de boucles découlent des valeurs constantes, ce qui permet d'effectuer ces appels librairies coûteux pendant la spécialisation.

FFT a été spécialisé en considérant que le nombre de points, n, en entrée de la fonction, est connu. Similairement, cette information contraint le flot de contrôle de l'algorithme et la spécialisation rend explicite ce flot de contrôle. Le déroulement de boucle et la propagation de constantes fournit quelque accélération. Comme pour l'approximation de Chebyshev, les fonctions librairies sinus et cosinus sont éliminées à la spécialisation, ce qui participe aux accélérations significatives obtenues (voir Tableau 6.3).

Le programme de *dithering* que nous avons spécialisé est constitué d'une procédure d'initialisation de tableaux utilisés dans le processus de *dithering* et la procédure de *dithering* elle-même qui calcule la nouvelle valeur de chaque *pixel* dans l'image. Ce programme prend des lignes de commande qui spécifie la taille de la matrice dither et le nombre de nuances de rouge, vert et bleu. Nous avons utilisé cette information pour spécialiser un programme avec une matrice 4x4, 5 nuances de rouge, 9 de vert et 5 de bleu. Cette information permet que la propagation de constante soit effectuée à la spécialisation. Une accélération substantielle vient, de plus, d'une transformation connue sous le nom de *réduction de force*, qui transforme les multiplications entières en opérations de décalage et d'addition et les divisions par une puissance de 2 par des opérations de décalage. Les accélérations obtenues par ces optimisations sont montrées dans le Tableau 6.3.

6.1.3 Génération d'applications

La spécialisation est aussi utilisée dans de nombreuses approches pour concevoir des *générateurs d'applications*, programmes qui traduisent automatiquement des spécifications en des applications [BVT+94, BBH+94, TC96]. Tempo joue un rôle clef dans l'approche présentée dans [TC96], qui définit un cadre de conception de générateur d'application structuré en deux niveaux. Le premier niveau est basé sur des idées bien établies de composants génériques, qui aident à la conception de la structure, et permet la réutilisation de code. Plus précisément, une machine abstraite est écrite, définissant une collection d'opérations adaptées à la construction d'applications dédiées à un domaine. Le second niveau consiste en la définition d'un micro-langage qui permet de composer ces opérations. Ce micro-langage fournit une interface à la machine abstraite, supportant la réutilisation de code entre des applications semblables.

La génération d'application permet de générer automatiquement des applications qui sont usuellement écrites à la main. Les applications générées automatiquement demandent moins de temps à la création, sont plus faciles à maintenir et sont moins enclines aux erreurs. Le désavantage est le sur-coût introduit par le processus de génération. Par exemple, la mise en œuvre de la machine abstraite ajoute un niveau d'interprétation. De plus, l'approche demande l'utilisation d'un interprète

pour le micro-langage, ce qui constitue aussi un sur-coût.

Le rôle de l'évaluation partielle dans cette approche est de supprimer automatiquement ces sur-coûts et de générer automatiquement une application qui soit compétitive en taille et rapidité par rapport à une version écrite à la main. Cette approche utilise Tempo comme évaluateur partiel et est actuellement appliquée pour générer automatiquement des pilotes de cartes vidéo comme les pilotes SVGA pour le serveur XFree86 X11. La mise en œuvre de la machine abstraite comprend environ 1000 lignes de code, l'interpréteur étant de grosso modo 5000 lignes. Les résultats des tests préliminaires sont encourageants, indiquant que la plupart si ce n'est l'intégralité du sur-coût est effectivement éliminé par Tempo.

6.2 Spécialisation au temps d'exécution basée sur des patrons

La spécialisation au temps d'exécution permet que des programmes puissent être optimisés en exploitant des informations dynamiques, ce qui a montré améliorer considérablement les performances de codes allant du système d'exploitation Synthesis[PMI88] aux programmes graphiques [Loc87, PLR85]. Cette amélioration est obtenue en exploitant les invariants d'exécution pour générer le code au fur et à mesure de l'exécution du programme. Un tel code peut être plus optimisé que ce qui peut être généré en se basant sur l'information statique disponible.

Notre approche basée sur des patrons de la spécialisation au temps d'exécution est efficace et produit du code de haute qualité [CN96]. Nous utilisons une extension génératrice cible, comme expliqué dans le Chapitre 1. Comme la génération du code a lieu lors de l'exécution du programme, le coût de la génération de code est minimisé. Le plus possible de calculs sont effectués à la compilation. Ceci permet d'identifier les patrons et de les pré-compiler au temps de compilation. Un spécialiseur hautement optimisé est de même généré, il est spécifiquement profilé pour ne générer que les spécialisations nécessaires pour assembler ces patrons. Puisque les patrons sont compilés au moyen de la technologie de compilation existante, la qualité du code de la spécialisation résultante hérite des techniques de génération de code de ces compilateurs, comme l'allocation de registre et l'enchaînement d'instructions.

Notre approche de la spécialisation au temps d'exécution est complètement automatique. Les analyses statiques font tout le travail pour déterminer quelles transformations appliquer et comment les appliquer. L'identification des patrons comme la production de l'extension génératrice sont toutes les deux basées sur les résultats des analyses de temps de liaison et de transformation. Il est crucial que ces analyses soient effectuées à la compilation pour que leur coût n'apparaisse pas à l'exécution. La précision de leurs résultats agit directement sur la qualité du code généré à l'exécution puisqu'ils sont utilisés pour identifier les patrons mais aussi pour générer l'extension génératrice.

Nous avons établi l'efficacité de cette approche en spécialisant (au temps d'exécution) des programmes numériques et des routines de traitement d'images mentionnées ci-dessus. Les résultats sont présentés dans le Tableau 6.4 [CN96]. Pour chaque application spécialisée, nous avons enregistré le temps passé à l'exécution à générer le code spécialisé et le temps pour exécuter le code spécialisé. L'accélération est calculée en comparant les temps d'exécution du code spécialisé avec le code compilé statiquement. Le point de rupture (la colonne étiquetée "=") est le nombre de fois que le programme spécialisé doit être exécuté pour amortir le temps de génération.

Le point de rupture est le facteur critique quand on fait le bilan du code spécialisé

Application	Invariant	Temps Original	Spécialisation RT Génération	Temps	**Speedup**	=	CT/RT **Ratio**
romberg	n=2	7.00	61.00	6.00	**1.17**	**61**	**83%**
	n=4	20.00	191.00	16.00	**1.25**	**48**	**88%**
	n=6	60.00	489.00	47.00	**1.28**	**38**	**85%**
	n=8	196.00	1375.00	165.00	**1.19**	**45**	**84%**
spline cubique	n=10	11.20	203.50	8.40	**1.33**	**73**	**74%**
	n=20	23.30	428.00	17.60	**1.32**	**76**	**81%**
	n=30	35.50	653.20	26.90	**1.32**	**76**	**78%**
chebyshev	n=5	68.00	164.00	13.00	**5.23**	**3**	**92%**
	n=10	246.00	561.00	30.00	**8.20**	**3**	**93%**
	n=15	520.00	1199.00	54.00	**9.63**	**3**	**91%**
	n=20	913.00	2110.00	88.00	**10.38**	**3**	**85%**
FFT	n=16	43.06	194.02	11.14	**3.87**	**7**	**72%**
	n=32	101.81	457.83	26.58	**3.83**	**7**	**73%**
	n=64	225.81	1054.98	61.55	**3.67**	**7**	**74%**
	n=128	483.34	2392.48	183.83	**2.63**	**8**	**73%**
dither	4,5,9,5	2.06	24.00	0.50	**4.12**	**17**	**78%**
in ppm	4,5,9,5	$3.03e^6$	24.00	$1.68e^6$	**1.81**	**17**	**103%**

TAB. 6.4 – *Spécialisation au temps d'exécution et comparaison au temps de compilation*

à l'exécution, puisqu'il détermine combien de fois une fonction doit être appelée pour que la spécialisation au temps d'exécution soit rentable. Un point de rupture bas est obtenu quand on combine un temps de génération bas et une forte accélération.

En regardant le Tableau 6.4, nous notons que les points de rupture les plus hauts sont ceux de l'intégration de Romberg et de l'interpolation par spline cubique, allant de 38 à 76. Les temps de génération ne sont pas particulièrement hauts mais les accélérations sont en même temps relativement basses. L'approximation de Chebyshev obtient les meilleurs points de rupture, allant de 3 à 8, à cause des accélérations fortes obtenues. Le *dithering* a aussi un point de rupture bas, 17, dû à son temps de génération bas et à une assez bonne accélération.

6.3 Exploitation des fonctionnalités d'analyse

Nous avons vu que l'application de Tempo à un ensemble d'applications réalistes, que ce soit au temps de compilation ou au temps d'exécution, produit des accélérations significatives. Donnons maintenant deux exemples illustrant les sensibilités au flot, au contexte, au retour et à l'utilisation et leur efficacité dans la spécialisation de ces applications.

Calcul partiel généralisé

Le premier exemple illustre une amélioration du temps de liaison issu d'une analyse sensible au flot. La Figure 6.1 montre un fragment de programme où une valeur dynamique est affectée à la variable `x`, puis où un certain nombre d'instructions utilisent `x`. Puisque l'affectation rend `x` dynamique, toutes ses utilisations ultérieures sont aussi dynamiques. Si, cependant, il est connu que certaines valeurs de `x` sont plus courantes que d'autres, le programme peut être transformé de façon à exploiter cette information. Précisément, une conditionnelle est introduite pour déterminer si `x` est en fait égal à une valeur courante. Si c'est le cas, en ajoutant alors explicitement une affectation dans la branche positive de la conditionnelle et en copiant les

Code source original

```
x = une expression dynamique;
instructions dans lequel x est considéré comme dynamique;
```

Code transformé

```
x = une expression dynamique;
if (x == common_case_value) {
  x = common_case_value;
  instructions dans lequel x est considéré comme statique
} else {
  instructions dans lequel x est considéré comme dynamique
}
```

FIG. 6.1 – *Exemple de calcul partiel généralisé basé sur une analyse sensible au flot*

instructions qui utilisent `x` dans les deux branches, on peut spécialiser la branche positive en fonction de cette valeur courante de `x`. Cet exemple de *calcul partiel généralisé* [Fut88, FN88] s'est avéré utile à la fois dans l'exemple du SUN RPC et dans la génération d'application. L'amélioration du temps de liaison est possible parce que l'analyse de temps de liaison est sensible au flot.

Sun RPC

Le second exemple montrent comment les sensibilités au retour et à l'utilisation sont cruciales pour spécialiser l'extrait du code client du DUN RPC[Mic89] que l'on voit en Fig. 6.2. Le code RPC est organisé comme suit. La fonction initiale `Xdr_bytes()` code la donnée dans le buffer client en faisant un appel à `Xdr_u_int()` et en vérifiant si la valeur de retour est un succès ou un échec. En suivant inter-procéduralement cet appel, nous arrivons à la fonction `Xdrmem_putlong` qui réalise le codage proprement dit. En plus de faire cet encodage (effectué par l'affectation à `*(xdrs->x_private)`), cette fonction décrémente aussi la taille du buffer du client `xdrs->x_handy`, incrémente le pointeur sur le buffer du client `xdrs->x_private`, et renvoie une erreur (`0`) si le buffer était vide (`xdrs->x_handy <0`) et un succès(`1`) sinon.

SUN RPC demande qu'un appel de procédure à distance soit initialisé avant d'être appelé. Beaucoup de valeurs, telles que savoir si on code ou décode une donnée, la taille du buffer client, deviennent connues après cette initialisation, ce qui crée des opportunités de spécialisation. En fait, la plupart des calculs de la Figure 6.2 dépendent de ces valeurs connues.

Si une analyse n'est pas assez précise, ces valeurs sont toutefois perdues. Une analyse *in*sensible à l'utilisation forcerait presque toutes les constructions à être résidualisées. La Figure 6.2 montre les annotations faites par une analyse insen-

```
int Xdr_bytes(...)
{
  ⋮
  if ((Xdr_u_int(xdrs, sizep) != 0) == 0)
    return 0;
  ⋮
}

int Xdr_u_int(struct str1 *xdrs, unsigned int *up)
{
  ⋮
  return Xdr_u_long(xdrs, up);
  ⋮
}

int Xdr_u_long(struct str1 *xdrs, unsigned int *ulp)
{
  ⋮
  if ((int)(xdrs→x_op) == 0)
    return Xdrmem_putlong(xdrs, (int *) ulp);

  if ((int)(xdrs→x_op) == 2)
    return 1;

  return 0;
  ⋮
}

int Xdrmem_putlong(struct str1 *xdrs, int *lp)
{
  xdrs→x_handy = xdrs→x_handy - 4u;
  if (xdrs→x_handy < 0)
    return 0;
  *(xdrs→x_private) = htonl(*lp);
  xdrs→x_private = 4u + xdrs→ x_private;
  return 1;
}
```

FIG. 6.2 – *Insensibilité à l'utilisation pour du code de systèmes d'exploitation*

```
int Xdr_bytes(...)
{
  ⋮
  if ((Xdr_u_int(xdrs, sizep) != 0) == 0)
    return 0;
  ⋮
}

int Xdr_u_int(struct str1 *xdrs, unsigned int *up)
{
  ⋮
  return Xdr_u_long(xdrs, up);
  ⋮
}

int Xdr_u_long(struct str1 *xdrs, unsigned int *ulp)
{
  ⋮
  if ((int)(xdrs->x_op) == 0)
    return Xdrmem_putlong(xdrs, (int *)ulp);

  if ((int)(xdrs->x_op) == 2)
    return 1;

  return 0;
  ⋮
}

int Xdrmem_putlong(struct str1 *xdrs, int *lp)
{
  xdrs->x_handy = xdrs->x_handy - 4u;
  if (xdrs->x_handy < 0)
    return 0;
  *(xdrs→x_private) = htonl(*lp);
  xdrs→x_private = 4u + xdrs→ x_private;
  return 1;
}
```

FIG. 6.3 – *Sensibilité à l'utilisation pour du code de systèmes d'exploitation*

```
int Xdr_bytes(...)
{
  ⋮
  *(xdrs→x_private) = htonl(*lp);
  xdrs→x_private = 4u + xdrs→ x_private;
  ⋮
}
```

FIG. 6.4 – *Exemple de SUN RPC spécialisé en fonction des annotations sensibles à l'utilisation*

sible à l'utilisation.[3] Comme le pointeur `xdrs` apparaît à la fois dans des contextes statiques et dynamiques, chaque utilisation sera considérée dynamique. Ces valeurs dynamiques sont propagées à travers le programme, provoquant la résidualisation d'à peu près tout.

Pour cet exemple, une analyse doit être sensible à l'utilisation pour pleinement exploiter ces valeurs connues, comme on le voit par les annotations sensibles à l'utilisation de la Figure 6.3. L'affectation à `*(xdrs->x_private)`, qui effectue le codage, sera résidualisée, ainsi que pointeur sur le buffer client. Ceci est dû au fait que la donnée à coder ne sera connue qu'à l'exécution. Toutefois, toutes les autres opérations, comme celles qui dépendent du buffer client, seront évaluées.

Par exemple, tester la valeur (`xdrs->x_handy <0`) détermine si un débordement de buffer apparaît et peut être calculé au temps de spécialisation. Si l'analyse est aussi sensible au retour, alors la valeur retournée résultante peut être propagée inter-procéduralement, causant la réduction de l'instruction `if` initiale. La sensibilité au retour permet de propager les valeurs de retour statiques inter-procéduralement, même si la fonction contient des effets de bord dynamiques. Le programme résiduel produit en spécialisant un programme annoté par une analyse sensible au retour et à l'utilisation est donné en Figure 6.4. Dans cet exemple, tous les appels de fonction intermédiaires ont été déroulés lors de la phase de post-traitement. Le programme résultat de deux lignes illustre clairement le potentiel de l'application de la spécialisation aux composants de systèmes d'exploitation.

On ne le voit pas sur cet exemple, mais la sensibilité au contexte a été aussi utile à la spécialisation de l'application RPC. Quand le codage d'un argument de fonction est spécialisé à l'initialisation, les types des arguments sont connus alors que leurs contenus ne le sont pas. Dans un contexte légèrement différent, un pointeur de fonction est encodé quand à la fois le type et le contenu du pointeur sont connus. Notre analyse sensible au contexte a créé un nouveau contexte pour ce second cas, où toutes les informations connues sont considérées statiques.

6.4 Sensibilité contre insensibilité à l'utilisation

Nous avons précédemment identifié les systèmes d'exploitation comme étant de bons candidats à la spécialisation en spécialisant à la main certains composants de systèmes d'exploitation existants et en obtenant des accélérations significatives[PAB+95, VMC96a]. Utiliser la technologie actuelle de l'évaluation par-

[3] Rappelons que d'après le Chapitre 2, les constructions en *italique* doivent être évaluées et celles en **gras** doivent être résidualisées

tielle pour obtenir automatiquement ces même accélérations n'est pas possible, car les analyses de temps de liaison existantes ne sont pas assez précises pour déterminer les transformations effectuées à la main. Il a toujours été nécessaire d'écrire les programmes à partir de rien ou de prendre du code existant et de l'adapter à la main, pour obtenir une spécialisation réussie. Si l'évaluation partielle veut être appliquée de façon réussie à du code existant, comme on l'a vu par la suite dans [MMV+97], une analyse de temps de liaison précise est nécessaire.

Pour donner un bilan quantitatif de la nécessité de la sensibilité à l'utilisation, nous comparons une analyse sensible à l'utilisation à une analyse insensible à l'utilisations sur plusieurs programmes systèmes. Nous commençons par une brève description de chaque programme considéré et présentons les invariants clefs utilisés pour sa spécialisation.

Le premier programme, `copy_elim`, comprend la manipulation de paquets de messages que l'on trouve typiquement dans les logiciels réseaux[VMC+96c]. Les paquets sont traités via des pointeurs sur les données. Certaines parties de ces données (typiquement, des en-têtes) sont statiques, alors que d'autres parties (le message lui-même) sont dynamiques. Le second programme, `minix_read`, est un fragment de la mise en œuvre du système de fichiers de Minix[Tan87]. Précisément, nous avons spécialisé les routines de haut niveau de l'appel système `read()` en fonction d'un fichier donné et d'une taille à lire donnée. Ici aussi, le descripteur de fichier n'est que partiellement statique (*e.g.,* le mode du fichier est statique alors que le déplacement ou offset du fichier est dynamique) ; cette structure est gérée par pointeur. Les troisième et quatrième programmes, `client_stub` et `marshaling`, sont deux fragments de code de la mise en œuvre du SUN RPC[Sun90]. Le programme `client_stub` contient la couche souche du client et le programme `marshaling` provient de la couche de marshaling. Nous avons spécialisé ces programmes en fonction d'une interface client/serveur donnée, où de nombreux descripteurs (descripteurs de fichier, descripteurs de socket, descripteurs de protocole,...) étaient partiellement statiques.

	nombre total de lignes	expressions			
		nombre total	% évaluées		
			insens.util.	sens.util.	gain
`copy_elim`	254	352	27%	85%	58%
`minix_read`	314	378	41%	65%	24%
`client_stub`	960	1732	46%	60%	14%
`marshalling`	910	887	38%	78%	40%

TAB. 6.5 – *Pourcentage du gain de l'évaluation d'instructions et d'expressions de programmes systèmes dû à la sensibilité à l'utilisation.*

Notre expérimentation consiste à analyser les quatre programmes deux fois : premièrement avec une analyse insensible à l'utilisation puis avec une analyse sensible à l'utilisation . Après chaque analyse, les instructions et les expressions annotées sont comptabilisées. Ces résultats sont donnés dans le Tableau 6.5 pour chaque programme considéré, avec le nombre de lignes, d'instructions et d'expressions. Le nombre d'instructions et d'expressions annotées *evaluate* sont exprimées en pourcentage. Les pourcentages obtenus par les analyses insensible et sensible à l'utilisation sont utilisés pour calculer le gain apporté par la dernière analyse. Dans le cas sensible, les instructions et expressions repérées comme étant à la fois évaluées et résidualisées (*i.e.*, *E,R*) ne sont pas comptabilisées puisqu'elles apparaissent dans le programme résiduel.

L'observation principale à faire sur le Tableau 6.5 est que l'analyse sensible à l'utilisation détecte sur tous les programmes entre 10% et 58% d'instructions et d'expressions à évaluer. Comme la spécialisation évalue les constructions statiques

et résidualise les constructions dynamiques, plus le pourcentage de construction statique est haut plus le programme résiduel sera optimisé. En effet, la spécialisation de ces programmes montre clairement que tous les invariants mentionnés plus haut sont exploités comme prévu. Dans certains cas, les programmes spécialisés résultants sont compétitifs avec leur version spécialisée manuellement.

Chapitre 7

Travaux connexes

Dans ce chapitre, nous examinons la recherche liée aux analyses statiques présentées dans ce document. Nous commençons par considérer les analyses de temps de liaison existantes et les comparons avec nos travaux. Nous présentons aussi d'autres analyses qui sont différentes mais ont des aspects communs avec nos analyses. Nous concluons par une étude de l'adaptation de programme, un domaine de recherche où les analyses statiques pourraient jouer un rôle important.

7.1 Analyse de temps de liaison

Il y a un certain nombre d'évaluateurs partiels hors-ligne existants pour les langages impératifs [And92, And94a, BGZ94, JGS93a, NP92], ainsi pour les langages fonctionnels [Con93c, HM94, JGS93a, RG92]. Certaines fonctionnalités de leurs analyses, comme la sensibilité au flot, ne s'appliquent qu'aux langages impératifs. D'autres aspects, comme la sensibilité au contexte et à l'utilisation peuvent aussi être d'intérêt pour les langages fonctionnels. Nous étudions chacune de ces fonctionnalités et les rapports qu'elles ont avec les analyses de temps de liaison existantes.

7.1.1 Sensibilité au flot

Toutes les analyses de temps de liaison impératives existantes sont *insensibles au flot*; c'est-à-dire qu'une description unique de l'état de temps de liaison est conservée pour tout le programme [And92, And94a, BGZ94, JGS93a, NP92]. Dans ce cas, si une variable est dynamique quelque part dans le programme, sa description unique sera dynamique et par conséquent, la variable sera considérée comme dynamique dans tout le programme. Dans ce document, nous avons obtenu la sensibilité au flot en écrivant une analyse sensible au flot.

Une approche alternative serait d'utiliser une analyse insensible au flot sur une représentation intermédiaire de flot de programme qui code explicitement les dépendances de flot, comme l'affectation statique unique (Single Static Assignement en anglais, abrégé par SSA) [CJR+91]. Par exemple, une analyse de temps de liaison a été décrite pour un langage impératif simple et elle obtient la sensibilité au flot en utilisant un graphe de représentation de programme (Program Representation Graph en anglais), une représentation qui contient certaines caractéristiques de SSA [DRVH95]. Ce travail se concentrait sur la donnée d'une sémantique formelle et les preuves de conditions de sûreté des analyses de temps de liaison afin d'établir une base sémantique. Par conséquent les aspects de mise en œuvre ou d'application n'ont pas été considérés. Il serait intéressant de déterminer si ce cadre peut être

adapté pour manipuler des programmes réels, par exemple en traitant un langage plus réaliste contenant des pointeurs, des structures de données ou des fonctions.

La sensibilité au flot ne s'applique pas aux langages fonctionnels puisqu'ils n'ont pas de notion d'état ou d'opérations de mise à jour.

7.1.2 Sensibilité au contexte

De même, toutes les analyses de temps de liaison existantes sont insensibles au contexte. Les contextes de tous les appels à une fonction sont approximés par un contexte unique et moins précis. Si un paramètre ou une variable non-locale est dynamique en un quelconque point d'appel, elle sera considérée comme dynamique pour tous les points d'appel. D'autre part, il y a de nombreuses analyses de temps de liaison pour les langages fonctionnels qui sont sensibles au contexte—on les appelle plus communément polyvariantes [Con93c, HM94, RG92]. Cependant, une analyse de temps de liaison sensible au contexte pour un langage impératif est plus compliquée puisque les contextes doivent comprendre les temps de liaison des variables non-locales lues par une fonction et l'état doit être mis à jour en fonction des variables non-locales écrites. Ceci est encore compliqué par la possibilité pour les définitions d'être ambiguës de par l'aliasing.

7.1.3 Sensibilité au retour

La sensibilité au retour, qui empêche le temps de liaison d'une fonction d'interférer par effet de bord sur le temps de liaison de son résultat, est un nouveau concept qui n'avait pas encore été étudié. Nous avons découvert le besoin de la sensibilité au retour en appliquant l'évaluation partielle au code de systèmes d'exploitation.

La sensibilité au retour n'est pas applicable aux langages fonctionnels puisque les fonctions pures ont une valeur de retour et ne provoquent aucun effet de bord. Cependant, des situations similaires surviennent dans les langages fonctionnels. Par exemple, une expression peut contenir des composants dynamiques mais retourner une valeur statique. Des techniques ont cependant été développées: elles passent la continuation de l'expression dans le corps de l'expression de façon à exploiter la valeur statique [Bon92].

7.1.4 Sensibilité à l'utilisation

La sensibilité à l'utilisation a été étudiée selon différentes perspectives et pour divers langages. Des transformations de programme et des représentations de valeurs ont été proposées pour obtenir certaines formes de sensibilité à l'utilisation. De même, des analyses ont été développées pour résoudre des problèmes de flot de données.

Représentations statiques et dynamiques pour la même valeur

Il existe de nombreuses analyses qui peuvent gérer à la fois une représentation statique (valeur concrète pour les utilisations statiques) et une représentation dynamique (représentation textuelle pour les utilisations dynamiques) pour la même valeur. Par exemple, les évaluateurs partiels FUSE [WCRS91] et Schism [Con93b, Con93d] maintiennent deux représentations de chaque fermeture, permettant qu'elle puisse être appliquée ou résidualisée selon le contexte. Danvy *et al.* montrent comment les transformations de programme préalables à l'évaluation partielle peuvent donner des résultats similaires [Dan95, DMP95].

Ces solutions sont suffisantes quand chaque valeur a une représentation dynamique, c'est-à-dire un fragment approprié de texte qui peut remplacer la valeur si

elle nécessite d'être résidualisée. Cela revient à considérer toutes les variables comme réifiables. Néanmoins, dans des langages impératifs comme C, les pointeurs, les tableaux et les structures de données ne peuvent pas être réifiées. Dans certains cas, les valeurs de pointeurs peuvent avoir des représentations dynamiques construites sur le nom d'une variable (*e.g.,* `&x` pour l'adresse de `x`), mais ceci n'est pas toujours possible. Par exemple, ces représentations ne sont pas valides inter-procéduralement (si les noms des variables passées sont hors de leur portée) ou avec une allocation de mémoire dynamique (où il n'y a pas de nom de variable). De plus, en C, il n'existe aucune semblable représentation dynamique pour les structures. Dans ces cas, la sensibilité à l'utilisation est nécessaire pour obtenir des résultats précis.

D'autre part, la sensibilité à l'utilisation peut aussi s'appliquer aux valeurs réifiables, ce qui dans certains cas produit une meilleure spécialisation. Par exemple, Schism incorpore une forme de sensibilité à l'utilisation pour les structures de données. Même si les structures de données peuvent toujours être réifiées, ce qui veut dire que les utilisations dynamiques ne perturbent pas les utilisations statiques, réifier plusieurs copies de la même structure de données introduit plus de code et d'utilisation de la mémoire dans le programme résiduel. Quand une structure de données statique de grande taille apparaît dans de nombreux contextes dynamiques, il n'est pas souhaitable de la réifier et ainsi de la résidualiser à de nombreux endroits. Schism, étant capable de détecter quand cette situation se produit, résidualise une unique instance de la structure de données parmi les multiples occurrences de celle-ci, évitant ainsi la duplication des données.

Il faut mentionner que toute autre analyse basée sur une sémantique similaire à deux niveaux (c'est-à-dire évaluer et résidualiser) sera confrontée aux mêmes problèmes. Par exemple, beaucoup de travail a été fait dans le domaine de la propagation de constantes, qui comprend une phase où les valeurs constantes sont propagées suivie d'une phase où les calculs qui ne dépendent que de valeurs constantes sont évalués[MS93]. Dans ces travaux, seules les valeurs réifiables sont prises en considération. Si des valeurs non-réifiables étaient propagées, une technique similaire devrait être développée pour résoudre les problèmes exposés dans ce document.

Scission de structure

C-Mix est un évaluateur partiel pour C qui traite les pointeurs, les tableaux et les structures de données [And92, And94a, JGS93a]. Toutefois, comme son analyse de temps de liaison est insensible à l'utilisation, les utilisations dynamiques de valeurs non-réifiables interfèrent avec les utilisations statiques. Nous avons vu que, pour certains programmes, une analyse de temps de liaison insensible à l'utilisation peut amener une perte significative de précision.

Comme mentionné plus tôt, une façon de pallier ces pertes est de récrire le code à la main, en séparant les utilisations statiques des utilisations dynamiques. C-Mix tâche d'automatiser une telle séparation par la *scission de structure.* Cette technique éclate une structure de données en composants séparés en créant une nouvelle variable pour chaque élément de structure. Ce procédé peut être répété récursivement (sur des structures imbriquées) jusqu'à ce que toutes les structures soient éliminées. Si les champs de la structure initiale sont des valeurs réifiables, alors toutes les nouvelles valeurs correspondantes sont réifiables. Et, puisque les valeurs réifiables ne provoquent pas de perte dans les analyses insensibles au flot, le problème est résolu.

Bien que cette approche soit généralement intra-procédurale, Andersen propose une extension inter-procédurale à la scission de structure qui introduirait un nouveau paramètre pour chaque champ de la structure [And94a]. Cette approche, cependant, s'avère ne pas être utilisable pour les applications réalistes. Comme déjà mentionné, les programmes systèmes maintiennent typiquement un état système

qui consiste en de nombreuses structures de données imbriquées. Par exemple, dans l'application `marshaling` considérée dans le Chapitre 6, l'état système est représenté par `struct cu_data`, une structure de données avec un total de 29 champs. Pour passer cette information inter-procéduralement, la structure est toujours passée via un pointeur, ce qui évite de copier chaque champ à chaque appel de la fonction. L'extension inter-procédurale proposée par Andersen n'utilise pas cette technique et chaque nouveau paramètre introduit une copie.

Il est aussi typique, quand on traite des programmes systèmes de grande taille qu'un petit fragment du système soit extrait et spécialisé. Après la spécialisation, la pièce nouvelle et spécialisée doit être réinsérée dans son contexte. Pour cette raison, il est nécessaire de préserver l'interface entre ces deux parties. L'extension inter-procédurale proposée par Andersen ne préserve pas cette interface.

7.2 Analyses connexes

Notre analyse de temps de liaison en deux temps, constituée d'une phase de temps de liaison suivie d'une phase de transformation, présente des similarités avec d'autres analyses. Nous comparons nos analyses avec celles utilisées dans le filtrage de programme, l'ajout d'argument et la spécialisation de représentations fonctionnelles de programmes impératifs.

7.2.1 Filtrage

Le filtrage de programme calcule les parties d'un programme qui affectent potentiellement les valeurs en un point de programme donné [Tip94]. Le filtrage de programme peut être vu comme une autre forme de spécialisation, semblable à l'évaluateur partiel SFAC décrit en Section 1.3. L'utilisateur fournit un critère de filtrage, à partir duquel le programme résiduel est produit. Parmi les applications du filtrage de programme se trouvent la maintenance et le débogage de logiciel.

Les techniques de filtrage avant, qui propagent de l'information des définitions de variables vers les utilisations de variables, ont été utilisées pour définir des analyses de temps de liaison pour des programmes impératifs [DRVH95]. Cette analyse en avant est très proche de notre phase de temps de liaison. Cependant, les valeurs non-réifiables ne sont pas abordées dans ce travail ; aucune phase de transformation n'est fournie pour traiter les utilisations dans des contextes différents.

Il y a de plus des techniques de filtrage arrière similaires à la phase de transformation de notre analyse de temps de liaison, propageant de l'information des utilisations de variables vers les définitions de variables [RT96]. On peut voir cela comme une sorte d'information de *nécessité*. À l'instar du filtrage qui calcule quelles commandes sont nécessaires dans un filtre, notre analyse calcule les transformations pour chaque construction. La différence principale est qu'au lieu d'utiliser un domaine à deux valeurs (needed, not needed), notre analyse travaille sur un domaine à quatre valeurs (evaluate, residualize, evaluate and residualize et $\{\}$) puisque certaines constructions peuvent être à la fois évaluées et résidualisées.

7.2.2 Ajout d'argument

L'ajout d'argument s'est avéré utile pour la spécialisation de programmes fonctionnels [Rom88, Rom90]. La motivation de ce sujet est autant l'élimination des constructeurs et sélecteurs de données inutiles que la réduction du sur-coût de l'appel de fonction. Par exemple, considérons un doublet passé à une fonction qui n'utilise qu'un de ses composants. L'ajout d'argument transforme le programme en passant les deux valeurs à la fonction plutôt que le doublet. Ceci élimine le constructeur

initial ainsi que le sélecteur qui en découle. De plus, plutôt que de passer les deux valeurs, seule la valeur utilisée a besoin d'être passée.

L'ajout d'argument, comme notre analyse de temps de liaison en deux étapes, est obtenue en combinant une analyse en avant et une analyse en arrière. Dans les deux cas, la phase en avant détermine la faisabilité d'une transformation particulière. Notre analyse de temps de liaison détermine si une construction peut être évaluée à la spécialisation tandis que l'ajout d'argument détermine si une structure de données passée inter-procéduralement peut être décomposée en ses sous-composants. De même, les deux phases en arrière effectuent une analyse de nécessité. L'analyse de temps de liaison collecte de l' information relative aux utilisations d'une variable de façon à déterminer l'annotation correspondante avec laquelle la définition de la variable doit être annotée. L'ajout d'argument détermine quels sous-composants sont utilisés par une fonction et par conséquent doivent être passés. Notons comment toutes les deux requièrent une analyse en arrière pour propager de l'information des utilisations de variables vers les définitions de variables.

7.2.3 Représentation fonctionnelle de programmes impératifs

Une approche différente pour obtenir une spécialisation efficace de programmes impératifs a été proposée [Mou97, MCL96]. Au lieu de traiter directement un programme impératif, le programme source original est transformé en une représentation fonctionnelle. Un évaluateur partiel existant pour un langage fonctionnel est alors utilisé pour spécialiser le programme, après quoi le programme résiduel est retransformé dans le langage impératif original. L'avantage principal de cette approche est que la réutilisation d'un évaluateur partiel existant et confirmé dispense de la conception et de la mise en œuvre d'un nouvel évaluateur partiel. Les premiers résultats montrent que cette approche peut atteindre un haut degré de spécialisation ; les sensibilités au flot, au contexte, au retour et à l'utilisation ont été démontrées sur de petits exemples. Une expérimentation plus approfondie serait nécessaire pour déterminer si cette approche peut aller jusqu'à gérer la taille et la complexité des programmes existants et réalistes.

7.3 Analyses statiques pour l'adaptation de programme

L'*adaptation de programme*, la capacité d'un programme à s'adapter au contexte où il est utilisé, a été proposée comme une application prometteuse de la transformation de programme [Con96]. Les transformations de programme telles que l'évaluation partielle sont particulièrement adéquates pour adapter un programme à son environnement. Il semble nécessaire d'utiliser un certain type d'analyse statiques, comme l'analyse de temps de liaison, pour que l'adaptation soit faisable et efficace. L'aspect important de l'analyse de temps de liaison est qu'elle identifie les dépendances entre les différentes parties d'un programme. Cette information peut alors être utilisée pour préparer les transformations qui sont par la suite déclenchées quand les valeurs réelles de l'environnement du programme changent. Deux applications concrètes de l'adaptation sont la génération de code à l'exécution et les systèmes d'exploitation adaptatifs.

7.3.1 Génération de code à l'exécution

La génération de code à l'exécution a récemment beaucoup attiré l'attention. Une raison primordiale pour ce regain d'intérêt est que les avancées récentes en

recherche sur les langages de programmation donnent maintenant des techniques plus portables et moins enclines aux erreurs pour générer du code au moment de l'exécution.

Certaines approches, cependant, requièrent toujours que le programmeur spécifie manuellement les spécialisations [EHK96, EP94]. La nécessité de l'intervention de l'usager est consommatrice en temps, assez compliquée et potentiellement propice aux erreurs. De plus, sans analyse pour produire une information globale sur le programme, le programme spécialisé résultant peut être moins efficace.

D'autres approches ont tenté d'automatiser le processus de spécialisation en fournissant des analyses statiques déterminant à la compilation les transformations qui seront effectuées à l'exécution. Par exemple, Fabius [LL94, LL96] et le système Dynamic Compilation [APC+96] comprennent une forme d'analyse de temps de liaison dans leurs approches.

Fabius traite un langage simple : un sous-ensemble du premier ordre de Standard ML. Par conséquent, il est probable que son analyse de temps de liaison ne requière pas les fonctionnalités avancées présentées dans ce document, comme la sensibilité au flot, au contexte, au retour et à l'utilisation. D'autre part, traiter un langage simple empêche Fabius de prendre en compte des applications existantes ou réalistes. Dynamic Compilation est dirigé vers les programmes réalistes écrits en C. De façon assez intéressante, il est aussi orienté vers le traitement de composants de systèmes d'exploitation. Par conséquent, il est probable que ses analyses statiques aient besoin de posséder les fonctionnalités présentées dans cette dissertation pour pleinement exploiter les opportunités de spécialisation.

7.3.2 Systèmes d'exploitation adaptatifs

Il existe actuellement un certain nombre de travaux sur les systèmes d'exploitation adaptatifs [BSP+95, EKO95, MMO+94]. Ces approches ont développé de nouvelles technologies pour procurer cette adaptativité. A l'opposé, nous proposons de réutiliser une technologie existante, à savoir l'évaluation partielle, pour atteindre les besoins des systèmes d'exploitation adaptatifs. Notre collaboration avec le groupe Synthetix crée un effet de synergie dont les deux groupes profitent par fertilisation mutuelle [PAB+95]. Le groupe système identifie les opportunités de spécialisation et utilise nos outils pour effectuer leurs spécialisation adaptative. En appliquant Tempo aux programmes systèmes, notre groupe peut continuer à affiner ses outils en se basant sur le feed-back reçu.

Chapitre 8

Conclusion

La spécialisation de programmes peut automatiquement et efficacement optimiser des applications existantes et réalistes. Des études précédentes ont démontré que spécialiser des applications de grande taille, tels que les programmes systèmes, produit des accélérations impressionnantes. Cependant, jusqu'à présent, cette spécialisation était faite à la main. Nous concevons de nouvelles analyses statiques qui permettre de spécialiser automatiquement et précisément ces programmes.

Nos analyses présentent des fonctionnalités spécifiques qui nous permettre d'exploiter pleinement les opportunités de spécialisation qui existent dans les programmes systèmes. La sensibilité au flot permet qu'une description d'analyse différente pour chaque instruction de mise à jour. La sensibilité au contexte permet d'analyser une fonction en fonction du contexte spécifique d'analyse de chaque site d'appel. La sensibilité au retour rend possible qu'une fonction qui a des effets de bord dynamiques retourne une valeur statique. La sensibilité à l'utilisation permet que différentes utilisations d'une variable aient différentes descriptions d'analyse. Nous montrons que si une seule de ces fonctionnalités fait défaut, le niveau de spécialisation décroît drastiquement.

Notre approche inclut un certain nombre d'aspects nouveaux. Contrairement aux analyses de temps de liaison existantes, celle que nous proposons consiste en une phase avant et une phase arrière. La phase de temps de liaison propage de l'information *vers l'avant*, d'une définition de variable vers ses utilisations, de façon à déterminer si les constructions sont statiques ou dynamiques. La phase de transformation propage de l'information *en arrière*, des utilisations de variables vers leurs définitions, pour déterminer une transformation pour chaque construction d'un programme. De plus, nous introduisons de nouvelles transformations, telles qu'à la fois évaluer *et* résidualiser une construction.

Nos analyses sont intégrées dans Tempo, notre évaluateur partiel pour C, et servent de base à la fois pour la spécialisation au temps de compilation et la spécialisation au temps d'exécution. Tempo a été appliqué à plusieurs systèmes d'application tournant sur différentes plates-formes matérielles. Par exemple, spécialiser le système d'appel de procédure à distance (Remote Procedure Call) de Sun sur une station de travail Sun fournit une accélération allant jusqu'à 3.75. Tempo s'est montré aussi efficace à spécialiser d'autres types de programmes, parmi eux, des algorithmes numériques et des procédures de traitement d'images. La spécialisation de ces programmes a donné des accélérations allant jusqu'à 12.

L'approche basée sur les patrons de Tempo pour la spécialisation au temps d'exécution s'est montrée hautement efficace. Les analyses statiques utilisées pour la spécialisation au temps de compilation sont aussi utilisées pour générer des patrons et une extension génératrice cible au temps de compilation, permettant une spécialisation sûre et efficace au temps d'exécution. Nos résultats montrent que les

programmes spécialisés au temps d'exécution tournent presque aussi rapidement que le même programme spécialisé et compilé au temps de compilation. La spécialisation ne requière pas plus de 3 exécutions pour être amortie. De plus, cette approche est automatique, portable et traite un langage réaliste, ce qui accroît le nombre d'applications potentielles auxquelles elle peut être appliquée.

Futurs travaux : analyses statiques

Nos résultats sont jusqu'à présent encourageants. Regardons maintenant l'évolution des analyses de Tempo et comment elles peuvent être étendues à l'avenir.

Au début de ce travail, nous savions qu'une analyse de temps de liaison serait nécessaire pour la spécialisation de programmes impératifs, mais nous ne savions pas quel type d'analyse. En étudiant les types de programmes que nous voulions spécialiser, nous avons déterminé les fonctionnalités que nous pensions nécessaires, telles que la sensibilité au flot et au contexte. De la même façon, nous avons identifié les fonctionnalités que nous ne jugions pas nécessaires, telles que le traitement des procédures récursives.

Une fois que des versions préliminaires de nos analyses ont été conçues, elles furent développées et expérimentées sur des programmes d'application de façon à faire un bilan de leur efficacité. Bien que nous ayons une idée de à quoi s'attendre, les grandes taille et complexité des programmes d'application rendaient impossible de prédire précisément les résultats. Les résultats initiaux révélèrent, par exemple, que nos décisions de conception originales n'étaient pas suffisantes pour exploiter pleinement d'importantes opportunités de spécialisation. Par conséquent, de nouvelles fonctionnalités, telles que la sensibilité à l'usage qui évalue *et* résidualise la même construction, ont été ajoutées pour améliorer la précision de nos analyses.

Avoir un prototype opérationnel nous a aidé à identifier de nouvelles opportunités de spécialisation. Par exemple, les valeurs de statut d'erreur n'étaient pas initialement considérées importantes. Après que la spécialisation ait avec succès éliminé la plupart des autres constructions d'un ensemble d'appels de fonctions, nous avons cependant réalisé que dans certains cas ces valeurs de retour résidualisées devenaient un goulot d'étranglement. Cette observation, par exemple, nous a conduit à l'addition de la sensibilité au retour dans nos analyses.

Comme nous continuons à traiter des programmes, nous trouvons que certaines fonctionnalités, que nous avions décidé de ne pas inclure, sont en fait souhaitables pour certaines applications. Un tel exemple est la récursivité. Initialement, nous avons choisi de la laisser de côté, parce que les programmes systèmes n'en font pas souvent usage et que cela complique les analyses et les transformations. Mais depuis que nous avons initié ce travail, nous avons en fait trouvé bon nombre d'applications intéressantes telles qu'un filtre de paquets dans un système d'exploitation ou un interprète de bytecode Java, qui utilisent la récursivité. C'est une fonctionnalité que nous projetons d'ajouter dans un futur proche.

Une autre décision que nous avions prise initialement est une approximation sur les structures de données. Nous traitons des structures de données *partiellement statiques*, permettant que des champs différents d'une structure de données aient des descriptions d'analyses différentes, mais nous ne traitons les structures de données que de façon monovariante. En d'autres termes, il y a une description pour chaque type de structure de données, qui s'applique à toutes les structures de données de ce type. Initialement, cette approximation était suffisante, une fois encore, parce que les programmes systèmes utilisent typiquement des structures de données d'une manière uniforme. Cependant, les applications ultérieures que nous considérons, telles que les langages à objets, requièrent que les structures de données soient traitées de façon *polyvariante*, permettant que des structures de données différentes

aient des descriptions différentes. Nous prévoyons donc d'ajouter cette fonctionnalité à nos analyses.

Comme on le voit sur ces exemples, les analyses statiques présentées dans cette thèse ont évolué au fil du temps. Certaines notions, telles que la sensibilité à l'utilisation et la sensibilité au retour, n'existaient même pas quand les analyses ont été initialement conçues. De même, d'autres fonctionnalités potentielles peuvent être incorporées plus tard, si le besoin survient. Ces possibilités incluent la duplication des continuations après les conditionnelles dynamiques ou bien la mémorisation en cache des spécialisations en tout point de programme plutôt que sur la base de la définition de fonction.

Le développement de Tempo a été réalisé avec la collaboration de chercheurs en systèmes d'exploitation. Cette collaboration a été mutuellement bénéfique. En tant que chercheurs en langage de programmation, nous avons reçu une riche expérience en systèmes qui nous a donné une cible vers laquelle nous continuons d'affiner et d'améliorer nos travaux. En retour, nous fournissons un outil qui permet aux systèmes d'exploitation d'être optimisés de manières jusqu'alors impossibles. L'échange entre les deux parties continue de créer des opportunités des deux côtés: si l'outil est amélioré, d'autres applications peuvent être optimisées, ce qui à son tour permet d'identifier de nouvelles opportunités pour améliorer encore l'outil.

Notre système est donc en perpétuelle évolution. Travailler avec un prototype changeant a des inconvénients. Par exemple, il serait souhaitable de prouver quelques aspects de corrections sur les analyses. Notamment, nous voudrions prouver que toute construction annotée comme statique dépend uniquement de valeurs d'entrée statiques. Il peut y avoir d'autres propriétés à prouver. Par exemple, puisque la transformation qui évalue et résidualise une construction est une nouvelle transformation de programmes, il serait bien de prouver sa correction. Prouver de telles propriétés devrait nous aider à comprendre les analyses, puisque cela obligerait à exprimer plus rigoureusement tous les aspects des analyses. De plus, ces preuves garantiraient que les propriétés fondamentales des analyses sont correctes.

Avec un prototype évoluant, prouver la correction devrait être assuré à chaque modification des analyses. Nous pensons nous atteler à cette tâche une fois que les données seront relativement stables, mais nous trouvons toujours de nouvelles façons d'améliorer notre système. Une possibilité serait de tenter de sélectionner un petit sous-ensemble du langage qui semble stable et de fournir des preuves pour ce sous-ensemble. De telles preuves seraient particulièrement importantes si le système était plus largement distribué.

Travaux futurs : Tempo

Tempo est actuellement appliqué à d'autres programmes de systèmes d'exploitation, tels que la délivrance de signaux et l'allocation de mémoire. Les résultats de ces expérimentations donneront un meilleur aperçu sur l'applicabilité de notre approche. De plus, nous explorons de nouveaux domaines, comme les programmes orientés objet. Ce domaine semble avoir beaucoup d'opportunités de spécialisation de la même façon que les programmes des systèmes d'exploitation. Nous dressons ci-dessous l'état actuel de ces études.

Les "connexions" transitoires entre modules dans les programmes systèmes fournissent des opportunités de spécialisation. Dans le chapitre 1, nous avons expliqué comment l'information communiquée entre modules pouvait contenir beaucoup de valeurs statiques, en prenant l'exemple d'un système de fichiers. Un autre exemple de communication apparaît quand un groupe de processus communiquent de façon répétitive pour atteindre un but commun. Des expériences sont actuellement faites où une connexion est partagée entre deux processus. Un processus envoie de façon répétée des signaux UNIX à l'autre processus. Les processus cible et destination

et leurs propriétés pertinentes sont considérées comme statiques. Spécialiser ce cas avec Tempo produit du code qui envoie le signal trois fois plus rapidement. Cette accélération n'est pas obtenue complètement automatiquement, mais avec une assistance humaine. Des études ultérieures montreront si cette intervention peut être automatisée et intégrée dans Tempo.

L'allocation dynamique de mémoire est une autre fonction qui est paramétrée et générique de telle façon que la même fonction puisse être utilisée dans une large variété de situations. Il s'avère que dans les programmes systèmes, elle est souvent utilisée de manière régulière. Tempo est aussi utilisé pour spécialiser la fonction d'allocation afin d'exploiter les invariants disponibles dans les cas communs.

Harissa est un environnement efficace pour l'exécution de programmes Java. Le compilateur Harissa traduit le bytecode Java en C. Une fois traduit en C, Tempo pourrait être utilisé pour évaluer partiellement le programme. Dans ce cas, le spécialiseur au temps de compilation de Tempo générerait des programmes résiduels écrits en code C. Ces programmes pourraient être compilés et exécutés ou peut-être retraduits en code Java ou en bytecode Java. Le spécialiseur au temps d'exécution de Tempo produit directement du code objet exécutable.

Au lieu de combiner Harissa et Tempo, un évaluateur partiel pourrait être développé pour traiter directement du code Java ou du bytecode Java. Dans ce cas, il est possible que les principes et techniques développés pour Tempo puissent être utiles. Par exemple, les analyses dans une approche hors-ligne peuvent inclure la sensibilité au flot, au contexte, au retour ou à l'utilisation.

La spécialisation déclarative est une technique qui permet à un programmeur de spécifier différents aspects de la spécialisation. Les déclarations sont faites dans un cadre orienté objet et expriment où, quand et comment la spécialisation est exécutée. Par exemple, le programmeur peut indiquer si la spécialisation est faite manuellement ou automatiquement, au temps de compilation ou au temps d'exécution. D'autres directives sont utilisées pour détecter quand le contexte d'utilisation change, ce qui veut dire que la version spécialisée ne peut plus être utilisée et pour indiquer quelles spécialisations peuvent être conservées et pendant combien de temps. Nous travaillons actuellement sur une interface de spécialisation déclarative avec Tempo.

Bibliographie

[AK82] S.M. Abramov and N.V. Kondratjev. A compiler based on partial evaluation. In *Problems of Applied Mathematics and Software Systems*, pages 66–69. Moscow State University, Moscow, USSR, 1982. (In Russian).

[And92] L.O. Andersen. Self-applicable C program specialization. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, San Francisco, CA, USA, June 1992. Yale University, New Haven, CT, USA. Technical Report YALEU/DCS/RR-909.

[And94a] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[And94b] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[APC+96] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [PLD96], pages 149–159.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AWS91] W.Y. Au, D. Weise, and S. Seligman. Generating compiled simulations using partial evaluation. In *28th Design Automation Conference*, pages 205–210. New York: IEEE, June 1991.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BBH+94] J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou. Software design for reliability and reuse: A proof-of-concept demonstration. In *Proceeding of TRI-Ada*, pages 396–404, 1994.

[BF93] S. Blazy and P. Facon. Partial evaluation for the understanding of fortran programs. In *Software Engineering and Knowledge Engineering, San Francisco, California, June 1993*, pages 517–525, 1993.

[BF96] S. Blazy and P. Facon. An automatic interprocedural analysis for the understanding of scientific application programs. volume 1110 of *Lecture Notes in Computer Science*, pages 1–16. Berlin: Springer-Verlag, 1996.

[BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In PEPM'94 [PEP94], pages 119–132.

[BM90] A. Bondorf and T. Mogensen. Logimix: A self-applicable partial evaluator for Prolog. DIKU, University of Copenhagen, Denmark, May 1990.

[Bon92] A. Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. New York: ACM, 1992.

[BS94] A.A. Berlin and R.J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In PEPM'94 [PEP94], pages 133–141.

[BSP+95] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In SOSP95 [SOS95], pages 267–283.

[BVT+94] D. Batory, S. Vivek, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.

[BW90] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[BW93a] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark, 1993. DIKU Research Report 93/22.

[BW93b] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Master's thesis, Computer Science Department, University of Copenhagen, 1993. Research Report 93/22.

[CD90] C. Consel and O. Danvy. From interpreting to compiling binding times. In N.D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 1990.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[CJR+91] R. Cytron, Ferrante J., B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems*, pages 451–490, 1991.

[CK91] C. Consel and S.C. Khoo. Semantics-directed generation of a Prolog compiler. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP '91, Passau, Germany, August 1991 (Lecture Notes in Computer Science, vol. 528)*, pages 135–146. Berlin: Springer-Verlag, 1991.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [POP96], pages 145–156.

[Con93a] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 66–77. New York: ACM, 1993.

[Con93b] C. Consel. Polyvariant binding-time analysis for applicative languages. In PEPM'93 [PEP93], pages 145–154.

[Con93c] C. Consel. Polyvariant binding-time analysis for applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 66–77. New York: ACM, 1993.

[Con93d] C. Consel. A tour of Schism. In PEPM'93 [PEP93], pages 66–77.

[Con96] Charles Consel. Program adaptation based on program transformation. *In ACM Workshop on Strategic Directions in Computing Research. ACM Computing Surveys*, 28A(4), December 1996.

[CPW93] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In PEPM'93 [PEP93], pages 44–46. Invited paper.

[CPW94] C. Consel, C. Pu, and J. Walpole. Making production OS kernel adaptive: Incremental specialization in practice. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 1994.

[CWKZ90] D.R. Chase, M. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, USA, June 1990. ACM SIGPLAN Notices, 25(6).

[Dan95] O. Danvy. Type-directed partial evaluation. Technical Report PB-494, Computer Science Department, Aarhus University, July 1995.

[DMP95] O. Danvy, K. Malmkjær, and J. Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–228, September 1995.

[DRVH95] M. Das, T. Reps, and P. Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 100–110, La Jolla, CA, USA, 1995. ACM Press.

[EGH94] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256. ACM SIGPLAN Notices, 29(6), June 1994.

[EH94] A.M. Erosa and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the IEEE 1994 International Conference on Computer Languages*, May 1994.

[EHK96] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [POP96], pages 131–144.

[EKO95] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In SOSP95 [SOS95], pages 251–266.

[EP94] D.R. Engler and T.A. Proebsting. DCG: An efficient retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 263–273. ACM Press, November 1994.

[Ers77] A.P. Ershov. On the essence of translation. *Computer Software and System Programming*, 3(5):332–346, 1977.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.

[Fut88] Y. Futamura. Program evaluation and generalized partial computation. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 1–8, 1988.

[GJ89] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, Amsterdam: North-Holland, 1989.

[GJ95] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Languages, Implementations, Logics, and Programs (PLILP'95). Lecture Notes in Computer Science, vol. ???* Berlin: Springer-Verlag, 1995. To appear.

[GNZ95] R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Berlin: Springer-Verlag, 1991.

[HM94] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 287–301. Berlin: Springer-Verlag, 1994.

[HN97] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM'97 [PEP97], pages 63–73.

[HNC96] L. Hornof, J. Noyé, and C. Consel. Accurate partial evaluation of realistic programs via use sensitivity. Research Report 1064, IRISA, Rennes, France, June 1996.

[JGS93a] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[JGS93b] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[Jør92] J. Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992. Student Project 92-1-4.

[JS80] N.D. Jones and D.A. Schmidt. Compiler generation from denotational semantics. In N.D. Jones, editor, *Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94)*, pages 70–93. Berlin: Springer-Verlag, 1980.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[KC91] D. Kincaid and W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole, 1991.

[KEH93] D. Keppel, S. Eggers, and R. Henry. Evaluating runtime compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[KKZG94] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. Meyer and G. Snelting, editors, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.

[LL94] M. Leone and P. Lee. Lightweight run-time code generation. In PEPM'94 [PEP94].

[LL96] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [PLD96], pages 137–148.

[Loc87] B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.

[Mas92] H. Massalin. *Synthesis: An Effictient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[MCL96] B. Moura, C. Consel, and J. Lawall. Bridging the gap between functional and imperative languages. Rapport de recherche, INRIA, Rennes, France, June 1996.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).

[Mic89] Sun Microsystem. NFS: Network file system protocol specification. RFC 1094, Sun Microsystem, March 1989. ftp://ds.internic.net/rfc/1094.txt.

[MMO+94] A.B. Montz, D. Mosberger, S.W. O'Malley, L.L. Peterson, T.A. Proebsting, and J.H. Hartman. Scout: A communications-oriented operating system. Technical Report 94–20, Department of Computer Science, The University of Arizona, 1994.

[MMV+97] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. Publication interne PI-1094, IRISA, Rennes, France, March 1997.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. Amsterdam: North-Holland, 1988.

[Mou97] B. Moura. *Bridging the Gap between Functional and Imperative Languages.* PhD thesis, University of Rennes I, April 1997.

[MP89] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, December 1990.

[MRB95] T.J. Marlowe, B.G. Ryder, and M. Burke. Defining flow sensitivity for data flow problems. Technical Report LCSR-TR-249, Computer Science Department, Rutgers University, July 1995.

[MS92] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992. Available from `ftp.diku.dk` as file `pub/diku/semantics/papers/D-152.ps.Z`.

[MS93] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letter on Programming Languages and Systems*, 2(1–4):213–232, March–December 1993.

[MVM97] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In PEPM'97 [PEP97], pages 116–125.

[NHCL96] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. Rapport de recherche 1065, IRISA, Rennes, France, November 1996.

[NN91] H.R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. John Wiley & Sons, 1991.

[NP92] V. Nirkhe and W. Pugh. Partial evaluation and high-level imperative programming languages with applications in hard real-time systems. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 269–280, Albuquerque, New Mexico, USA, January 1992. ACM Press.

[PAB+95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In SOSP95 [SOS95], pages 314–324.

[PEP93] *Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

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

[PEP97] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.

[PLD95] *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 30(6), June 1995.

[PLD96] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 31(5), May 1996.

[PLR85] R. Pike, B. N. Locanthi, and J.F. Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software - Practice and Experience*, 15(2):131–151, 1985.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[POP96] *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[PTVF93] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2nd edition, 1993.

[RAA+92] V. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX - Workshop Proceedings - Micro-kernels and Other Kernel Architectures*, pages 39–70, Seattle, WA, USA, April 1992.

[RG92] B. Rytz and M. Gengler. A polyvariant binding time analysis. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992 (Technical Report YALEU/DCS/RR-909)*, pages 21–28. New Haven, CT: Yale University, 1992.

[Rom88] S.A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. Amsterdam: North-Holland, 1988.

[Rom90] S.A. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Berlin: Springer-Verlag, 1990.

[RT96] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 409–429, February 1996.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In PLDI'95 [PLD95], pages 13–22.

[SOS95] *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[Sun90] Sun Microsystems. *Network Programming Guide*, March 1990.

[Sur95] Rajeev Surati. Practical partial evaluation. Master's thesis, Cambridge, MA: MIT Press, 1995.

[Tan87] A.S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[TC96] S. Thibault and C. Consel. A framework of application generator design. Rapport de recherche RR-3005, INRIA, Rennes, France, December 1996. To appear in ACM SIGSOFT Symposium on Software Reusability (SSR'97).

[Tip94] F. Tip. A survey of program slicing techniques. Report CS-R9438, Computer Science, Centrum voor Wiskunde en Informatica, 1994.

[VMC96a] E.N. Volanschi, G. Muller, and C. Consel. Safe operating system specialization: the RPC case study. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 24–28, Tucson, AZ, USA, February 1996.

[VMC+96b] E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Publication interne 1021, IRISA, Rennes, France, June 1996.

[VMC+96c] E.N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noyé, and C. Pu. A uniform automatic approach to copy elimination in system extensions via program specialization. Research Report 2903, INRIA, Rennes, France, June 1996.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191, Cambridge, MA, USA, August 1991. Springer-Verlag.

[WFW+94] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 94.

[WL95] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis of C programs. In PLDI'95 [PLD95], pages 1–12.

Résumé

Cette thèse montre comment la spécialisation de programmes peut optimiser automatiquement et efficacement des applications existantes et réalistes. On considère une classe d'applications existantes, les programmes des systèmes d'exploitation. Les études précédentes ont démontré que la spécialisation de ces programmes introduit des accélérations impressionnantes mais la spécialisation était faite à la main. Nous introduisons des nouvelles analyses statiques qui traitent automatiquement et précisément les motifs de programmes communs à tous ces programmes. Les fonctionnalités spécifiques incluent :

La sensibilité au flot. Une description d'analyse différente est faite pour chaque point de programme.

La sensibilité au contexte. Une fonction est analysée selon le contexte spécifique d'analyse de chaque site d'appel.

La sensibilité au retour. Une description d'analyse différente est faite pour les effets de bord et la valeur retournée d'une fonction.

La sensibilité à l'utilisation. Les différentes utilisations d'une variable peuvent avoir des descriptions d'analyse différentes au même point de programme.

Une analyse en deux phases. Une analyse de temps de liaison suivie d'une phase de transformation.

De nouvelles transformations de programmes. Une construction de programme peut se voir associer plusieurs transformations.

Notre analyse est utilisée à la fois pour la spécialisation à la compilation et aussi pour la spécialisation à l'exécution. Nous intégrons notre analyse dans un évaluateur partiel et l'appliquons à un ensemble d'application existantes, incluant un composant de système d'exploitation commercial. Nous obtenons des accélérations significatives tant durant la spécialisation à la compilation que durant celle à l'exécution.