THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6

Spécialité :

Systèmes Informatiques

présentée

par Monsieur *Pierre DAVID*

pour obtenir le titre de DOCTEUR DE L'UNIVERSITÉ PARIS 6

Sujet de la thèse :

Analyse sémantique des programmes en langage C en vue de leur parallélisation

soutenue le *30 septembre 1991*

devant le jury composé de :

Monsieur	Girault	Président, Rapporteur
Monsieur	Rifflet	Rapporteur
Monsieur	Estraillier	Examinateur
Monsieur	Feautrier	Examinateur
Madame	Lhuillier	Examinateur
Monsieur	Perrot	Examinateur

Résumé

Les recherches dans le domaine de la parallélisation automatique se sont concentrées sur le langage Fortran. Or, ce langage souffre de défauts qui le rendent inadapté à un certain nombre de problèmes. Le but de notre étude est la détection automatique de parallélisme en langage C.

Le problème est complexe puisque le langage C dispose de nombreuses constructions destinées à faciliter une écriture optimisée sur une architecture traditionnelle, et en particulier les pointeurs, l'arithmétique sur ces pointeurs et les expressions avec effets de bord. Notre approche consiste à “désoptimiser” le programme pour en faciliter l'analyse sémantique.

Dans un premier temps, nous définissons un sous-ensemble du langage C sans appel de fonctions ni coercitions, mais en acceptant toutes les expressions avec effets de bord. L'absence de coercitions nous permet de définir une structuration de la mémoire suivant le système de types du langage.

Dans un deuxième temps, nous transformons les boucles *for*, qui ont en langage C une sémantique très complexe, en boucles *while*, puis nous retirons les effets de bord des expressions.

Enfin, nous procédons à une évaluation symbolique du programme pour en extraire une suite de transformations, c'est à dire des substitutions gardées par des expressions logiques. Ce calcul est direct et précis pour les séquences linéaires, mais nous devons procéder à des approximations pour les boucles.

Un des sous-produits de cette analyse est l'identification de celles des boucles *while* qui ont une sémantique de boucle *do* de Fortran. A partir de là, il nous est possible de nous interfacer avec un paralléliseur classique.

Mots-clés : parallélisation automatique, compilation, évaluation symbolique, effets de bord, types

Abstract

Research in the field of automatic parallelization has been focused on the Fortran language. However, this language suffers from draw-backs which make it unadequate for certain problems. Our goal, with this work, is the automatic detection of parallelism in C language.

This is not a trivial matter, as C language has numerous constructs used to make optimized coding easier for traditional architectures, in particular pointers, arithmetic on these pointers, and expressions with side effects. Our approach is to "desoptimize" the program to make the semantic analysis easier.

In the first phase, we define a subset of the C language, minus all function calls and casts, but including expressions and their side effects. The absence of casts allows us to define structuring of memory according to the type system of the language.

In the second phase, we transform *for* loops, whose semantics are complex in C, into *while* loops, and we remove side effects from expressions.

Finally, we proceed to a symbolic evaluation of the program in order to extract a series of transformations, which are substitutions guarded by logical expressions. This calculation is direct and precise for linear sequences, but we have to make some approximations for loops.

On of the by-product of this analysis is the identification of those *while* loops which have the semantic of a Fortran *do* loop. From here, we can interface with a classical parallelizer.

Key-words : automatic parallelization, compilation, symbolic evaluation, side effects, types

À Claudio, Marc-Antoine, Georg-Philipp, Jean-Philippe, Georg-Friedrich, Joseph et Amadeo.

Remerciements

Le travail présenté dans cette thèse n'aurait certainement pas été ce qu'il est sans le concours de nombreuses personnes que je tiens à remercier ici :

- le professeur Guy Pujolle, qui m'a accueilli dans son laboratoire "Méthodologie et Architecture des Systèmes Informatiques" (MASI) à l'Université Pierre et Marie Curie ;
- le professeur Claude Girault, qui a accepté de présider mon jury, et dont les remarques opportunes m'ont été fort utiles pour finaliser le présent document ;
- le professeur Jean-Marie Rifflet qui, après m'avoir initié par ses ouvrages aux intéressantes possibilités du système Unix et du langage C, a accepté d'être rapporteur de cette thèse ;
- le professeur Paul Feautrier, sous la direction duquel j'ai travaillé, et dont les passionnantes discussions m'ont ouvert les yeux sur des domaines fascinants ;
- le professeur Jean-François Perrot, qui après avoir été mon professeur lors de mes débuts à l'Université, a bien voulu accepter de participer à mon jury ;
- le professeur Claire Lhuillier, qui a accepté avec gentillesse de participer à mon jury ;
- Pascal Estraillier, qui m'a toujours encouragé et qui m'a fait l'amitié de participer à mon jury ;
- les professeurs Pierre Lallemand (DRET), Michel Moreau (LPTL) et Pronob Mitter (LPTHE) pour leur contribution financière ;
- Rémi Triolet et François Irigoin de l'École des Mines de Paris qui, par l'intermédiaire des "séminaires du jeudi", ont élargi ma vision du parallélisme ;

Je tiens en outre à remercier toutes les personnes, trop nombreuses pour être citées ici, qui ont contribué de près ou de loin au bon déroulement de ce travail par leurs remarques critiques ou par un simple sourire ou des paroles réconfortantes. Je les remercie chaleureusement.

Enfin, je tiens à remercier tout spécialement Janick Taillandier pour son amitié indéfectible.

Table des Matières

Liste des figures

Introduction

L'essor actuel des architectures parallèles et vectorielles a été largement motivé par les demandes de calcul émanant de diverses disciplines telles que physique, chimie, ou sciences de la terre. Le langage de programmation couramment utilisé est le vénérable Fortran, ce qui explique que de nombreuses recherches dans le domaine de la détection automatique du parallélisme aient pris comme cible ce langage. La conséquence logique est que la très grande majorité des outils disponibles dans ce domaine est orientée vers l'utilisation de Fortran. Malheureusement, malgré son grand âge synonyme de respectabilité, ce langage n'est pas exempt de défauts. Ses innombrables cures de jouvence n'ont pas éliminé toutes ses structures archaïques et n'ont pas ajouté certaines caractéristiques très utiles des langages de programmation modernes telles que les structures de données évoluées et les pointeurs.

L'utilisateur d'un super-calculateur a le choix entre coder son modèle dans un langage qui le contraint afin de tirer parti de toutes les ressources de l'architecture sous-jacente, ou coder son modèle dans un langage évolué certes, mais pour lequel aucun outil n'existe pour paralléliser son programme.

Le but de notre étude est d'offrir une alternative viable, par la confection d'un paralléliseur automatique pour un langage plus riche en structures de données, le langage C [ANSI88]. Les caractéristiques de ce langage sont la présence de types de données construits, de pointeurs et surtout d'arithmétique sur ces pointeurs en plus des structures de contrôle habituelles.

Le langage C est un bon candidat pour la parallélisation automatique. Parmi ses nombreux avantages, nous pouvons citer :

- c'est un langage maintenant largement répandu et enseigné dans bon nombre de disciplines, scientifiques ou non ;
- c'est un langage stable, normalisé par l'ANSI ;
- c'est un langage permettant l'écriture de programmes portables sur une vaste gamme d'architectures ;
- c'est un langage performant, utilisé par ceux qui recherchent un maximum d'efficacité ;
- c'est un langage adapté à une large classe de problèmes, depuis les simulations numériques jusqu'aux algorithmes graphiques, grand consommateurs de parallélisation.

Cependant, malgré ces avantages reconnus, la parallélisation et le langage C forment un couple nouveau, rarement abordé dans la littérature. Le problème, dans son ensemble, est donc très peu défriché. Les difficultés spécifiques à ce langage sont en effet nombreuses. Allen et Johnson [AJ88] en citent quelques-unes :

- les effets de bord, y compris l'affectation, à l'intérieur des expressions posent un problème d'analyse évident ;
- certains opérateurs, comme `&&` sont intrinsèquement séquentiels : ils imposent une dépendance de contrôle dans les expressions ;
- la structure de contrôle essentielle en parallélisation n'est pas fiable : les boucles *for* ne sont que des boucles *while* déguisées, elles ne peuvent donc pas servir de base à la parallélisation comme les boucles *do* en Fortran ;
- l'utilisation de petites fonctions visant à la structuration des programmes est encouragée, posant ainsi de nombreux obstacles à la parallélisation des boucles contenant des appels ;
- l'opérateur d'extraction d'adresse permet la modification des variables du programme de manière très subtile, rendant impraticables les méthodes de construction de graphes de dépendances utilisées en Fortran.

Nous pouvons également ajouter à cette liste le problème du calcul d'adresses, c'est à dire l'arithmétique sur les pointeurs, qui complique encore la détermination des variables modifiées. Un pointeur, en langage C, peut adresser n'importe quelle variable. La présence des *casts* et des *unions* rend le problème encore plus délicat par une déstructuration catastrophique de la mémoire.

Si nous analysons plus finement le problème et recherchons la cause fondamentale de ces difficultés, nous constatons qu'une grande partie provient de la connaissance et de l'exploitation par le programmeur de l'architecture sous-jacente. Par exemple, le mot-clef `register` suppose des registres scalaires, l'opérateur `++` requiert une instruction d'incrémentation pour être efficace. En ajoutant à cela les opérateurs avec effets de bord, le programmeur peut rédiger son programme de manière extrêmement efficace pour une architecture séquentielle classique en réalisant lui-même les optimisations que le compilateur devrait détecter. Ce faisant, il rend son programme très difficile à paralléliser.

Notre approche pour résoudre ce problème a reposé sur une idée essentielle : les techniques de détection automatique de parallélisme développées pour Fortran [Feau88d] sont des techniques éprouvées, efficaces et maintenant largement répandues. Nous avons supposé l'existence d'un paralléliseur [TDF87] "traditionnel" (c'est à dire conçu pour Fortran) modifié pour accepter notre syntaxe et capable de générer du code parallèle.

Face à l'ampleur du problème à résoudre, et partant de l'analyse des difficultés ci-dessus, nous avons pris le parti de *désoptimiser* les programmes, parti inverse des techniques de compilation classiques [ASU86], mais déjà utilisé en parallélisation avec l'expansion de scalaires par exemple [PW86]. Ainsi, beaucoup de programmes sont rédigés en utilisant des pointeurs en lieu et place d'indices pour parcourir des tableaux. Nous détectons ces pointeurs, nous les remplaçons par des indices, et nous pouvons détecter les *vraies* boucles *for*, c'est à dire celles qui sont équivalentes aux boucles *for* de Pascal ou *do* de Fortran, ce qui nous permet alors de paralléliser en exploitant directement toutes les techniques sophistiquées développées à l'origine pour Fortran.

Notre démarche s'est scindée en quatre étapes, chacune ayant donné lieu à une réalisation :

1. nous avons défini un sous-ensemble du langage C, que nous avons voulu le plus opérationnel possible, mais rejetant les constructions inenvisageables dans une première approche du problème comme les *casts*. En particulier, tous les types de données (sauf les champs de bits, les structures, les unions et les fonctions) sont admis, toutes les expressions (sauf celles contenant des appels de fonctions ou des *casts*) sont reconnues, ainsi que les constructions `while`, `for` et `if` ;

2. nous avons ensuite défini des règles de transformation du programme afin de le *désoptimiser* au niveau de la syntaxe, c'est à dire que nous retirons les boucles *for* et tous les opérateurs avec effets de bord qui gênent l'analyse des expressions : c'est la *normalisation* du programme ;

3. nous avons alors procédé à une évaluation symbolique [Feau84] en présence de pointeurs, dans le but de synthétiser l'effet d'une séquence d'instructions par une *transformation*, c'est à dire un ensemble de modifications de la mémoire conditionnées par des *gardes* afin de lever les ambiguïtés qui surgissent dès qu'un pointeur intervient. La présence de ces gardes réduit à une opération triviale le calcul de la transformation associée à un test. Pour limiter le nombre de gardes et conserver néanmoins une finesse d'analyse importante malgré les optimisations opérées par le programmeur, nous avons formalisé une structuration de la mémoire induite par le système de types du langage C ;

4. enfin, nous avons étendu cette évaluation symbolique aux itérations par l'introduction d'approximations de transformations. Au cours de ce calcul, nous avons reconnu les variables inductives et les boucles pouvant être converties en vraies boucles de type *do*. Dès lors, nous pouvons procéder à la parallélisation.

Nous avons considéré les éléments indispensables pour le calcul scientifique (tableaux et boucles) en présence de pointeurs et d'arithmétique sur ces pointeurs. Les méthodes que nous avons développées peuvent être étendues à certaines constructions du langage, non traitées ici, comme les appels de fonction [Trio84] ou les structures.

Notre étude s'est finalisée par la réalisation d'un analyseur sémantique pour le langage C. Un analyseur syntaxique, un normaliseur de programme et une phase de recherche des transformations pour les séquences linéaires ont été rédigés en C et en Common Lisp et permettent l'analyse d'une fonction de taille moyenne sans boucle en moins de 10 secondes. Lorsque l'implémentation des boucles sera complètement terminée, l'intégration dans un paralléliseur existant, et notamment la construction du graphe de dépendances, pourra être réalisée aisément en s'inspirant des techniques que nous avons développées pour l'analyseur sémantique.

Nous débutons notre étude par une description du problème. En prenant comme base les travaux déjà effectués dans le domaine du parallélisme, nous cernons progressivement l'objectif que nous nous assignons.

Le deuxième chapitre est consacré aux deux premiers points ci-dessus. En premier lieu, nous définissons le langage que nous admettons en entrée. Ce langage comprenant en particulier tous les opérateurs avec effets de bord, nous définissons en second lieu un nouveau langage pour la phase d'analyse sémantique, sans effet de bord dans les expressions. La description de la réalisation nous fournit l'opportunité d'introduire la représentation intermédiaire utilisée dans toutes les étapes de notre travail et de décrire le principe du normaliseur.

Le troisième chapitre décrit l'évaluation symbolique. Il introduit, dans sa première partie, les notations et les conventions que nous avons adoptées dans le reste de notre étude. Nous y introduisons notamment la structuration de la mémoire grâce aux types du langage. La suite du chapitre est consacrée à l'étude des séquences linéaires, c'est à dire les affectations, les alternatives et les suites d'instructions. Nous définissons ce qu'est une transformation de la mémoire, puis nous introduisons la nécessité d'y adjoindre des gardes. La réalisation devant manipuler les transformations, nous en donnons une nouvelle forme facilitant leur implémentation, et nous prouvons que les deux formes sont équivalentes.

La quatrième et dernier chapitre est consacré à l'étude des boucles. Après avoir constaté l'impossibilité d'utiliser directement la méthode du chapitre précédent, nous définissons ce qu'est une approximation de la transformation associée à une boucle, puis nous décrivons une méthode pour obtenir une approximation de manière non itérative.

Nous concluons enfin en récapitulant les principaux résultats exposés dans cette étude, puis en évoquant certains axes de recherche, dans le prolongement de ce que nous avons étudié, qu'il nous paraît souhaitable d'approfondir.

Chapitre 1

Présentation du problème

Les architectures parallèles ont connu un essor tel que nous devons admettre que le matériel a pris de l'avance par rapport au logiciel. Les constructeurs annoncent aujourd'hui des performances spectaculaires, mais en prenant soin d'employer le terme de *performances théoriques*. Malheureusement, il est bien rare d'atteindre ces niveaux de performances. Le terme *performance théorique* signifie-t-il donc "niveau de performance inaccessible" ? Ou peut-être, en étant plus optimiste, les constructeurs expriment le fait que "la théorie du logiciel n'est pas encore arrivée à exploiter suffisamment les architectures que nous définissons, et la puissance que nous documentons est celle que l'on pourrait atteindre si la théorie était suffisamment avancée" ?

Quelle que soit la signification qu'il nous plaise d'affecter à ce terme, cette étude a pour but de rendre ces *performances théoriques* un peu moins inaccessibles. Le présent chapitre a pour rôle de présenter le problème que nous allons nous attacher à résoudre.

1.1 Architectures parallèles et vectorielles

De nombreux efforts de classification des architectures parallèles ont été menés, une bonne synthèse étant [HB85]. La classification de Flynn [Flyn66] est certainement la plus répandue, mais également la moins précise. Elle est basée sur l'exécution d'un flot d'instructions sur un ensemble de données, et répertorie quatre catégories d'architectures :

- SISD (Single Instruction stream, Single Data stream)

  Le flot d'instructions est exécuté par un seul processeur, qui traite une seule donnée à la fois. C'est l'architecture séquentielle classique. Rentrent aussi dans cette catégorie les architectures avec pipeline et/ou plusieurs unités fonctionnelles, comme les processeurs vectoriels CDC Cyber 205 ou Cray 1 ;

- SIMD (Single Instruction stream, Multiple Data stream)

De multiples processeurs élémentaires exécutent le même programme sur des données différentes. Ce sont les *array processors* ;

- MISD (Multiple Instruction stream, Single Data stream)

  Une seule donnée est transmise à plusieurs processeurs exécutant des instructions différentes. Cette catégorie n'a pas tellement de sens et n'a jamais connu de représentant réellement réalisé [Taba90] ;

- MIMD (Multiple Instruction stream, Multiple Data stream)

  Les processeurs élémentaires exécutant des programmes différents traitent des données différentes. C'est une architecture nommée multi-processeurs, comme l'IBM ES 9000, le Cray 2 ou le Multimax de Encore.

Ces différentes architectures nécessitent souvent des approches spécifiques lors de l'adaptation à la machine cible. Nous ne nous préoccupons pas de ces problèmes architecturaux, ce qui nous permet d'utiliser dans la suite le terme de *parallélisation* pour englober tous les aspects du parallélisme, y compris la vectorisation.

1.2 Programmation et parallélisme

Face à un ordinateur parallèle, il n'y a pas de miracle : pour obtenir de bonnes performances, le programme doit être parallèle ou parallélisable. Ces deux adjectifs symbolisent les deux approches traditionnelles du parallélisme.

1.2.1 Parallélisme explicite

La première approche consiste à utiliser un langage permettant au programmeur de définir lui-même les sections de programme aptes à être parallélisées ou vectorisées. Il faut pour cela un nouveau langage de programmation ou des extensions à un langage existant, et une formulation des problèmes avec les outils du parallélisme [Bana91]. Parmi les exemples de langages ou d'extensions de langages existants, nous pouvons citer Occam, pour les architectures à base de Transputers, Fortran 90 et Vector C pour leurs extensions vectorielles.

Occam

Issu du langage CSP [Hoar78], défini par le professeur Hoare et la société anglaise Inmos, le langage Occam [Inmo82] doit son nom à William d'Occam qui enseignait la philosophie à l'université d'Oxford au quatorzième siècle. Ce langage, conçu à l'origine pour être le langage de programmation du Transputer [Stre85], fournit les abstractions de processus[1] et d'envoi de message.

[1] La notion de processus est ici élémentaire : elle est à Occam ce que l'instruction est à Pascal et n'est donc pas à comparer avec la notion de processus dans un système d'exploitation.

Il est aisé de créer des processus concurrents en Occam. Grâce à des mécanismes simples, le langage est très pur. Le parallélisme est totalement exprimé avec le constructeur `PAR`, et la synchronisation est assurée par l'envoi et la réception de messages dans des canaux de communication.

Malgré cette simplicité, Occam reste toutefois cantonné à quelques applications et architectures particulières. La remise en cause des habitudes de programmation qu'il implique le prédispose peu aux grands programmes scientifiques.

Fortran 90

Longtemps appelée Fortran 8x, la nouvelle forme de Fortran a été normalisée en 1990 par l'American National Standards Institute. Fortran 90 introduit, entre autres, la notion de paquetage avec les modules, l'imbrication et la récursivité des procédures. En ce qui nous concerne, l'aspect le plus important est l'extension vectorielle, qui simplifie et optimise les opérations en étendant les opérations scalaires aux vecteurs et aux matrices.

Par exemple, `A(1:N) = B(1:N) + C(2:N+1)` revient à affecter au sous-ensemble du tableau `A` délimité par les indices `1` et `N` la somme des sous-ensembles des tableaux `B` et `C`, somme prise élément par élément. Ce type d'expression tire parti au maximum des architectures vectorielles telles que le Cray.

On peut, de plus, utiliser certaines fonctions internes étendues aux vecteurs, comme par exemple : `A(N1:N2) = 7 * SQRT(A(N1:N2))`.

De nouvelles structures de contrôle ont été introduites telles que `WHERE`, qui permet de faire une affectation conditionnelle sur les éléments d'un tableau, comme par exemple dans l'exemple suivant :

```
WHERE (B .NE. 0)
    C = A/B
    ELSEWHERE
    C = A
END WHERE
```

Fortran 90, compatible avec Fortran 77, permet de conserver les énormes bibliothèques de programmes scientifiques, tout en permettant une évolution en douceur vers des programmes utilisant au mieux les architectures vectorielles.

Vector C

Le langage Vector C [Li86] a été conçu et implémenté en 1984 à l'Université de Purdue, pour l'ordinateur vectoriel CDC Cyber 205. C'est une extension du langage C [KR78] orientée vers le traitement des tableaux comme Fortran 90, avec quelques opérations particulièrement bien adaptées à l'architecture du Cyber 205.

La première nouveauté intéressante est l'apparition de la notion d'intervalle d'indices (comme en Fortran 90, mais étendue) : on peut écrire par exemple :

- `a [*] = b [*] ;`
  Cette forme recopie tous les éléments du tableau `b` dans le tableau `a` ;
- `a [0:n:3] = b [0#m:2] ;`
  Cette forme recopie m éléments du tableau `b` à partir de l'indice 0 par pas de 2, dans les éléments entre 0 et n par pas de 3 du tableau `a` ;
- `a [5:*:2] = b [0:n-1] ;`
  Cette forme recopie tous les éléments du tableau `b` dont les indices sont compris entre 0 et $n-1$ (le pas est ici de 1 par défaut) dans les éléments du tableau `a` à partir de l'indice 5 par pas de 2 ;
- `a [ve [*]] = b [n#p] ;`
  Cette forme recopie les p éléments du tableau `b` à partir de l'indice n dans les éléments du tableau `a` sélectionnés par les indices du tableau d'entiers `ve` ;
- `m1 [*][*] = m2 [0#50][n:p] ;`
  Cette forme recopie le sous-ensemble de la matrice `m2` délimité par ses expressions d'indices (c'est à dire les 50 premières colonnes à partir de la colonne 0 et les lignes de n à p) dans la matrice `m1` prise dans sa totalité.

Deux nouveaux types de données sont introduits : il s'agit du vecteur de bits et du descripteur de vecteur. Le vecteur de bits est déclaré avec le nouveau mot-clef `bit`, il jouera le rôle de sélecteur d'éléments dans un tableau. Le descripteur de vecteur, déclaré avec le symbole `@` (comme `*` pour un pointeur) mémorise l'adresse et les indices sélectionnés (comme l'adresse pour un pointeur) afin d'optimiser les accès suivants, car la sélection est coûteuse sur le Cyber 205. Avec ces types de données, nous pouvons alors écrire :

- `bit vb [8] = b"11011001" ;`
  Cette ligne déclare un tableau `vb` de 8 bits et l'initialise avec la succession de bits `11011001` ;
- `float @dv ; dv = & a [5:53:7] ;`
  La première partie de cette ligne déclare un descripteur de vecteur `dv`. La deuxième partie est une instruction d'affectation : le descripteur de vecteur `dv` reçoit une description d'intervalle du tableau `a` ;
- `@dv = b [*] ;`
  Cette affectation recopie tous les éléments du tableau `b` dans les éléments du tableau (ou plutôt du sous-tableau) référencé par le descripteur de vecteur `dv`. Si nous gardons la définition de `dv` précédente, cette ligne est équivalente à `a [5:53:7] = b [*] ;`

- `a [@dve] = b [vb [*]] ;`
  Cette ligne a pour effet de recopier les éléments du tableau `b` sélectionnés par les éléments correspondants du tableau de bits `vb` (sont sélectionnés les éléments dont le bit correspondant est à 1) dans les éléments du tableau `a` sélectionnés par le descripteur de vecteur d'entiers ou de bits `dve`.

Les opérateurs classiques du langage C ont été étendus aux vecteurs. Ainsi, par exemple :

- `a [*] = b [*] + 5 + c [*] ;`
  Les éléments du tableau `a` reçoivent la somme des éléments des deux tableaux `b` et `c` et de la constante 5 ;
- `a [*] = vb [*] ? b [*] : c [*] ;`
  Les éléments du tableau `a` reçoivent les éléments du tableau `b` ou du tableau `c` suivant la valeur des bits du tableau `vb` ;
- `vb [*] = (a [*] >= 0) & (b [*] >= 0) ;`
  Les éléments du vecteur de bits `vb` reçoivent le bit 1 si les éléments correspondants des tableaux `a` et `b` sont positifs.

De nouveaux et puissants opérateurs sont introduits. Par exemple :

- `@~` : partie entière inférieure des éléments du vecteur ;
- `@|` : valeur absolue des éléments du vecteur ;
- `@%` : $v[i+1] - v[i]$ avec $0 \leq i \leq n-2$;
- `@+` : somme des éléments du vecteur ;
- `@*` : produit des éléments du vecteur ;
- `@<` : minimum des éléments du vecteur ;
- `@.` : produit scalaire des vecteurs ;
- etc.

Cette description est très rapide, mais nous pouvons toutefois constater que Vector C est une extension très intéressante du langage C. Malheureusement, certaines opérations sont très liées à l'architecture du Cyber 205, ce qui limite sa portée.

1.2.2 Parallélisme implicite

La deuxième approche de la programmation parallèle consiste à laisser faire par le compilateur (ou la chaîne complexe que le programmeur met en œuvre en demandant une traduction de son programme) le travail d'extraction du parallélisme. Cette approche peut être appliquée à deux sortes de langages de programmation :

- des langages non impératifs, tels que :
  - les langages fonctionnels, comme Fp [Back78],
  - les langages logiques, comme Prolog,
  - les langages pour architectures data-flow,

  Dans ces langages, on ne réalise pas un programme composé d'affectations, de boucles et d'appels de procédures, mais on énonce des équations, la formalisation d'un problème, des combinaisons d'opérateurs. Le compilateur ou l'interprète tente de trouver une solution au problème posé, à travers un enchaînement d'actions. Le programmeur ne spécifiant pas ces actions, le compilateur ou l'interprète est libre de les choisir, et des les agencer. Dans le cas d'une utilisation sur architecture parallèle, ces langages se révèlent de bons candidats pour la parallélisation.

  Le problème est que ces langages sont encore l'apanage de spécialistes, et ne connaissent pas encore le succès qu'a Fortran, par exemple, dans le domaine scientifique.

- des langages impératifs (comme Fortran, Pascal, C, etc.). Il s'agit là de prendre un programme séquentiel classique, puis d'en extraire le parallélisme s'il existe. La phase de compilation est donc secondée d'une phase de parallélisation automatique. De nombreux efforts de recherche se sont portés sur les paralléliseurs automatiques, comme les paralléliseurs PAF [TDF87] ou Parafrase [KKLW80].

  Cette approche ne dispense pas le programmeur de "penser parallèle", mais elle offre la possibilité de "programmer parallèle portable", puisque c'est le paralléliseur qui détermine le meilleur choix en fonction de l'architecture. C'est aussi la seule méthode qui permette de paralléliser les bibliothèques scientifiques écrites en Fortran sans les réécrire, ce qui est irréalisable lorsque l'on considère l'ampleur de la tâche.

1.2.3 Objectifs

Le domaine qui a retenu notre attention est celui de la détection automatique du parallélisme. En particulier, nous nous intéressons à la parallélisation automatique de programmes en langage C.

A l'heure actuelle, la majorité des efforts se sont concentrés sur Fortran : pour des raisons historiques, c'est le langage le plus utilisé dans le monde scientifique, gros consommateur de calcul.

Toutefois, le langage C dispose de plusieurs caractéristiques qui font de lui un bon candidat pour de nouvelles recherches :

- c'est un langage répandu ; certes, pas aussi répandu que le Fortran ou le Cobol dans leurs domaines respectifs mais, malgré ses arcanes et ses recoins obscurs, beaucoup de programmeurs utilisent aujourd'hui ce langage et en font un point de passage obligé ;
- c'est un langage maintenant stable : il a été normalisé par l'ANSI en 1989, et est en voie de devenir un standard international (ISO) en 1991 ;
- c'est un langage de bas niveau, adapté au système. Unix [Bach86][LMKQ89], Mach, Xinu sont des exemples de systèmes d'exploitation écrits en langage C ;
- c'est un langage adapté aux problèmes graphiques [SGN88] où les besoins en parallélisme sont importants ;
- c'est un langage disposant de structures de données et de contrôle plus riches que celles de Fortran ;
- c'est un langage qui s'interface relativement bien avec d'autres langages tels que Pascal ou Fortran, ce qui permet d'utiliser des bibliothèques écrites dans un autre langage, ou de constituer des bibliothèques pour d'autres langages ;
- c'est un langage efficace et portable, adapté pour constituer la cible de systèmes de compilation novateurs tels que C++ [Stro86][ES90], ou servir de base à des travaux généraux de compilation [Seni91] entre autres.

Ces avantages font que le langage C est un langage mieux adapté que Fortran à traiter toute une classe de problèmes (implémentation d'interprètes, de systèmes, d'algorithmes non numériques nécessitant des structures de données évoluées). Ces problèmes, bien que non numériques, peuvent tirer profit d'une architecture parallèle.

Thornburg [Thor86] a analysé les problèmes posés par l'utilisation du langage C dans le calcul numérique. En substance, il s'agit de :

- le mode de passage des arguments à une fonction est différent entre Fortran (tous les arguments sont passés par adresse, y compris les constantes) et C (le passage par adresse est explicite, comme en Pascal). Ceci nécessite que le programmeur soit averti de cette différence ;
- l'ordonnancement des tableaux multi-dimensionnels est différent entre Fortran et la plupart des autres langages. En Fortran, les tableaux sont rangés par colonnes (l'élément suivant $M_{i,j}$ est $M_{i+1,j}$), alors qu'en C, les tableaux sont rangés par lignes (l'élément suivant $M_{i,j}$ est $M_{i,j+1}$) ;
- le langage C calcule systématiquement les expressions flottantes en double précision, même lorsque tous les opérandes sont en simple précision ;
- le langage C n'admet pas le type `COMPLEX` de Fortran ;
- le langage C n'admet pas l'opérateur d'exponentiation.

Si certains points sont peu importants, les autres sont liés aux choix pris lors des premières implémentations de Fortran, et ne sont pas liés particulièrement au langage C. Sous réserve que les programmeurs soient avertis de ces différences, ils peuvent utiliser les bibliothèques Fortran.

1.3 Détection automatique du parallélisme en Fortran

Les recherches se sont longtemps focalisées sur Fortran. Nous disposons alors d'un certain nombre de techniques et d'outils les exploitant.

Nous présentons, dans cette partie, ces techniques afin de comprendre comment s'insère notre travail dans les paralléliseurs.

1.3.1 Techniques de parallélisation automatique

Le principe fondamental de ces techniques est le même que celui des compilateurs et des optimiseurs : la correction. Le traitement ne doit pas changer la sémantique du programme. Que dirait-on si un optimiseur modifiait le comportement du programme qui lui est soumis ? Il en va de même pour un paralléliseur : s'il n'est pas faux de déclarer séquentiel un programme qui aurait pu être parallélisé, il est en revanche faux de paralléliser un programme qui n'est pas parallélisable, puisque le résultat sera vraisemblablement différent de la version séquentielle. Autrement dit, en face d'un doute, on choisit toujours la version séquentielle, la moins optimiste.

Les conditions qui sont à la base de tous les paralléliseurs actuels sont les conditions de Bernstein [Bern66]. Appelons $m(i)$ et $l(i)$ les variables respectivement modifiées et lues par l'instruction i. Supposons que nous ayons deux instructions r et s, r étant la première dans le programme. Ces deux instructions peuvent être exécutées en parallèle si le résultat ne dépend pas de l'ordre de leur exécution, c'est à dire si l'exécution de r puis s conduit au même résultat que l'exécution de s puis r. Pour qu'il n'y ait pas dépendance entre r et s, il suffit que les trois conditions suivantes soient simultanément vérifiées :

- $m(r) \cap l(s) = \emptyset$
  C'est une dépendance PC (producteur-consommateur). C'est la seule vraie dépendance : elle exprime le fait qu'on calcule un résultat (dans r) que l'on veut réutiliser ultérieurement (dans s). Les deux instructions r et s doivent vraiment être exécutées en séquence ;
  Par exemple :

  ```
  r     x = y ;
  s     z = x ;
  ```

- $m(r) \cap m(s) = \emptyset$
  C'est une dépendance PP, qui exprime le fait qu'on réutilise une variable dont l'ancienne valeur, initialisée dans r, ne sert plus à partir de l'instruction s. Ces dépendances peuvent être éliminées à l'aide d'une technique appelée "expansion de scalaire" (voir para-

graphe 1.3.2) ;
Par exemple :

```
r    x = y ;
s    x = z ;
```

- $l(r) \cap m(s) = \emptyset$
  C'est une dépendance CP, qui exprime le fait qu'une variable utilisée dans r est réutilisée pour mémoriser une autre valeur dans s.
  Par exemple :

```
r    x = y ;
s    y = z ;
```

Les anglophones utilisent les termes *true-dependence* pour dépendance PC, *output-dependence* pour dépendance PP et *anti-dependence* pour dépendance CP. Notons que, dans [PH90], les auteurs utilisent les termes WAR (*write after read*), WAW (*write after write*) et RAW (*read after write*), ce qui est tout de même plus mnémotechnique !

Si les conditions de Bernstein sont assez faciles à détecter lorsque le segment de programme étudié est une séquence linéaire d'instructions, il n'en va pas de même lorsqu'il s'agit d'une boucle (ou d'un nid de boucles, c'est à dire de boucles imbriquées à plus d'un niveau) avec des tableaux. Il est alors plus difficile d'analyser les dépendances, puisque les boucles peuvent elles-mêmes en générer (on parle alors de *loop carried dependences*), comme par exemple [PH90] :

```
      do 10 i=1,100
r           a (i+1) = a(i) + b(i)
s           b (i+1) = b(i) + a(i+1)
10    continue
```

Cet exemple illustre quelques dépendances qui n'existeraient pas s'il n'y avait pas de boucle :

- la case mémoire `a(i+1)` écrite dans l'instruction r au tour i de la boucle correspond à la case mémoire `a(i)` lue dans l'instruction r au tour $i+1$, c'est une dépendance de type PC ;
- la case mémoire `b(i)` lue dans les instructions r et s au tour i de la boucle correspond à la case mémoire `b(i+1)` écrite dans l'instruction s au tour $i-1$ de la boucle, c'est une dépendance de type CP ;

Il n'y a pas de dépendance de type PP dans cet exemple.

Dans le cas général d'un nid de boucles, l'instance d'une instruction est référencée par sa position lexicale dans le programme et par un vecteur de compte-tours $v = [i_1 \dots i_n]$ où les i_j

sont les valeurs des compte-tours des n boucles englobant l'instruction. Soient deux instances d'instructions r (vecteur de compte-tours $v = [i_1 \dots i_n]$) et s (vecteur de compte-tours $v' = [i'_1 \dots i'_{n'}]$). Le nombre de boucles englobantes n'est pas forcément le même. Nommons p le nombre de boucles communes à r et s.

Les deux instructions r et s sont en collision à la profondeur p si [TDF87] :

- (r, v) s'exécute avant (s, v') dans le programme séquentiel, ce qui s'exprime par :

$$\left\{\begin{array}{ll} & i_1 < i'_1 \\ \text{ou} & i_1 = i'_1 \text{ et } i_2 < i'_2 \\ & \quad\vdots \\ \text{ou} & i_1 = i'_1 \text{ et } \dots \text{ et } i_{p-1} = i'_{p-1} \text{ et } i_p < i'_p \\ \text{ou} & i_1 = i'_1 \text{ et } \dots \text{ et } i_p = i'_p \text{ et } T_{rs} \end{array}\right.$$

  où T_{rs} est un booléen indiquant si r précède s dans le texte du programme ; et

- une des conditions de Bernstein est violée, c'est à dire si les deux instructions accèdent à la même case d'un tableau. Le problème est que les indices sont des fonctions différentes des comptes-tours. On a alors le système d'équations $\Phi(v) = \Phi'(v')$ à résoudre, où les fonctions Φ représentent le vecteur des indices du tableau en question, ces indices étant calculés à partir des compte-tours.

Ces deux conditions se traduisent en un système d'équations aux contraintes que nous pouvons compléter par des conditions telles que la validité des comptes-tours (ils doivent être compris entre la borne inférieure et la borne supérieure), ou d'autres conditions issues d'une analyse sémantique plus poussée. Dans le cas où les fonctions Φ sont des fonctions linéaires des compte-tours de boucles (ce qui est la très grande majorité), on peut alors résoudre ce système en nombres entiers. La méthode des coupes de Gomory fournit une méthode générale et précise.

La résolution de ce système étant une opération précise, mais coûteuse en temps de calcul, [Bane76] utilise un test moins précis (donc adoptant une attitude peut-être pessimiste), mais beaucoup plus rapide. On vérifie d'abord que le plus grand commun dénominateur des coefficients divise le terme constant. Puis, en s'appuyant sur les bornes des inconnues, on vérifie que l'expression change de signe d'une extrémité à l'autre de l'intervalle de variation. Si l'un de ces tests échoue, l'équation n'a pas de solution, et il n'y a donc pas de dépendance. Dans le cas contraire, on ne peut rien affirmer, et l'on se place donc dans le cas pessimiste : il y a dépendance entre les instructions.

Les dépendances sont synthétisées dans un graphe de dépendances, dans lequel les sommets sont les instructions. Il y a arc entre deux sommets s_1 et s_2 si et seulement si il y a dépendance (ou soupçon de dépendance) entre s_1 et s_2 (quels que soient les tours de boucle éventuels). Deux instructions reliées par un arc ne peuvent donc pas être exécutées en parallèle.

Le graphe de dépendances synthétise tout le parallélisme du programme. Le problème est ensuite de l'exploiter. Par exemple, l'algorithme de Kennedy et Allen [AK85] parcourt le graphe de dépendances enrichi d'informations telles que la profondeur (c'est à dire le nombre de boucles communes englobant les instructions) de collision, en extrait des composantes fortement connexes pour extraire les boucles parallèles au premier niveau, supprime les arcs correspondant à la profondeur, et itère pour chaque profondeur.

1.3.2 Transformations de programme

Pour augmenter le parallélisme ou l'adéquation à la machine cible, il est possible de transformer le programme pour en donner une version sémantiquement équivalente, mais où certaines dépendances peuvent être évitées. Padua et Wolfe [PW86] en proposent certaines, parmi lesquelles nous avons sélectionné quelques exemples, particulièrement en ce qui concerne les transformations visant à accroître le parallélisme.

Reconnaissance des variables d'induction

Les variables d'induction sont les variables qui, à l'intérieur d'une boucle, suivent une progression arithmétique. Détecter ces variables et exprimer leur valeur en fonction du compte-tours de la boucle permet de connaître les indices des tableaux, et de procéder à l'analyse des dépendances. Par exemple :

```
         j = n
         do 10 i=1,n
               k = 2*i - 1
               x(j) = y(i) + z(k)
               j = j - 1
   10    continue
```

peut être transformé, après élimination des variables d'induction j et k, en :

```
         do 10 i=1,n
               x(n-i+1) = y(i) + z(2*i-1)
   10    continue
```

qui peut être vectorisé en `x(n:1:-1) = y(1:n) + z(1:2*n-1:2)`. Les variables d'induction sont purement et simplement retirées si elles ne sont plus utilisées ultérieurement.

Expansion de scalaires et de tableaux

L'expansion de scalaire est une technique utilisée pour supprimer les dépendances PP générées par la réutilisation de la même variable scalaire dans une boucle. Par exemple :

```
      do 10 i=1,n
            x = a(i) + b(i)
            c(i) = x ** 2
10    continue
```

contient une dépendance PP car la variable x est affectée à chaque tour de boucle, sans être lue. On peut convertir cet exemple en :

```
      do 10 i=1,n
            tmpx(i) = a(i) + b(i)
            c(i) = tmpx(i) ** 2
10    continue
      x = tmpx(n)
```

En remplaçant cette unique boucle par deux nouvelles, on peut vectoriser en :

```
      tmpx(1:n) = a(1:n) + b(1:n)
      c(1:n) = tmpx(1:n) ** 2
      x = tmpx(n)
```

Cette technique a été généralisée aux tableaux dans [Feau88a]. L'expansion elle-même est relativement simple puisqu'il s'agit, si la dépendance PP est entre deux instructions différentes, d'utiliser un nouveau tableau dans l'une des instructions. Si la dépendance PP est provoquée par deux exécutions différentes de la même instruction (la dépendance est alors générée par la ou les boucles), l'expansion est réalisée en utilisant un tableau indicé par les n indices des boucles englobantes. En revanche, le problème ardu réside dans le remplacement des utilisations des cases du tableau original par des références au nouveau tableau. En effet, il faut localiser toutes les instructions où le tableau est modifié, puis extraire la dernière valeur mémorisée, c'est à dire calculer l'instruction et le vecteur d'itération correspondant à la dernière modification de la case du tableau, ce qui se traduit par la résolution d'un système de contraintes paramétrique.

Échange de boucles

Dans des nids de boucle, l'ordre peut souvent être modifié sans affecter la sémantique du programme. Ainsi, la boucle :

```
      do 10 j=1,n
            do 10 i=2,n
                  a(i,j) = a(i-1,j) + b(i)
10    continue
```

n'est pas vectorisable, car il y a dépendance PC entre $a(i,j)$ affectée au tour i et $a(i-1,j)$ utilisée au tour $i+1$. Si nous réécrivons cette boucle en inversant l'ordre des boucles, nous obtenons le programme équivalent :

```
      do 10 i=2,n
            do 10 j=1,n
                  a(i,j) = a(i-1,j) + b(i)
10    continue
```

où la dépendance PC est supprimée, et la boucle interne peut être vectorisée en :

```
      do 10 i=2,n
            a(i,1:n) = a(i-1,1:n) + b(i)
10    continue
```

1.3.3 Quelques paralléliseurs automatiques

Ces techniques de base ont été mises en œuvre dans les paralléliseurs automatiques. Nous décrivons ici deux paralléliseurs, à titre d'exemple d'intégration des techniques que nous avons vues.

PAF

Le paralléliseur *PAF* [TDF87] (pour Paralléliseur Automatique de Fortran) est le paralléliseur développé au laboratoire MASI dans l'équipe du Professeur Feautrier. Sa structure [Davi87] est représentée en figure 1.1.

L'analyseur syntaxique est décrit dans [Amma86]. Il prend en entrée un programme Fortran et le transforme en une représentation intermédiaire très proche de Le_Lisp [Chai87]. Le programme est ensuite restructuré [Amma88] pour éliminer les `goto` et normaliser les boucles.

Le calcul du graphe de dépendance extrait ensuite les dépendances à l'aide d'un module de résolution de systèmes d'équations et d'inéquations paramétriques en nombres entiers [Feau88b]. Le graphe de dépendances est représenté par une liste d'instructions, chaque instruction faisant référence aux boucles et tests dans laquelle elle figure.

Ce graphe de dépendance est ensuite compilé par une extension de l'algorithme de Kennedy et Allen [AK87]. L'ordonnancement optimal des tâches parallèles étant un problème NP-complet, des heuristiques sont employées pour résoudre statiquement le problème [Tawb91].

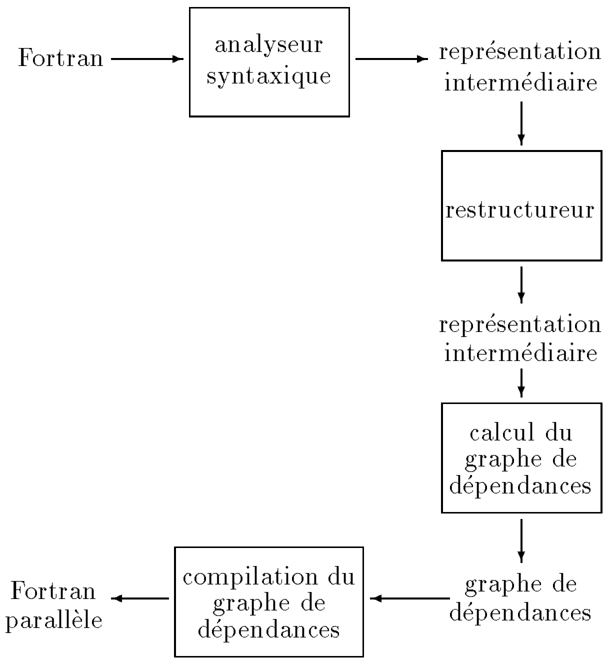

Figure 1.1: Paralléliseur PAF

Parafrase-2

Le paralléliseur Parafrase-2 [PGHL89] est le successeur de Parafrase [PKL80][KKLW80]. Le cœur du paralléliseur utilise une représentation intermédiaire, ce qui lui permet d'être indépendant du langage réel. Des pré-processeur et post-processeur sont rédigés pour traduire le langage réel en représentation intermédiaire et réciproquement.

Le calcul des dépendances est réalisé avec les tests de Banerjee [Bane76], complété de quelques simplifications symboliques.

Un module d'analyse statique des programmes est ensuite invoqué pour estimer les durées d'exécution du programme. Ce module suppose défini un modèle d'architecture. À présent, seul le modèle de multi-processeurs à mémoire partagée est implémenté. L'intérêt d'un module d'estimation des durées d'exécution est pour sélectionner des transformations et optimisations. Par exemple, si une tâche est courte, le coût de gestion de la synchronisation peut devenir prohibitif et il vaut mieux renoncer à la parallélisation.

Parafrase-2 procède également à l'analyse inter-procédurale pour obtenir des informations sur les alias et pour propager les constantes. Cette analyse est résumée, puis conservée pour utilisation ultérieure, évitant ainsi de procéder à l'analyse à chaque site d'appel.

Une fois le graphe de dépendances établi, certaines des transformations vues en 1.3.2 sont appliquées.

Enfin, l'ordonnancement est statique et dynamique : le partitionnement du programme est réalisé statiquement, mais chaque tâche peut se décomposer dynamiquement en sous-tâches. Une fois le programme partitionné statiquement, Parafrase-2 peut insérer des suggestions d'ordonnancement suivant les informations qu'il a pu extraire statiquement du programme. Dynamiquement, un ordonnanceur traite les tâches comme des unités élémentaires d'exécution.

1.3.4 Conclusion

Les techniques de parallélisation automatique sont arrivées à maturité pour ce qui est du Fortran. Les paralléliseurs modernes sont devenus des ensembles complexes mettant en œuvre un nombre croissant de traitements afin d'obtenir le degré de parallélisme (*speedup*) le plus élevé au vu du texte du programme.

1.4 Détection automatique du parallélisme en C

La section précédente nous a permis de constater que la parallélisation automatique de programmes en Fortran a fait l'objet de nombreuses recherches fructueuses.

Cependant, qu'en est-il pour les programmes en langage C ? Quels sont les travaux réalisés ? Nous proposons d'examiner d'abord les problèmes qui font la difficulté de l'analyse du C, puis nous verrons les solutions qui y ont été apportées. Enfin, nous situerons notre approche dans ce contexte.

1.4.1 Problèmes posés

Un des premiers articles évoquant les problèmes posés par le langage C est dû à Lori Grob [Grob88]. Certains points rendent l'analyse des dépendances difficiles, à savoir la récursivité, la présence d'unions (c'est à dire de zones mémoire accédées suivant des types différents à plusieurs endroits du programme), les paramètres, les variables globales et les pointeurs non contraints (ce qui signifie qu'ils peuvent adresser n'importe quelle variable et que leur type peut être forcé). De tous ces points, seule la présence de pointeurs non contraints est caractéristique du langage C. Lori Grob pense qu'il n'y a pas de solution générale au problème, mais cite quelques voies éventuelles de recherche. En particulier, elle pense que les pointeurs utilisés pour indexer des tableaux est un domaine où il est possible de détecter les dépendances.

Allen et Jonhson [AJ88] soulignent plus précisément les difficultés de l'optimisation de C sur des architectures parallèles. Ils citent, entre autres points délicats :

- les affectations et les effets de bord en général (par exemple l'opérateur `++`), ainsi que les opérateurs conditionnels (par exemple les opérateurs `&&` et `?:` se compilent facilement et efficacement sur des architectures séquentielles, mais rendent l'analyse très difficile. De plus, ces opérateurs sont intrinsèquement séquentiels, ce qui nécessite un retrait ou une transformation préalable ;

- les boucles *for* en C sont considérablement moins restrictives que les boucles *do* de Fortran. Les bornes et l'incrément de la boucle peuvent être modifiées entre deux itérations, des sauts peuvent être effectués légalement dans le corps de la boucle et, pour couronner le tout, il est d'usage de placer des opérateurs avec effet de bord pour incrémenter la variable de boucle, quand on a la chance d'avoir une variable de boucle ;
- le langage encourage l'utilisation de petites fonctions, obstacles à la parallélisation des boucles contenant des appels ;
- l'utilisation de C dans du code de très bas niveau a imposé des constructions difficiles à optimiser, comme en particulier les variables *volatiles*, dont les changements de valeur ne sont pas provoqués par l'exécution du programme, et dont la sémantique n'est pas toujours spécifiée par le texte de la norme ANSI ;
- l'opérateur `&` d'extraction d'adresse permet la modification des variables du programme de manière très subtile, compliquant ainsi la tâche d'analyse.

De plus, nous pouvons ajouter à ces points le très épineux calcul d'adresses, c'est à dire l'arithmétique sur les pointeurs, caractéristique du langage C et qui l'a fait surnommer *assembleur fortement typé*.

1.4.2 Solutions apportées

La difficulté principale de l'analyse du langage provenant des pointeurs, certaines recherches se sont focalisées sur l'étude des structures de données, notamment les structures de données récursives telles que arbres ou listes, sujet déjà étudié pour des langages autres que C, comme les langages fonctionnels.

Weihl [Weih80] propose, dans un contexte non lié au langage C, un algorithme pour l'analyse inter-procédurale en présence de pointeurs et de pointeurs sur des fonctions. Il propose de créer des associations variable-valeur, où la valeur peut elle même être une variable, puis à propager ces relations d'alias par fermeture transitive. Toutefois, cet algorithme ne traite ni les structures, ni les tableaux, ce qui limite sa pertinence pour l'étude du langage C.

D'autres travaux [HPR89][Bodi90] ont pour but de condenser le graphe des données en une représentation finie. Seul le travail de François Bodin [Bodi90] est appliqué directement au langage C. L'objectif est la parallélisation de l'utilisation de données dynamiques : les programmes traités contiennent des pointeurs (mais pas d'arithmétique), des structures, des allocations de mémoire (fonction `malloc`) et des boucles. À chaque adresse utilisée est associé un nom, ce qui permet de construire un graphe dans lequel les sommets sont les adresses (de structures) et les arcs les champs permettant de passer d'un sommet à l'autre. Ce graphe, dans les cas des boucles, n'est pas fini. On conçoit donc un *graphe résumé* dans lequel les sommets sont *toutes* les adresses (de structures) allouées à un point de programme donné et les arcs (s_1, s_2) sont étiquetés par le champ de la structure permettant d'accéder à une des adresses du sommet

s_2 à partir d'une des adresses du sommet s_1 et par le nombre de tours de boucle séparant la création de s_1 de la création de s_2. Le graphe est construit par analyse data-flow [ASU86] pour les séquences linéaires, puis interprétation itérative (jusqu'à obtenir un graphe stable) dans le cas des boucles. Une fois ce graphe obtenu, il est possible d'analyser les chemins d'accès aux données, et d'extraire les dépendances entre deux chemins afin de procéder à la parallélisation. Malgré l'intérêt certain de cette étude, les problèmes que nous avons précédemment répertoriés ne sont pas évoqués, ce qui laisse notre problème intact.

La construction itérative de ces résumés de graphes de données pouvant s'avérer coûteuse en temps, [CWZ90] propose de construire un graphe SSG (*storage shape graph*) dans lequel les sommets sont les variables simples ou les structures dynamiquement allouées, et les arcs modélisent les valeurs des pointeurs. Il peut y avoir plusieurs arcs sortant d'un même sommet signifiant ainsi que la valeur réelle correspond à l'un des sommets adressés. La construction du graphe est réalisée par un algorithme efficace tirant profit d'une part du nombre réduit de références à une structure (en général pas plus d'une), et d'autre part du fait que le langage traité est à affectation unique.

Un grand nombre de travaux a donc été publié dans le domaine de l'analyse des structures de données. Cependant, un faible nombre est applicable aux programmes exprimés en langage C, et aucun ne prend en compte l'arithmétique sur les pointeurs.

Qu'en est-il des réalisations complètes, c'est à dire des paralléliseurs ?

En 1988, Allen et Jonhson [AJ88] décrivent un compilateur C pour une machine (*titan*) multiprocesseurs (RISC) et vectorielle. Face à la diversité des problèmes posés par le langage C, et confrontés à une réalité commerciale, leur approche pragmatique a été d'utiliser le compilateur `pcc` d'AT&T, avec des ajouts de la norme ANSI, comme frontal d'un paralléliseur. En utilisant une représentation intermédiaire de plus haut niveau que celle que l'on trouve dans les compilateurs traditionnels, ils transforment le programme pour enlever des expressions tous les effets de bord. Les boucles *for* sont converties en boucles *while* par le frontal, puis les boucles *while* sont analysées pour en extraire des boucles *do* à la Fortran, et les variables d'induction sont substituées. Enfin, les auteurs décrivent la technique de l'expansion des procédures aux sites d'appels comme une solution au problème de l'utilisation des fonctions évoqué dans le paragraphe 1.4.1. Malheureusement, les auteurs, sans doute confrontés à des contraintes commerciales, ne donnent pas beaucoup de détails sur leurs travaux.

Le paralléliseur Parafrase-2 [PGHL89], quant à lui, dispose d'un pré-processeur et d'un post-processeur pour le langage C, ce qui n'est pas suffisant pour constituer un paralléliseur pour C. Les auteurs sont assez discrets sur le traitement de ce langage, et ils reconnaissent que l'analyse de dépendances en présence de pointeurs n'a pas été implémentée.

Après ce rapide survol des travaux publiés sur le sujet, on s'aperçoit que le langage C suscite depuis peu l'intérêt des chercheurs, mais qu'il y a malheureusement très peu de solutions satisfaisantes apportées au problème de la parallélisation.

1.5 Conclusion

Quelle voie allons-nous donc suivre ? L'approche consistant à analyser les structures de données souffre à nos yeux de plusieurs défauts :

- le parallélisme issu de l'analyse des structures de données semble étroit : par exemple, il y a vraisemblablement très peu de cas où cette analyse conduit à une vectorisation des programmes, ce qui la prive d'une large classe d'applications ;
- il s'agit de travaux issus principalement de recherches sur la parallélisation de langages non numériques (comme par exemple les langages fonctionnels), et qui utilisent assez peu l'ensemble des techniques développées pour Fortran ;
- l'application directe de l'analyse des structures de données au langage C est tentante, mais elle nous semble prématurée. Nous pensons en effet qu'il y a des problèmes à résoudre en priorité, tels les effets de bord dans les expressions, la situation des boucles *for*, ou encore l'arithmétique sur les pointeurs. Sans avoir traité au préalable ces problèmes, on risquerait de n'accepter qu'un langage réduit, qui n'aurait plus rien en commun avec le vrai langage C sinon les accolades...

Nous allons donc reprendre le problème au début, méthodiquement, de manière analogue aux premiers travaux sur le parallélisation automatique en Fortran. Loin de vouloir tout reconstruire, ce qui ne serait pas réaliste, nous cherchons au contraire à utiliser au maximum tout l'attirail de la parallélisation automatique en Fortran dans un nouveau contexte.

L'application directe des méthodes de Fortran n'est pas envisageable dans le cas général. Notre but est donc dans cette étude d'identifier et de reconstruire les boucles de type *do* dans les programmes en langage C, lorsque cela est possible bien entendu. Le problème est rendu difficile par la présence des pointeurs et de l'arithmétique sur ces pointeurs : un pointeur peut servir de compte-tours par exemple...

La manière la plus efficace d'utiliser les techniques de parallélisation automatiques est d'intégrer ce que nous allons développer dans un paralléliseur existant comme PAF (voir les éléments en pointillés dans la figure 1.2). Ce n'est malheureusement pas directement possible, pour beaucoup de raisons principalement dûes en la différence entre Fortran et C (par exemple, les paralléliseurs de Fortran n'ont aucun support pour les pointeurs), sans une refonte des premières passes du paralléliseur. Même si cette étape n'est pas réalisée à l'heure actuelle, car elle n'apporte pas vraiment d'élément nouveau à la compréhension du problème, elle permet de situer notre approche et ses hypothèses.

Résumons notre démarche :

- nous désirons admettre un sous-ensemble représentatif du langage C, incluant en particulier les expressions avec effet de bord, les tests et les boucles ;

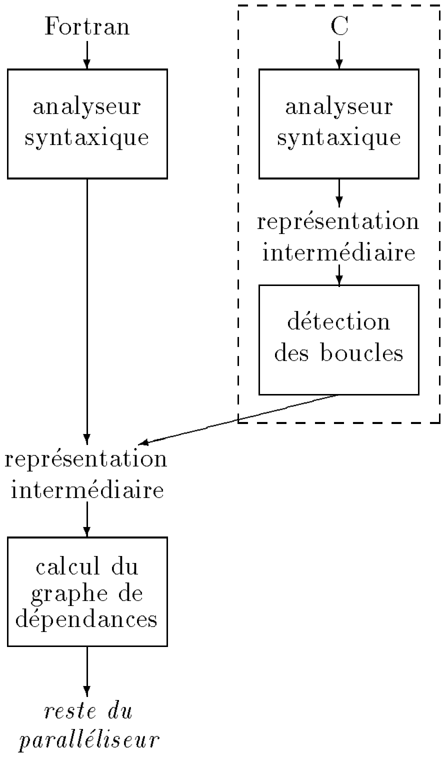

Figure 1.2: Paralléliseur automatique pour C

- nous ne pouvons pas faire confiance aux boucles *for*, aussi nous cherchons à détecter les boucles analogues aux boucles *do* de Fortran ;
- le résultat est ensuite traité par un paralléliseur "classique", légèrement modifié, pour générer du code parallèle.

Chapitre 2

Analyse syntaxique et normalisation des programmes

Ce chapitre est consacré aux premiers traitements de notre analyseur : l'analyse syntaxique du programme et sa normalisation.

Avant toute chose, il faut définir le langage avec lequel nous envisageons de faire notre analyse sémantique. Il est en effet irréalisable de vouloir aborder de front l'ensemble du langage C : nous préférons traiter avec succès une partie du C plutôt qu'avoir de mauvais résultats, voire une absence de résultat, sur la totalité du langage !

Mais il ne faut pas simplifier le problème à l'extrême, et traiter un "Fortran avec des accolades". Nous avons identifié un certain nombre de difficultés dans le chapitre précédent, et certaines dépassent largement le cadre de notre étude. Nous devons donc trouver un compromis viable entre exhaustivité et difficulté.

Les restrictions que nous définissons sont de trois ordres :

- les constructions posant trop de problèmes nouveaux pour faire l'objet d'une première étude, comme les *casts* ou les unions ;
- les sujets déjà étudiés comme l'analyse inter-procédurale ou les constructions déstructurantes (`goto`, `break`, etc) ;
- les constructions dont le support n'est qu'une extension triviale des mécanismes mis en œuvre, comme la boucle `do ... while`, les types structurés, le choix multiple ou l'instruction `continue` ;

La première partie de ce chapitre est consacrée à la définition du sous-ensemble du langage C que nous prenons en entrée (le langage d'*entrée*). Cette définition est essentiellement restrictive, c'est à dire que nous partons du langage C et nous lui enlevons des constructions.

La deuxième partie de ce chapitre détaille le langage que la phase d'analyse décrite dans les chapitres suivants admet (le langage d'*analyse*). Autrement dit, il s'agit du langage en sortie de l'analyseur syntaxique et du normaliseur. La définition que nous en donnons est explicite, c'est à dire que nous ne spécifions plus des restrictions, mais au contraire un langage.

La dernière partie est consacrée à la structure générale de l'implémentation de l'analyseur syntaxique et du normaliseur. Nous mettons en particulier l'accent sur quelques points qui nous paraissent intéressants, comme la représentation intermédiaire, fil conducteur de l'implémentation des chapitres suivants, et les performances.

2.1 Langage d'entrée

Le langage utilisé dans cet exposé est un sous-ensemble du langage C, tel que normalisé par l'ANSI [ANSI88]. Un des objectifs de la norme était de garder la compatibilité avec le langage C traditionnel [KR78], en précisant certains points et en apportant de nouvelles constructions telles que les prototypes ou les qualificateurs [KR88, annexe C].

Une description précise et nettement plus pédagogique que le texte de norme a été rédigée par Harbison et Steele [HS87]. Elle contient en outre de précieuses informations sur les problèmes de portabilité avec les compilateurs antérieurs. [Riff86] est également un texte très pédagogique pour le C traditionnel.

Cette section a pour but de détailler les différences entre le C "ANSI" et le langage que nous nous proposons d'admettre.

2.1.1 Types de données

Par rapport aux paralléliseurs et vectoriseurs traditionnels ne manipulant que des types de base (entiers et flottants) et des tableaux, notre langage présente un nouvel élément : le pointeur.

Les types de données sont donc :

- les types de base ; nous prenons ceux du langage C, c'est à dire `int`, `long int`, `short int`, `char`, dans leurs versions signées ou non signées, ainsi que les types flottants, c'est à dire `float`, `double` et `long double` ;
- les pointeurs ; comme en langage C, les pointeurs sont fortement typés. Un pointeur repère un objet d'un type déterminé et ne peut adresser un objet d'un autre type. Les conversions de types n'étant pas supportées, un pointeur ne peut que respecter son type ;
- les tableaux ;

Un type complexe peut être vu comme un chemin d'accès à un type de base. Ainsi, la déclaration classique en C :

```
char *argv [10] ;
```

déclare un tableau nommé `argv` de 10 pointeurs sur des caractères. Nous pouvons représenter ceci comme une chaîne constituée des maillons *tableau de*, *pointeur sur* et *caractère*.

Les champs de bits ne sont pas traités.

2.1.2 Fonctions et procédures

Notre langage, comme le langage C, possède des fonctions, ce qui nous permet d'écrire des programmes : les fonctions délimitent un segment de programme et lui donnent un nom.

En revanche, l'appel de fonction n'est pas traité. Notre approche ne vise pas pour le moment les problèmes posés par la parallélisation inter-procédurale.

2.1.3 Expressions

Les expressions de notre langage sont celles du langage C, à l'exception de deux constructions notables :

- l'appel de fonction ;
- la conversion de type explicite (*cast*).

Bien qu'il n'y ait pas d'obstacle majeur au traitement des fonctions, nous avons préféré éviter de les introduire dans notre formalisme par souci de clarifier l'exposé.

La conversion de type explicite, quant à elle, pose des problèmes bien plus difficiles. En effet, la notion de typage fort introduit une structuration de la mémoire, objet central de notre étude : nous nous servons plus d'une fois du fait qu'un pointeur ne peut pointer en dehors du sous-ensemble de la mémoire défini par son type (ou plus précisément par le type des objets pointés). L'introduction de la conversion de type explicite détruirait cette structuration et anéantirait nos efforts en vue d'une parallélisation efficace. Nous avons donc pris le parti de conserver cette structuration et d'interdire les conversions de type.

Les conversions explicites ne sont pas traitées. Il reste les conversions implicites (comme dans `int i = 'A' ;`). Ces conversions ne nous gênent pas car elles ne sont effectuées qu'entre types de base, et n'affectent pas la structuration dont nous avons parlé.

Notons que le langage C traditionnel n'impose pas la vérification des types des arguments des fonctions. Si nous traitions les fonctions, il faudrait faire cette vérification pour garder intacte la structuration de la mémoire.

Hormis ces deux exceptions, toutes les expressions, y compris les plus obscures, sont reconnues et traitées, comme par exemple :

```
*a++ = i % (n=3) && ((j || ! k) ? *(b=tab)[c+=5] : 5[tab],*y),z ;
```
[1]

Nous laissons au lecteur le soin de vérifier sa compréhension des expressions du langage C sur cet exemple, beau par sa complexité inutile. Il pourra en particulier constater que tous les opérateurs disponibles n'ont pas été utilisés...

2.1.4 Instructions

Les instructions sont celles du langage C, à l'exception de :

- `do ... while`.
- `goto`, `break`, `continue` et `return` ;
- `switch` et `case` ;

Les boucles du type `do ... while` n'apportent rien de plus que les boucles `while` classiques. En fait, les premières pourraient être traitées et être remplacées par les secondes, comme dans :

```
do
{
      x++ ;
} while (x < n) ;
```

qui pourrait être remplacé par :

```
x++ ;
while (x < n) {
      x++ ;
}
```

[1]Ce n'est pas une coquille, vous avez bien lu `5[tab]`...

Les instructions de rupture de séquence `goto` et `break` ne sont pas reconnues. Les instructions `continue` pourraient être reconnues et générer un test regroupant le reste de la boucle, comme dans :

```
while (x < n)
{
      x++ ;
      if (x % 2 == 0)
            continue ;
      y++ ;
}
```

qui pourrait être remplacé par :

```
while (x < n)
{
      x++ ;
      if (! (x % 2 == 0))
      {
            y++ ;
      }
}
```

L'instruction `return` serait utile si nous traitions l'appel de fonctions. Mais, étant donné que ce n'est pas le cas, et que cette instruction provoque une rupture de séquence, elle n'est pas reconnue.

Les sélections multiples `switch` et `case` sont une des constructions les plus bizarres du langage C. Dans les cas habituels, on peut transformer un `switch` en une succession de tests `if`. Mais la norme permet les pires égarements, illustrés par l'exemple suivant :

```
switch (c)
{
      if (i == 0)
      {
            case 1 :
                  i = 1 ;
                  if (j == 0)
                        default: break ;
      }
      case 2 :
            ...
}
```

Face à de telles aberrations qui, de toutes manières, sont difficilement parallélisables, nous avons pris le parti d'ignorer cette construction. Notons qu'elle est d'ailleurs plus du ressort d'un restructureur analogue à celui décrit dans [Amma88] que d'un paralléliseur.

2.1.5 Conclusion

Pour résumer et synthétiser, le langage d'entrée est le langage C, avec les traits suivants :

- tous les types de données, sauf les champs de bits, les fonctions, les structures et les unions sont admis ;
- toutes les expressions sauf celles contenant des conversions de type explicite ou des appels à des fonctions sont admis ;
- les constructions `while`, `for`, et `if` sont admises.

2.2 Langage d'analyse

Le sous-ensemble du langage C que nous avons défini dans la section précédente dispose d'un certain nombre de traits qui le rendent très difficile à analyser dans le cadre de la parallélisation. En particulier, les expressions gardent toute la complexité du langage C, non pas par le nombre d'opérateurs qui est conséquent, mais par leurs nombreux effets de bord.

Face à cette complexité, trois solutions sont envisageables :

1. Une solution simple serait de restreindre le langage d'entrée et d'en retirer tous les traits qui gênent l'analyse. Cette approche est traditionnelle et employée par de nombreux outils de parallélisation : peu de paralléliseurs de Fortran traitent les fameux `EQUIVALENCE` par exemple. Dans notre cas, toutefois, retirer tous les traits contraignants n'est pas une bonne solution : nous avons déjà imposé certaines contraintes (pas d'appels de fonctions, pas de structures de données évoluées, etc.). En retirer encore, et notamment tout ce qui fait la richesse du langage C c'est à dire les effets de bord dans les expressions, et le langage sera tellement appauvri que nous ne pourrons plus parler de langage C, mais tout juste de "Fortran avec des pointeurs" ;
2. Une solution complexe serait d'accepter le langage tel que nous l'avons défini, avec ses restrictions raisonnables, et d'essayer de le traiter dans les phases ultérieures. Toutefois, comme nous l'avons mentionné, les effets de bord sont complexes, et il faudrait vraisemblablement appliquer à chaque terme d'une expression des techniques et des représentations évoluées. La complexité résultant de cet effort ne nous a pas paru adaptée au problème, et nous avons préféré une troisième solution ;
3. La solution envisagée repose sur deux constatations :

- le langage ne doit pas avoir trop de restrictions,
- le langage ne doit pas être trop complexe.

Il est évident que ces deux constatations ne font pas référence au même langage !

Si nous conservons notre langage tel que décrit précédemment, le langage servant de base à l'analyse est différent. Une série de transformations d'ordre syntaxique vont nous permettre de passer de l'un à l'autre. On simplifie le langage tout en gardant son pouvoir expressif.

Il est bien sûr évident qu'on ne peut passer de n'importe quel langage à n'importe quel autre (surtout plus simple) par une série de transformations d'ordre syntaxique. La méthode que nous proposons n'est donc pas généralisable à tous les problèmes.

Nous décrivons ici le langage servant de base aux phases d'analyse ultérieures. Nous le définissons non plus en tant que sous-ensemble du langage C, mais comme un nouveau langage, tant les restrictions sont sévères. Ceci nous sert de référence pour les chapitres suivants.

Nous étudions dans la section consacrée à l'implémentation comment passer du langage réel à notre nouveau langage.

2.2.1 Types de données

Les types de données sont en tout point similaires aux types de données du langage original. Les restrictions que nous lui avons apportées sont identiques. Pour résumer, sont autorisés, à l'exclusion de tout type de données :

- les types de base ;
- les pointeurs ;
- les tableaux.

2.2.2 Fonctions et procédures

Comme nous l'avons mentionné, les fonctions ne sont qu'un cadre pour l'analyse en donnant un nom à un segment de programme. On peut donc écrire des fonctions, mais pas les appeler...

Dans ce contexte, la différence entre les notions de variables locales ou globales n'a plus de sens. Pour simplifier, nous supposons qu'il n'existe qu'un seul type de variable : la variable globale.

Il faut noter, bien que cela ne soit pas implémenté, que la méthode décrite dans cette étude se généralise aisément au cas de l'analyse inter-procédurale.

2.2.3 Expressions

C'est dans les expressions que les restrictions sont les plus marquées. Il s'agit d'avoir un jeu d'opérateurs permettant l'écriture d'expressions sans aucun effet de bord.

Seuls les opérateurs suivants sont admis :

Opérateurs unaires	
Opérateur	**Signification**
`+`	plus unaire
`-`	négation
`!`	négation logique
`~`	négation bit à bit
`&`	adresse de
`*`	objet pointé par

Opérateurs binaires	
Opérateur	**Signification**
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	reste de division entière
`&`	et bit à bit
`|`	ou bit à bit
`^`	ou exclusif bit à bit
`<<`	décalage à gauche
`>>`	décalage à droite
`==`	égalité
`!=`	différence
`<`	infériorité stricte
`<=`	infériorité large
`>`	supériorité stricte
`>=`	supériorité large

La priorité de ces opérateurs est celle du langage C.

Les opérateurs ne figurant pas dans cette liste sont :

- l'affectation : c'est maintenant une instruction du langage, ce qui interdit les affectations multiples telles que `x = y = 0`. Ces affectations multiples peuvent être remplacées par deux instructions d'affectation sans aucune difficulté.
- les opérateurs à effet de bord (`++`, `+=` etc.) : ces opérateurs sont des facilités d'écriture et sont aisément remplacés par une séquence de code équivalente.
- les opérateurs déclenchant une rupture de séquence implicite (`&&`, `||` et l'opérateur ternaire `?:`) : en effet, l'évaluation de l'expression `x||y` entraîne l'évaluation de `y` si et seulement si l'évaluation de `x` donne la valeur 0.
- l'opérateur d'accès à un tableau : l'accès aux cases d'un tableau est remplacé par une lecture via un pointeur puisque les deux écritures sont équivalentes et interchangeables en langage C. Par exemple, `tab[5]` est équivalent à `*(tab+5)`.
- l'opérateur "exotique" dont le support n'apporte rien : l'opérateur de séquencement virgule.

Les expressions de notre langage sont donc épurées de tout effet de bord (modification de mémoire ou rupture de séquence). Nous voyons au paragraphe 2.3.4 comment passer automatiquement des expressions d'un langage tel que C aux expressions de notre mini-langage, incomparablement plus faciles à étudier...

La syntaxe des expressions est récapitulée dans la figure 2.1. Notons l'élément non terminal *expression-logique* utile dans les tests des alternatives et des itérations.

2.2.4 Instructions

Les instructions sont réduites à :

- la séquence ;
- l'affectation ;
- l'alternative ;
- l'itération.

La syntaxe complète est récapitulée en figure 2.2.

La séquence d'instructions permet de grouper plusieurs instructions en une seule.

L'affectation n'est donc plus un opérateur des expressions de notre mini-langage, mais une instruction à part entière. Le membre de gauche doit être la référence à une case mémoire

```
expression          ::= ( expression )
                      | constante
                      | identificateur
                      | operateur-unaire expression
                      | expression operateur-binaire expression

operateur-unaire    ::= +
                      | -
                      | !
                      | ~
                      | &
                      | *

operateur-binaire   ::= operateur-relation
                      | +
                      | -
                      | *
                      | /
                      | %
                      | &
                      | |
                      | ^
                      | <<
                      | >>

operateur-relation  ::= ==
                      | !=
                      | <
                      | <=
                      | >
                      | >=

expression-logique  ::= expression operateur-relation expression
```

Figure 2.1: Syntaxe des expressions

instruction	::= { instruction* }	
	| affectation	
	| alternative	
	| iteration	
affectation	::= expression `=` expression ;	
alternative	::= **if** (expression-logique) instruction	
	| **if** (expression-logique) instruction **else** instruction	
iteration	::= **while** (expression-logique) instruction	

Figure 2.2: Syntaxe des instructions

accessible (ce qu'on nomme en langage C un *lhs*, ou *left-hand side*). Le membre de droite est une expression sans restriction.

Notons que l'alternative et l'itération nécessitent un test de la forme *expression-logique*. Ceci ne restreint pas l'éventail des possibilités, car il suffit de réécrire des tests comme `if (x)` en `if (x != 0)` pour se conformer à la syntaxe. L'avantage est l'homogénéité induite par cette modification dans la mesure où nous pouvons considérer, par la suite, la présence d'objets de type booléen.

2.2.5 Conclusion

Le langage d'analyse est essentiellement le langage d'entrée sans effets de bord, c'est à dire sans modification de mémoire et sans rupture de séquence à l'intérieur des expressions. Pour cela, nous avons dissocié l'affectation des expressions et nous l'avons rangée parmi les instructions. Les autres opérateurs générant des effets de bord ont été supprimés du langage.

L'autre point caractérisant le langage d'analyse est l'unification des différentes classes de variables (suivant les classes d'allocation du langage C) en une seule classe, éliminant ainsi les arcanes du langage d'origine. Ceci est rendu possible par la suppression des appels de fonctions.

2.3 Implémentation

Ce sous-chapitre décrit l'implémentation effectivement réalisée. Nous commençons par situer ce qui a été réalisé par rapport au schéma simplifié de la figure 1.2 (page 31). Nous décrivons ensuite la représentation intermédiaire, liant tous les éléments que nous présentons dans cette étude, puis expliquons la structure générale des deux composants décrits dans ce chapitre.

2.3.1 Introduction

La figure 1.2 (voir page 31) présente en pointillés les éléments que nous proposons de greffer à un paralléliseur existant. Comment se situent les réalisations de ce chapitre ? Détaillons la boîte nommée *analyseur syntaxique* : nous l'avons scindée en deux parties :

- l'*analyseur syntaxique* proprement dit, traduisant le programme en langage C dans une forme intermédiaire (analogue au langage d'entrée) ;
- le *normaliseur de programme*, prenant en entrée cette forme intermédiaire, supprimant les effets de bord dans les expressions, et donnant en sortie une nouvelle forme intermédiaire (analogue au langage d'analyse). Cette nouvelle forme est très proche de la précédente, la différence étant essentiellement un retrait des opérateurs à effets de bord.

Détaillons à présent la forme intermédiaire.

2.3.2 Représentation intermédiaire

Un paralléliseur est, en quelque sorte, un compilateur un peu spécial : au lieu d'optimiser pour un processeur classique, il optimise pour une machine vectorielle ou multi-processeurs. Comme tout compilateur, il utilise une forme intermédiaire pour représenter le programme afin de l'analyser. Cette section a pour but de familiariser le lecteur avec la représentation intermédiaire utilisée dans les exemples de ce chapitre et des suivants.

Niveau de la représentation

Chaque compilateur dispose de sa propre représentation intermédiaire : il n'y a pas pour cela de consensus. Mais le niveau de la représentation est différent pour un compilateur devant générer immédiatement du code efficace et pour un paralléliseur devant procéder à une analyse sémantique du code. Examinons brièvement deux exemples de formes intermédiaires :

- Le compilateur GNU [Stal88] utilise une forme nommée RTL (pour *Register Transfer Language*). Cette forme ressemble à un assembleur portable avec une syntaxe "à la Lisp". Un exemple de ce curieux mélange est la représentation de l'expression `x = y * 3 ;` :

```
(insn 6 4 7
      (set (reg:SI 56)
           (ashift:SI (mem:SI (symbol_ref:SI ("y")))
                      (const_int 1)))
```

```
        -1 (nil) (nil))
(insn 7 6 8
        (set (mem:SI (symbol_ref:SI ("x")))
             (plus:SI (reg:SI 56)
                      (mem:SI (symbol_ref:SI ("y")))))
        -1 (nil) (nil))
```

La première instruction (numérotée 6) est chaînée aux instructions précédente (numérotée 4) et suivante (numéroté 7). Son effet est d'affecter au registre numéro 56, accédé comme un entier simple (*Single Integer*) la valeur calculée par le décalage d'un bit à gauche de la valeur lue dans la mémoire à l'adresse référencée par le symbole `y`. La deuxième instruction affecte à la case mémoire référencée par le symbole `x` la somme de la valeur calculée précédemment (dans le registre numéro 56) et de la valeur lue en mémoire à l'adresse référencée par le symbole `y`.

Les trois derniers champs de chaque instruction sont plutôt des images de la structure de données utilisée par le compilateur pour ses optimisations (quelle instruction ou suite d'instructions se rapproche de ce calcul, liste des effets sur les registres et liste des instructions affectant ce registre)

Cette forme intermédiaire est intéressante. Cependant, bien que de haut niveau pour un compilateur classique, elle reste de trop bas niveau pour servir de support à une analyse sémantique.

- Le paralléliseur *PAF* utilise une forme intermédiaire [Amma86] de beaucoup plus haut niveau. En effet, il s'agit d'une traduction de l'arbre de syntaxe du programme en un dialecte de Lisp [Chai87]. Par exemple, le programme :

```
        subroutine exemple (x, n)
        real x(n,n)
        integer i
        do 10 i=1,n
              x(i, 1) = x(1,n)
10      continue
        return
        end
```

est traduit en :

```
(df #:_:exemple (#:exemple:x #:exemple:n)
    (tag sortie
         (prog ()
               (integer #:exemple:n)
               (integer #:exemple:i)
               (real #:exemple:x (#:exemple:n #:exemple:n))
               (for (#:exemple:i 1 1 #:exemple:n)
                    (prog ()
                          (aset (eval #:exemple:x)
                                #:exemple:i 1
                                (aref (eval #:exemple:x)
                                      1 #:exemple:n))
```

```
                  _10 () ))
        (exit sortie)) ))
```

Cette forme intermédiaire est intéressante, mais souffre du défaut principal de n'être absolument pas adaptée aux contraintes du langage C. En particulier, les accès mémoire sont très simplifiés, à l'image des accès mémoire de Fortran.

Notons que cette forme devait, pendant les premiers développements de PAF, permettre l'exécution directe en Le_Lisp à des fins de vérification, ce qui explique certains traits tels que `df`, `tag`, utilisation de `eval` pour accéder aux tableaux, etc.

Il faut donc trouver une forme intermédiaire de haut niveau, tout en permettant d'exprimer toutes les subtilités du langage C, comme les accès mémoire. Toutefois, nous ne devons pas tomber dans l'excès d'une représentation explicitant des détails non utiles pour la parallélisation (comme l'utilisation de registres, même virtuels).

Principes

La forme intermédiaire est un arbre représenté par une suite d'expressions Lisp. Il y a une expression par objet C (variable globale, fonction, modèle de structure ou d'union) déclaré au niveau global.

La représentation que nous proposons est décrite en détail dans [Davi89], et nous ne faisons ici que présenter ses principales caractéristiques. On trouvera en annexe une spécification de cette représentation intermédiaire. Notons qu'elle est plus générale que les langages décrits dans ce chapitre, car elle a été conçue pour être réutilisable dans des extensions futures de notre travail.

Les paragraphes suivants sont consacrés à la définition des types, caractéristique très importante de notre étude, et leur utilisation dans les déclarations, puis aux structures de contrôle et enfin aux expressions.

Types et déclarations

Les types sont un élément fondamental du langage C. Les chapitres ultérieurs utilisant les types pour affiner l'analyse, il est très important de trouver une représentation simple. Notons que la représentation des types n'est pas un élément essentiel des paralléliseurs automatiques de Fortran.

La représentation que nous proposons est basée sur la notion de constructeur. Nous pouvons dissocier :

- les types de base (types entier `int`, `unsigned long int`, `signed char`, etc.), types flottants `float`, `double` et `long double`) et le type nul `void` ;

- les tableaux d'objets, les pointeurs sur les objets et les qualificateurs.

Une déclaration en C consiste à spécifier un type de base et une méthode d'accès au type par les constructeurs. Ainsi, la déclaration classique `char *const argv []` doit être lue comme : "`argv` est un tableau de taille inconnue de pointeurs constants sur des objets de type `char`". Explicitons ce chemin par une liste : `((adecl nil) (const) (pdecl) (simple-type char))`.

Le type de base peut éventuellement être une structure ou une union. Dans ce cas, le type de base est remplacé par une référence à un modèle de structure. Par exemple : `struct s *v` déclare une variable `v` comme étant un pointeur sur une structure conforme au modèle `s`. Ceci nous donne la liste : `((pdecl) (struct s))`.

Le problème se complique lorsqu'on traite les fonctions. En effet, celles-ci ont un type, composé du type de retour et des types des paramètres. Nos chemins d'accès au type de base, linéaires, deviennent donc arborescents. Par exemple, le type de la fonction `main` déclarée par :

```
void main (const int argc, const char *const argv [])
```

est l'arbre :

```
(fun ((void))
     (proto (vdecl _argc
                   ((const) (simple-type int))
                   auto nil)
            (vdecl _argv
                   ((adecl nil) (const) (pdecl) (const) (simple-type char))
                   auto nil) ))
```

Dans ce type, `fun` agit comme `simple-type`, c'est à dire que nous pouvons décrire des tableaux de pointeurs sur des fonctions de la même manière que des tableaux de pointeurs sur des entiers par exemple. Le mot-clef `proto` indique la présence d'un prototype conforme à la norme ANSI, et `vdecl` indique une déclaration de paramètre. Si les noms des paramètres ne sont pas fournis, le nom est remplacé par `nil`.

Il peut y avoir quatre sortes de déclarations :

- `sdecl` : déclaration de modèle de structure
- `udecl` : déclaration de modèle d'union
- `vdecl` : déclaration de variable
- `fdecl` : déclaration de fonction

Les deux premières sont simplement constituées par le nom du modèle et par la liste des champs (c'est à dire des associations formées par le nom du champ et par son type).

Les déclarations de variable sont constituées par le nom de la variable, son type, sa classe d'allocation et sa valeur initiale s'il y a lieu. La classe d'allocation (`static`, `extern`, `register` ou `auto`[2]) n'est pas liée au type, elle spécifie la zone mémoire dans laquelle la variable est rangée, ainsi que sa visibilité dans l'environnement. Les classes d'allocation sont simplifiées dans la représentation intermédiaire, les variables globales pouvant être `static` (locales au fichier) ou `global` (variables exportées dans d'autres fichiers), et les variables locales aux fonctions pouvant être `auto` (variables déclarées dans la pile, obligatoire pour les paramètres de fonction), `static` (variables placées dans la zone des données communes). La classe `register`, dont la prise en compte est facultative [ANSI88], n'est pas utilisée. Par exemple, la déclaration de variable :

```
typedef unsigned int uint ;
static const uint x = 5 ;
```

est traduite en :

```
(vdecl _x
       ((const) (simple-type unsigned int))
       static
       5)
```

Les déclarations de fonction ne diffèrent des déclarations de variable que par le mot-clef, le type (qui doit forcément commencer par `fun`), et le corps qui remplace la valeur initiale. La classe d'allocation des fonctions ne fait que spécifier la visibilité, de manière analogue aux variables globales. Ainsi, la déclaration de la fonction `main` que nous avons vue plus haut est traduite en :

(fdecl _main
       type vu plus haut
       global
       corps ou nil)

Notons que les variables locales (même celles qui sont internes à des blocs) sont toutes placées en tête du corps de la fonction. Un dispositif de renommage permet d'éviter des collisions dans l'espace des noms. De plus, toutes les variables locales sont préfixées par le nom de la fonction, ce qui permet de traiter toutes les variables dans un seul espace de noms unifié.

[2] `typedef`, souvent référencé comme une classe d'allocation, ne fait que donner un nom à un type. Les déclarations `typedef` ne sont donc pas reproduites dans la forme intermédiaire, et les types contenant une référence à un `typedef` sont expansés.

Instructions

Les structures de contrôle sont l'image directe des structures correspondantes en C, mise à part la boucle `for`. Par exemple, le segment de code en langage C :

> `while` (*expression*)
> *corps*

est représenté par :

> (`while` *expression corps*)

Le bloc est représenté par le classique `prog`. Rappelons que les variables locales à un bloc sont rejetées en tête de la fonction courante.

La traduction des boucles `for` se fait par l'intermédiaire de la très classique transformation :

`for` (e_1 ; e_2 ; e_3)
    inst
	$\Longleftrightarrow$	e_1 ;
		`while` (e_2)
		{
		    inst
		    e_3 ;
		}

Notons la présence d'une boucle *do* dans la forme intermédiaire. Celle-ci est nécessaire pour la reconnaissance des boucles de ce type décrite au chapitre 4.

L'affectation est, dans le langage d'analyse, une instruction et non plus un opérateur. L'affectation e_1 `=` e_2 s'écrit en représentation intermédiaire : (`store` *type* e_1 e_2), où *type* est le type (et donc la taille) de la case mémoire affectée. L'affectation étant à l'origine un opérateur, nous le décrivons de manière plus précise dans le paragraphe suivant 2.3.2.

Nous renvoyons le lecteur à la description complète pour les structures de contrôle non présentes dans cette rapide description. Rappelons toutefois que seuls le bloc, les boucles `while` et les tests `if` sont présents dans le langage d'analyse.

Expressions

Les premiers constituants des expressions sont les constantes. Les constantes entières peuvent être spécifiées de quatre manières différentes en C (par exemple `65`, `'A'`, `0x41` ou `0101`), mais seule la forme décimale est utilisée dans la représentation intermédiaire.

Le point intéressant de notre représentation intermédiaire est la représentation des accès mémoire (lecture et écriture). Il faut en effet pouvoir transcrire toute les possibilités des accès du langage C. Pour comprendre les principes, prenons comme exemple les deux lignes suivantes :

```
x = 3 ;
y = x ;
```

Ces deux utilisations du symbole `x` sont très différentes : la première place 3 à l'adresse identifiée par le symbole `x`, alors que la seconde lit ce qu'il y a à cette même adresse. Traditionnellement, on parle de *lhs* (*left hand side*) et de *rhs* (*right hand side*) pour différencier ces deux types d'utilisation. Notre représentation intermédiaire tient compte de cette différence, tout en supprimant les cas particuliers qu'elle aurait pu entraîner : *un identificateur dans cette représentation s'évalue comme son adresse.*

Les identificateurs peuvent représenter des objets dans la pile (variables automatiques), dans la zone de données communes (variables globales) ou dans le code (fonctions). Comment allons-nous les utiliser ? Trois cas peuvent se présenter :

- la référence, ou lecture d'une case mémoire ; la syntaxe est :
  `(fetch` *type adresse*`)`
- l'affectation, ou écriture dans une case mémoire ; la syntaxe est :
  `(store` *type adresse expression*`)`
- l'appel de fonctions (bien que ce cas ne se puisse se présenter dans le langage d'analyse, nous le prenons en compte pour les travaux futurs) ; la syntaxe est :
  `(fcall` *type adresse liste-d'arguments*`)`

Dans les trois cas, *adresse* est l'adresse où se fait la référence, c'est à dire dans la plupart des cas l'identificateur puisqu'il s'évalue en son adresse. Le *type* est un type comme ceux que nous avons déjà vus. Il spécifie la manière dont se fait l'accès. Par exemple :

```
(fetch ((simple-type int)) x)
(fetch ((simple-type char)) x)
```

sont deux accès aux informations stockées à l'adresse x, différant par la taille de ce qui est lu. Pour l'appel de fonction, le *type* est le type du résultat.

L'opérateur `fetch` s'évalue en le contenu de la case mémoire adressée et l'opérateur `fcall` en la valeur retournée par la fonction appelée. `store` est maintenant une instruction.

Prenons un exemple complet (voir figure 2.3). La traduction en représentation intermédiaire de cet exemple est (déclarations non comprises) :

```
int i, *p ;

i = *p ;
p = &i ;
*p = 5 ;
```

Figure 2.3: Exemple de programme

```
(prog (store ((simple-type int))
             _i
             (fetch ((simple-type int))
                    (fetch ((pdecl) (simple-type int))
                           _p)))
      (store ((pdecl) (simple-type int))
             _p
             _i)
      (store ((simple-type int))
             (fetch ((pdecl) (simple-type int))
                    _p)
             5))
```

L'adresse utilisée lors de la première affectation est l'adresse de la variable `i`. En revanche, la valeur affectée nécessite une lecture pour la valeur du pointeur `p` (à l'adresse de la variable `p`, puis une deuxième lecture pour la valeur entière. La deuxième affectation est beaucoup plus simple puisqu'il s'agit d'affecter l'adresse de la variable `i` à la variable `p`. Il n'y a donc aucune lecture en mémoire. En revanche, la troisième affectation nécessite une lecture pour avoir l'adresse où stocker la valeur entière.

L'accès aux cases d'un tableau se fait naturellement avec l'arithmétique sur les pointeurs. En revanche, l'accès aux champs de structure nécessite l'introduction d'un opérateur spécial `sfield` pour représenter le déplacement du champ accédé par rapport au premier champ.

La liste des opérateurs est donnée dans la grammaire fournie en annexe. Notons que cette liste est très complète, prenant en compte des constructions dont nous n'avons pas besoin dans notre étude. Rappelons que cette représentation doit pouvoir supporter des travaux ultérieurs.

Conclusion

La représentation intermédiaire que nous avons brièvement introduite permet de représenter un langage plus complet que le langage d'analyse. Ceci permettra des travaux futurs sans revenir sur la définition actuelle de cette forme.

Le point non traité dans cette forme est la représentation des champs de bits. On peut aussi noter l'absence de précision en ce qui concerne la représentation des constantes flottantes.

La simplicité de la représentation des accès à la mémoire assure la souplesse et l'adaptation à tous les langages. La représentation est générale et peut servir de base à un paralléliseur multi-langages admettant au minimum C et Fortran.

2.3.3 Analyseur syntaxique

L'analyseur syntaxique est rédigé en langage C, en utilisant les outils [SF85] `yacc` [John75] [John79] et `lex` [Lesk75]. Il totalise près de 15 000 lignes fortement commentées.

Il permet de construire l'arbre de syntaxe du programme qui lui est soumis, puis d'en sortir une représentation sous forme de listes Lisp. Cette représentation contient des opérateurs ne figurant pas dans la description précédente car les effets de bord des expressions n'ont pas encore été supprimés, et `store` est donc encore un opérateur.

Sans rentrer dans les détails de l'implémentation, signalons que la structure de données interne pour représenter l'arbre, ainsi que les opérations qui s'y rapportent, ressemblent autant que possible à l'implémentation existant dans des interprètes Lisp [CDH84].

Bien que très complet, notre analyseur ne gère pas encore les classes d'allocation `global` et les doubles déclarations, courantes en C, ne sont pas reconnues.

Exemple

Prenons l'exemple de la figure 2.4 extrait de [KR88]. Après analyse et *pretty-print*, cela donne :

```
(fdecl _strlen ((fun ((simple-type int)) (noproto))) nil nil)

(fdecl _reverse
  ((fun ((void))
        (proto (vdecl _reverse-s ((adecl nil) (simple-type char)) nil nil))))
  nil
  (prog (vdecl _reverse-7 ((pdecl) (simple-type int)) nil nil)
        (vdecl _reverse-8 ((simple-type int)) nil nil)
        (vdecl _reverse-9 ((pdecl) (simple-type int)) nil nil)
        (vdecl _reverse-10 ((simple-type int)) nil nil)
        (vdecl _reverse-1-j ((simple-type int)) nil nil)
```

```
void reverse (char s [])
{
      int c, i, j ;

      for (i = 0, j = strlen (s)-1 ; i < j ; i++, j--)
            c = s [i], s [i] = s [j], s [j] = c ;
}
```

Figure 2.4: Exemple de programme

```
(vdecl _reverse-1-i ((simple-type int)) nil nil)
(vdecl _reverse-1-c ((simple-type int)) nil nil)
(comma (store ((simple-type int)) _reverse-1-i 0)
       (store ((simple-type int)) _reverse-1-j
              (sub (fcall ((simple-type int))
                          _strlen
                          ((fetch ((adecl nil) (simple-type char)) _reverse-s)))
                   1)))
(while (lt (fetch ((simple-type int)) _reverse-1-i)
           (fetch ((simple-type int)) _reverse-1-j))
       (prog (comma (comma (store ((simple-type int))
                                  _reverse-1-c
                                  (fetch ((simple-type char))
                                         (add (fetch ((adecl nil) (simple-type char))
                                                     _reverse-s)
                                              (mul (fetch ((simple-type int))
                                                          _reverse-1-i)
                                                   1))))
                           (store ((simple-type char))
                                  (add (fetch ((adecl nil) (simple-type char))
                                              _reverse-s)
                                       (mul (fetch ((simple-type int))
                                                   _reverse-1-i)
                                            1))
                                  (fetch ((simple-type char))
                                         (add (fetch ((adecl nil) (simple-type char))
                                                     _reverse-s)
                                              (mul (fetch ((simple-type int))
                                                          _reverse-1-j)
                                                   1)))))
                    (store ((simple-type char))
                           (add (fetch ((adecl nil) (simple-type char))
                                       _reverse-s)
                                (mul (fetch ((simple-type int))
                                            _reverse-1-j)
                                     1))
                           (fetch ((simple-type int))
                                  _reverse-1-c)))
             (comma (comma (store ((pdecl) (simple-type int))
```

```
                  _reverse-7
                  _reverse-1-i)
           (store ((simple-type int))
                  _reverse-8
                  (fetch ((simple-type int))
                         (fetch ((pdecl) (simple-type int))
                                _reverse-7)))
           (store ((simple-type int))
                  (fetch ((pdecl) (simple-type int))
                         _reverse-7)
                  (add (fetch ((simple-type int))
                              _reverse-8)
                       1))
           (fetch ((simple-type int)) _reverse-8))
    (comma (store ((pdecl) (simple-type int))
                  _reverse-9
                  _reverse-1-j)
           (store ((simple-type int))
                  _reverse-10
                  (fetch ((simple-type int))
                         (fetch ((pdecl) (simple-type int))
                                _reverse-9)))
           (store ((simple-type int))
                  (fetch ((pdecl) (simple-type int))
                         _reverse-9)
                  (sub (fetch ((simple-type int))
                              _reverse-10)
                       1))
           (fetch ((simple-type int)) _reverse-10)))))))
```

Nous pouvons noter sur cet exemple que les opérateurs `++`, `--` ont été transformés par notre analyseur syntaxique. Cette transformation devrait logiquement être traitée par le normaliseur d'expressions (voir 2.3.4), mais a été placée ici pour améliorer les performances de cette modification très fréquente.

Performances

Le temps pris par l'analyseur syntaxique est négligeable par rapport au temps pris par les autres passes. Néanmoins, nous avons construit un programme composé de 10 fonctions totalisant chacune 100 affectations, toutes égales :

```
void f1 (void)
{
      int x1, x2, *p1, *p2 ;

      *p1 = x1 + x2 + *p1 + *p2 ;
         ⋮
      *p1 = x1 + x2 + *p1 + *p2 ;
}
```

Pour établir une comparaison équitable, nous avons compilé ce programme de 1000 lignes utiles avec notre analyseur syntaxique et le préprocesseur natif d'une part, et le compilateur natif sans assemblage du fichier généré d'autre part. Les temps mesurés sont les temps cpu en mode "usager" sur une station de travail basée sur un processeur MC 68040 à 25 MHz :

- compilateur natif : 10,70 secondes ;
- analyseur syntaxique : 6,60 secondes.

Même si les tâches réalisées par le compilateur natif sont plus importantes, notre analyseur syntaxique soutient favorablement la comparaison.

2.3.4 Normaliseur

Le normaliseur, à partir de la traduction opérée par l'analyseur syntaxique, supprime les effets de bord des expressions. Tout d'abord, définissons ce que sont les effets de bord :

- les opérateurs qui modifient la mémoire : en effet, nous ne voulons de modifications de mémoire que dans l'affectation qui doit devenir une instruction du langage et plus opérateur ;
- les opérateurs qui provoquent une rupture de séquence : par exemple, dans l'expression `a && b`, `b` n'est évalué que si l'évaluation de `a` donne un résultat non nul. De même, l'opérateur ternaire `?:` a pour vocation de provoquer une rupture de séquence ;
- les appels de fonction (bien que ce ne soit pas dans le langage d'analyse) : ils provoquent un déroutement, et peuvent également provoquer une modification de la mémoire ;
- l'opérateur *virgule*, bien que ne générant aucun effet de bord, doit être supprimé : en effet, sa présence ne se justifie plus lorsque tous les effets de bord ont été supprimés.

La démarche intuitive sera donc : l'opérateur `store` étant une expression, il peut être n'importe où dans l'arbre de syntaxe. Il faut donc le faire remonter en position d'instruction, c'est à dire le placer comme fils direct d'un `prog`. Cette remontée doit être faite à l'aide d'une série de transformations ne modifiant pas la sémantique du programme, comme :

```
                                  tmp = *r++,
*p++ = *q++ = *r++        ⟺      *q++ = tmp,
                                  *p++ = tmp
```

où `tmp` est une variable temporaire créée pour l'occasion, nécessaire pour garantir une seule évaluation des deux adresses affectées. L'opérateur *virgule* est utilisé dans la transformation, car cette expression peut elle-même être une partie d'une expression plus importante.

Les opérateurs provoquant une rupture de séquence doivent être convertis avec les instructions du langage. Ainsi, les transformations doivent faire intervenir des instructions `if`, comme par exemple :

```
a = (b && c) ;
⇕
a = (b ? (c ? 1 : 0) : 0) ;
⇕
b ? (c ? (a = 1) : (a = 0)) : (a = 0) ;
⇕
if (b)
{
     if (c)
          a = 1 ;
     else a = 0 ;
}
else a = 0 ;
```

Avec le programme sous forme d'un arbre, il suffit d'utiliser un algorithme de filtrage [HO82] pour opérer ces transformations. Une règle du moteur est constituée de trois parties : le motif recherché, le motif de remplacement et les variables temporaires à créer éventuellement.

Le normaliseur est écrit en Common Lisp compilé [Stee84] et occupe environ 800 lignes, dont plus de la moitié est consacrée aux règles du moteur de filtrage.

Prenons comme exemple le programme de la figure 2.5 et passons le résultat de l'analyseur syntaxique dans notre normaliseur, ce qui nous conduit à la représentation intermédiaire suivante :

```
(fdecl _eb
  ((fun ((void)) (proto void)))
  nil
  (prog (vdecl _eb-1 ((pdecl) (simple-type int)) nil nil)
        (vdecl _eb-2 ((simple-type int)) nil nil)
        (vdecl _eb-3 ((pdecl) (simple-type int)) nil nil)
        (vdecl _eb-4 ((pdecl) (simple-type int)) nil nil)
        (vdecl _eb-5 ((simple-type int)) nil nil)
        (vdecl _eb-1-tab ((adecl 5) (simple-type int)) nil nil)
        (vdecl _eb-1-z ((simple-type int)) nil nil)
        (vdecl _eb-1-y ((simple-type int)) nil nil)
        (vdecl _eb-1-x ((simple-type int)) nil nil)
        (vdecl #:|$rmse-1021| ((simple-type int)) nil nil)
        (vdecl #:|$rmse-1020| ((simple-type int)) nil nil)
        (vdecl #:|$rmse-1019| ((simple-type int)) nil nil)
        (store ((simple-type int)) _eb-1-y 1)
        (store ((pdecl) (simple-type int)) _eb-1 _eb-1-y)
        (store ((simple-type int))
               _eb-2
               (fetch ((simple-type int))
```

```
void eb (void)
{
      int x, y, z ;
      int tab [5] ;

      y = 1 ;
      y++ ;
      while ((x = y) && ++z)
      {
            x || (x = 3), z++ [tab] ;
      }
}
```

Figure 2.5: Exemple de programme

```
            (fetch ((pdecl) (simple-type int)) _eb-1)))
(store ((simple-type int))
       (fetch ((pdecl) (simple-type int)) _eb-1)
       (add (fetch ((simple-type int)) _eb-2) 1))
(store ((simple-type int))
       #:|$rmse-1020|
       (fetch ((simple-type int)) _eb-1-y))
(store ((simple-type int))
       _eb-1-x
       (fetch ((simple-type int)) #:|$rmse-1020|))
(while 1
       (prog (if (fetch ((simple-type int)) #:|$rmse-1020|)
                 (prog (store ((pdecl) (simple-type int))
                              _eb-3
                              _eb-1-z)
                       (store ((simple-type int))
                              #:|$rmse-1019|
                              (add (fetch ((simple-type int))
                                          (fetch ((pdecl) (simple-type int)) _eb-3))
                                   1))
                       (store ((simple-type int))
                              (fetch ((pdecl) (simple-type int)) _eb-3)
                              (fetch ((simple-type int)) #:|$rmse-1019|))
                       (if (fetch ((simple-type int)) #:|$rmse-1019|)
                           (prog (break)))))
             (if (lognot (fetch ((simple-type int)) _eb-1-x))
                 (prog (store ((simple-type int)) #:|$rmse-1021| 3)
                       (store ((simple-type int))
                              _eb-1-x
                              (fetch ((simple-type int)) #:|$rmse-1021|))))
             (store ((pdecl) (simple-type int)) _eb-4 _eb-1-z)
             (store ((simple-type int))
```

```
                   _eb-5
                   (fetch ((simple-type int))
                          (fetch ((pdecl) (simple-type int)) _eb-4)))
            (store ((simple-type int))
                   (fetch ((pdecl) (simple-type int)) _eb-4)
                   (add (fetch ((simple-type int)) _eb-5) 1))
            (store ((simple-type int))
                   #:|$rmse-1020|
                   (fetch ((simple-type int)) _eb-1-y))
            (store ((simple-type int))
                   _eb-1-x
                   (fetch ((simple-type int)) #:|$rmse-1020|))))))
```

Si nous convertissons ceci en langage C pour faciliter la lecture, nous obtenons :

```
void eb (void)
{
      int *t1, t2, *t3, *t4, t5 ;
      int tab [5] ;
      int z, y, x ;
      int t1021, t1020, t1019 ;

      y = 1 ;

      t1 = &y ;
      t2 = *t1 ;
      *t1 = t2 + 1 ;

      t1020 = y ;
      x = t1020 ;

      while (1)
      {
            if (t1020)
            {
                  t3 = &z ;

                  t1019 = *t3 + 1 ;
                  *t3 = t1019 ;
                  if (t1019)
                        break ;
            }
            if (! x)
            {
                  t1021 = 3 ;
                  x = t1021 ;
            }
```

```
                t4 = &z ;
                t5 = *t4 ;
                *t4 = t5 + 1 ;

                t1020 = y ;
                x = t1020 ;

        }
}
```

On constate que chaque affectation est maintenant une instruction autonome, il ne reste plus aucun effet de bord dans ces expressions.

L'opération a duré 3,18 secondes.

2.4 Conclusion

Le langage admis en entrée de notre outil est donc un sous-ensemble très représentatif du langage C. Certaines constructions n'ont pas été jugées utiles, comme les boucles `do ... while` ou le choix multiple `switch`. D'autres traits ne sont pas admis, comme l'utilisation de fonctions ou de structures.

Les expressions avec effets de bord constituant un problème lors des phases ultérieures, nous avons défini un sous-ensemble du langage d'entrée, appelé langage d'analyse, dans lequel l'affectation n'est plus un opérateur, mais une instruction du langage. Les autres opérateurs provoquant des effets de bord ont été supprimés de ce nouveau langage, ce qui implique que les nouvelles expressions sont exemptes de tout effet de bord.

Un analyseur syntaxique pour le langage d'entrée, et un normaliseur d'expression pour transformer le langage d'entrée en langage d'analyse ont été réalisés. Ils utilisent une forme intermédiaire qui représente un programme par son arbre de syntaxe exprimé sous forme de liste Lisp. Cette représentation intermédiaire spécifie plus de constructions que les langages définis dans ce chapitre ne l'imposent, préparant ainsi les travaux ultérieurs à cette étude.

Les restrictions que nous avons imposées au langage C, ainsi que les modifications que nous avons fait subir aux expressions pour leur retirer les effets de bord, définissent un contexte pour l'analyse sémantique. En particulier, l'absence de *casts* et d'unions garantissent une structuration de la mémoire, dans laquelle le système de types n'est jamais contourné. Ceci va s'avérer très important dans les chapitres suivants.

Chapitre 3

Séquences linéaires

À ce stade de l'étude, nous disposons d'un programme exprimé avec le langage d'analyse. Il s'agit alors d'une suite d'affectations sans effet de bord, d'alternatives et de boucles. Nous avons donc un programme *presque* admissible par un paralléliseur.

Le dernier obstacle est la forme des boucles. Comme nous l'avons vu dans le chapitre précédent, nous ne disposons pas des boucles de type *do* de Fortran ou *for* de Pascal, les seules à présenter une forme adaptée à la parallélisation.

Le problème de la conversion des boucles de type *while* en boucles de type *do* n'étant pas un problème simple, nous proposons de le résoudre en deux temps :

1. dans un premier temps, étudier les séquences linéaires (affectation, alternative et suites composant ces deux éléments de base) dans le but de restreindre la portée des pointeurs ;
2. puis, dans un deuxième temps, analyser les boucles.

Dans ce chapitre, nous étudions les séquences linéaires. La démarche que nous proposons peut servir en outre d'exemple de détection de conflits d'accès mémoire en présence de pointeurs et de calcul d'adresses, détection qui s'avère nécessaire dans un paralléliseur pour la phase de construction du graphe de dépendances.

Le formalisme que nous adoptons ici pour décrire notre analyse repose sur la notion de fonction. Les notations que nous avons adoptées sont, certes, moins concises que celles du lambda-calcul [Seth89][BC88], mais ne nous plaçant pas dans le même contexte, nous avons préféré une notation plus adaptée et plus claire.

Dans les deux premières parties de ce chapitre, nous proposons quelques notations et définitions qui nous serviront dans le reste de notre étude. En particulier, nous introduisons la structuration de la mémoire, c'est à dire une partition de la mémoire suivant le système de types du langage C, utilisée dans la suite pour diminuer le nombre de gardes.

La troisième partie est consacrée à l'analyse de l'affectation, composant de base de notre étude. Nous amenons quelques notions utiles, comme l'interprétation des expressions dans le contexte d'une mémoire structurée. Puis, nous modélisons une affectation comme une transformation, c'est à dire une suite de fonctions modifiant une mémoire, sélectionnées par des gardes, c'est à dire des expressions logiques.

Les quatrième et cinquième parties introduisent la combinaison de transformation, c'est à dire la séquence d'instructions et l'alternative. L'analyse d'une séquence d'instructions est une composition de transformations dont nous donnons la forme. L'analyse de l'alternative est rendue triviale par la présence des gardes.

La sixième partie décrit la réalisation effectuée, et présente quelques exemples destinés à mettre en pratique les idées de ce chapitre et à évaluer l'implémentation.

3.1 Notations

Les noms d'ensembles seront en caractères calligraphiés (comme $\mathcal{E}$), et la cardinalité d'un ensemble $\mathcal{E}$ est notée $|\mathcal{E}|$.

Nous notons :

$$\begin{array}{rlll} f: & \mathcal{E} & \longrightarrow & \mathcal{E}' \\ & e & \longmapsto & f(e) = e' \end{array}$$

la fonction f, dont l'ensemble de départ est $\mathcal{E}$ et l'ensemble d'arrivée est $\mathcal{E}'$, faisant correspondre à $e \in \mathcal{E}$ la valeur $f(e) = e' \in \mathcal{E}'$.

Nous aurons besoin d'utiliser des fonctions sans donner leur nom, c'est à dire des fonctions anonymes. Nous notons :

$$\left(\begin{array}{lll} \mathcal{E} & \longrightarrow & \mathcal{E}' \\ e & \longmapsto & e' \end{array} \right) (a)$$

l'application de la fonction à la valeur a, élément de $\mathcal{E}$.

Nous notons $\times$ le produit cartésien et $\langle e_1, e_2, \ldots e_n \rangle$ un élément de $\mathcal{E}_1 \times \mathcal{E}_2 \times \ldots \mathcal{E}_n$.

Nous utilisons l'opérateur $+$ pour représenter la somme de deux ensembles, c'est à dire $\mathcal{E}_1 + \mathcal{E}_2 = \mathcal{E}_1 \cup \mathcal{E}_2$ si $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$.

Afin d'éviter de définir des fonctions partielles, c'est à dire des fonctions ayant un domaine de définition inclus strictement dans l'ensemble de départ, nous utilisons l'ensemble $\top$ (contenant un unique élément $\top$) comme constituant de l'ensemble d'arrivée. Ceci nous permet d'associer à chaque élément de l'ensemble de départ sa valeur ou $\top$ s'il ne fait pas partie du domaine de définition.

Nous utilisons ensuite le symbole $\cup$ pour représenter la fonction constituée de la famille de fonctions $f_i : \mathcal{E}_i \longrightarrow \mathcal{E}'_i$ telle que :

$$\begin{array}{rlll} f: & \mathcal{E}_1 + \ldots + \mathcal{E}_n & \longrightarrow & \mathcal{E}' + \ldots + \mathcal{E}'_n \\ & e & \longmapsto & f(e) = \begin{cases} f_1(e) & \text{si } e \in \mathcal{E}_1 \\ \vdots & \\ f_n(e) & \text{si } e \in \mathcal{E}_n \end{cases} \end{array}$$

Nous notons ensuite $P(\mathcal{E})$ l'ensemble des parties (ensembles d'éléments) de $\mathcal{E}$. Par exemple, si $\mathcal{E} = \{a, b, c, d, e\}$, $\{a\} \in P(\mathcal{E})$, $\{a, b\} \in P(\mathcal{E})$, $\{a, b, c\} \in P(\mathcal{E})$.

Nous utilisons les opérateurs logiques $\wedge$ (*et*), $\vee$ (*ou*) et $\neg$ (*non*).

3.2 Définitions

Les objets de base présentés ici définissent un modèle abstrait d'ordinateur.

Valeurs de base

Définition 3.1 *L'ensemble $\mathcal{E}$ est l'ensemble des valeurs entières*

L'ensemble $\mathcal{E}$ est l'ensemble des valeurs entières manipulables dans notre langage, ceci indépendamment des tailles physiques de la représentation des données dans un ordinateur. Cet ensemble, contrairement à l'ensemble des valeurs représentables en arithmétique entière dans les ordinateurs, n'est pas borné.

Nous pourrions de même définir l'ensemble $\mathcal{R}$ des valeurs réelles (et non flottantes) manipulables et en tenir compte dans la suite. Cependant, pour la clarté de l'exposé, nous n'utiliserons pas cet ensemble. Nous invitons le lecteur à garder en mémoire qu'il s'agit uniquement d'une simplification d'écriture, et que la généralisation de l'exposé à cet ensemble $\mathcal{R}$ est immédiate.

Définition 3.2 $\mathcal{B} = \{vrai, faux\}$

L'ensemble $\mathcal{B}$ est l'ensemble des valeurs booléennes. Il possède donc les deux éléments *vrai* et *faux*.

Dans ce contexte, nous pouvons maintenant définir l'ensemble $\mathcal{V}$ des valeurs comme étant :

Définition 3.3 $\mathcal{V} = \mathcal{E} + \mathcal{B}$

Cet ensemble $\mathcal{V}$ représente toutes les valeurs pouvant être affectées à une variable de l'un des types de base. Par exemple, $5 \in \mathcal{V}$ car on peut affecter la valeur 5 à une variable de type `int`. Le cas des booléens est particulier puisqu'il n'existe pas de type booléen en C, mais il est plus simple d'avoir un ensemble des valeurs qui les regroupe également.

On définit un certain nombre d'opérations sur les éléments de cet ensemble $\mathcal{V}$. Ces opérations (sémantiques) correspondent aux opérateurs (syntaxiques) définis la section 2.2.3 à l'exception des opérateurs unaires `&` et `*`. Notons que ces opérations sont stables sur $\mathcal{V}$, puisque pour un (respectivement deux) opérande élément de $\mathcal{V}$, le résultat de l'opération unaire (respectivement binaire) est élément de $\mathcal{V}$.

Identificateurs

Définition 3.4 *L'ensemble $\mathcal{I}$ est l'ensemble non fini (mais dénombrable) des identificateurs du langage.*

Notons ici une convention utilisée dans la suite : les identificateurs fournis par l'utilisateur sont implicitement préfixés par le caractère "_" pour les différencier automatiquement et sans ambiguïté des identificateurs que nous sommes amenés à créer.

Par exemple, l'identificateur `tab` fourni par l'utilisateur correspond à l'identificateur `_tab` $\in \mathcal{I}$.

Adresses

La mémoire d'un ordinateur est constituée d'une suite de cases, repérées par une suite croissante d'adresses. Nous ne représenterons pas ces adresses sous forme numérique, ce qui nous contraindrait à nous lier à une architecture donnée et à perdre en généralité, mais nous utiliserons les repères que sont les identificateurs. Informellement, à chaque identificateur est associée une adresse, mais à chaque adresse n'est pas lié un identificateur. Pour identifier une adresse, nous utiliserons l'identificateur dont l'adresse est la plus grande inférieure à l'adresse désirée, et nous associerons une valeur représentant la distance entre l'adresse de l'identificateur et l'adresse désirée.

Définition 3.5 *L'ensemble $\mathcal{A} \subset \mathcal{I} \times \mathcal{E}$ est l'ensemble des adresses.*

Chaque adresse est donc un élément du produit cartésien, c'est à dire un couple identificateur-déplacement. Le cas le plus simple des variables scalaires et pointeurs se résume au couple $\langle$identificateur$, 0\rangle$. Les variables scalaires et pointeurs sont toujours utilisées comme elles sont définies (pas de recouvrement). Le cas des tableaux est représenté par un déplacement éventuellement non nul. Cette notation ne permet pas une grande finesse dans l'analyse des tableaux, mais ce n'est pas le but recherché dans cette étude.

L'identificateur est associé à un objet, éventuellement complexe. Le déplacement représente le décalage par rapport au début de l'objet. La seule restriction apportée au décalage est que l'adresse résultante désigne bien une partie de l'objet. On peut donc associer un déplacement maximum d_m à chaque identificateur tel que :

Définition 3.6 *l'adresse* $\langle i, d\rangle$ *est valide si et seulement si* $0 \leq d < d_m(i)$

Cette définition exprime les règles sémantiques du langage C, c'est à dire qu'un pointeur ne peut sortir de l'objet (variable, tableau) dans lequel il pointe. La norme légitime cette attitude lorsqu'elle précise (voir paragraphe 3.3.6 de [ANSI88]) que l'addition d'une valeur entière à un pointeur n'est *définie* que si le résultat et le pointeur original pointent dans le même objet[1].

À titre d'exemple, le fragment de programme de la figure 3.1), dans lequel la variable désignée par l'identificateur `_x`, les éléments du tableau désigné par l'identificateur `_t` et l'objet pointé par le pointeur désigné par l'identificateur `_p` sont des entiers, utilise quatre adresses. La première correspond au doublet $\langle$`_x`$, 0\rangle$, la deuxième au doublet $\langle$`_t`$, 0\rangle$, la troisième au doublet $\langle$`_t`$, 1\rangle$ et la quatrième au doublet $\langle$`_p`$, 0\rangle$. L'affectation aux éléments du tableau illustre le fait que, par convention, la distance entre deux éléments consécutifs d'un tableau correspond à une unité d'adresse.

```
x = 1 ;
* (t + 0) = 2 ;          /* ⟺ t [0] = 2 */
* (t + 1) = 3 ;          /* ⟺ t [1] = 3 */
p = t ;
```

Figure 3.1: Exemple de programme

Définissons à présent certaines opérations sur les éléments de $\mathcal{A}$:

[1] Pour être précis, la norme indique que si le résultat pointe juste après la dernière case d'un tableau, le résultat est encore défini. Nous pourrions considérer, si cela posait un problème, que chaque tableau contient une case inutilisée à la fin, et augmenter d_m en conséquence. Mais nous n'aurons pas besoin de cet artifice.

- l'addition et la soustraction d'adresse avec un entier sont des fonctions :

$$\begin{array}{rll} + : & \mathcal{A} \times \mathcal{E} & \longrightarrow \mathcal{A} \\ & \langle\langle x, e_1\rangle, e_2\rangle & \longmapsto \langle x, e_1\rangle + e_2 = \langle x, e_1 + e_2\rangle \end{array}$$

$$\begin{array}{rll} - : & \mathcal{A} \times \mathcal{E} & \longrightarrow \mathcal{A} \\ & \langle\langle x, e_1\rangle, e_2\rangle & \longmapsto \langle x, e_1\rangle - e_2 = \langle x, e_1 - e_2\rangle \end{array}$$

  Ces fonctions obéissent à la règle de validité des adresses, c'est à dire que pour l'addition, par exemple, on doit avoir $0 \leq e1 + e2 < d_m(x)$.

- la soustraction d'adresses est une fonction :

$$\begin{array}{rll} - : & \mathcal{A} \times \mathcal{A} & \longrightarrow \mathcal{E} \\ & \langle\langle x, e_1\rangle, \langle x, e_2\rangle\rangle & \longmapsto \langle x, e_1\rangle - \langle x, e_2\rangle = e_1 - e_2 \end{array}$$

  Le domaine de définition de cette fonction est très réduit puisque nous ne pouvons faire la différence d'adresses que sur des adresses à l'intérieur du même objet. Notons que le résultat peut être négatif. Plus précisément, $-d_m(x) < \langle x, e_1\rangle - \langle x, e_2\rangle < d_m(x)$.

- les relations d'ordre sur les pointeurs doivent obligatoirement être restreintes : il ne s'agit pas d'un ordre total, mais d'un ordre partiel à l'intérieur de chaque objet. Ainsi, nous pouvons définir les opérateurs $<, \leq, >, \text{et} \geq$:

$$\begin{array}{rll} op : & \mathcal{A} \times \mathcal{A} & \longrightarrow \mathcal{B} \\ & \langle\langle x, e_1\rangle, \langle x, e_2\rangle\rangle & \longmapsto \langle x, e_1\rangle \text{ op } \langle x, e_2\rangle = (\langle x, e_1\rangle - \langle x, e_2\rangle \text{ op } 0) \end{array}$$

  Nous définissons les opérateurs d'ordre à l'aide des opérateurs d'ordre sur $\mathcal{V}$.

- les relations d'égalité et de non-égalité sont définies simplement par les fonctions :

$$\begin{array}{rll} =: & \mathcal{A} \times \mathcal{A} & \longrightarrow \mathcal{B} \\ & \langle\langle x_1, e_1\rangle, \langle x_2, e_2\rangle\rangle & \longmapsto (\langle x_1, e_1\rangle = \langle x_2, e_2\rangle) = (x_1 = x_2 \wedge e_1 = e_2) \\ \neq: & \mathcal{A} \times \mathcal{A} & \longrightarrow \mathcal{B} \\ & \langle\langle x_1, e_1\rangle, \langle x_2, e_2\rangle\rangle & \longmapsto (\langle x_1, e_1\rangle \neq \langle x_2, e_2\rangle) = (x_1 \neq x_2 \vee e_1 \neq e_2) \end{array}$$

Notons que chaque objet (variable) du programme est de taille finie, $d_m(i)$ existe pour tout identificateur i. L'ensemble des identificateurs $\mathcal{I}$ étant dénombrable, l'ensemble $\mathcal{A}$ est donc lui aussi dénombrable.

Partition des adresses

La description de $\mathcal{A}$ ci-dessus est simplifiée, et nous allons en donner une définition plus précise en accord avec les règles sémantiques du langage C. Pour cela, il est nécessaire de postuler une partition des adresses.

Axiome 3.1 *Il existe une partition de $\mathcal{A}$ en sous-ensembles $\mathcal{A}_i$, c'est à dire des sous-ensembles $\mathcal{A}_i$ de $\mathcal{A}$ tels que :*

$$\mathcal{A} = \bigcup_{i \geq 0} \mathcal{A}_i \textit{ et } \forall i \neq j, \mathcal{A}_i \cap \mathcal{A}_j = \emptyset$$

```
int i ;                    /* entier */
int *pi ;                  /* pointeur sur entier */
int **ppi ;                /* pointeur sur pointeur sur entier */
```

Figure 3.2: Exemple de programme

Que représente cette partition de $\mathcal{A}$? Il s'agit d'intégrer les types du langage C, et en particulier les pointeurs, à notre formalisme. De manière informelle et imagée, le i dans $\mathcal{A}_i$ (avec $i \geq 0$) correspond au nombre d'étoiles dans la déclaration de l'objet. Par exemple, les déclarations de la figure 3.2 impliquent que :

$$\begin{aligned} \langle \_\mathtt{i}, 0 \rangle &\in \mathcal{A}_0 \\ \langle \_\mathtt{pi}, 0 \rangle &\in \mathcal{A}_1 \\ \langle \_\mathtt{ppi}, 0 \rangle &\in \mathcal{A}_2 \end{aligned}$$

La raison d'être de cette partition est de mettre en évidence les pointeurs. Nous pourrions affiner davantage et partitionner $\mathcal{A}_0$ en ensembles disjoints suivant les types (par exemple $\mathcal{A}_0^{\mathtt{int}}$, $\mathcal{A}_0^{\mathtt{char}}$, etc, idem pour les $\mathcal{A}_i$ avec $i > 0$), mais ceci induirait une complexité d'écriture inutile.

Les règles sémantiques du langage C, ainsi que les restrictions que nous apportons pour définir notre mini-langage, garantissent la partition des adresses, et en particulier le non recouvrement des zones mémoire. Dans notre mini-langage, une variable est forcément utilisée comme elle est déclarée.

Notons que nous pouvons formaliser ces règles sémantiques en une propriété importante des pointeurs :

Propriété 3.1 $\forall i \in \mathcal{I}, \langle i, 0 \rangle \in \mathcal{A}_n \implies \forall k \in \mathcal{E}(0 \leq k < d_m(i)), \langle i, k \rangle \in \mathcal{A}_n$

Cette propriété stipule que tous les éléments d'un tableau déclaré avec un type sont utilisés avec ce type. En particulier, cette propriété ne donne aucune indication sur les objets *au dessus* ou *en dessous* d'un tableau (c'est à dire à l'extérieur de l'objet), puisque ce n'est pas valide.

Qu'en est-il des tableaux dans cette partition ? Comment se situent-ils parmi les ensembles $\mathcal{A}_i$? Il est tentant de généraliser la règle du nombre d'étoiles au nombre de dimensions. Toutefois, il est prudent d'étudier les tableaux d'un peu plus près.

```
int tab [10], *p, *q ;

p = q ;
p = tab ;
```

Figure 3.3: Exemple de programme

En effet, les tableaux se distinguent des pointeurs. Ainsi, dans l'exemple de la figure 3.3, les deux affectations ne sont pas équivalentes. La première consiste à lire le contenu de la case mémoire référencée par l'identificateur `q`, alors que la deuxième est simplement l'adresse du tableau, ou plus exactement de son premier élément. Les tableaux sont donc évalués comme leur adresse, sans accès mémoire supplémentaire : un identificateur de tableau est donc une constante "pointeur". On voit donc que les tableaux ne peuvent pas être traités comme les autres identificateurs.

Le langage C, par souci de faciliter l'écriture, confond pointeurs et tableaux jusqu'à un certain point. Par exemple, les deux écritures `tab [i]` et `*(tab + i)` sont équivalentes pour accéder au $(i + 1)$-ème élément du tableau. Cette règle est vraie pour les tableaux multi-dimensionnels, ou plus exactement pour la dernière dimension d'un tableau. Prenons comme exemple le tableau déclaré par `int t [10][20][30]`. Si nous appliquions ce principe à tous les éléments, `t [i][j][k]` serait équivalent à `*(t [i][j] + k)`, donc à `*(*(t [i] + j) + k)`, et enfin à `*(*(*(t + i) + j) + k)` ce qui est évidemment faux, puisqu'un seul accès mémoire suffit. Seule la première équivalence est vraie, donc `t [i][j][k]` est équivalent à `*(t [i][j] + k)` et `t [i][j]` est une adresse calculée à partir des valeurs de `i` et `j` sans nécessiter aucun accès mémoire supplémentaire. Nous avons raisonné avec trois dimensions, mais nous pouvons généraliser sans difficulté :

Propriété 3.2 *Quel que soit le nombre de dimensions d'un tableau, l'accès à un élément se fait avec un seul accès mémoire.*

Qu'est-ce qui différencie alors les tableaux des variables simples, pour lesquelles un accès se fait avec un accès mémoire (implicite) ? Pour éclairer ce point, supposons les déclarations suivantes :

```
int i ;                  /* entier */
int *pi ;                /* pointeur sur entier */
int ti [10] ;            /* tableau d'entiers */
```

La case mémoire d'adresse $\langle \texttt{\_i}, 0\rangle$ contient un entier, de même que la case mémoire $\langle \texttt{\_ti}, 0\rangle$. La case mémoire d'adresse $\langle \texttt{\_pi}, 0\rangle$ contient, quant à elle, un pointeur sur un entier. Pour lire la valeur entière, un accès mémoire suffit pour `i` (implicite) et pour `ti` (explicite). En revanche, il faut deux accès (un implicite et un explicite) pour l'objet entier adressé par `pi`.

On peut donc considérer un tableau comme une association de variables simples. Le tableau `ti` ne fait que contenir 10 exemplaires de variables comme `i`.

Dans ces conditions, nous pouvons énoncer la règle d'appartenance aux ensembles $\mathcal{A}_i$ (avec $i \geq 0$) :

Définition 3.7 *Une variable d'adresses $\langle i, k\rangle$ $(0 \leq k < d_m(i))$ est dans $\mathcal{A}_n$ si et seulement si la déclaration de i contient n étoiles.*

En d'autres termes, le nombre d'accès mémoire (implicites ou explicites avec l'opérateur `*`) pour arriver au type de base est égal à $n + 1$.

```
int i ;                  /* entier */
int *pi ;                /* pointeur sur entier */
int **ppi ;              /* pointeur sur pointeur sur entier */
int ti [10] ;            /* tableau d'entiers */
int *tpi [10] ;          /* tableau de pointeurs sur entiers */
int (*pti) [10] ;        /* pointeur sur tableau d'entiers */
int *(*ptpi) [10] ;      /* pointeur sur tableau de pointeurs sur entiers */
int (*tpti [10]) [20] ; /* tableau de pointeurs sur tableaux d'entiers */
```

Figure 3.4: Exemple de programme

Pour synthétiser finalement ce que sont les sous-ensembles $\mathcal{A}_i$, prenons un exemple complet. Les déclarations de la figure 3.4 induisent les appartenances suivantes :

$$
\begin{aligned}
\langle \_\texttt{i}, 0\rangle &\in \mathcal{A}_0 \\
\langle \_\texttt{pi}, 0\rangle &\in \mathcal{A}_1 \\
\langle \_\texttt{ppi}, 0\rangle &\in \mathcal{A}_2 \\
\forall 0 \le k < 10, \langle \_\texttt{ti}, k\rangle &\in \mathcal{A}_0 \\
\forall 0 \le k < 10, \langle \_\texttt{tpi}, k\rangle &\in \mathcal{A}_1 \\
\langle \_\texttt{pti}, 0\rangle &\in \mathcal{A}_1 \\
\langle \_\texttt{ptpi}, 0\rangle &\in \mathcal{A}_2 \\
\forall 0 \le k < 10, \langle \_\texttt{tpti}, k\rangle &\in \mathcal{A}_1
\end{aligned}
$$

Le tableau `ti`, comme nous l'avons vu, appartient à $\mathcal{A}_0$ comme la variable simple `i`. Le tableau `tpi` ajoute un accès mémoire, donc est dans $\mathcal{A}_1$, comme `pti`. Le pointeur `ptpi`, quant à lui, est un pointeur sur un tableau de 10 pointeurs sur des entiers. Le nombre d'accès mémoire nécessaire pour arriver aux entiers est 3 (1 pour la lecture du pointeur sur le tableau + 1 pour la lecture du pointeur dans le tableau + 1 pour la lecture de l'entier), donc $n = 3 - 1 = 2$, comme l'indique le nombre d'étoiles dans la déclaration. Le dernier cas, `tpti`, se résume à 10 pointeurs sur des tableaux de 20 entiers, donc c'est le même nombre d'accès mémoire que pour `pti`.

Pour être complet, signalons la présence en langage C de *faux* tableaux. En effet, les paramètres des fonctions déclarés sous la forme de tableaux à n dimensions ne sont pas des vrais tableaux, mais des pointeurs sur des tableaux à $n - 1$ dimensions. Ainsi, si nous avons le paramètre `int param [10][20]`, nous devons le réécrire sous la forme équivalente `int (*param)[20]`. Cette dernière forme implique : $\langle \_\texttt{param}, 0\rangle \in \mathcal{A}_1$, ce qui est bien compatible avec les règles de passage de paramètres.

Notons enfin que les opérations que nous avons définies sur l'ensemble $\mathcal{A}$ (addition, soustraction, autres opérations) se décomposent naturellement suivant les ensembles $\mathcal{A}_i$. De plus, ceux-ci sont stables pour ces opérations.

Mémoire

La mémoire contient des variables déclarées avec un des types de notre langage réduit. Chaque case de la mémoire contient soit une variable scalaire, donc un objet d'un type de base, soit un pointeur, c'est à dire une adresse, soit un tableaux de scalaires ou de pointeurs.

Définissons la famille de fonctions M_i comme étant :

$$
\begin{aligned}
& M_0 : \quad \mathcal{A}_0 \longrightarrow \mathcal{V} + \top \\
\forall i > 0, \quad & M_i : \quad \mathcal{A}_i \longrightarrow \mathcal{A}_{i-1} + \top
\end{aligned}
$$

Cette famille de fonctions fait correspondre à une adresse le contenu d'une case mémoire, c'est à dire une valeur (M_0) ou une adresse (M_i, avec $i > 0$). Si le contenu de la case mémoire n'est pas défini, nous notons $\top$ (*top*) sa valeur.

Définition 3.8 *Une mémoire est une fonction*

$$M = \bigcup_{i \geq 0} M_i$$

Par exemple, supposons que la mémoire soit dans l'état résultant de l'exécution des trois affectations du segment de programme précédent (figure 3.1). Nous pouvons représenter la fonction M par :

$$\begin{array}{rcl} M: \ \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & M(a) = \begin{cases} 1 & \text{si } a = \langle \texttt{\_x}, 0 \rangle \\ 2 & \text{si } a = \langle \texttt{\_t}, 0 \rangle \\ 3 & \text{si } a = \langle \texttt{\_t}, 1 \rangle \\ \langle \texttt{\_t}, 0 \rangle & \text{si } a = \langle \texttt{\_p}, 0 \rangle \\ \top & \text{sinon} \end{cases} \end{array}$$

Nommons à présent $\mathcal{M}$ l'ensemble de toutes les mémoires. $\mathcal{M}$ est donc un ensemble de fonctions. La fonction M décrite ci-dessus appartient à $\mathcal{M}$.

3.3 Affectation

L'affectation, opérateur d'expression dans le langage C, est maintenant une instruction dans notre langage d'étude. Cette mise à l'écart correspond à la volonté d'éliminer les effets de bord dans les expressions, ce qui facilite ainsi l'analyse des instructions. Pour citer un exemple, nous n'avons plus à traiter l'expression `a = b = 0`, mais deux instructions différentes `b = 0` et `a = 0`, plus simples.

L'affectation étant la première instruction étudiée, nous serons amenés à définir un modèle pour l'étude des instructions. Ce modèle nécessite une étude préalable de l'évaluation des expressions.

3.3.1 Évaluation des expressions

Nous avons vu en 2.2.3 une définition des expressions. Malheureusement, cette définition est grossière, car orientée vers la présentation de notre langage en supposant applicables les règles sémantiques du langage C. En particulier, la définition des expressions laisse de côté deux éléments importants dans le cadre de cette étude :

- les types de données ne figurent pas dans la syntaxe ; en particulier, la notion de *pointeur*, notion centrale de l'étude, n'y figure pas ;
- l'affectation distingue deux sortes d'expressions ; cette distinction a été évoquée lorsque nous avons présenté l'affectation, mais de manière trop informelle.

Il nous faut donc une définition plus précise des expressions pour servir de support à une modélisation des instructions.

Partition des identificateurs

De même que nous avons partitionné l'ensemble $\mathcal{A}$ des adresses, nous allons partitionner l'ensemble $\mathcal{I}$ des identificateurs suivant la relation :

$$\text{soit}(i,j) \in \mathcal{I}^2, i \text{ est en relation avec } j \Longleftrightarrow \exists n \text{ tel que } \langle i,0\rangle \in \mathcal{A}_n \text{ et } \langle j,0\rangle \in \mathcal{A}_n$$

On peut montrer simplement que cette relation est réflexive, symétrique et transitive, donc que cette relation est une relation d'équivalence. Elle définit une partition car tout couple $\langle i,0\rangle$ où $i \in \mathcal{I}$ appartient à un ensemble $\mathcal{A}_n$.

Nous pouvons donc associer un ensemble $\mathcal{I}_n$ à tout ensemble $\mathcal{A}_n$.

Nous avons vu que les tableaux posaient un problème car leur évaluation était différente de celles des variables simples. Nommons identificateur$_n$ les identificateurs des variables simples de $\mathcal{I}_n$. Nous noterons constante$_n$ (avec forcément $n \geq 1$) les identificateurs des tableaux de $\mathcal{I}_n$. Notons qu'un pointeur sur un tableau est un identificateur$_n$ et pas une constante$_n$.

Nouvelle syntaxe des expressions

Tous les langages impératifs traditionnels distinguent la partie gauche de la partie droite d'une expression. La partie droite s'évalue en une valeur, la partie gauche en une adresse où écrire la valeur. Ainsi, si on écrit :

```
i = i ;
```

Le `i` de droite s'évalue en le contenu de la case mémoire nommée `_i`, alors que le `i` de gauche s'évalue en l'adresse de cette même case mémoire. On parle alors de *right hand side* et de *left hand side* pour identifier les deux membres de l'affectation. On voit que leur évaluation diffère, ainsi que leur syntaxe : par exemple, on ne peut écrire `&i` en partie gauche, mais on peut l'écrire en partie droite.

expression	::= rhs
	| lhs
rhs	::= rhs_0
	| rhs_1
	| ...
	| rhs_n
	| ...
rhs_0	::= constante
	| identificateur_0
	| `(` rhs_0 `)`
	| rhs_0 operateur-binaire rhs_0
	| operateur-unaire rhs_0
	| rhs_i `-` rhs_i
	| `*` rsh_1
rhs_n	::= constante_{n-1}
	| identificateur_n
	| `(` rhs_n `)`
	| rhs_n `+` rhs_0
	| rhs_n `-` rhs_0
	| `*` rsh_{n+1}
	| `&` lsh_{n-1}
lhs	::= lhs_0
	| lhs_1
	| ...
	| lhs_n
	| ...
lhs_n	::= identificateur_n
	| `(` lhs_n `)`
	| `*` rhs_{n+1}

Figure 3.5: Nouvelle syntaxe des expressions

Cette différence de syntaxe est reprise en figure 3.5. Notons quelques remarques à propos de cette nouvelle syntaxe :

- on a dissocié les deux membres en familles : les rhs_n et les lhs_n ;
- lorsqu'une valeur est ajoutée à une adresse, on place d'abord l'adresse, bien que l'opérateur d'addition soit commutatif ;
- le n de rhs_n ou lhs_n correspond au nombre d'étoiles dans le type résultat.

```
int *tab [10] ;
int *p ;

* (tab + 1) = & *(p + 2) ;
```

Figure 3.6: Exemple de programme

Prenons un exemple. Dans le segment de programme de la figure 3.6, les deux expressions sont :

- `* (tab + 1)` : c'est un lhs_1 et son arbre de dérivation est représenté en figure 3.7 ;
- `& * (p + 2)` : c'est un rhs_1 (identique à l'expression `p + 2`), dont l'arbre de dérivation est représenté en figure 3.8.

Définissons l'ensemble $\mathcal{X}$ comme étant l'ensemble des expressions satisfaisant la syntaxe ci-dessus. Pour plus de précision, nous serons amenés à définir $\mathcal{X}_l$ et $\mathcal{X}_r$ les sous-ensembles correspondant respectivement aux expressions *lhs* et *rhs* ($\mathcal{X} = \mathcal{X}_l \cup \mathcal{X}_r$). On notera que l'ambiguïté de la grammaire fait que $\mathcal{X}_l \cap \mathcal{X}_r \neq \emptyset$. En effet, l'expression `i` que nous avons vu dans l'exemple initial de ce paragraphe est à la fois *lhs* lorsque placée en partie gauche et *rhs* lorsque placée en partie droite de l'affectation. L'expression `i` est donc élément de $\mathcal{X}_l$ aussi bien que de $\mathcal{X}_r$.

Évaluation des expressions

Étant donnée une expression (*lhs* ou *rhs*), quelle est sa valeur ? Pour répondre à cette question, nous devons définir des familles L_n et R_n ($n \geq 0$) de fonctions d'*interprétation*. Ces fonctions évaluent une expression suivant la syntaxe définie en 3.5 dans le contexte d'une mémoire, pour donner une valeur ou une adresse.

En reprenant la syntaxe définie en 3.5, nous pouvons définir la famille de fonctions L_n :

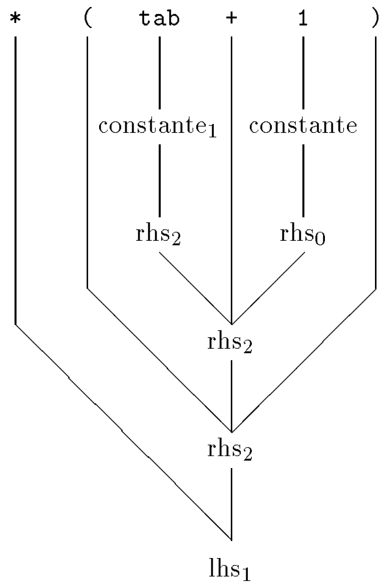

Figure 3.7: Arbre de dérivation de `* (tab + 1)`

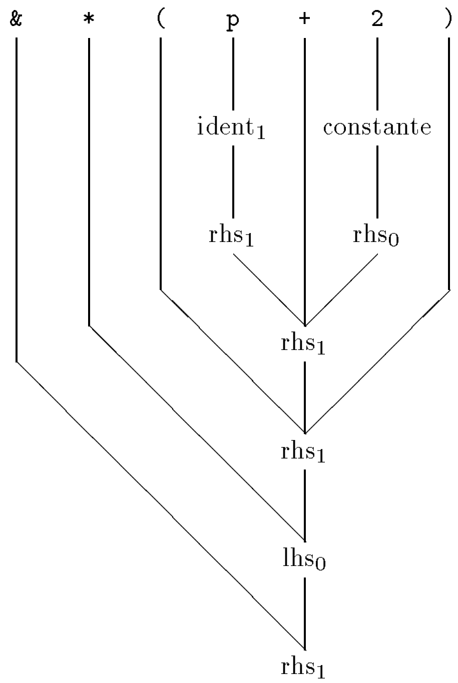

Figure 3.8: Arbre de dérivation de `& * (p + 2)`

- $L_n(\text{identificateur}_n, M) = \langle \text{identificateur}_n, 0\rangle$: l'évaluation d'un identificateur dans un lhs_n revient à former l'adresse élément de $\mathcal{A}_n$ correspondant ;
- $L_n((\ \text{lhs}_n), M) = L_n(\text{lhs}_n, M)$: la parenthèse n'est qu'un artifice pour forcer l'ordre des évaluations. La valeur d'une expression entre parenthèses n'est donc que la valeur de l'expression ;
- $L_n(*\ \text{rhs}_{n+1}, M) = R_{n+1}(\text{rhs}_{n+1}, M)$: l'évaluation d'un lhs_n de la forme `*p` revient à calculer la valeur de `p`.

De même, nous pouvons définir la fonction R_0 :

- $R_0(\text{constante}, M) = V(\text{constante})$: la valeur d'une constante est la traduction de sa représentation textuelle dans l'ensemble $\mathcal{V}$ des valeurs, opérée par la fonction V de projection dans l'ensemble $\mathcal{V}$;
- $R_0(\text{identificateur}_0, M) = M(\langle \text{identificateur}_0, 0\rangle)$: la valeur d'un identificateur est simplement le contenu de la mémoire repérée par l'identificateur (ou plus exactement son adresse) ;
- $R_0((\ \text{rhs}_0), M) = R_0(\text{rhs}_0, M)$: la valeur d'une expression entre parenthèses n'est que la valeur de l'expression ;
- $R_0(\text{rhs}_0 \text{ operateur-binaire } \text{rhs}_0, M) = R_0(\text{rhs}_0, M) \text{ operateur-binaire } R_0(\text{rhs}_0, M)$: une opération entre deux rhs_0 (donc éléments de X_r) s'évalue en l'opération (sur les éléments de $\mathcal{V} + \mathcal{A}$) appliquée aux deux membres évalués ;
- $R_0(\text{operateur-unaire } \text{rhs}_0, M) = \text{operateur-unaire } R_0(\text{rhs}_0, M)$: un opérateur unaire appliqué à un rhs_0 (donc élément de X_r) s'évalue en l'opération (sur un élément de $\mathcal{V} + \mathcal{A}$) appliquée à l'évaluation du rhs_0 ;
- $R_0(\text{rhs}_i \texttt{ - } \text{rhs}_i, M) = R_i(\text{rhs}_i, M) - R_i(\text{rhs}_i, M)$: une valeur peut être le résultat de la soustraction de deux adresses (compatibles au sens du langage C) ;
- $R_0(*\ \text{rhs}_1, M) = M(R_1(\text{rhs}_1, M))$: l'évaluation d'une indirection revient à une lecture en mémoire.

Puis, la fonction R_n pour $n \geq 1$:

- $R_n(\text{constante}_{n-1}, M) = \langle \text{constante}_{n-1}, 0\rangle$: l'évaluation d'un nom de tableau retourne simplement l'adresse correspondant au premier élément ;
- $R_n(\text{identificateur}_n, M) = M(\langle \text{identificateur}_n, 0\rangle)$: comme pour R_0, la valeur d'un identificateur est simplement le contenu de la mémoire repérée par l'adresse correspondant à l'identificateur ;
- $R_n((\ \text{rhs}_n), M) = R_n(\text{rhs}_n, M)$: là encore, la valeur d'une expression entre parenthèses n'est que la valeur de l'expression ;

- $R_n(\mathrm{rhs}_n \texttt{ + } \mathrm{rhs}_0, M) = R_n(\mathrm{rhs}_n, M) + R_0(\mathrm{rhs}_0, M)$: une adresse plus une valeur est évaluée en la somme de l'évaluation de l'adresse et de l'évaluation de la valeur ;
- $R_n(\mathrm{rhs}_n \texttt{ - } \mathrm{rhs}_0, M) = R_n(\mathrm{rhs}_n, M) - R_0(\mathrm{rhs}_0, M)$: comme pour l'addition, la valeur est ici l'évaluation de l'adresse moins l'évaluation de la valeur ;
- $R_n(\texttt{*}\ \mathrm{rhs}_{n+1}, M) = M(R_{n+1}(\mathrm{rhs}_{n+1}, M))$: comme pour R_0, l'évaluation d'une indirection revient à une lecture en mémoire ;
- $R_n(\texttt{\&}\ \mathrm{lhs}_{n-1}, M) = L_{n-1}(\mathrm{lhs}_{n-1}, M)$: prendre l'adresse d'une variable revient à évaluer son adresse, comme s'il s'agissait d'un membre gauche d'affectation, sans autre accès mémoire.

Afin d'assimiler cette définition, examinons le segment de programme de la figure 3.6 qui nous avait servi d'exemple lors de la définition de la nouvelle syntaxe des expressions. Les valeurs de la fonction d'interprétation pour ces deux expressions sont :

$$
\begin{aligned}
L_1(\texttt{* (tab + 1)}, M) &= R_2(\texttt{(tab + 1)}, M)\\
&= R_2(\texttt{tab + 1}, M)\\
&= R_2(\texttt{tab}) + R_0(\texttt{1}, M)\\
&= \langle \texttt{\_tab}, 0\rangle + 1\\
&= \langle \texttt{\_tab}, 1\rangle
\end{aligned}
$$

$$
\begin{aligned}
R_1(\texttt{\& * (p + 2)}, M) &= L_2(\texttt{* (p + 2)}, M)\\
&= R_1(\texttt{(p + 2)}, M)\\
&= R_1(\texttt{p + 2}, M)\\
&= R_1(\texttt{p}, M) + R_0(\texttt{2}, M)\\
&= M(\langle \texttt{\_p}, 0\rangle) + 2
\end{aligned}
$$

Pouvons-nous caractériser ces fonctions d'interprétation, et en particulier affirmer que :

$$
\begin{aligned}
\forall n \geq 0, L_n : \quad & \mathcal{X}_l \times \mathcal{M} && \longmapsto \mathcal{A}_n\\
R_0 : \quad & \mathcal{X}_r \times \mathcal{M} && \longmapsto \mathcal{V}\\
\forall n \geq 1, R_n : \quad & \mathcal{X}_r \times \mathcal{M} && \longmapsto \mathcal{A}_{n-1}
\end{aligned}
$$

C'est ce que nous allons nous attacher à prouver, en reprenant la définition de ces fonctions d'interprétation. Notre raisonnement prendra appui sur les arbres de dérivation d'une expression : en partant des feuilles de l'arbre (les éléments terminaux du langage, ou assimilés comme les constantes et les identificateurs), nous prouverons que chaque élément de l'arbre conserve les caractéristiques que nous avons énoncées ci-dessus.

Partons des feuilles des arbres de dérivation :

- $R_0(\text{constante}, M) = V(\text{constante})$, donc :

$$R_0(\text{constante}, M) \in \mathcal{V}$$

- $R_0(\text{identificateur}_0, M) = M(\langle \text{identificateur}_0, 0\rangle)$ or nous savons, M_0 étant la restriction de M à $\mathcal{A}_0$, que $M(\langle \text{identificateur}_0, 0\rangle) = M_0(\langle \text{identificateur}_0, 0\rangle)$ puisque identificateur$_0 \in \mathcal{A}_0$ (par définition de identificateur$_0$), donc, l'ensemble d'arrivée de M_0 est $\mathcal{V}$, ce qui entraîne :

$$R_0(\text{identificateur}_0, M) \in \mathcal{V}$$

- $L_n(\text{identificateur}_n, M) = \langle \text{identificateur}_n, 0\rangle$. Or, par définition de identificateur$_n$, l'adresse $\langle \text{identificateur}_n, 0\rangle \in \mathcal{A}_n$. Donc :

$$L_n(\text{identificateur}_n, M) \in \mathcal{A}_n$$

- $R_n(\text{constante}_{n-1}, M) = \langle \text{constante}_{n-1}, 0\rangle$ donc, par définition de constante$_{n-1}$,

$$R_n(\text{constante}_{n-1}, M) \in \mathcal{A}_{n-1}$$

- $R_n(\text{identificateur}_n, M) = M(\langle \text{identificateur}_n, 0\rangle)$, ce qui entraîne naturellement par définition de identificateur$_n$ et de M_n :

$$R_n(\text{identificateur}_n, M) \in \mathcal{A}_{n-1}$$

Démontrons maintenant que ces propriétés sont toujours vérifiées à chaque étape de la progression dans l'arbre de dérivation :

- $L_n((\ \text{lhs}_n), M) = L_n(\text{lhs}_n, M)$, c'est à dire que l'adjonction de parenthèses est une opération neutre et ne change donc pas l'ensemble d'arrivée de L_n.

- $L_n(*\ \text{rhs}_{n+1}, M) = R_{n+1}(\text{rhs}_{n+1}, M)$, or par hypothèse, $R_{n+1}(x, M) \in \mathcal{A}_n$, donc :

$$L_n(*\ \text{rhs}_{n+1}, M) \in \mathcal{A}_n$$

- $R_0((\ \text{rhs}_0), M) = R_0(\text{rhs}_0, M)$. Comme pour L_n, l'adjonction de parenthèses est une opération neutre.

- $R_0(\text{rhs}_0 \text{ operateur-binaire rhs}_0, M) = R_0(\text{rhs}_0, M)$ operateur-binaire $R_0(\text{rhs}_0, M)$: par hypothèse, $R_0(x, M) \in \mathcal{V}$, et comme nous utilisons les opérateurs sur V qui sont stables, le résultat est donc élément de $\mathcal{V}$:

$$R_0(\text{rhs}_0 \text{ operateur-binaire rhs}_0, M) \in \mathcal{V}$$

- $R_0(\text{operateur-unaire rhs}_0, M) =$ operateur-unaire $R_0(\text{rhs}_0, M)$: comme nous l'avons vu précédemment, nous utilisons des opérateurs stables sur $\mathcal{V}$, donc :

$$R_0(\text{operateur-unaire rhs}_0, M) \in \mathcal{V}$$

- $R_0(\text{rhs}_i \texttt{ - } \text{rhs}_i, M) = R_i(\text{rhs}_i, M) - R_i(\text{rhs}_i, M)$: nous utilisons là l'opérateur de soustraction d'adresses de $\mathcal{A}_i \times \mathcal{A}_i$ dans $\mathcal{E}$, or $\mathcal{E} \subset \mathcal{V}$, donc :

$$R_0(\text{rhs}_i \texttt{ - } \text{rhs}_i, M) \in \mathcal{V}$$

- $R_0(* \text{ rhs}_1, M) = M(R_1(\text{rhs}_1, M))$: puisque par hypothèse de travail, $\forall x \in \mathcal{X}_r, R_1(x, M) \in \mathcal{A}_0$, il vient naturellement que $M(R_1(x, M)) = M_0(R_1(x, M))$, or $\forall a \in \mathcal{A}_0, M_0(a) \in \mathcal{V}$ par définition de M_0, donc :

$$R_0(* \text{ rhs}_1, M) \in \mathcal{V}$$

- $R_n((\text{ rhs}_n), M) = R_n(\text{rhs}_n, M)$. Comme pour L_n et R_0, l'adjonction de parenthèses est une opération neutre.

- $R_n(\text{rhs}_n \texttt{ + } \text{rhs}_0, M) = R_n(\text{rhs}_n, M) + R_0(\text{rhs}_0, M)$. Par hypothèse, $\forall x \in \mathcal{X}_r, R_n(x, M) \in \mathcal{A}_{n-1}$ et $R_0(x, M) \in \mathcal{V}$. L'addition utilisée est donc l'addition entre une adresse et un entier, son résultat est une adresse de même nature, donc :

$$R_n(\text{rhs}_n \texttt{ + } \text{rhs}_0, M) \in \mathcal{A}_{n-1}$$

- $R_n(\text{rhs}_n \texttt{ - } \text{rhs}_0, M) = R_n(\text{rhs}_n, M) - R_0(\text{rhs}_0, M)$. Comme précédemment, il s'agit de la soustraction d'un entier à une adresse, et donc :

$$R_n(\text{rhs}_n \texttt{ - } \text{rhs}_0, M) \in \mathcal{A}_{n-1}$$

- $R_n(* \text{ rhs}_{n+1}, M) = M(R_{n+1}(\text{rhs}_{n+1}, M))$. Par hypothèse, $R_{n+1}(x, M) \in \mathcal{A}_n$, M égale ici M_n, sa restriction à $\mathcal{A}_n$, et donc $\forall a \in \mathcal{A}_n, M_n(a) \in \mathcal{A}_{n-1}$, donc :

$$R_n(* \text{ rhs}_{n+1}, M) \in \mathcal{A}_{n-1}$$

- $R_n(\texttt{\&} \text{ lhs}_{n-1}, M) = L_{n-1}(\text{lhs}_{n-1}, M)$. Les hypothèses impliquent simplement que :

$$R_n(\texttt{\&} \text{ lhs}_{n-1}, M) \in \mathcal{A}_{n-1}$$

Nouvelle syntaxe de l'affectation

Avec cette nouvelle syntaxe des expressions, la syntaxe de l'affectation telle que nous l'avons définie en 3.3.1 doit être réécrite (voir figure 3.9). Il en va de même pour les autres instructions qui sont elles aussi affectées par cette nouvelle syntaxe.

Conclusion

Dans cette section, nous avons proposé une formalisation du langage d'analyse introduit au chapitre précédent. Nous avons précisé notamment ses expressions dans le but de fournir une ossature pour l'étude des pointeurs. Nous avons reflété dans les expressions la partition de la mémoire que nous avions définie dans la section précédente. Ainsi, nous avons structuré la mémoire et cela nous permettra une plus grande précision dans la délimitation des zones modifiées par les instructions.

instruction	::=	{ instruction* }
	|	affectation
	|	alternative
	|	iteration
affectation	::=	lhs$_n$ = rhs$_n$;
alternative	::=	**if** (rhs$_n$) instruction
	|	**if** (rhs$_n$) instruction **else** instruction
iteration	::=	**while** (rhs$_n$) instruction

Figure 3.9: Nouvelle syntaxe des instructions

3.3.2 Modélisation par transformations

Il faut maintenant trouver un moyen pour modéliser l'effet d'un segment de programme, puis une représentation de ce modèle afin de l'exploiter.

Dans un premier temps, nous nous attacherons à définir un modèle pour les affectations. Nous serons alors conduit naturellement à sa représentation, mais nous mettrons en lumière les insuffisances de celle-ci lors de sa généralisation, ce qui nous mènera à une nouvelle représentation.

Formes abstraite et concrète

La mémoire est définie ici comme une fonction. L'effet d'une affectation, ou plus généralement d'un segment de programme, sera aussi défini comme une fonction. C'est ce que nous appellerons la *forme abstraite* d'une transformation.

Par opposition, la représentation que nous allons chercher sera définie comme une série d'expressions, éléments de l'ensemble $\mathcal{X}$, c'est à dire formées à partir des éléments syntaxiques du langage étudié. C'est ce que nous appellerons la *forme concrète* d'une transformation.

Les passages d'une forme à l'autre, que nous nous emploierons à préciser, s'appellent donc *abstraction* et *concrétisation.*

Ces notions d'abstraction et de concrétisation ne doivent pas être confondues avec les opérations correspondantes liées à l'interprétation abstraite [Cous77][Cous81]. En effet, dans le cadre de l'interprétation abstraite, on définit une sémantique concrète pour formaliser l'exécution d'un programme et les environnements (associations mémoire-valeur) que l'on peut associer à chaque point. Ces environnements étant difficiles à calculer, on définit un sous ensemble des valeurs, dont la définition basée sur des approximations dépend du problème à traiter, et des règles d'interprétation en conséquence, ce qui conduit à une sémantique abstraite. Pour vérifier la validité de l'interprétation abstraite, on définit deux fonctions (abstraction α et concrétisation γ) opérant sur des vecteurs de contexte (c'est à dire des fonctions associant à un point du programme une liste d'environnements abstraits ou concrets suivant le cas). La validité est assurée si, étant donné un vecteur concret c et un vecteur abstrait a, $\gamma(\alpha(c))$ est moins précis que (ou égal à) c et $\alpha(\gamma(a))$ est moins précis que (ou égal à) a.

Transformation de la mémoire

Soit une affectation $l = r$. Cette instruction a pour effet de modifier la case mémoire définie par l'expression l. Nous voyons donc que l'effet de l'affectation est de modifier une mémoire. Si nous nommons M la mémoire avant l'affectation, la nouvelle mémoire est définie par :

$$\begin{array}{rlll} M': & \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ & a & \longmapsto M'(a) = & \left\{ \begin{array}{ll} \text{si } a = L(l, M) & \text{alors } R(r, M) \\ & \text{sinon } M(a) \end{array} \right. \end{array}$$

où L et R sont les fonctions d'interprétation générales ($L = \bigcup_{n \geq 0} L_n$ et $R = \bigcup_{n \geq 0} R_n$).

La forme extrêmement simple de cette fonction M' est dûe au fait que les expressions de l'affectation ne contiennent pas d'effets de bord (comme ceux qu'une expression typique du langage C peut contenir, telle `p++`), et donc la même mémoire M sert à interpréter l'intégralité de l aussi bien que de r.

L'affectation peut donc être vue comme une fonction faisant passer d'un état mémoire M dans un autre état mémoire M'.

Définition 3.9 *On nomme transformation associée à une affectation la fonction T telle que :*

$$\begin{array}{rll} T: & \mathcal{M} & \longrightarrow \mathcal{M} \\ & M & \longmapsto T(M) = M' \end{array}$$

Nous avons défini une transformation comme une fonction. Comment concrétiser cette transformation ? Quelle notation pouvons-nous trouver pour la représenter simplement ?

Représentation d'une transformation

En examinant de près la définition de la fonction M', nous constatons que, outre la fonction M, les seuls éléments définissant la nouvelle mémoire sont les expressions l et r, et les fonctions d'interprétation L et R. Or, ces fonctions ne varient pas : leur définition a été étudiée en 3.3.1. La nouvelle fonction M' dépend donc uniquement de M et des deux expressions l et r. Ceci nous conduit naturellement à la représentation des transformations par des *substitutions* :

Définition 3.10 *On représente la transformation T associée à une affectation $l = r$ par le couple $\langle l, r\rangle \in \mathcal{X}_l \times \mathcal{X}_r$, que l'on note $\sigma = [l \leftarrow r]$.*

En d'autres termes, une substitution n'est que l'écriture (mécanique) des deux expressions intervenant dans une affectation. Ainsi, si nous considérons l'affectation `i = 5`, nous pouvons immédiatement écrire la substitution $[\mathtt{i} \leftarrow \mathtt{5}]$.

De manière générale, pour une affectation, lorsqu'on a une fonction mémoire de la forme :

$$\begin{array}{rll} M': & \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ & a & \longmapsto M'(a) = \left\{ \begin{array}{ll} \text{si } a = e_1 & \text{alors } e_2 \\ & \text{sinon } M(a) \end{array} \right. \end{array}$$

la modification de la mémoire illustrée par la nouvelle fonction M' en fonction de la fonction M peut être représentée par un couple $\langle l, r\rangle \in \mathcal{X}_l \times \mathcal{X}_r$, où $L(l, M) = e_1$ et $R(r, M) = e_2$. Les opérations d'abstraction et de concrétisation sont triviales.

Par abus de langage, nous utiliserons σ à la place de la transformation qu'elle représente : l'opération d'abstraction est implicite, et nous noterons la fonction $\sigma : \mathcal{M} \longrightarrow \mathcal{M}$.

Notons $\mathcal{S} = \mathcal{X}_l \times \mathcal{X}_r$ l'ensemble de toutes les substitutions.

Insuffisance des substitutions

Anticipons la suite de l'étude : il arrivera bien peu souvent qu'un programme entier se résume à une seule affectation. Les transformations ne vont donc plus être associées à une affectation, mais à une ou plusieurs instructions. La conséquence logique est qu'elles ne pourront plus être représentées par un couple $\langle l, r\rangle \in \mathcal{X}_l \times \mathcal{X}_r$, mais par un ensemble de couples.

Par exemple, le segment de programme :

```
i = 1 ;
j = 2 ;
```

sera représenté par la substitution $\sigma = [\mathtt{i} \leftarrow \mathtt{1}, \mathtt{j} \leftarrow \mathtt{2}]$.

Avec cette extension de la définition des substitutions, l'ensemble $\mathcal{S}$ des substitutions n'est plus le produit cartésien $\mathcal{X}_l \times \mathcal{X}_r$, mais $P(\mathcal{X}_l \times \mathcal{X}_r)$, l'ensemble des ensembles de couples éléments du produit $\mathcal{X}_l \times \mathcal{X}_r$.

Le fait de devoir traiter plusieurs couples, et en particulier en présence de pointeurs, pose un problème intéressant, celui de la consistance d'une substitution.

Définition 3.11 *Une substitution $\sigma = [l_1 \leftarrow r_1, \ldots l_n \leftarrow r_n]$ est consistante si et seulement si :*

$$\forall M \in \mathcal{M}, L(l_i, M) = L(l_j, M) \Longrightarrow i = j$$

En d'autres termes, une substitution est consistante si toutes ses expressions en partie gauche (c'est-à-dire tous les l des couples $\langle l, r \rangle \in \mathcal{X}_l \times \mathcal{X}_r$) désignent des cases mémoire différentes (quelle que soit la configuration de la mémoire).

Par exemple, la substitution $[\mathtt{i} \leftarrow \mathtt{1}, \mathtt{j} \leftarrow \mathtt{2}]$ est consistante, car les deux adresses $\langle \mathtt{\_i}, 0 \rangle$ et $\langle \mathtt{\_j}, 0 \rangle$ sont distinctes. En revanche, la substitution $[\mathtt{i} \leftarrow \mathtt{1}, \mathtt{*p} \leftarrow \mathtt{2}]$ n'est pas consistante, car si $M(\langle \mathtt{\_p}, 0 \rangle) = \langle \mathtt{\_i}, 0 \rangle$, les deux lhs de la substitution ne sont pas systématiquement identiques. On peut alors trouver une famille infinie de fonctions mémoire qui satisfont le critère de non-consistance.

Le problème est qu'il est impossible dans le cas général d'assurer la consistance. Tant que nous avons à traiter des variables simples ou des tableaux, cela reste possible, puisque nous n'autorisons pas le recouvrement des zones mémoire. Mais en face de pointeurs, ça ne l'est plus. Il y a alors ambiguïté sur la case mémoire modifiée.

Par exemple, il peut y avoir ambiguïté lorsque :

- les interprétations de deux lhs sont éléments du même ensemble $\mathcal{A}_n$: par exemple, si `i` est une variable entière et `p` et `q` des pointeurs sur entier, alors :
  - la substitution $[\mathtt{i} \leftarrow e_1, \mathtt{p} \leftarrow e_2]$ est consistante quelle que soit la configuration de la mémoire,
  - la substitution $[\mathtt{*p} \leftarrow e_1, \mathtt{*q} \leftarrow e_2]$ n'est pas consistante,

  et
- au moins une des interprétations nécessite l'utilisation d'un pointeur, c'est à dire une interprétation de la forme $L(e, M) = M(R(e'))$. On voit bien, par définition de la fonction L, que le lhs doit être de la forme $e = *e'$. Par exemple, si `i` est une variable entière et `p` et `q` sont des pointeurs sur entiers, alors :
  - la substitution $[\mathtt{p} \leftarrow e_1, \mathtt{q} \leftarrow e_2]$ est consistante,
  - la substitution $[\mathtt{*p} \leftarrow e_1, \mathtt{*q} \leftarrow e_2]$ n'est pas consistante,
  - la substitution $[\mathtt{i} \leftarrow e_1, \mathtt{*p} \leftarrow e_2]$ n'est pas consistante.

Mais il peut aussi y avoir ambiguïté lorsque deux éléments d'un tableau sont accédés avec des indices dont la valeur nécessite au moins un accès mémoire. Ainsi, la substitution $[\mathtt{*(tab + 1)} \leftarrow e_1, \mathtt{*(tab + i)} \leftarrow e$ n'est pas consistante, car il y a incertitude sur la valeur de la variable `i`.

La consistance est donc loin d'être assurée lorsqu'on étend les substitutions à plusieurs instructions.

Extension des substitutions

Nous avons vu au paragraphe précédent que la consistance de la représentation d'une substitution n'était pas assurée dans le cas général, et nous avons exposé quelques cas de non-consistance. Cependant, les substitutions et leurs représentations sont un outil remarquable pour modéliser les instructions d'affectation, et en particulier les séquences d'instructions. Nous avons donc besoin d'étendre cet outil.

Le problème qui se pose est l'ambiguïté des lhs dans les représentations des substitutions. Cette ambiguïté est dûe au fait que les représentations sont d'origine purement syntaxique, et que deux expressions peuvent avoir la même valeur dans certaines configurations de la mémoire. On peut alors généraliser les représentations des substitutions en distingant suivant les différentes configurations de la mémoire pour avoir de nouveau la consistance.

L'outil de base qui nous permettra de faire cette distinction est la garde que nous définissons comme :

Définition 3.12 *Une garde est une fonction G qui à une mémoire M associe une valeur booléenne $G(M) \in \mathcal{B}$.*

Autrement exprimé, une garde est une fonction :

$$\begin{aligned} G: \ \mathcal{M} &\longrightarrow \mathcal{B} \\ M &\longmapsto G(M) \end{aligned}$$

De manière analogue à la représentation des transformations, nous devons passer d'un modèle abstrait de garde à un modèle concret. Quelle représentation devons nous donc choisir ?

Les configurations de mémoire que nous utiliserons par la suite seront des conjonctions de prédicats, donc une catégorie particulière de gardes. Nous allons les représenter par des ensembles d'expressions logiques. Une expression logique est un cas particulier de rhs_0. Pour alléger l'exposé, nous nous permettrons un petit abus en n'introduisant pas de nouvelle notation pour les sous-ensembles de $\mathcal{X}_r$ que sont les expressions logiques. Le lecteur notera que les expressions logiques sont caractérisées par un opérateur logique (voir figure 3.9), et nous utiliserons ultérieurement ce fait.

Définition 3.13 *La représentation γ d'une garde est un sous-ensemble de $P(\mathcal{X}_r) + \mathcal{B}$.*

Cette représentation est valide, car si on considère une mémoire M et une représentation de garde $\gamma = \{e_i \in \mathcal{X}_r + \mathcal{B} / 1 \leq i \leq n\}$, la fonction G correspondante s'écrit :

$$\begin{aligned} G: \ \mathcal{M} &\longrightarrow \mathcal{B} \\ M &\longmapsto G(M) = \textstyle\bigwedge_{1 \leq i \leq n} B(e_i, M) \end{aligned}$$

où $\bigwedge$ correspond à la conjonction (*et*) des prédicats, et B correspond à la fonction :

$$\begin{array}{llll} B: & (\mathcal{X}_r + \mathcal{B}) \times \mathcal{M} & \longrightarrow \mathcal{B} & \\ & \langle x, M \rangle & \longmapsto B(\langle x, M \rangle) = & \left\{ \begin{array}{ll} R_0(x, M) \neq 0 & \text{si } x \in \mathcal{X}_r \\ x & \text{si } x \in \mathcal{B} \end{array} \right. \end{array}$$

L'abstraction de la représentation est établie. En revanche, l'opération de concrétisation n'est pas possible, car notre définition de la garde n'est pas assez précise.

Là encore, par abus de langage, nous utiliserons γ à la place de la garde qu'elle représente. Nous noterons la fonction $\gamma : \mathcal{M} \longrightarrow \mathcal{B}$. L'opération d'abstraction est implicite.

Par exemple, L'ensemble $\{$`*p != i`, `p != q`$\}$ est une représentation de garde, de même que l'ensemble $\{$vrai$\}$.

Nous noterons $\mathcal{G} \subset P(\mathcal{X}_r) + \mathcal{B}$ l'ensemble de toutes les représentations de gardes.

Ces gardes nous donnent l'outil pour lever les ambiguïtés et représenter plus précisément les transformations :

Définition 3.14 *On représente une transformation associée à un segment de programme par un ensemble τ de doublets $\langle \gamma_i, \sigma_i \rangle \in \mathcal{G} \times \mathcal{S}$ ($\tau \in P(\mathcal{G} \times \mathcal{S})$).*

Par exemple, nous pourrons représenter la transformation associée au segment de programme de la figure 3.10 par :

```
int i, *p ;

i = 1 ;
*p = 2 ;
```

Figure 3.10: Exemple de programme

$$\tau = \left| \begin{array}{l} \{\texttt{p == \&i}\}/[\texttt{i} \leftarrow \texttt{2}] \\ \{\texttt{p != \&i}\}/[\texttt{i} \leftarrow \texttt{1}, \texttt{*p} \leftarrow \texttt{2}] \end{array} \right.$$

Ceci pose tout de suite le problème de la consistance : qu'advient-il s'il existe une configuration mémoire telle qu'aucune garde ne soit vraie, ou encore si deux gardes différentes sont vraies ? Nous sommes amenés à généraliser la définition de la consistance aux transformations :

Définition 3.15 *La représentation d'une transformation* $\tau = \{\langle \gamma_i, \sigma_i \rangle \in \mathcal{G} \times \mathcal{S} / 1 \leq i \leq n\}$ *est consistante si et seulement si :*

$$\forall M \in \mathcal{M}, \exists i (1 \leq i \leq n) \text{ } tel\ que\ \gamma_i(M) = vrai$$

$$et$$

$$\forall M \in \mathcal{M}, \gamma_i(M) = vrai \Longrightarrow \Big(\forall \langle l_j, r_j \rangle \in \sigma_i, \forall \langle l_k, r_k \rangle \in \sigma_i, L(l_j, M) = L(l_k, M) \Longrightarrow j = k\Big)$$

$$et$$

$$\forall M \in \mathcal{M}, \exists i \neq j, \gamma_i(M) = vrai\ et\ \gamma_j(M) = vrai \Longrightarrow \sigma_i(M) = \sigma_j(M)$$

La première condition exprime qu'il doit y avoir au moins une garde vraie. La deuxième condition exprime la consistance de la substitution associée à la garde vraie, sous réserve que la mémoire soit dans cette configuration précise. La troisième condition exprime que s'il y a plus d'une garde possible pour une configuration mémoire, les substitutions doivent être équivalentes.

Il est temps de prendre un exemple. Considérons le segment de programme de la figure 3.10 :

Notre but dans cet exemple n'est pas de déflorer le sujet de la section suivante (3.4), mais de montrer comment on passe d'une substitution non consistante à une transformation consistante.

La substitution que l'on trouve naïvement pour représenter ce segment de programme correspond à $\sigma = [\mathtt{i} \leftarrow \mathtt{1}, \mathtt{*p} \leftarrow \mathtt{2}]$. Cette substitution n'est pas consistante, car si $M(\langle \mathtt{\_p}, 0 \rangle) = \langle \mathtt{\_i}, 0 \rangle$, les deux lhs sont égaux. Cette substitution ne modélise donc pas ce programme.

Nous voulons donc en faire une transformation consistante. La configuration mémoire exposant l'ambiguïté est celle où `p` pointe sur `i`. La garde n'est que l'expression concrète de cette configuration, c'est à dire {`p == &i`}. Il ne nous reste plus qu'à écrire la représentation de la transformation :

$$\tau = \left| \begin{array}{l} \{\mathtt{p\ ==\ \&i}\}/[\mathtt{i} \leftarrow \mathtt{2}] \\ \{\mathtt{p\ !=\ \&i}\}/[\mathtt{i} \leftarrow \mathtt{1}, \mathtt{*p} \leftarrow \mathtt{2}] \end{array} \right.$$

Si on abstrait τ en une transformation T, nous avons :

$$\begin{array}{rlll} T: & \mathcal{M} & \longrightarrow \mathcal{M} & \\ & M & \longmapsto T(M) = & \left\{ \begin{array}{ll} \text{si } M(\langle \mathtt{\_p}, 0 \rangle) = \langle \mathtt{\_i}, 0 \rangle & \text{alors } [\mathtt{i} \leftarrow \mathtt{2}] \\ & \text{sinon } [\mathtt{i} \leftarrow \mathtt{1}, \mathtt{*p} \leftarrow \mathtt{2}] \end{array} \right. \end{array}$$

Nous reviendrons plus en détail dans la section suivante (3.4) sur les opérations d'abstraction et de concrétisation.

Conclusion

Nous avons défini un modèle pour l'étude des segments de programme. Il s'agit des transformations, c'est à dire de fonctions transformant la mémoire, que l'on représente par un ensemble de gardes et substitutions.

La propriété essentielle que doit posséder une représentation de transformation est la consistance, condition qui permet l'abstraction de la forme concrète. À une configuration mémoire, notre représentation fait correspondre une et une seule configuration mémoire.

3.3.3 Modélisation d'une affectation

Notre modèle étant défini, nous pouvons modéliser une affectation $l = r$ par la transformation :

$$
\begin{array}{lll}
T: & \mathcal{M} & \longrightarrow \mathcal{M} \\
 & M & \longmapsto T(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = L(l, M) & \text{alors } R(r, M) \\ & \text{sinon } M(a) \end{array} \right. \end{array} \right)
\end{array}
$$

que nous représentons par la forme concrète :

$$\tau = \{\text{vrai}\}/[l \leftarrow r]$$

La garde est toujours vraie, puisqu'il n'y a pas besoin de lever une ambiguïté entre deux lhs, étant donné qu'il n'y en a qu'un. La substitution associée est analogue à celle que nous avons vue lorsque nous avons introduit les substitutions.

La propriété importante est :

Propriété 3.3 *La représentation d'une transformation associée à* une *affectation est consistante.*

La preuve est évidente, par construction.

3.4 Séquences

En supposant que nous ayons le segment de programme de la figure 3.11, les transformations associées aux deux instructions sont :

$$
\begin{array}{llll}
T_1 : & \mathcal{M} & \longrightarrow \mathcal{M} & \\
& M & \longmapsto T_1(M) = & \left(\begin{array}{lll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = \langle \_\texttt{p}, 0\rangle \text{ alors} & \langle \_\texttt{i}, 0\rangle \\ \text{sinon} & M(a) \end{array} \right. & \end{array} \right)
\end{array}
$$

et

$$
\begin{array}{llll}
T_2 : & \mathcal{M} & \longrightarrow \mathcal{M} & \\
& M & \longmapsto T_2(M) = & \left(\begin{array}{lll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = M(\langle \_\texttt{p}, 0\rangle) \text{ alors} & 2 \\ \text{sinon} & M(a) \end{array} \right. & \end{array} \right)
\end{array}
$$

```
int *p, i ;

p = &i ;
*p = 2 ;
```

Figure 3.11: Exemple de programme

Comment, à partir de ces deux transformations élémentaires T_1 et T_2, arriver à la transformation T_3 qui nous apprendra que les variables `i` et `p` reçoivent respectivement 2 et l'adresse de `i` ? C'est ce que nous proposons d'étudier maintenant.

3.4.1 Composition de transformations

Nous avons mentionné qu'une transformation T est une fonction faisant passer d'un état mémoire M à un état mémoire $M' = T(M)$. Lorsqu'on exécute deux instructions en séquence, on opère deux modifications sur la mémoire. Plus précisément, la deuxième instruction modifie la mémoire déjà modifiée par la première. La transformation d'une séquence de deux instructions correspond donc à la composition des transformations des deux instructions. Autrement dit :

$$T_3 = T_1 \circ T_2$$

où l'opérateur $\circ$ représente la composition de fonctions, c'est à dire $(f \circ g)(x) = g(f(x))$.

En reprenant notre exemple, calculons $T_3 = T_1 \circ T_2$:

$$
\begin{aligned}
T_3(M) &= (T_1 \circ T_2)(M) \\
&= T_2(T_1(M)) \\
&= \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{l} \text{si } a = (T_1(M))(\langle \texttt{\_p}, 0 \rangle) \text{ alors } 2 \\ \text{sinon } (T_1(M))(a) \end{array} \right. \end{array} \right) \\
&= \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{l} \text{si } a = \langle \texttt{\_i}, 0 \rangle \text{ alors } 2 \\ \text{sinon } (T_1(M))(a) \end{array} \right. \end{array} \right) \\
&= \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{l} \text{si } a = \langle \texttt{\_i}, 0 \rangle \text{ alors } 2 \\ \text{sinon } \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{l} \text{si } a = \langle \texttt{\_p}, 0 \rangle \text{ alors } \langle \texttt{\_i}, 0 \rangle \\ \text{sinon } M(a) \end{array} \right. \end{array} \right) (a) \end{array} \right. \end{array} \right) \\
&= \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{l} \text{si } a = \langle \texttt{\_i}, 0 \rangle \text{ alors } 2 \\ \text{sinon } \left\{ \begin{array}{l} \text{si } a = \langle \texttt{\_p}, 0 \rangle \text{ alors } \langle \texttt{\_i}, 0 \rangle \\ \text{sinon } M(a) \end{array} \right. \end{array} \right. \end{array} \right) \\
&= \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{ll} \text{si } a = \langle \texttt{\_i}, 0 \rangle \text{ alors} & 2 \\ \text{sinon si } a = \langle \texttt{\_p}, 0 \rangle \text{ alors} & \langle \texttt{\_i}, 0 \rangle \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right)
\end{aligned}
$$

Au cours du calcul, nous n'avons pas détaillé le développement de $(T_1(M))(\langle \texttt{\_p}, 0 \rangle)$. En effet, nous avons directement évalué la fonction T_1 avec l'argument $\langle \texttt{\_p}, 0 \rangle$ afin d'alléger l'écriture.

La forme finale du calcul de $T_1 \circ T_2$ nous donne bien le résultat cherché.

Reprenons maintenant un second exemple (voir figure 3.10) plus complexe, non pas dans sa rédaction ou son calcul, mais par les conséquences qu'il impliquera. Les transformations associées aux deux instructions sont :

$$T_4 : \begin{array}{lll} \mathcal{M} & \longrightarrow \mathcal{M} \\ M & \longmapsto T_4(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = \langle \mathtt{\_i}, 0 \rangle \text{ alors} & 1 \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right) \end{array}$$

et

$$T_5 : \begin{array}{lll} \mathcal{M} & \longrightarrow \mathcal{M} \\ M & \longmapsto T_5(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = M(\langle \mathtt{\_p}, 0 \rangle) \text{ alors} & 2 \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right) \end{array}$$

Notons que $T_5 = T_2$ de l'exemple précédent, ce qui était à prévoir puisque les instructions correspondantes de chaque exemple sont identiques.

Calculons maintenant la transformation $T_6 = T_4 \circ T_5$ associée à l'exemple entier :

$$\begin{array}{rcl} T_6(M) & = & (T_4 \circ T_5)(M) \\ & = & T_5(T_4(M)) \\ & = & \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{l} \text{si } a = (T_4(M))(\langle \mathtt{\_p}, 0 \rangle) \text{ alors } 2 \\ \text{sinon } (T_4(M))(a) \end{array} \right. \end{array} \right) \\ & = & \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{l} \text{si } a = M(\langle \mathtt{\_p}, 0 \rangle) \text{ alors } 2 \\ \text{sinon } (T_4(M))(a) \end{array} \right. \end{array} \right) \\ & = & \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{l} \text{si } a = M(\langle \mathtt{\_p}, 0 \rangle) \text{ alors } 2 \\ \text{sinon } \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{l} \text{si } a = \langle \mathtt{\_i}, 0 \rangle \text{ alors } 1 \\ \text{sinon } M(a) \end{array} \right. \end{array} \right) (a) \end{array} \right. \end{array} \right) \\ & = & \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{l} \text{si } a = M(\langle \mathtt{\_p}, 0 \rangle) \text{ alors } 2 \\ \text{sinon } \left\{ \begin{array}{l} \text{si } a = \langle \mathtt{\_i}, 0 \rangle \text{ alors } 1 \\ \text{sinon } M(a) \end{array} \right. \end{array} \right. \end{array} \right) \end{array}$$

$$= \left(\begin{array}{lll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = M(\langle \texttt{\_p}, 0 \rangle) \text{ alors} & 2 \\ \text{sinon si } a = \langle \texttt{\_i}, 0 \rangle \text{ alors} & 1 \\ \text{sinon} & M(a) \end{array} \right. & \end{array} \right)$$

Que pouvons-nous observer sur cet exemple ? Nous avons vu ce segment de programme à plusieurs reprises dans des sections précédentes (notamment en 3.3.2) et nous avons conclu qu'il recelait des ambiguïtés qui nécessitaient l'introduction de gardes avec les substitutions pour représenter les transformations. Mais en développant la composition des transformations, nous pouvons constater que la nouvelle transformation T_5 ne contient aucune ambiguïté. En effet, la nouvelle mémoire se lit comme :

- si l'adresse correspond à $M(\langle \texttt{\_p}, 0 \rangle)$, c'est à dire si on accède à cette adresse par l'expression `*p`, le résultat est 2. Si `p = &i` et si on accède à cette adresse par l'expression `i`, le résultat est toujours 2 ;
- si l'adresse correspond à $\langle \texttt{\_i}, 0 \rangle$, ce ne peut être que si `p != &i` puisque nous sommes dans le cas *sinon* du cas précédent, et donc le résultat est 1 car la deuxième instruction n'a pas touché à la case mémoire $\langle \texttt{\_i}, 0 \rangle$;
- si l'adresse ne correspond à aucune des deux ci-dessus, le résultat correspond au contenu de la case mémoire adressée avant l'exécution de ces deux instructions.

Si le problème de consistance se pose pour les représentations des transformations, il n'en va pas de même pour les transformations elles-mêmes.

3.4.2 Forme abstraite d'une transformation

Généralisons la forme que nous avons vue lors des exemples précédents :

Propriété 3.4 *La forme générale (forme abstraite) d'une transformation est une fonction :*

$$\begin{array}{lll} T: & \mathcal{M} & \longrightarrow \mathcal{M} \\ & M & \longmapsto T(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \textit{si } a = f_1(M) \wedge g_1(M) \textit{ alors} & h_1(M) \\ \quad \vdots & \\ \textit{sinon si } a = f_n(M) \wedge g_n(M) \textit{ alors} & h_n(M) \\ \textit{sinon} & M(a) \end{array} \right. \end{array} \right) \end{array}$$

où, pour $1 \leq i \leq n$, f_i est une fonction dont le résultat est élément de $\mathcal{A}$, g_i est une fonction dont le résultat est élément de $\mathcal{B}$ et h_i est une fonction dont le résultat est élément de $\mathcal{A} + \mathcal{V}$

Prenons l'exemple de la figure 3.12. La première transformation s'écrit :

```
int tab [5], i, *q ;

*q = 1 ;
*(tab + i) = 2 ;
```

Figure 3.12: Exemple de programme

$$
\begin{array}{llll}
T_1 : & \mathcal{M} & \longrightarrow \mathcal{M} \\
& M & \longmapsto T_1(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = M(\langle \texttt{\_q}, 0 \rangle) \text{ alors} & 1 \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right)
\end{array}
$$

c'est à dire que $n = 1$, $f_1(M) = M(\langle \texttt{\_q}, 0\rangle)$, $g_1(M) =$ vrai et $h_1(M) = 1$.

La deuxième transformation est légèrement plus complexe dans l'expression de f_1 :

$$
\begin{array}{llll}
T_2 : & \mathcal{M} & \longrightarrow \mathcal{M} \\
& M & \longmapsto T_2(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = \langle \texttt{\_tab}, 0 \rangle + M(\langle \texttt{\_i}, 0 \rangle) \text{ alors} & 2 \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right)
\end{array}
$$

ici encore, $n = 1$. En revanche, $f_1(M) = \langle \texttt{\_tab}, 0\rangle + M(\langle \texttt{\_i}, 0\rangle)$, $g_1(M) =$ vrai et $h_1(M) = 2$.

Et la composition $T_1 \circ T_2$ s'écrit (nous épargnons au lecteur les fastidieux calculs) :

$$
(T_1 \circ T_2)(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = \langle \texttt{\_tab}, 1 \rangle \wedge \langle \texttt{\_i}, 0 \rangle = M(\langle \texttt{\_q}, 0 \rangle) \text{ alors} & 2 \\ \text{sinon si } a = \langle \texttt{\_tab}, 0 \rangle + M(\langle \texttt{\_i}, 0 \rangle) \text{ alors} & 2 \\ \text{sinon si } a = M(\langle \texttt{\_q}, 0 \rangle) \text{ alors} & 1 \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right)
$$

Preuve

Nous allons suivre un raisonnement par récurrence : nous allons prouver la propriété pour la transformation minimum, c'est à dire l'affectation, puis nous allons vérifier que la composition conserve la propriété.

Par définition (voir 3.3.3), cette forme est celle de la transformation associée à une affectation $l = r$ avec $n = 1$, $f_1(M) = L(l, M)$, $g_1(M) =$ vrai et $h_1 = R(r, M)$.

Le cas général est bien plus complexe. La démonstration passe par une énumération exhaustive de la syntaxe des expressions. Ne voulant pas lasser le lecteur par une démonstration longue et indigeste, nous définissons pour la circonstance une syntaxe réduite des expressions (voir figure 3.13) qui nous servira à esquisser la démonstration complète. Nous laisserons au lecteur le soin de traiter la syntaxe entière.

expression-logique	::= expression = expression
	| expression-logique ∧ expression-logique
	| **vrai**
	| **faux**
expression	::= adresse
	| constante
	| $M($ expression $)$
	| expression + expression

Figure 3.13: Syntaxe réduite des expressions

Cette syntaxe nous permet de formaliser les expressions f_i, g_i et h_i : f_i et h_i respectent la syntaxe des *expressions*, g_i respecte la syntaxe des *expressions logiques*.

Nous allons donc supposer que la transformation a la forme citée ci-dessus. Nous pourrons composer deux transformations T et T', nous détaillerons chaque sous-expression de la composition, en donnerons la forme, puis nous les reporterons dans la composition de T et T' pour vérifier *in fine* qu'elle respecte la forme désirée.

Composons donc T et T' :

$$
\begin{aligned}
(T \circ T')(M) = \;& T'(T(M)) \\
= \;& \left(\begin{array}{lcl}
\mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\
a & \longmapsto & \begin{cases}
\text{si } a = f'_1(T(M)) \wedge g'_1(T(M)) \text{ alors} & h'_1(T(M)) \\
\qquad \vdots & \\
\text{sinon si } a = f'_{n'}(T(M)) \wedge g'_{n'}(T(M)) \text{ alors} & h'_{n'}(T(M)) \\
\text{sinon} & (T(M))(a)
\end{cases}
\end{array} \right)
\end{aligned}
$$

Concentrons-nous sur la ligne : sinon si $a = f_i'(T(M)) \wedge g_i'(T(M))$ alors $h_i'(T(M))$ et étudions la forme des fonctions f_i', g_i' et h_i' lorsque leur est appliquée la mémoire $T(M)$. En réalité, il nous suffit de connaître les formes d'une expression (pour f_i' et h_i') et d'une expression logique (pour g_i') lorsqu'on les évalue en présence d'une mémoire $T(M)$ qui a une forme conditionnelle.

Étudions d'abord la forme des expressions, pour prouver qu'elles sont de la forme :

$$\begin{cases} \text{si } a_1(M) \text{ alors} & b_1(M) \\ \quad\vdots & \\ \text{sinon si } a_m(M) \text{ alors} & b_m(M) \end{cases}$$

où les a_i sont des expressions logiques et les b_i sont des expressions (sans inclusion de conditionnelles). Pour cela, considérons les différents cas de la syntaxe des expressions :

- adresse

  Une adresse est constante. Son évaluation en présence d'une mémoire $T(M)$ ne varie donc pas. Sa forme correspond bien à ce qu'on veut prouver, si on considère que $a_1(M) =$ vrai et $b_1(M) =$ adresse.

- $M(e)$ où e est une expression

  La mémoire considérée ici est $T(M)$. Si nous développons, cela donne :

$$\begin{aligned}
&(T(M))(e) = \\
&= \left(\begin{array}{lll} \mathcal{M} & \longrightarrow \mathcal{M} & \\ M & \longmapsto & \left(\begin{array}{lll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ a & \longmapsto & \begin{cases} \text{si } a = f_1(M) \wedge g_1(M) \text{ alors} & h_1(M) \\ \quad\vdots & \\ \text{sinon si } a = f_n(M) \wedge g_n(M) \text{ alors} & h_n(M) \\ \text{sinon} & M(a) \end{cases} \end{array} \right) \end{array} \right)(e) \\
&= \begin{cases} \text{si } e = f_1(M) \wedge g_1(M) \text{ alors} & h_1(M) \\ \quad\vdots & \\ \text{sinon si } e = f_n(M) \wedge g_n(M) \text{ alors} & h_n(M) \\ \text{sinon} & M(e) \end{cases}
\end{aligned}$$

  Isolons la ligne : sinon si $e = f_i(M) \wedge g_i(M)$ alors $h_i(M)$. L'expression $e = f_i(M)$ est une expression logique car e est une expression, $f_i(M)$ aussi par hypothèse, donc $e = f_i(M) \wedge g_i(M)$ est une expression logique que nous pouvons nommer $a_i(M)$. Quant à $h_i(M)$, c'est une expression (par hypothèse) que nous pouvons renommer $b_i(M)$. La dernière ligne est particulière car $a_m(M) =$ vrai $b_m(M) = M(e)$.

- $e_1 + e_2$ où e_1 et e_2 sont des expressions

  Il s'agit donc de réaliser l'opération :

$$\left\{\begin{array}{ll} \text{si } a_1^1(M) \text{ alors} & b_1^1(M) \\ \quad\vdots & \\ \text{sinon si } a_{n_1}^1(M) \text{ alors} & b_{n_1}^1(M) \end{array}\right. + \left\{\begin{array}{ll} \text{si } a_1^2(M) \text{ alors} & b_1^2(M) \\ \quad\vdots & \\ \text{sinon si } a_{n_2}^2(M) \text{ alors} & b_{n_2}^2(M) \end{array}\right.$$

  qui signifie qu'il faut trouver le premier $a_i^1(M)$ vrai et le premier $a_j^2(M)$ vrai, puis faire la somme $b_i^1(M)+b_j^2(M)$. La ligne générale sera donc : si $a_i^1(M) \wedge a_j^2(M)$ alors $b_i^1(M)+b_j^2(M)$. Mais il faut respecter un ordre dans les lignes, car nous devons être sûrs de trouver i et j et non d'autres indices sinon le résultat risque d'être faux.

  Nous introduisons pour respecter cet ordre deux fonctions d'indices φ_1 et φ_2 qui nous permettent d'écrire $e_1 + e_2$ en présence de $T(M)$:

$$\left\{\begin{array}{ll} \text{si } a_{\varphi_1(1)}^1(M) \wedge a_{\varphi_2(1)}^2(M) \text{ alors} & b_{\varphi_1(1)}^1(M) + b_{\varphi_2(1)}^2 \\ \quad\vdots & \\ \text{sinon si } a_{\varphi_1(n_1 n_2)}^1(M) \wedge a_{\varphi_2(n_1 n_2)}^2(M) \text{ alors} & b_{\varphi_1(n_1 n_2)}^1(M) + b_{\varphi_2(n_1 n_2)}^2 \end{array}\right.$$

  Les valeurs de $\varphi_1(n)$ et $\varphi_2(n)$ nous importent peu (bien qu'elles soient intéressantes à calculer). En revanche, la forme de l'expression finale est intéressante, car si nous posons $a_i(M) = a_{\varphi_1(i)}^1(M) \wedge a_{\varphi_2(i)}^2(M)$ et $b_i(M) = b_{\varphi_1(i)}^1(M) + b_{\varphi_2(i)}^2$, nous nous ramenons à la forme cherchée.

Nous avons donc vu la forme générale des expressions lorsqu'elles sont évaluées en présence d'une transformation.

On prouve par le même raisonnement que les expressions logiques, dans les mêmes circonstances, sont écrites sous la forme :

$$\left\{\begin{array}{ll} \text{si } a_1(M) \text{ alors} & b_1(M) \\ \quad\vdots & \\ \text{sinon si } a_m(M) \text{ alors} & b_m(M) \end{array}\right.$$

où les a_i et les b_i sont des expressions logiques (sans inclusion de conditionnelles).

Nous avons donc maintenant les écritures des fonctions $f_i'(T(M))$, $g_i'(T(M))$ et $h_i'(T(M))$. Reportons-les dans $(T \circ T')(M)$, et plus précisément dans la ligne d'indice i :

$$\text{sinon si } a = f_i'(T(M)) \wedge g_i'(T(M)) \text{ alors } h_i'(T(M))$$

$$\text{sinon si } a = \begin{cases} \text{si } a_1^f(M) \text{ alors } b_1^f(M) \\ \quad\vdots \\ \text{sinon si } a_{n_f}^f(M) \text{ alors } b_{n_f}^f(M) \end{cases} \wedge \begin{cases} \text{si } a_1^g(M) \text{ alors } b_1^g(M) \\ \quad\vdots \\ \text{sinon si } a_{n_g}^g(M) \text{ alors } b_{n_g}^g(M) \end{cases} \text{ alors } h_i'(T(M))$$

$$\text{sinon} \begin{cases} \text{si } a = b_{\varphi_1(1)}^f(M) \wedge a_{\varphi_1(1)}^f(M) \wedge a_{\varphi_2(1)}^g(M) \wedge b_{\varphi_2(1)}^g(M) \\ \quad \text{alors } h_1'(T(M)) \\ \quad\vdots \\ \text{sinon si } a = b_{\varphi_1(n_f n_g)}^f(M) \wedge a_{\varphi_1(n_f n_g)}^f(M) \wedge a_{\varphi_2(n_f n_g)}^g(M) \wedge b_{\varphi_2(n_f n_g)}^g(M) \\ \quad \text{alors } h_1'(T(M)) \end{cases}$$

Posons maintenant $g_j''(M) = a_{\varphi_1(j)}^f(M) \wedge a_{\varphi_2(j)}^g(M) \wedge b_{\varphi_2(j)}^g(M)$ (avec $1 \leq j \leq n_f n_g$), et reportons maintenant $h_i'(M)$ dans la ligne j de la ligne i de $(T \circ T')(M)$:

$$\text{sinon si } a = b_{\varphi_1(j)}^f(T(M)) \wedge g_j''(T(M)) \text{ alors } h_i'(T(M))$$

$$\text{sinon si } a = b_{\varphi_1(j)}^f(T(M)) \wedge g_j''(T(M)) \text{ alors } \begin{cases} \text{si } a_1^h(M) \text{ alors } b_1^h(M) \\ \quad\vdots \\ \text{si } a_{n_h}^h(M) \text{ alors } b_{n_h}^h(M) \end{cases}$$

$$\text{sinon} \begin{cases} \text{si } a = b_{\varphi_1(j)}^f(T(M)) \wedge g_j''(T(M)) \wedge a_1^h(M) \text{ alors } b_1^h(M) \\ \quad\vdots \\ \text{sinon si } a = b_{\varphi_1(j)}^f(T(M)) \wedge g_j''(T(M)) \wedge a_{n_h}^h(M) \text{ alors } b_{n_h}^h(M) \end{cases}$$

Si nous posons $g_k'''(M) = g_j''(T(M)) \wedge a_k^h(M)$, nous constatons que chaque ligne (exceptée la dernière) de $(T \circ T')(M)$ est :

$$\text{sinon si } a = b_z^f(M) \wedge g_z'''(M) \text{ alors } b_z^h(M)$$

avec $1 \leq z \leq n_f n_g n_h$.

Nous avons donc retrouvé après de multiples développements la ligne générale de la forme de la transformation que nous cherchons à démontrer.

Il reste à développer la ligne finale de $(T \circ T')(M)$, c'est à dire la ligne : sinon $(T(M))(a)$; il suffit de reporter la définition de $T(M)$, ce qui nous donne n lignes (les n lignes du développement de $(T(M))(a)$) supplémentaires à ajouter aux lignes générales de la forme.

La dernière ligne de $(T \circ T')(M)$ est enfin la dernière ligne du développement de $(T(M))(a)$, c'est à dire sinon $M(a)$ et nous retrouvons ainsi l'intégralité de la forme que nous cherchions.

□

3.4.3 Concrétisation

Nous avons vu jusqu'ici les transformations d'une part, et les représentations des transformations d'autre part. Comment passer des transformations à leur représentation ? C'est ce que nous proposons de détailler dans ce paragraphe.

Supposons une transformation T. Le problème est de la représenter par un ensemble τ de doublets $\langle \gamma_i, \sigma_i \rangle \in \mathcal{G} \times \mathcal{S}$. Comment passer d'une forme fonctionnelle, donnant pour chaque adresse la modification apportée à la mémoire, à un ensemble de substitutions gardées ?

Méthode

Pour obtenir les gardes, il faut une détermination plus précise de la condition $g_i(M)$ de chaque ligne de la forme abstraite. Sachant que chaque ligne commence par *sinon*, il n'est pas faux de dire que la ligne d'indice i n'est abordée que si les tests de toutes les lignes d'indice j ($j < i$) ont échoué. Ceci nous permet de reporter la négation de tous les tests précédents dans la ligne i :

$$\text{sinon si } a = f_i(M) \wedge g_i(M) \wedge \left(\bigwedge_{j<i} \neg(a = f_j(M) \wedge g_j(M)) \right) \text{ alors } h_i(M) \tag{3.1}$$

Oublions la ligne finale “sinon $M(a)$” car elle ne nous apporte rien pour la concrétisation. Cet ajout de termes nous permet de nous affranchir du “*sinon*” : celui-ci assurait la consistance de la transformation en sélectionnant le premier terme vrai, c'est à dire sans souffrir l'ambiguïté. Dorénavant, aucune ambiguïté ne peut plus surgir, nous pouvons retirer le “*sinon*” :

$$\text{si } a = f_i(M) \wedge g_i(M) \wedge \left(\bigwedge_{j<i} \neg(a = f_j(M) \wedge g_j(M)) \right) \text{ alors } h_i(M) \tag{3.2}$$

Le premier terme impose $a = f_i(M)$. Nous pouvons donc substituer a par sa valeur dans tous les autres termes :

$$\text{si } a = f_i(M) \wedge g_i(M) \wedge \left(\bigwedge_{j<i} \neg(f_i(M) = f_j(M) \wedge g_j(M)) \right) \text{ alors } h_i(M) \tag{3.3}$$

Explicitons maintenant la négation des termes précédant i :

$$\text{si } a = f_i(M) \wedge g_i(M) \wedge \left(\bigwedge_{j<i} (f_i(M) \neq f_j(M) \vee \neg g_j(M)) \right) \text{ alors } h_i(M) \tag{3.4}$$

L'intégration de la négation dans les clauses j a pour effet d'introduire l'opérateur $\vee$ ("*ou*"). Nous ne savons pas le prendre en compte, donc faisons-le disparaître ! En effet, nous pouvons éclater une condition "*ou*" sur plusieurs lignes :

$$\text{si } x \vee y \text{ alors } z \Longleftrightarrow \begin{cases} \text{si } x \wedge y \text{ alors } z \\ \text{si } x \wedge \neg y \text{ alors } z \\ \text{si } \neg x \wedge y \text{ alors } z \end{cases} \tag{3.5}$$

Nous pouvons donc éclater les $i-1$ opérateurs $\vee$ (*ou*) en remplaçant la ligne d'indice i par 3^{i-1} lignes. D'autre part, certains termes $g_j(M)$ recèlent des opérateurs $\wedge$ (*et*), donc les expressions $\neg g_j(M)$ contiennent elles-aussi des opérateurs "*ou*" ; il faut donc leur appliquer le même traitement. Ne connaissant ni le nombre de g_j contenant des opérateurs "*ou*", ni le nombre d'opérateurs pour chaque g_j, nous ne pouvons calculer exactement le nombre de lignes dans T une fois toutes ces duplications traitées. La seule chose que nous pouvons affirmer est que ce nombre est important en regard du nombre original de lignes.

Nous arrivons donc à une nouvelle écriture où chaque ligne est autonome : elle ne dépend pas pour sa sélection des précédentes, l'ordre d'écriture des lignes n'est plus important. La transformation s'écrit donc avec le système :

$$\begin{cases} \text{si } a = \alpha_1(M) \wedge \left(\bigwedge_{1 \leq j \leq m_1} \beta_1^j(M) \right) \text{ alors } \rho_1(M) \\ \qquad \vdots \\ \text{si } a = \alpha_m(M) \wedge \left(\bigwedge_{1 \leq j \leq m_m} \beta_m^j(M) \right) \text{ alors } \rho_m(M) \end{cases} \tag{3.6}$$

où, avec la syntaxe vue en figure 3.13 :

- chaque α_i est une expression ;
- chaque β_i^j est de la forme $e_1 = e_2$ ou $e_1 \neq e_2$, avec e_1 et e_2 étant des expressions (fonctions de M) ;
- chaque ρ_i est une expression.

Intuitivement, on sent que les substitutions vont être construites à partir de "choses" du genre de $\alpha_i(M) \leftarrow \rho_i(M)$, et les gardes à partir de $\bigwedge_{1 \leq j \leq m_i} \beta_i^j(M)$. Cependant, il faut réunir plusieurs

$\alpha_i(M) \leftarrow \rho_i(M)$ suivant des gardes compatibles. Pour cela, nous allons récapituler l'ensemble des paires d'expressions $\mathcal{P} = \{(e_1, e_2) / \exists i, j \text{ tel que } \beta_i^j = (e_1 = e_2) \text{ ou } \beta_i^j = (e_1 \neq e_2)\}$. L'expérience prouve que la cardinalité de $\mathcal{P}$ est souvent très inférieure à $\sum_{i=1}^m m_i$.

Chaque élément (e_1, e_2) de $\mathcal{P}$ nous permet de générer une expression $e_1 = e_2 \vee e_1 \neq e_2$ dont la valeur est toujours vraie, quelles que soient les expressions e_1 et e_2. En alignant toutes ces expressions vraies, nous obtenons l'expression :

$$\bigwedge_{(e_1,e_2)\in\mathcal{P}} (e_1 = e_2 \vee e_1 \neq e_2)$$

elle-même toujours vraie. Si nous introduisons cette expression toujours vraie dans chaque ligne de notre système, nous ne changeons pas sa validité, donc la ligne générale s'écrit :

$$\text{si } a = \alpha_i(M) \wedge \left(\bigwedge_{1\leq j\leq m_i} \beta_i^j(M) \right) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}} (e_1 = e_2 \vee e_1 \neq e_2) \right) \text{ alors } \rho_i(M) \tag{3.7}$$

Il suffit de dupliquer les lignes pour faire disparaître les "*ou*" suivant la méthode déjà utilisée plus haut, pour donner après simplification triviale (puisque les (e_1, e_2) sont définies par rapport aux β_i^j) :

$$\text{si } a = \alpha_i(M) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_i'} (e_1 = e_2) \right) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_i''} (e_1 \neq e_2) \right) \text{ alors } \rho_i(M) \tag{3.8}$$

où $\mathcal{P}_i' + \mathcal{P}_i'' = \mathcal{P}$.

Il ne reste plus qu'à regrouper les lignes autour des gardes communes pour donner le système correspondant toujours à notre transformation initiale :

$$\left\{ \begin{array}{ll}
\text{si } \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_1'}(e_1 = e_2)\right) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_1''}(e_1 \neq e_2)\right) \wedge a = \alpha_1^1(M) \text{ alors} & \rho_1^1(M) \\
\quad \vdots & \\
\text{si } \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_1'}(e_1 = e_2)\right) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_1''}(e_1 \neq e_2)\right) \wedge a = \alpha_{n_1}^1(M) \text{ alors} & \rho_{n_1}^1(M) \\
\quad \vdots & \\
\text{si } \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_n'}(e_1 = e_2)\right) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_n''}(e_1 \neq e_2)\right) \wedge a = \alpha_1^n(M) \text{ alors} & \rho_1^n(M) \\
\quad \vdots & \\
\text{si } \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_n'}(e_1 = e_2)\right) \wedge \left(\bigwedge_{(e_1,e_2)\in\mathcal{P}_n''}(e_1 \neq e_2)\right) \wedge a = \alpha_{n_n}^n(M) \text{ alors} & \rho_{n_n}^n(M)
\end{array} \right. \tag{3.9}$$

Notons que cette forme est équivalente à la forme abstraite initiale puisqu'il suffit de remettre des "sinon" devant toutes les lignes (sauf la première bien sûr) et la ligne "sinon $M(a)$" à la fin.

Le dernier problème à résoudre est l'écriture des gardes et des substitutions à partir d'éléments de $\mathcal{X}$, c'est à dire à partir d'expressions concrètes.

Nous proposons pour cela de construire des fonctions L^{-1} et R^{-1}, symétriques aux fonctions L et R vues dans la section 3.3.1, associant à une expression selon la syntaxe de la figure 3.13 (c'est à dire une fonction de $\mathcal{M}$ vers $\mathcal{A}+\mathcal{V}$) une expression de $\mathcal{X}_l$ (pour L^{-1}) ou $\mathcal{X}_r$ (pour R^{-1}).

On pourrait faire une analogie entre L (ou R) et abstraction, et L^{-1} (ou R^{-1}) et concrétisation.

Comment calculer ces nouvelles fonctions ? C'est très simple. Suivons la syntaxe des expressions :

- $L^{-1}(e) = \texttt{*}\ R^{-1}(e)$: l'évaluation de la fonction L^{-1} est réduite au plus simple ;
- $R^{-1}(\text{adresse}) = R^{-1}(\langle id, k\rangle)$: plusieurs cas sont à envisager suivant que id est un identificateur$_n$ ou une constante$_n$, et suivant la nullité de k :

id	k	$R^{-1}(\text{adresse})$
constante$_n$	$= 0$	id
constante$_n$	$\neq 0$	$\texttt{(}id\ \texttt{+}\ k\texttt{)}$
identificateur$_n$	$= 0$	$\texttt{(\&}\ id\ \texttt{)}$
identificateur$_n$	$\neq 0$	$\texttt{(\&}\ id\ \texttt{+}\ k\ \texttt{)}$

- $R^{-1}(\text{constante}) = V^{-1}(\text{constante})$ où la fonction V^{-1} est telle que $V^{-1}(V(c)) = c$;
- $R^{-1}(M(e)) = \texttt{*}\ e$;
- $R^{-1}(e_1 + e_2) = \texttt{(}\ R^{-1}(e_1)\ \texttt{+}\ R^{-1}(e_2)\ \texttt{)}$

Par exemple :

- $R^{-1}(\langle \texttt{\_i}, 0\rangle) = \texttt{(\& i)}$;
- $L^{-1}(\langle \texttt{\_i}, 0\rangle) = \texttt{* (\& i)}$, ce qui revient à l'expression `i` ;
- $R^{-1}(M(\langle \texttt{\_q}, 0\rangle)) = \texttt{* (\& q)}$, ce qui revient à `q` ;
- $L^{-1}(M(\langle \texttt{\_q}, 0\rangle)) = \texttt{* * (\& q)}$, ce qui revient à `*q` ;
- $R^{-1}(\langle \texttt{\_tab}, 0\rangle + M(\langle \texttt{\_i}, 0\rangle)) = \texttt{(tab + * (\& i))}$, ce qui revient à `tab + i` ;
- $L^{-1}(\langle \texttt{\_tab}, 0\rangle + M(\langle \texttt{\_i}, 0\rangle)) = \texttt{* (tab + * (\& i))}$, ce qui revient à `*(tab + i)`.

Nous avons donc fabriqué deux fonctions de base qui nous permettront de fabriquer les gardes et les substitutions à partir des expressions α_i, β_i^j et ρ_i. La démonstration de $L^{-1}(L(x, M)) =$

x n'est pas facile, car l'expression $x \in \mathcal{X}_l$ peut être écrite sous de nombreuses formes et nous ne savons pas passer facilement d'une forme à l'autre. En revanche, la démonstration de $L(L^{-1}(e), M) = e$ est plus facile et est laissée à la sagacité du lecteur.

Disposant maintenant de tous les éléments, la forme concrète des transformations s'écrit donc finalement en se basant sur l'équation 3.9 :

$$\tau = \left| \begin{array}{l} \{R^{-1}(e_1)\texttt{==}R^{-1}(e_2)/(e_1,e_2) \in \mathcal{P}'_1\} \cup \{R^{-1}(e_1)\texttt{!=}R^{-1}(e_2)/(e_1,e_2) \in \mathcal{P}''_1\} \\ \quad [L^{-1}(\alpha^1_1) \leftarrow R^{-1}(\rho^1_1), \ldots, L^{-1}(\alpha^1_{n_1}) \leftarrow R^{-1}(\rho^1_{n_1})] \\ \vdots \\ \{R^{-1}(e_1)\texttt{==}R^{-1}(e_2)/(e_1,e_2) \in \mathcal{P}'_n\} \cup \{R^{-1}(e_1)\texttt{!=}R^{-1}(e_2)/(e_1,e_2) \in \mathcal{P}''_n\} \\ \quad [L^{-1}(\alpha^n_1) \leftarrow R^{-1}(\rho^n_1), \ldots, L^{-1}(\alpha^n_{n_n}) \leftarrow R^{-1}(\rho^n_{n_n})] \end{array} \right. \tag{3.10}$$

Exemple

```
int i, *p, *q ;

i = 1 ;
*p = 2 ;
*q = 3 ;
```

Figure 3.14: Exemple de programme

Le moment est venu de prendre comme exemple le programme de la figure 3.14. La transformation associée donne, en composant brutalement :

$$\left\{ \begin{array}{ll} \text{si } a = 2 \wedge \langle \texttt{\_q}, 0\rangle = \langle \texttt{\_i}, 0\rangle \wedge \langle \texttt{\_i}, 0\rangle = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 3 \\ \text{sinon si } a = 1 \wedge \langle \texttt{\_q}, 0\rangle = \langle \texttt{\_i}, 0\rangle \text{ alors} & 3 \\ \text{sinon si } a = 2 \wedge \langle \texttt{\_q}, 0\rangle = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 3 \\ \text{sinon si } a = M(\langle \texttt{\_q}, 0\rangle) \text{ alors} & 3 \\ \text{sinon si } a = \langle \texttt{\_i}, 0\rangle \wedge \langle \texttt{\_i}, 0\rangle = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 2 \\ \text{sinon si } a = \langle \texttt{\_i}, 0\rangle \text{ alors} & 1 \\ \text{sinon si } a = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 2 \\ \text{sinon} & M(a) \end{array} \right. \tag{3.11}$$

Nous pouvons retirer les trois premières lignes qui ne sont jamais vraies, pour donner :

$$\begin{cases} \text{sinon si } a = M(\langle \texttt{\_q}, 0\rangle) \text{ alors} & 3 \\ \text{sinon si } a = \langle \texttt{\_i}, 0\rangle \wedge \langle \texttt{\_i}, 0\rangle = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 2 \\ \text{sinon si } a = \langle \texttt{\_i}, 0\rangle \text{ alors} & 1 \\ \text{sinon si } a = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 2 \\ \text{sinon} & M(a) \end{cases} \tag{3.12}$$

Introduisons maintenant les gardes supplémentaires de manière à retirer les "sinon" :

$$\begin{cases} \text{si } a = M(\langle \texttt{\_q}, 0\rangle) \text{ alors} & 3 \\ \text{si } a = \langle \texttt{\_i}, 0\rangle \wedge \langle \texttt{\_i}, 0\rangle = M(\langle \texttt{\_p}, 0\rangle) & \\ \qquad\qquad \wedge(\langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_q}, 0\rangle) \vee \neg \text{ vrai) alors} & 2 \\ \text{si } a = \langle \texttt{\_i}, 0\rangle & \\ \qquad\qquad \wedge(\langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_q}, 0\rangle) \vee \neg \text{ vrai)} & \\ \qquad\qquad \wedge(\langle \texttt{\_i}, 0\rangle \neq \langle \texttt{\_i}, 0\rangle \vee \langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_p}, 0\rangle)) \text{ alors} & 1 \\ \text{si } a = M(\langle \texttt{\_p}, 0\rangle) & \\ \qquad\qquad \wedge(M(\langle \texttt{\_p}, 0\rangle) \neq M(\langle \texttt{\_q}, 0\rangle) \vee \neg\text{vrai)} & \\ \qquad\qquad \wedge((M(\langle \texttt{\_p}, 0\rangle) \neq \langle \texttt{\_i}, 0\rangle) \vee \langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_p}, 0\rangle)) & \\ \qquad\qquad \wedge((M(\langle \texttt{\_p}, 0\rangle) \neq \langle \texttt{\_i}, 0\rangle) \vee \neg \text{ vrai) alors} & 2 \end{cases} \tag{3.13}$$

Les clauses "$\vee\neg$ vrai" sont explicitées afin de respecter l'équation 3.1. Pour alléger l'écriture, nous allons les retirer, puisqu'elles ne sont jamais vraies. De même, la clause "$\langle \texttt{\_i}, 0\rangle \neq \langle \texttt{\_i}, 0\rangle$" (à la troisième ligne) est toujours fausse et peut donc être retirée du "*ou*". Enfin, la dernière ligne contient trois fois la même clause "$\langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_p}, 0\rangle$" et peut être réduite aisément. Ces trois simplifications triviales donnent :

$$\begin{cases} \text{si } a = M(\langle \texttt{\_q}, 0\rangle) \text{ alors} & 3 \\ \text{si } a = \langle \texttt{\_i}, 0\rangle \wedge \langle \texttt{\_i}, 0\rangle = M(\langle \texttt{\_p}, 0\rangle) \wedge \langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_q}, 0\rangle) \text{ alors} & 2 \\ \text{si } a = \langle \texttt{\_i}, 0\rangle \wedge \langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_q}, 0\rangle) \wedge \langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & 1 \\ \text{si } a = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_p}, 0\rangle) \neq M(\langle \texttt{\_q}, 0\rangle) \wedge \langle \texttt{\_i}, 0\rangle \neq M(\langle \texttt{\_p}, 0\rangle) \text{ vrai) alors} & 2 \end{cases} \tag{3.14}$$

Construisons l'ensemble des paires d'expressions $\mathcal{P}$ associé à cette transformation :

$$\mathcal{P} = \{(\langle \texttt{\_i}, 0\rangle, M(\langle \texttt{\_p}, 0\rangle)), (\langle \texttt{\_i}, 0\rangle, M(\langle \texttt{\_q}, 0\rangle)), (M(\langle \texttt{\_p}, 0\rangle), M(\langle \texttt{\_q}, 0\rangle))\}$$

Introduisons pour chaque ligne l'expression "$\wedge(\langle \texttt{\_i},0\rangle = M(\langle \texttt{\_p},0\rangle) \vee \langle \texttt{\_i},0\rangle \neq M(\langle \texttt{\_p},0\rangle)) \wedge (\langle \texttt{\_i},0\rangle = M(\langle \texttt{\_q},0\rangle) \vee \langle \texttt{\_i},0\rangle \neq M(\langle \texttt{\_q},0\rangle)) \wedge (M(\langle \texttt{\_p},0\rangle) = M(\langle \texttt{\_q},0\rangle) \vee M(\langle \texttt{\_p},0\rangle) \neq M(\langle \texttt{\_q},0\rangle))$", simplifions et éclatons sur autant de lignes que nécessaire :

$$\left\{\begin{array}{ll}
\text{si } a = M(\langle \texttt{_q},0\rangle) & \\
\quad \wedge\langle \texttt{_i},0\rangle = M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle = M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) = M(\langle \texttt{_q},0\rangle \text{ alors} & 3 \\
\text{si } a = M(\langle \texttt{_q},0\rangle) & \\
\quad \wedge\langle \texttt{_i},0\rangle = M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle = M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) \neq M(\langle \texttt{_q},0\rangle \text{ alors} & 3 \\
\text{si } a = M(\langle \texttt{_q},0\rangle) & \\
\quad \wedge\langle \texttt{_i},0\rangle = M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) = M(\langle \texttt{_q},0\rangle \text{ alors} & 3 \\
\qquad \vdots & \\
\text{si } a = M(\langle \texttt{_q},0\rangle) & \\
\quad \wedge\langle \texttt{_i},0\rangle \neq M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) \neq M(\langle \texttt{_q},0\rangle \text{ alors} & 3 \\
\text{si } a = \langle \texttt{_i},0\rangle & \\
\quad \wedge\langle \texttt{_i},0\rangle = M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) = M(\langle \texttt{_q},0\rangle \text{ alors} & 2 \\
\text{si } a = \langle \texttt{_i},0\rangle & \\
\quad \wedge\langle \texttt{_i},0\rangle = M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) \neq M(\langle \texttt{_q},0\rangle \text{ alors} & 2 \\
\text{si } a = \langle \texttt{_i},0\rangle & \\
\quad \wedge\langle \texttt{_i},0\rangle \neq M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) = M(\langle \texttt{_q},0\rangle \text{ alors} & 1 \\
\text{si } a = \langle \texttt{_i},0\rangle & \\
\quad \wedge\langle \texttt{_i},0\rangle \neq M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) \neq M(\langle \texttt{_q},0\rangle \text{ alors} & 1 \\
\text{si } a = M(\langle \texttt{_p},0\rangle) & \\
\quad \wedge\langle \texttt{_i},0\rangle \neq M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle = M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) \neq M(\langle \texttt{_q},0\rangle) \text{ alors} & 2 \\
\text{si } a = M(\langle \texttt{_p},0\rangle) & \\
\quad \wedge\langle \texttt{_i},0\rangle \neq M(\langle \texttt{_p},0\rangle) \wedge \langle \texttt{_i},0\rangle \neq M(\langle \texttt{_q},0\rangle) \wedge M(\langle \texttt{_p},0\rangle) \neq M(\langle \texttt{_q},0\rangle) \text{ alors} & 2
\end{array}\right. \tag{3.15}$$

Nous pouvons encore simplifier quelques lignes : par exemple, si p égale q et p pointe sur i, q ne peut pas ne pas pointer sur i. De même, si p pointe sur i et p est différent de q, alors q ne peut pointer sur i. Sur les $n2^{|\mathcal{P}|} = 4 * 8 = 32$ lignes théoriques de cette transformation, nous n'en aurons au plus que $5n = 20$. De plus, certaines de ces lignes ayant déjà des clauses limitantes, nous arrivons *in-fine* à 10 lignes dans la forme ultime avant l'écriture sous forme concrète :

$$
\begin{cases}
\text{si } a = M(\langle \mathtt{\_q}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle = M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle = M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) = M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 3 \\
\text{si } a = M(\langle \mathtt{\_q}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle = M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 3 \\
\text{si } a = M(\langle \mathtt{\_q}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle = M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 3 \\
\text{si } a = M(\langle \mathtt{\_q}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) = M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 3 \\
\text{si } a = M(\langle \mathtt{\_q}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 3 \\
\text{si } a = \langle \mathtt{\_i}, 0\rangle & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle = M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 2 \\
\text{si } a = \langle \mathtt{\_i}, 0\rangle & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) = M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 1 \\
\text{si } a = \langle \mathtt{\_i}, 0\rangle & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle \text{ alors} & 1 \\
\text{si } a = M(\langle \mathtt{\_p}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle = M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle) \text{ alors} & 2 \\
\text{si } a = M(\langle \mathtt{\_p}, 0\rangle) & \\
\quad \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_p}, 0\rangle) \wedge \langle \mathtt{\_i}, 0\rangle \neq M(\langle \mathtt{\_q}, 0\rangle) \wedge M(\langle \mathtt{\_p}, 0\rangle) \neq M(\langle \mathtt{\_q}, 0\rangle) \text{ alors} & 2
\end{cases}
\tag{3.16}
$$

Il ne reste plus qu'à regrouper toutes les lignes pour lesquelles les clauses sont identiques, à convertir les expressions en éléments de $\mathcal{X}$, et nous obtenons la forme concrète :

$$
\tau = \left|
\begin{array}{l}
\{\mathtt{p\ ==\ \&i}, \mathtt{q\ ==\ \&i}, \mathtt{p\ ==\ q}\} \ / \ [\mathtt{*q} \leftarrow \mathtt{3}] \\
\{\mathtt{p\ ==\ \&i}, \mathtt{q\ !=\ \&i}, \mathtt{p\ !=\ q}\} \ / \ [\mathtt{*q} \leftarrow \mathtt{3}, \mathtt{i} \leftarrow \mathtt{2}] \\
\{\mathtt{p\ !=\ \&i}, \mathtt{q\ ==\ \&i}, \mathtt{p\ !=\ q}\} \ / \ [\mathtt{*q} \leftarrow \mathtt{3}, \mathtt{*p} \leftarrow \mathtt{2}] \\
\{\mathtt{p\ !=\ \&i}, \mathtt{q\ !=\ \&i}, \mathtt{p\ ==\ q}\} \ / \ [\mathtt{*q} \leftarrow \mathtt{3}, \mathtt{i} \leftarrow \mathtt{1}] \\
\{\mathtt{p\ !=\ \&i}, \mathtt{q\ !=\ \&i}, \mathtt{p\ !=\ q}\} \ / \ [\mathtt{*q} \leftarrow \mathtt{3}, \mathtt{i} \leftarrow \mathtt{1}, \mathtt{*p} \leftarrow \mathtt{2}]
\end{array}
\right.
\tag{3.17}
$$

qui corrobore l'intuition que nous pouvions avoir à partir du programme original.

Simplifications

Dans l'exemple précédent, nous avons procédé à un certain nombre de simplifications triviales. Nous n'avions pas pour but de simplifier *au maximum*, ni d'écarter *toutes* les clauses toujours fausses, mais simplement de diminuer le nombre de clauses et, par là, le nombre de lignes en présence d'opérateurs *ou*. En fait, ces simplifications ne sont pas nécessaires pour la validité de la démarche, mais sont juste un outil pour rendre plus lisibles les exemples.

Nous présentons donc ici quelques simplifications triviales dont l'implémentation se fait sans problème :

- $e \wedge$ vrai $= e$, $e \wedge$ faux $=$ faux, $e \vee$ vrai $=$ vrai et $e \vee$ faux $= e$

  Ces règles simples sont la base des simplifications ci-après ;
- $(e \neq e) =$ faux ;
- $(e_1 = e_2 \wedge e_1 \neq e_2) =$ faux ;
- $(a = v) =$ faux, où $v \in \mathcal{V}$, ne peut jamais arriver puisque a ne serait alors pas une adresse ;
- $(e_1 = e_2) =$ faux, où $\langle e_1, e_2 \rangle \in \mathcal{A}_i \times \mathcal{A}_j$ et $i \neq j$: nous profitons de la structuration de la mémoire introduite au début de ce chapitre. Nous pourrions pousser plus loin dans la précision offerte par le typage[2] ;
- $(\langle id_1, k_1 \rangle = \langle id_2, k_2 \rangle) =$ faux si $id_1 \neq id_2$ ou $k_1 \neq k_2$.

D'autre part, une simplification supplémentaire va, dans la pratique, économiser le traitement de beaucoup de cas inutiles : un pointeur ne peut adresser une variable simple ($d_m(v) = 1$) et être incrémenté. Nous supposons qu'un pointeur ne peut sortir de l'objet dans lequel il pointe, supposition renforcée par le texte de la norme.

Ce ne sont là que quelques simplifications ne prétendant absolument pas à l'exhaustivité. Notre but n'est pas de faire un outil général de calcul symbolique mais, répétons-le, de restreindre l'explosion de la complexité des transformations. L'efficacité de ces quelques simplifications est remarquable puisque, dans l'exemple précédent, nous sommes parvenus à une réduction de plus d'un tiers des clauses théoriques (de 32 à 10) simplement par quelques simplifications triviales !

3.4.4 Abstraction

Reconstituer une transformation à partir de sa représentation est simple. Considérons la transformation représentée par :

[2] L'implémentation utilise la structuration plus complète offerte par le typage.

$$\tau = \left| \begin{array}{l} \{g_1^1 \ldots g_1^{p_1}\} \ / \ [l_1^1 \leftarrow r_1^1, \ldots l_1^{q_1} \leftarrow r_1^{q_1}] \\ \quad \vdots \\ \{g_n^1 \ldots g_n^{p_n}\} \ / \ [l_n^1 \leftarrow r_n^1, \ldots l_n^{q_n} \leftarrow r_n^{q_n}] \end{array} \right. \tag{3.18}$$

Cette transformation se lit, en prenant la première ligne comme exemple : “si $g_1^1 \wedge \ldots \wedge g_1^{p_1}$, alors l_1^1 reçoit $r_1^{q_1}$ et ... et $l_1^{q_1}$ reçoit $r_1^{q_1}$”.

Il suffit donc de transcrire ce qu'on lit en traduisant les *reçoit*, pour obtenir la transformation :

$$T(M) = \left(\begin{array}{lll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ a & \longmapsto & \left\{ \begin{array}{ll} \text{si } g_1(M) \wedge a = L(l_1^1, M) \text{ alors} & R(r_1^1, M) \\ \quad \vdots & \\ \text{sinon si } g_1(M) \wedge a = L(l_1^{q_1}, M) \text{ alors} & R(r_1^{q_1}, M) \\ \quad \vdots & \\ \text{sinon si } g_n(M) \wedge a = L(l_n^1, M) \text{ alors} & R(r_n^1, M) \\ \quad \vdots & \\ \text{sinon si } g_n(M) \wedge a = L(l_n^{q_n}, M) \text{ alors} & R(r_n^{q_n}, M) \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right) \tag{3.19}$$

où :

$$g_i(M) = \bigwedge_{1 \le j \le p_i} \left(R(g_i^j, M) \neq 0 \right) \tag{3.20}$$

Notons que les *sinon* (hormis le dernier) sont placés ici uniquement pour respecter la forme générale des transformations. Le fait d'avoir conservé toutes les gardes retire le risque d'ambiguïté et le besoin de respecter l'ordre des lignes.

3.4.5 Conclusion

Dans cette section, nous avons défini plus précisément les transformations que dans la section précédente. En particulier, nous avons vu comment composer les transformations pour modéliser une séquence d'instructions, puis quelle est la forme générale des transformations, ce qui nous a permis de proposer une méthode pour passer de la forme que nous avons appelé *abstraite* à sa représentation (forme que nous avons appelé *concrète*).

Nous avons défini l'opération de séquence sur la forme abstraite.

L'implémentation peut donc être réalisée au niveau de la forme concrète, puisque nous avons démontré les passages de la forme abstraite à la forme concrète et réciproquement.

3.5 Alternative

```
int *p, n, i, j ;

if (n < 3)
        p = &i ;
else
        p = &j ;
```

Figure 3.15: Exemple de programme

Supposons que nous ayons le programme de la figure 3.15. Quelle est la transformation modélisant cette portion de programme ?

Ne nous limitons pas à un exemple. Considérons la syntaxe de l'alternative (voir figure 3.9).

Nommons T_v (respectivement T_f) la transformation correspondant au cas *test vrai* (respectivement *test faux*). S'il n'y a pas d'alternant si faux, T_f est la transformation vide, c'est à dire $Id_{\mathcal{M}}$, la fonction identité sur $\mathcal{M}$, qui est un cas dégénéré de forme abstraite.

La transformation correspondant au test (expression e) et à ses deux alternants (ou un seul si $T_f = Id_{\mathcal{M}}$) s'écrit donc :

$$
\begin{aligned}
T : \quad \mathcal{M} & \longrightarrow \mathcal{M} \\
M & \longmapsto T(M) = \begin{cases} \text{si } R(e, M) \neq 0 \text{ alors} & T_v(M) \\ \text{sinon} & T_f(M) \end{cases}
\end{aligned}
\tag{3.21}
$$

Développons cette expression :

$$T(M) = \begin{cases} \text{si } R(e,M) \neq 0 \text{ alors} & \begin{pmatrix} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \begin{cases} \text{si } a = f_1^v(M) \wedge g_1^v(M) \text{ alors} & h_1^v(M) \\ \quad\vdots & \\ \text{sinon si } a = f_{n_v}^v(M) \wedge g_{n_v}^v(M) \text{ alors} & h_{n_v}^v(M) \\ \text{sinon} & M(a) \end{cases} \end{pmatrix} \\ \text{sinon} & \begin{pmatrix} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \begin{cases} \text{si } a = f_1^f(M) \wedge g_1^f(M) \text{ alors} & h_1^f(M) \\ \quad\vdots & \\ \text{sinon si } a = f_{n_f}^f(M) \wedge g_{n_f}^f(M) \text{ alors} & h_{n_f}^f(M) \\ \text{sinon} & M(a) \end{cases} \end{pmatrix} \end{cases}$$

Si nous intégrons le $R(e,M)$ dans les transformations des alternants, cela donne :

$$T(M) = \begin{pmatrix} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \begin{cases} \text{si } a = f_1^v(M) \wedge R(e,M) \neq 0 \wedge g_1^v(M) \text{ alors} & h_1^v(M) \\ \quad\vdots & \\ \text{sinon si } a = f_{n_v}^v(M) \wedge R(e,M) \neq 0 \wedge g_{n_v}^v(M) \text{ alors} & h_{n_v}^v(M) \\ \text{sinon si } R(e,M) \neq 0 \text{ alors} & M(a) \\ \text{sinon si } a = f_1^f(M) \wedge g_1^f(M) \text{ alors} & h_1^f(M) \\ \quad\vdots & \\ \text{sinon si } a = f_{n_f}^f(M) \wedge g_{n_f}^f(M) \text{ alors} & h_{n_f}^f(M) \\ \text{sinon} & M(a) \end{cases} \end{pmatrix}$$

La ligne “sinon si $R(e,M) \neq 0$ alors $M(a)$” pose un problème : cette fonction ne respecte plus la forme abstraite des transformations. Munis de notre fer à souder logique, nous allons arranger ces expressions en introduisant explicitement la dépendance vis-à-vis de la nullité de $R(e,M)$:

$$T(M) = \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{ll} \text{si } a = f_1^v(M) \wedge R(e,M) \neq 0 \wedge g_1^v(M) \text{ alors} & h_1^v(M) \\ \quad \vdots & \\ \text{sinon si } a = f_{n_v}^v(M) \wedge R(e,M) \neq 0 \wedge g_{n_v}^v(M) \text{ alors} & h_{n_v}^v(M) \\ \text{sinon si } R(e,M) \neq 0 \text{ alors} & M(a) \\ \text{sinon si } a = f_1^f(M) \wedge g_1^f(M) \wedge R(e,M) = 0 \text{ alors} & h_1^f(M) \\ \quad \vdots & \\ \text{sinon si } a = f_{n_f}^f(M) \wedge g_{n_f}^f(M) \wedge R(e,M) = 0 \text{ alors} & h_{n_f}^f(M) \\ \text{sinon si } R(e,M) = 0 \text{ alors} & M(a) \end{array} \right. \end{array} \right)$$

Il ne nous reste plus qu'à descendre $M(a)$ pour donner :

$$T(M) = \left(\begin{array}{lll} \mathcal{A} & \longrightarrow & \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto & \left\{ \begin{array}{ll} \text{si } a = f_1^v(M) \wedge R(e,M) \neq 0 \wedge g_1^v(M) \text{ alors} & h_1^v(M) \\ \quad \vdots & \\ \text{sinon si } a = f_{n_v}^v(M) \wedge R(e,M) \neq 0 \wedge g_{n_v}^v(M) \text{ alors} & h_{n_v}^v(M) \\ \text{sinon si } a = f_1^f(M) \wedge g_1^f(M) \wedge R(e,M) = 0 \text{ alors} & h_1^f(M) \\ \quad \vdots & \\ \text{sinon si } a = f_{n_f}^f(M) \wedge g_{n_f}^f(M) \wedge R(e,M) = 0 \text{ alors} & h_{n_f}^f(M) \\ \text{sinon si } R(e,M) \neq 0 \text{ alors} & M(a) \\ \text{sinon si } R(e,M) = 0 \text{ alors} & M(a) \end{array} \right. \end{array} \right)$$

Les deux dernières lignes peuvent maintenant être fondues en une seule :

$$T(M) = \left(\begin{array}{lll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top & \\ a & \longmapsto \left\{ \begin{array}{ll} \text{si } a = f_1^v(M) \wedge R(e,M) \neq 0 \wedge g_1^v(M) \text{ alors} & h_1^v(M) \\ \quad \vdots & \\ \text{sinon si } a = f_{n_v}^v(M) \wedge R(e,M) \neq 0 \wedge g_{n_v}^v(M) \text{ alors} & h_{n_v}^v(M) \\ \text{sinon si } a = f_1^f(M) \wedge g_1^f(M) \wedge R(e,M) = 0 \text{ alors} & h_1^f(M) \\ \quad \vdots & \\ \text{sinon si } a = f_{n_f}^f(M) \wedge g_{n_f}^f(M) \wedge R(e,M) = 0 \text{ alors} & h_{n_f}^f(M) \\ \text{sinon} & M(a) \end{array} \right. \end{array} \right) \tag{3.22}$$

Notre fonction est donc maintenant conforme aux formes abstraites des transformations.

3.6 Réalisation

L'implémentation est réalisée suivant la méthode décrite dans ce chapitre. Toutefois, les transformations restent sous forme concrète.

3.6.1 Représentation d'une transformation

Comment est représentée une transformation ? C'est une liste de la forme :

```
(
    (
        ( relop expression expression )
        ⋮
        ( relop expression expression )
    )
    (
        ( store type expression expression )
        ⋮
        ( store type expression expression )
    )
)
```

On voit que la représentation d'une transformation associée à une affectation est triviale, puisqu'il s'agit purement et simplement de la forme intermédiaire représentant cette affectation.

Notons la présence d'un élément supplémentaire dans les gardes. Il s'agit de listes de la forme :

(`intab` *type expression*)

qui spécifie que l'expression pointe dans un tableau (et précise le type de ses éléments). Ces éléments spéciaux ne sont pas indispensables. En fait, ce ne sont pas des gardes, mais simplement des optimisations pour éviter d'explorer les substitutions à chaque fois qu'un conflit avec une variable scalaire est détecté. Puisque ces éléments ne sont pas indispensables, nous les avons retiré des exemples.

3.6.2 Implémentation

L'analyseur de séquences linéaires se compose de quatre modules principaux :

- analyse des instructions : ce module opère par compositions successives des transformations associées aux instructions. Étant donné un segment de programme, il renvoie la transformation associée ;
- simplification des gardes : étant donnée une garde, combinaison de prédicats, ce module simplifie ces prédicats jusqu'à donner une garde toujours vraie, toujours fausse ou une version simplifiée dans laquelle les prédicats redondants ont été retirés ;
- simplification des expressions : les expressions sont simplifiées en les normalisant ;
- substitution des expressions : ce module reçoit une transformation (où figurent donc des expressions) et une substitution, et renvoie une nouvelle transformation dans laquelle le maximum d'expressions ont été substituées.

Notons que l'analyseur dans son ensemble n'utilise que très peu de données globales. Les transformations associées à chaque nœud de l'arbre (du programme) sont construites au fur et à mesure. L'analyseur prend donc en entrée un programme exprimé en repésentation intermédiaire et renvoie donc une transformation sous la forme décrite dans la section précédente.

3.6.3 Exemples

Tous les exemples de ce chapitre utilisent les déclarations globales suivantes :

```
int a ;
int i1, i2 ;

int *p1, *p2 ;

int t1 [10], t2 [10] ;
```

Chaque exemple est composé :

- de la fonction en C ;
- du temps pris par l'analyse (temps cpu) ;
- de la transformation (en forme concrète) résultat ;
- d'un commentaire soulignant les points intéressants de l'exemple.

Il faut souligner que la transformation issue directement de cette passe est traduite par un "décompilateur de transformation", c'est à dire un simple filtre qui transforme la représentation intermédiaire en syntaxe C. Les seules éditions effectuées ont un but cosmétique, d'intégration dans le présent document et de retrait des "gardes" `intab`.

Exemple

```
void ts00 (void)
{
      p1 = &i1 ;
      i1 = 1 ;
      *p1 = 0 ;
}
```

Temps : 0,040 secondes

$$\tau_{00} = \left|\ \{\text{vrai}\}/[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i1} \leftarrow \texttt{0}]\right.$$

Commentaire : l'affectation de 1 à la variable `i1` est anéantie par l'affectation à `*p1`.

Exemple

```
void ts01 (void)
{
      i1 = 0 ;
      i1 = 2 ;
}
```

Temps : 0,000 secondes

$$\tau_{01} = \left|\ \{\text{vrai}\}/[\texttt{i1} \leftarrow \texttt{2}]\right.$$

Commentaire : le temps de l'analyse est non mesurable. Ce cas est particulièrement simple, puisque l'adresse $\langle \_\mathtt{i1}, 0\rangle$ égale l'adresse $\langle \_\mathtt{i1}, 0\rangle$, il n'y a pas besoin de traitement complexe pour s'en apercevoir.

Exemple

```
void ts02 (void)
{
      i1 = 0 ;
      i1 = i1 + 2 ;
      i2 = i1 + 1 ;
}
```

Temps : 0,020 secondes

$$\tau_{02} = \left| \; \{\text{vrai}\}/[\mathtt{i1} \leftarrow 2, \mathtt{i2} \leftarrow 3] \right.$$

Commentaire : il s'agit d'une composition de transformations pour vérifier que le simplificateur d'expressions constantes fonctionne bien.

Exemple

```
void ts03 (void)
{
      p1 = &i1 ;
      i1 = 1 ;
      i2 = 2 ;
      *p1 = 3 ;
}
```

Temps : 0,040 secondes

$$\tau_{03} = \left| \; \{\text{vrai}\}/[\mathtt{p1} \leftarrow \mathtt{\&i1}, \mathtt{i1} \leftarrow 3, \mathtt{i2} \leftarrow 2] \right.$$

Commentaire : ce cas est plus complexe que τ_{01}, car il faut propager l'information que $M(\langle \_\mathtt{p1}, 0\rangle) = \langle \_\mathtt{i1}, 0\rangle$ pour anéantir l'affectation à `i1`. Notons que l'algorithme compose les transformations *à partir de la fin*, c'est à dire que lorsqu'il compose les transformations correspondant à `p1` d'une part, à `i1`, `i2` et `*p2` d'autre part, les gardes `p1 == &i1` et `p1 == &i2` sont supprimées.

Exemple

```
void ts04 (void)
{
      p1 = &i1 ;
      i1 = 1 ;
      i2 = 2 ;
      *p1 = 3 ;
      *p2 = 4 ;
}
```

Temps : 0.760 secondes

$$\tau_{04} = \left| \begin{array}{c} \{\texttt{p2 == \&i1}\} \quad /[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i2} \leftarrow \texttt{2}, \texttt{*p2} \leftarrow \texttt{4}] \\ \{\texttt{p2 == \&i2}\} \quad /[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i1} \leftarrow \texttt{3}, \texttt{*p2} \leftarrow \texttt{4}] \\ \{\texttt{p2 != \&i1}, \texttt{p2 != \&i2}\}/[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i1} \leftarrow \texttt{3}, \texttt{i2} \leftarrow \texttt{2}, \texttt{*p2} \leftarrow \texttt{4}] \end{array} \right.$$

Commentaire : cet exemple est le plus long en temps d'exécution. N'ayant aucune information sur `p2`, il faut supposer que `p2` peut pointer n'importe où dans la mémoire (restreinte aux objets entiers), d'où la présence de nombreuses gardes.

Exemple

```
void ts05 (void)
{
      t1 [i1] = 0 ;
      i2 = 0 ;
}
```

Temps : 0,000 secondes

$$\tau_{05} = \left| \ \{\text{vrai}\}/[\texttt{*(t1+i1)} \leftarrow \texttt{0}, \texttt{i2} \leftarrow \texttt{0}] \right.$$

Commentaire : le tableau `t1` est à une adresse fixe, donc il ne peut y avoir confusion entre une case du tableau et la variable `i2`.

Exemple

```
void ts06 (void)
{
      t1 [i1] = 1 ;
      a = 0 ;
```

```
        t2 [i1] = 2 ;
        t2 [i2] = 3 ;
}
```

Temps : 0,060 secondes

$$\tau_{06} = \left| \begin{array}{l} \{\texttt{i1 == i2}\}/[*(\texttt{t1+i1}) \leftarrow 1, *(\texttt{t2+i2}) \leftarrow 3, \texttt{a} \leftarrow 0] \\ \{\texttt{i1 != i2}\}/[*(\texttt{t1+i1}) \leftarrow 1, *(\texttt{t2+i2}) \leftarrow 3, \texttt{a} \leftarrow 0, *(\texttt{t2+i1}) \leftarrow 2] \end{array} \right.$$

Commentaire : le calcul de gardes est suffisamment fin pour éviter des gardes comme `t2+i2 == t2+i1`, il les remplace par l'égalité des deux indices. D'autre part, les accès aux deux tableaux `t1` et `t2` ne peuvent provoquer de conflit.

Exemple

```
void ts07 (void)
{
        if (a == 1)
                p1 = &i1 ;
        else  p1 = &i2 ;
        *p1 = 5 ;
}
```

Temps : 0,020 secondes

$$\tau_{07} = \left| \begin{array}{l} \{\texttt{a == 1}\}/[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i1} \leftarrow 5] \\ \{\texttt{a != 1}\}/[\texttt{p1} \leftarrow \texttt{\&i2}, \texttt{i2} \leftarrow 5] \end{array} \right.$$

Commentaire : vérification simple de l'alternative.

Exemple

```
void ts08 (void)
{
        p1 = &i1 ;
        i1 = 3 ;
        *p1 = *p1 + 1 ;
}
```

Temps : 0,020 secondes

$$\tau_{08} = \left| \ \{\text{vrai}\}/[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i1} \leftarrow \texttt{4}] \right.$$

Commentaire : un nouveau test du simplificateur d'expressions, compliqué par la présence de `*p1` en partie droite d'affectation.

Exemple

```
void ts09 (void)
{
      *p1 = 3 ;
      *p1 = *p1 + 1 ;
}
```

Temps : 0,000 secondes

$$\tau_{09} = \left| \ \{\text{vrai}\}/[\texttt{*p1} \leftarrow \texttt{4}] \right.$$

Commentaire : le pointeur `p1` peut pointer n'importe où en mémoire (restreinte aux objets entiers), mais il ne peut y avoir aucun conflit, d'où aucune garde dans cette transformation.

Exemple

```
void ts10 (void)
{
      p1 = t1 ;
      p1 [i1] = p1 [i1] + 1 ;
}
```

Temps : 0,000 secondes

$$\tau_{10} = \left| \ \{\text{vrai}\}/[\texttt{p1} \leftarrow \texttt{t1}, \texttt{*(t1+i1)} \leftarrow \texttt{*(t1+i1) + 1}] \right.$$

Commentaire : le pointeur `p1` est utilisé comme un faux tableau.

Exemple

```
void ts11 (void)
{
```

```
        i1 = i1 + 1 ;
        *p1 = *p1 + 1 ;
}
```

Temps : 0,020 secondes

$$\tau_{11} = \left| \begin{array}{l} \{\texttt{p1 == \&i1}\}/[\texttt{*p1} \leftarrow \texttt{i1 + 1 + 1}] \\ \{\texttt{p1 != \&i1}\}/[\texttt{*p1} \leftarrow \texttt{*p1 + 1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \end{array} \right.$$

Commentaire : contrairement à τ_{09}, le pointeur `p1` est en conflit avec la variable `i1`.

Exemple

```
void ts12 (void)
{
        a = *p1 ;
        *p1 = *p2 ;
        *p2 = a ;
}
```

Temps : 0,100 secondes

$$\tau_{12} = \left| \begin{array}{rl} \{\texttt{p1 == \&a}, \texttt{p2 == \&a}\} & /[\texttt{*p2} \leftarrow \texttt{*p1}] \\ \{\texttt{p1 == p2}, \texttt{p1 != \&a}\} & /[\texttt{a} \leftarrow \texttt{*p1}, \texttt{*p2} \leftarrow \texttt{*p2}] \\ \{\texttt{p1 == \&a}, \texttt{p2 != \&a}\} & /[\texttt{*p1} \leftarrow \texttt{*p2}, \texttt{*p2} \leftarrow \texttt{*p2}] \\ \{\texttt{p2 == \&a}, \texttt{p1 != \&a}\} & /[\texttt{*p1} \leftarrow \texttt{*p1}, \texttt{*p2} \leftarrow \texttt{*p1}] \\ \{\texttt{p1 != p2}, \texttt{p1 != \&a}, \texttt{p2 != \&a}\} & /[\texttt{a} \leftarrow \texttt{*p1}, \texttt{*p1} \leftarrow \texttt{*p2}, \texttt{*p2} \leftarrow \texttt{*p1}] \end{array} \right.$$

Commentaire : un des exemples les plus difficiles à vérifier à la main... On note certains cas (`*p2` $\leftarrow$ `*p2` par exemple) de substitutions “inutiles”. Elles ne sont pas fausses, mais elles n'apportent aucune information.

Exemple

```
void ts13 (void)
{
        a = *p1 ;
        *p1 = *p2 ;
        *p2 = a ;
```

```
        p1 = p1 + 1 ;
        p2 = p2 + 1 ;
}
```

Temps : 0,380 secondes

$$\tau_{13} = \left| \begin{array}{l} \{\texttt{p1==p2}, \texttt{p1!=\&a}\} \quad /[\texttt{a} \leftarrow \texttt{*p1}, \texttt{p1} \leftarrow \texttt{p1+1}, \texttt{p2} \leftarrow \texttt{p2+1}, \texttt{*p2} \leftarrow \texttt{*p1}] \\ \{\texttt{p1!=p2}, \texttt{p1!=\&a}, \texttt{p2!=\&a}\}/[\texttt{a} \leftarrow \texttt{*p1}, \texttt{p1} \leftarrow \texttt{p1+1}, \texttt{p2} \leftarrow \texttt{p2+1}, \texttt{*p1} \leftarrow \texttt{*p2}, \texttt{*p2} \leftarrow \texttt{*p1}] \end{array} \right.$$

Commentaire : les pointeurs `p1` et `p2` ne peuvent pointer dans la variable `a`. Puisqu'ils sont incrémentés, ils pointent forcément dans un tableau, mais ne peuvent pas adresser une variable scalaire. Les gardes `p1!=&a` sont donc toujours vraies, mais les substitutions gardées par `p1==&a` ont été retirées de la transformation.

Exemple

```
void ts14 (void)
{
        p1 = 1 + p1 + 1 ;
        i1 = 1 ;
        p1 = p1 + 1 ;
        *p1 = 2 ;
}
```

Temps : 0,020 secondes

$$\tau_{14} = \left| \ \{\text{vrai}\}/[\texttt{i1} \leftarrow 1, \texttt{p1} \leftarrow \texttt{1+p1+1+1}, *\texttt{(1+p1+1+1)} \leftarrow 2] \right.$$

Commentaire : là encore, le pointeur `p1` ne peut adresser la variable `i1` puisqu'il est incrémenté. Signalons ici une limitation du simplificateur d'expressions qui ne sait pas simplifier `1+p1+1+1` en `p1+3`...

Exemple

```
void ts15 (void)
{
        p1 = p1 + 1 ;
        p1 = &i1 ;
        *p1 = 2 ;
}
```

Temps : 0,020 secondes

$$\tau_{15} = \left| \; \{\text{vrai}\}/[\texttt{p1} \leftarrow \texttt{\&i1}, \texttt{i1} \leftarrow 2] \right.$$

Commentaire : l'incrémentation du pointeur `p1` ne gène pas la détection de l'affectation de 2 à `i1`.

Exemple

```
void ts16 (void)
{
      p1 = p1 + 1 ;
      i1 = 3 ;
      *p1 = 5 ;
}
```

Temps : 0,020 secondes

$$\tau_{16} = \left| \; \{\text{vrai}\}/[\texttt{p1} \leftarrow \texttt{p1 + 1}, \texttt{i1} \leftarrow 3, \texttt{*(p1+1)} \leftarrow 5] \right.$$

Commentaire : la valeur pointée par `p1` dans la troisième affectation correspond à l'adresse `p1+1` puisqu'on considère la valeur de `p1` *au début* de la séquence.

Exemple

```
void ts17 (void)
{
      *p1 = 1 ;
      *(t1 + i1) = 2 ;
}
```

Temps : 0,020 secondes

$$\tau_{17} = \left| \begin{array}{l} \{\texttt{p1 == t1 + i1}\}/[\texttt{*(t1+i1)} \leftarrow 2] \\ \{\texttt{p1 != t1 + i1}\}/[\texttt{*p1} \leftarrow 1, \texttt{*(t1+i1)} \leftarrow 2] \end{array} \right.$$

Commentaire : il n'y a pas de conflit entre `p1` et la variable `i1` puisque `i1` n'est que lue et pas modifiée.

Exemple

```
void ts18 (void)
{
      *p1 = *p1 + 1 ;
      *p2 = *p2 + 1 ;
      i1 = i1 + 1 ;
      p1 = p1 + 1 ;
      p2 = p2 + 1 ;
}
```

Temps : 0,300 secondes

$$\tau_{18} = \left|\begin{array}{l} \{\mathtt{p1==p2}, \mathtt{p1!=\&i1}\} \quad /[\mathtt{i1} \leftarrow \mathtt{i1+1}, \mathtt{p1} \leftarrow \mathtt{p1+1}, \mathtt{p2} \leftarrow \mathtt{p2+1}, \mathtt{*p2} \leftarrow \mathtt{*p1+1+1}] \\ \{\mathtt{p1!=p2}, \mathtt{p1!=\&i1}, \mathtt{p2!=\&i1}\}/[\mathtt{i1} \leftarrow \mathtt{i1+1}, \mathtt{p1} \leftarrow \mathtt{p1+1}, \mathtt{p2} \leftarrow \mathtt{p2+1}, \mathtt{*p1} \leftarrow \mathtt{*p1+1}, \mathtt{*p2} \leftarrow \mathtt{*p2+1}] \end{array}\right.$$

Commentaire : `p1` et `p2` ne peuvent pointer sur `i1`. Cette constatation est très souvent utile dans l'analyse des corps de boucle.

3.6.4 Remarque

Les gardes calculées dans ces exemples sont toujours minimales : il n'y a en effet jamais de garde toujours fausse, donc de ligne inutile. Il est possible que, dans certains cas, certaines lignes soient superflues. Ce phénomène pourrait être évité par la confection d'un module de vérification de gardes plus perfectionné, mais aussi plus coûteux en temps de calcul. Le compromis entre précision et temps d'analyse que nous avons choisi avantage le temps d'analyse.

3.6.5 Performances

Il est difficile d'évaluer la complexité de notre analyseur, car il est constitué de plusieurs modules non triviaux. L'évolution des temps de calcul en fonction des paramètres du problème est pourtant nécessaire pour jauger notre algorithme. Nous proposons donc, dans cette section, de déterminer expérimentalement les paramètres et leur influence sur le temps d'analyse.

Nombre d'affectations

Le premier paramètre qui vient à l'esprit est le nombre d'affectations dans le programme. Ce nombre d'affectations correspond directement au nombre de compositions de transformations. Malheureusement, ce paramètre n'est pas assez précis car il regroupe beaucoup de cas différents comme par exemple :

- la composition de deux affectations à des variables simples ne génère aucune garde alors que la composition de deux affectations dont l'une au moins est un accès via un pointeur génère une garde ;
- la composition de deux affectations à la même variable donne une substitution à un seul terme, alors que la composition de deux affectations à des variables différentes donne une substitution à deux termes.

Mesurons l'influence du nombre d'affectations (c'est à dire la variation du temps de calcul suivant le nombre d'affectations) dans le cas le plus simple. Supposons que nous ayons le programme :

```
int x ;

void affect (void)
{
      x = 0 ;
      x = x + 1 ;
      x = x + 1 ;
        ⋮
      x = x + 1 ;
}
```

qui se traduit par la transformation :

$$\tau = \left| \ \{\text{vrai}\}/[\mathtt{x} \leftarrow n] \right.$$

où n est le nombre de lignes `x = x + 1` dans le programme. C'est le cas le plus simple car il n'y a aucune garde supplémentaire après chaque ligne, et la substitution n'a qu'un seul terme. La simplification nécessaire à chaque ligne est minimale. Nous avons donc les compositions les plus simples. Faisons varier n :

n	temps (s)	n	temps (s)	n	temps (s)	n	temps (s)
10	0,1	60	1,2	110	3,3	160	6,1
20	0,1	70	1,3	120	3,6	170	7,0
30	0,2	80	1,6	130	4,5	180	8,0
40	0,9	90	2,3	140	4,8	190	8,4
50	1,0	100	2,5	150	5,7	200	9,9

On voit se dessiner sur la figure 3.16 une courbe non linéaire. Bien que les fonctions C testées soient énormes, les nombres de lignes considérés sont trop faibles pour donner la forme exacte de cette courbe.

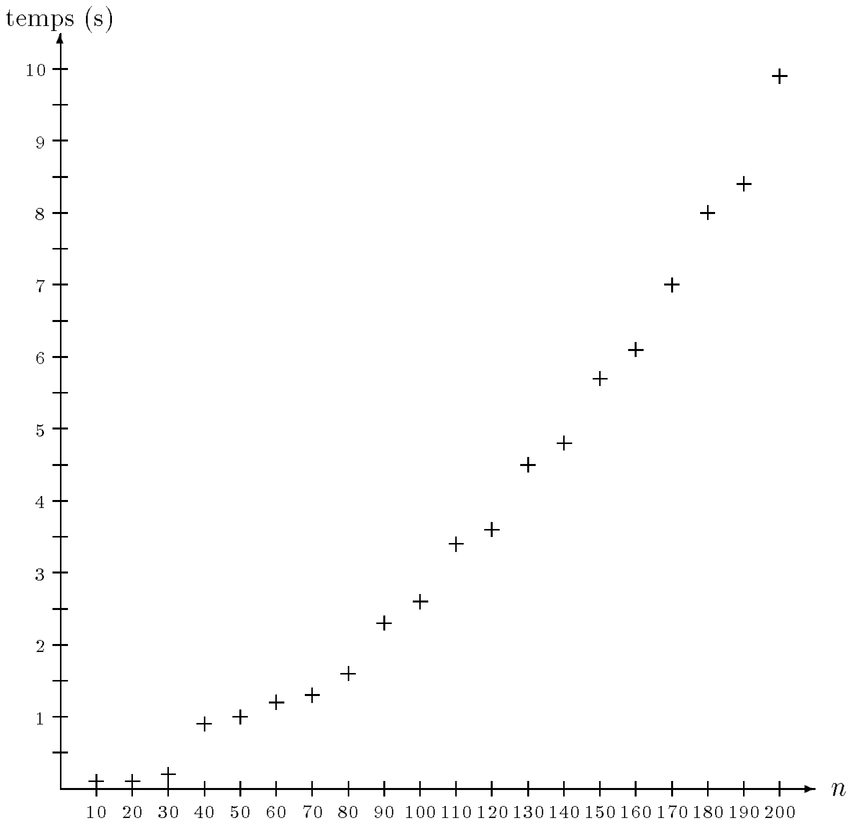

Figure 3.16: Temps d'exécution en fonction du nombre d'affectations

Nombre de termes dans une substitution

Étudions maintenant l'évolution du temps d'analyse en fonction du nombre de termes dans une substitution. Prenons le programme :

```
int x₁, x₂, ... xₙ ;

void substn (void)
{
        x₁ = 0 ;
        x₂ = 0 ;
           ⋮
        xₙ = 0 ;
}
```

dont l'effet est synthétisé par la transformation :

$$\tau = \Big| \; \{\text{vrai}\}/[\mathtt{x}_1 \leftarrow 0, \mathtt{x}_2 \leftarrow 0, \ldots \mathtt{x}_n \leftarrow 0]$$

Ce cas est plus complexe puisqu'il faut manipuler des substitutions à plusieurs termes, chaque composition ajoutant un terme. Faisons varier n :

n	temps (s)	n	temps (s)	n	temps (s)	n	temps (s)
10	0,1	60	5,0	110	17,6	160	40,2
20	0,7	70	7,1	120	21,1	170	46,7
30	1,0	80	8,9	130	24,9	180	53,6
40	2,4	90	11,0	140	29,6	190	61,0
50	3,6	100	14,4	150	35,3	200	68,8

L'examen de la figure 3.17 amène les remarques suivantes :

- du fait de l'importance relative des temps, les aléas dûs à l'intervention du ramasse-miettes, par exemple, sont lissés et la courbe est nettement plus régulière ;
- l'influence du nombre de termes dans une substitution domine l'influence du nombre de compositions ;
- bien que l'augmentation sur 200 compositions soit d'un facteur 10, les temps d'analyse restent très faibles : de l'ordre d'une demi-seconde par ligne pour une fonction de moins de 200 lignes.

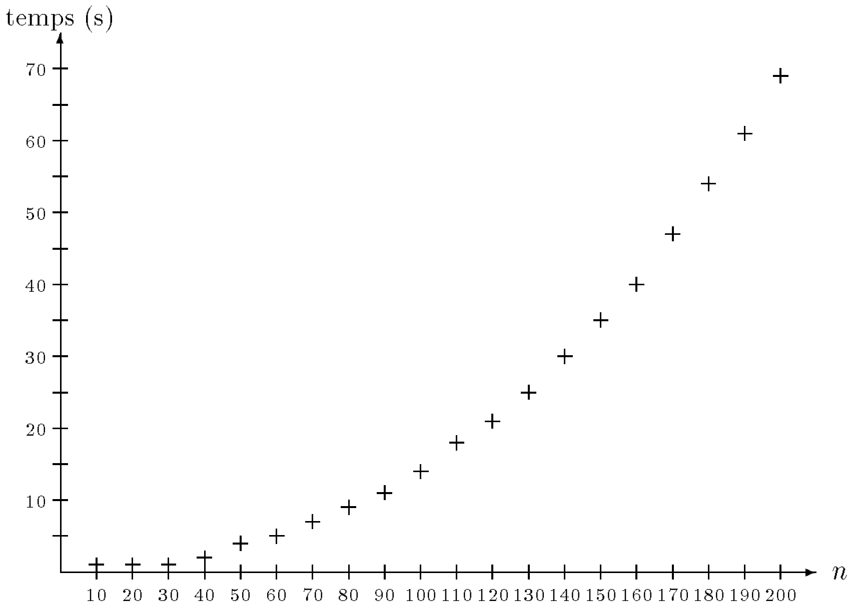

Figure 3.17: Temps d'exécution en fonction du nombre de termes d'une substitution

La complexité est vraisemblablement d'ordre polynomial. Cependant, nous ne cherchons pas à la calculer de manière plus précise, car bien que plus important, le nombre de termes dans une substitution n'est toujours pas le paramètre essentiel.

Nombre de gardes

Le paramètre qui influe le plus sur le temps de l'analyse est sans conteste le nombre de gardes générées à chaque composition. En effet, de nouvelles gardes sont éventuellement générées à chaque composition, puis vérifiées avec les gardes de chaque transformation composée. La vérification des gardes est un module complexe qui en deux phases :

- élimine les égalités ;
- vérifie la cohérence des inégalités.

Les gardes sont le résultat du retrait des ambiguïtés causées par les fonctions f_i des formes abstraites des transformations. Ces fonctions f_i servent à sélectionner l'adresse de la case mémoire cherchée, c'est à dire que chaque f_i correspond à une case mémoire modifiée par une affectation. Le nombre de gardes dépend donc directement de la quantité d'ambiguïtés, c'est à dire de la quantité de variables modifiées de même type, accédées via une lecture mémoire.

Pour quantifier cet effet, plaçons-nous dans le cas le plus défavorable, c'est à dire celui du programme :

```
int *p₁, *p₂, ... *pₙ ;

void garde (void)
{
      *p₁ = 0 ;
      *p₂ = 0 ;
          ⋮
      *pₙ = 0 ;
}
```

Ce programme est le cas le plus défavorable pour ce qui concerne le nombre d'ambiguïtés :

- toutes les variables modifiées sont de même type ;
- chaque ligne accède à la variable modifiée par une lecture mémoire, générant ainsi une nouvelle ambiguïté ;
- il n'y a aucune information initiale sur les $\mathtt{p}_i$, forçant ainsi les ambiguïtés.

La transformation associée à cette fonction est :

$$\tau = \left| \begin{array}{l} \{\mathtt{p}_1\mathtt{==}\mathtt{p}_2, \ldots \mathtt{p}_1\mathtt{==}\mathtt{p}_n, \mathtt{p}_2\mathtt{==}\mathtt{p}_3, \ldots \mathtt{p}_2\mathtt{==}\mathtt{p}_n, \ldots \mathtt{p}_{n-1}\mathtt{==}\mathtt{p}_n\}/[\mathtt{*p}_1 \leftarrow 0,] \\ \qquad\qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad / \\ \{\mathtt{p}_1\mathtt{!=}\mathtt{p}_2, \ldots \mathtt{p}_1\mathtt{!=}\mathtt{p}_n, \mathtt{p}_2\mathtt{!=}\mathtt{p}_3, \ldots \mathtt{p}_2\mathtt{!=}\mathtt{p}_n, \ldots \mathtt{p}_{n-1}\mathtt{!=}\mathtt{p}_n\}/[\mathtt{*p}_1 \leftarrow 0, \mathtt{*p}_2 \leftarrow 0, \ldots \mathtt{*p}_n \leftarrow 0] \end{array} \right.$$

Affinons le calcul du nombre de gardes générées par cette fonction : chaque affectation provoque la modification d'une variable, potentiellement en conflit avec toutes les autres variables modifiées par les affectations précédentes. Le nombre de termes dans chaque garde est donc le nombre de combinaisons de deux éléments pris parmi n :

$$n_t = \mathrm{C}_n^2 = \frac{n!}{2!(n-2)!} = \frac{n(n-1)(n-2)!}{2(n-2)!} = \frac{n(n-1)}{2} \tag{3.23}$$

Une autre approche consiste à compter les variables en conflit avec $\mathtt{p}_1$ (il s'agit de $\mathtt{p}_2 \ldots \mathtt{p}_n$), puis les variables en conflit avec $\mathtt{p}_2$ (il s'agit de $\mathtt{p}_3 \ldots \mathtt{p}_n$), ainsi de suite jusqu'à la simple variable en conflit avec $\mathtt{p}_{n-1}$ (il s'agit de $\mathtt{p}_n$). Nous obtenons alors la somme des $n-1$ premiers entiers :

$$n_t = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \tag{3.24}$$

Les équations 3.23 et 3.24 donnent la même valeur pour n_t, ce qui est heureux !

S'il y a n_t termes dans les gardes, le nombre théorique de gardes est donc 2^{n_t}. Ce nombre est très pessimiste, puisque des gardes comme $a = b \wedge b = c \wedge a \neq c$ sont impossibles. Nous comparons plus loin l'écart entre le nombre de gardes théorique et le nombre de gardes réel.

Ce nombre de gardes est le résultat final et non le nombre de gardes vérifiées par le module de vérification des gardes dont nous avons mentionné ci-dessus la difficulté. Calculons le nombre total de vérifications de gardes : l'algorithme procède par composition successive des gardes, c'est à dire compose les première et deuxième, puis compose le résultat avec la troisième, ainsi de suite. C'est à dire que pour obtenir la composition de n affectations, on compose la composée des $n-1$ premières avec la n-ième affectation. Le nombre total de vérifications est donc la somme des nombres de gardes des compositions précédentes, ce qui donne :

$$n_v = \sum_{i=2}^{n} 2^{\frac{i(i-1)}{2}} \tag{3.25}$$

Encore une fois, ce nombre est pessimiste, puisque basé sur le calcul du nombre de gardes théorique. Notons que $i \geq 2$ puisque la première affectation ne génère aucune garde à elle seule.

Examinons maintenant les résultats issus de l'expérience :

	Expérimental				Théorique		
n	temps (s)	vérifications	valides	n_g	n_t	2^{n_t}	n_v
1	0,0	0	0	1	0	1	1
2	0,0	2	2	2	1	2	3
3	0,1	8	7	5	3	8	11
4	1,0	30	22	15	6	32	43
5	7,2	124	74	52	10	1 024	1 067
6	52,8	578	277	203	15	32 768	33 835
7	523,4	3008	1154	877	21	2 097 152	2 130 987
8	> 6000	?	?	?	28	268 435 456	270 566 443

Ce tableau appelle plusieurs commentaires. Tout d'abord, nous avons séparé les valeurs purement expérimentales des valeurs issues des calculs précédents. Dans la partie expérimentale, nous pouvons constater que le cas $n = 8$ contient des points d'interrogation : en effet, la mémoire de Common Lisp s'est trouvée saturée au bout de deux heures d'analyse environ, et l'analyse n'a donc pu être menée à terme.

Nous voyons immédiatement que l'influence du nombre de gardes est absolument prépondérante sur les autres paramètres.

Cependant, il est intéressant de mettre en relation les valeurs expérimentales et les valeurs issues du calcul :

- nombre de gardes

  Les colonnes n_g (nombre de gardes réel, c'est à dire nombre de lignes dans la transformation résultat) et 2^{n_t} (nombre de gardes théorique) doivent être comparées. On voit immédiatement que le nombre de gardes théorique est une valeur très pessimiste, corroborant ainsi les commentaires précédents. Notons que, pour $n = 1$, la garde correspond à *vrai* ; il s'agit d'une valeur aberrante.

- nombre de vérifications conduisant à des gardes valides

  La colonne *vérifications* correspond au nombre total de vérifications de gardes effectuées. Sur ces vérifications, la colonne *valides* indique le nombre de vérifications qui se sont soldées par une garde valide. Notons que :

  $$\text{valides}(n) = \sum_{i=2}^{n} n_g(i) \tag{3.26}$$

  Nous avons écarté la valeur $n = 1$, car il n'y a aucune vérification dans ce cas.

- nombre de vérifications

  Les colonnes *vérification* et n_v (nombre théorique de vérifications de gardes) montrent, là encore, que le pessimisme de nos calculs n'a rien à voir avec l'optimisme de l'expérience ! La propagation de 2^{n_t} donne des résultats disproportionnés, non confirmés dans cas le plus défavorable de programme.

La mesure des temps de calcul est implacable. Le nombre de gardes est le facteur prédominant pour le calcul de la complexité de notre analyseur. Cependant, il faut tempérer les temps d'analyse :

- la complexité théorique est exponentielle, mais l'expérience montre que, dans le cas le plus défavorable, la complexité est très en deçà ;
- nous nous sommes placés dans le cas le plus défavorable : il y a en effet bien peu de fonctions C où le programmeur jongle avec plus de 5 pointeurs différents et non initialisés ;

Comment pouvons-nous remédier, toutefois, à ces temps de calcul élevés ?

- le module de vérification des gardes est, intuitivement, très lourd en temps de calcul : une optimisation permettrait peut-être de réduire sa complexité interne, et de permettre le calcul de quelques milliers de gardes ;
- la composition des transformations est faite avec un algorithme naïf : il y a $n-1$ compositions pour n affectations ; une voie d'amélioration consiste à utiliser un algorithme dichotomique permettant de ne faire que $\lceil \log_2 n \rceil$ compositions, puisque le produit de compositions est associatif[3] ;

[3] Il serait même possible de faire ces compositions en parallèle !

- il y a beaucoup de vérifications de gardes impossibles (environ deux tiers pour $n = 5$) : il est peut-être possible, dans certains cas, de conclure à l'impossibilité dès la formation des gardes par des tests plus légers, et ainsi éviter de coûteuses vérifications inutiles ;
- enfin, un dispositif simple de "chien de garde" arrêtant l'analyse lorsque le nombre de termes dans une garde et le nombre de gardes deviennent trop importants permettrait d'intégrer notre analyseur dans un paralléliseur "commercial".

3.6.6 Conclusion

Cette étude expérimentale de la complexité de notre analyseur a montré que le facteur influant sur le temps de calcul est le nombre de gardes, c'est à dire la quantité d'ambiguïtés dans la transformation. Nous nous sommes placés dans un cas défavorable, puis nous avons calculé *a priori* le nombre de termes et le nombre de gardes dans la transformation résultat, ainsi que le nombre de vérifications de gardes nécessaires.

L'expérience a montré que, même dans le cas le plus défavorable, cette complexité exponentielle théorique est très pessimiste par rapport aux temps de calcul et des nombre de vérifications constatés, du fait de l'impossibilité de très nombreuses gardes. Malgré cette diminution, la quantité d'ambiguïtés reste de très loin le facteur prédominant.

3.7 Conclusion

Ce chapitre a permis d'aborder le calcul des transformations sur le cas simple des séquences linéaires.

Pour ce faire, nous avons été amenés à définir un modèle d'interprétation, basé sur une mémoire structurée en espaces disjoints suivant les types. Cette caractéristique du langage C nous permet une finesse de détection de conflits relativement importante.

Nous avons ensuite défini un modèle pour représenter les transformations opérées sur la mémoire par une instruction, puis par une séquence d'instructions. Nous avons affiné ce modèle, nous en avons donné deux formes, une forme *abstraite* et une forme *concrète* et nous avons cherché à passer de l'une à l'autre.

Une fois ces points acquis, nous avons cherché la transformation associée à une séquence de transformations en forme abstraite.

L'implémentation, elle, n'utilise que les formes concrètes. Dans la plupart des cas, elle est très efficace puisqu'aucun des exemples fournis n'a dépassé une seconde de temps de calcul, et puisque dans les cas énormes et sans ambiguïté, les temps d'analyse sont de l'ordre d'une demi-seconde par composition d'affectation. Toutefois, en présence de nombreuses ambiguïtés, les temps d'analyse deviennent importants et il faut chercher des voies d'amélioration.

Nous avons dorénavant les briques de base, c'est à dire l'affectation et la combinaison par la séquence et l'alternative, pour construire l'analyse des boucles. C'est l'objet du chapitre suivant.

Chapitre 4

Itérations

Nous avons raisonné jusqu'ici sur des éléments de programme simples : les affectations, les séquences et les alternatives. Tous ces éléments ont en commun la facilité d'obtention d'une tranformation synthétisant *exactement* leur effet : étant donnée une mémoire, nous sommes capables de fournir de manière exhaustive les modifications opérées sur cette mémoire.

Supposons que les programmes candidats à la parallélisation ne soient composés que de ces éléments : si nous prenons un ordinateur actuel (du type station de travail ou mini-ordinateur par exemple) avec une mémoire de 256 millions de mots, capable d'exécuter 10 millions d'instructions par secondes. Le plus lent programme que nous pourrions former sur cet ordinateur serait constitué de 256 millions d'instructions à la file, totalisant ainsi un temps d'exécution d'environ 25 secondes. Dans ces conditions, plus besoin d'architecture parallèle, plus besoin de super-ordinateurs, plus besoin de paralléliseurs ! Dans ce contexte, le temps d'analyse du programme par la méthode décrite dans le chapitre précédent serait très largement supérieur au temps d'exécution... Doit-on conclure que ce que nous avons développé ne sert à rien ? Heureusement non.

Dans la très grande majorité des cas, les programmes candidats à la parallélisation contiennent des boucles. À la différence des structures étudiées dans le précédent chapitre, les structures itératives sont un obstacle à la compréhension automatique du programme au niveau de chaque variable : nous ne pouvons en effet pas prédire l'état exact de la mémoire, variable par variable, en fin d'exécution. Nous sommes donc obligés de nous placer à un niveau supérieur : nous n'essayerons plus d'affecter une valeur, fût-elle symbolique, à une variable, mais nous tenterons de synthétiser l'effet d'une boucle, c'est à dire étudier la variation des valeurs d'une variable et cerner l'ensemble des variables modifiées. Ceci n'est possible que grâce à l'étude que nous avons effectuée dans le chapitre précédent.

Rappelons que le langage C n'offre pas de vraie boucle de type *for* comme en Pascal ou en Fortran (boucle de type *do*). Ceci nous a conduit à ne considérer que les boucles de type *while*, ce qui est contraignant en parallélisation où l'essentiel des travaux porte sur les boucles *do*.

Dans ce chapitre, nous allons donc chercher à synthétiser l'effet des boucles, en nous fixant comme but de retrouver les boucles de type *for* déguisées en boucles de type *while*. Ceci étant fait, nous serons à même d'appliquer au langage C tous les travaux majeurs sur la parallélisation.

Nous construisons, dans la première partie de ce chapitre, la transformation associée à la boucle. Bien évidemment, celle-ci n'est pas calculable dans le cas général, mais ce calcul va servir de point de repère pour le reste du chapitre.

La deuxième partie introduit des notations nécessaires pour définir, dans la troisième partie, la notion d'approximation de transformation. Une fois cette notion appliquée à la transformation de plusieurs exécution du corps de la boucle, nous pouvons construire une méthode pour trouver une telle approximation. La méthode que nous proposons conduit à un calcul non itératif (donc rapide) de l'approximation, par l'approximation successive de tous les composants de la transformation associée au corps de la boucle.

La quatrième partie, utilisant les résultats de la précédente, achève l'analyse par la construction de l'approximation de la transformation associée à la boucle dans sa totalité.

La cinquième partie est consacrée à l'étude de quelques exemples et à la description de l'état actuel de la réalisation.

4.1 Transformation associée à la boucle

La syntaxe du langage d'analyse n'autorise que les boucles du type :

```
while (e)
      c
```

où e est l'expression qui gouverne la boucle, et c est une ou plusieurs instructions constituant le corps.

Dans la suite, nous supposons que le programme est valide, c'est à dire qu'il s'arrête en un temps fini, ce qui nous permet d'affirmer qu'une boucle comme celle-ci est parcourue un nombre θ de fois. Dans le cas général, nous ne pouvons pas connaître la valeur de θ.

Une boucle parcourue θ fois peut être "déroulée", c'est à dire transformée en une succession de θ tests et corps :

```
if (e)
{
      c
      if (e)
      {
```

```
        ...
            if (e)
            {
                c
                if (e)
                    c
            }

    }
}
```

Calculons la transformation T_b associée à cette boucle. Pour cela, supposons que la transformation liée au corps c soit T_c, et partons du niveau le plus profond, c'est à dire du dernier tour de la boucle.

Le test se traduit par :

$$\begin{array}{rcl} T_{\theta,c}(M) & = & T_c(M) \\ T_{\theta,e}(M) & = & \left\{ \begin{array}{ll} \text{si } L(e,M) \neq 0 \text{ alors} & T_{\theta,c}(M) \\ \text{sinon} & M \end{array} \right. \end{array}$$

Comme nous savons que la boucle est parcourue θ fois exactement, le cas *sinon* n'a pas lieu d'être. Composons ensuite avec les $(\theta-1)$-èmes corps c et test e :

$$\begin{array}{rcl} T_{\theta-1,c}(M) & = & (T_c \circ T_{\theta,e})(M) \\ & = & T_{\theta,e}(T_c(M)) \\ & = & \text{si } L(e,T_c(M)) \neq 0 \text{ alors } T_{\theta,c}(T_c(M)) \\ & = & \text{si } L(e,T_c(M)) \neq 0 \text{ alors } (T_c \circ T_{\theta,c})(M) \\ & = & \text{si } L(e,T_c(M)) \neq 0 \text{ alors } (T_c \circ T_c)(M) \\ & = & \text{si } L(e,T_c(M)) \neq 0 \text{ alors } T_c^2(M) \\ T_{\theta-1,e}(M) & = & \text{si } L(e,M) \neq 0 \text{ alors } T_{\theta-1,c}(M) \\ & = & \text{si } L(e,M) \neq 0 \wedge L(e,T_c(M)) \neq 0 \text{ alors } T_c^2(M) \end{array}$$

Nous notons T^i les i compositions de la fonction T. La fonction T^0 sera par convention $Id_{\mathcal{M}}$, la fonction identité sur $\mathcal{M}$.

Nous avons simplifié les cas *sinon* pour la raison énoncée ci-dessus. En généralisant ce calcul jusqu'au niveau du premier test effectué, cela donne :

$$T_{1,c}(M) \quad = \quad \text{si} \left(\bigwedge_{0 \leq i < \theta} (L(e, T_c^i(M)) \neq 0) \right) \text{ alors } T_c^\theta(M)$$

Ce qui correspond à l'idée que nous nous faisions *a priori* : pour que la boucle soit effectuée θ fois, il faut que le test e soit vrai θ fois. Mais le deuxième test est effectué sur la mémoire transformée par le corps, le troisième est effectué sur la mémoire transformée par les deux premières exécutions du corps, etc. Grâce à la suppression des effets de bord, les tests ne modifient pas la mémoire, donc la transformation finale n'est que le fruit de θ exécutions du corps c, c'est à dire θ fois la composition de T_c.

Nous voyons que le développement de $T_{1,c}$ nous conduira à une forme abstraite de transformation comme celles que nous avons vues lors du chapitre précédent. Le $\bigwedge_{0 \leq i < \theta}(L(e, T_c^i(M)) \neq 0)$ figurera directement comme une garde.

Nous pouvons ajouter un élément à cette garde : en effet, la boucle n'est pas parcourue une $(\theta + 1)$-ème fois, ce qui nous autorise à écrire enfin la transformation T :

$$T_b(M) = \text{si} \left(\bigwedge_{0 \leq i < \theta} (L(e, T_c^i(M)) \neq 0) \right) \wedge L(e, T_c^\theta(M)) = 0 \text{ alors } T_c^\theta(M) \tag{4.1}$$

Cette équation nous donne la forme abstraite de la transformation correspondant à la boucle. Cependant, nous ne sommes guère plus avancés, car elle est bien évidemment incalculable dans le cas général !

Nous allons donc *approximer* la transformation T_b. Notre objectif est de trouver, au vu de la transformation T_c et de l'expression e, une forme approximative de la transformation, synthétisant ainsi l'effet de la boucle. Le reste de ce chapitre est consacré au développement de cette idée.

4.2 Notations

4.2.1 Nouveaux éléments

Pour procéder à ces approximations, introduisons de nouveaux éléments dans les ensembles définis dans le chapitre précédent. Il s'agit de :

- $\perp_n \in \mathcal{A}_n$, avec $n \geq 0$;
- $\perp_{-1} \in \mathcal{V}$;

De manière informelle, une valeur $\perp$ (prononcer *bottom*) indique que nous n'avons pas de certitude quant à la valeur exacte. Ceci ne pouvait arriver dans le chapitre précédent, où nous avions en permanence toutes les informations disponibles pour lever toutes les ambiguïtés. Ce n'est plus le cas dans les boucles, où nous devons au contraire disposer d'un outil nous permettant de dire : "nous ne savons plus".

La partition des adresses que nous avons introduite permet d'éviter l'introduction d'un élément $\perp \in \mathcal{A} + \mathcal{V}$, *bottom* absolu. En effet, le typage du langage C nous indique en permanence de manière statique (c'est à dire au vu du texte du programme) à quel ensemble $\mathcal{A}_n$ ou $\mathcal{V}$ appartient le résultat d'un calcul. Nous pouvons d'ailleurs, en utilisant le système complet de typage offert par le langage C, profiter encore plus de cette structuration de la mémoire. Nous ne l'avons pas fait dans l'exposé, pour alléger l'écriture, mais c'est ce qui a été réalisé.

Les opérations sur ces ensembles $\mathcal{A}_n$ et $\mathcal{V}$ doivent être étendues à ces nouveaux éléments :

- Pour l'ensemble $\mathcal{V}$:
  - $\perp_{-1}$ operateur-binaire $v = \perp_{-1}$ (avec $v \in \mathcal{V}$),
  - operateur-unaire $\perp_{-1} = \perp_{-1}$;
- Pour les ensembles $\mathcal{A}_n$ (avec $n \geq 0$) :
  - $M(\perp_n) = \perp_{n-1}$
  - $\perp_n$ operateur-binaire $a = \perp_{-1}$ (avec $a \in \mathcal{A}_n$),
  - $\perp_n + v = \perp_n$ (avec $v \in \mathcal{V}$),
  - $\perp_n - v = \perp_n$ (avec $v \in \mathcal{V}$) ;

  L'opérateur de différence d'adresses est particulier : $\perp_n - a$ n'est pas définie puisque la différence n'a de sens que lorsque les deux adresses sont dans le même objet.

  D'autre part, lorsqu'on ajoute une quantité inconnue à une adresse, la structuration de la mémoire nous assure que le résultat pointe toujours dans l'objet initial. Nous pouvons donc en déduire que l'expression $a + \perp_{-1}$ (où a est une adresse appartenant à $\mathcal{A}_n$) reste inchangée.

  Lorsqu'on a quelques précisions supplémentaires sur l'adresse, c'est à dire lorsqu'on sait à partir de quels éléments du produit cartésien $\mathcal{I} \times \mathcal{E}$ elle est formée, nous pouvons déduire certaines règles (avec i étant un élément de $\mathcal{I}$ tel que $\langle i, 0\rangle$ soit un élément de $\mathcal{A}_n$) :
  - $\langle i, v\rangle + \perp_{-1} = \langle i, \perp_{-1}\rangle + v = \langle i, \perp_{-1}\rangle$ (avec $v \in \mathcal{V}$),
  - $M(\langle i, \perp_{-1}\rangle) = \perp_{n-1}$;

  La différence d'adresses, dans le cas d'une adresse inconnue à l'intérieur d'un objet, n'est là encore définie que si les deux adresses sont à l'intérieur du même objet. Dans ce cas, elle équivaut à la différence des déplacements à l'intérieur de l'objet.

Pour simplifier l'écriture, nous utiliserons $\perp$ lorsqu'une valeur est $\perp_n$, $\perp_{-1}$, ou une adresse de la forme $\langle i, \perp_{-1}\rangle$.

Nous aurons besoin d'un outil pour la détection des ambiguïtés dans nos transformations, et pour permettre d'assurer la consistance. Partant de la constatation que l'adresse $\perp_n$ regroupe toutes les adresses de l'ensemble $\mathcal{A}_n$ (pour $n \geq 0$), et que les adresses $\langle i, \perp_{-1}\rangle$ sont en fait toutes les adresses de l'ensemble $\{\langle i, v\rangle \in \mathcal{I} \times \mathcal{V}/0 \leq v < d_m(i)\}$, nous pouvons généraliser et affirmer pour l'occasion que toute adresse représente un ensemble de case mémoire. Si a est une adresse, notons $E(a)$ l'ensemble associé. Avec cette vision ensembliste, nous pouvons définir l'égalité (à ne pas confondre avec l'opérateur `==` du langage) de deux adresses comme :

Définition 4.1 $\forall\langle a_1, a_2\rangle \in \mathcal{A}_n^2, a_1 = a_2 \Longleftrightarrow E(a_1) \cap E(a_2) \neq \emptyset$

La définition de l'opérateur $\neq$ est la négation de cette définition. En termes simples, deux adresses ne sont pas égales si on peut affirmer en toute sécurité qu'elles ne pourront jamais être égales, quelque soient les valeurs exactes que l'on peut mettre à la place des $\perp$. Dans le cas contraire, elle sont égales.

Par exemple :

- $\langle \texttt{\_tab}, \perp_{-1}\rangle = \langle \texttt{\_tab}, 0\rangle$
- $\langle \texttt{\_t1}, \perp_{-1}\rangle \neq \langle \texttt{\_t2}, \perp_{-1}\rangle$
- $\langle \texttt{\_t1}, \perp_{-1}\rangle = \perp_0$ (avec $\langle \texttt{\_t1}, 0\rangle \in \mathcal{A}_0$)

On peut vérifier que la définition 4.1, dans le cas de deux adresses exactes, correspond bien au résultat attendu, étant donné que l'ensemble $E(a)$ correspond alors au singleton $\{a\}$.

4.2.2 Relation d'ordre

Définissons à présent une relation d'ordre partielle sur $\mathcal{A} + \mathcal{V}$. Cette relation d'ordre va nous permettre de formaliser la notion de précision que l'introduction de ces nouveaux éléments a introduite.

Cette relation, notée $x \sqsubseteq y$ (prononcer x est moins précis que y), réflexive et transitive, est définie par :

$$\begin{array}{rcl}
\perp_{-1} & \sqsubseteq & v \\
\perp_{-1} & \sqsubseteq & \perp_{-1} \\
\langle i, \perp_{-1}\rangle & \sqsubseteq & \langle i, v\rangle \\
\langle i, \perp_{-1}\rangle & \sqsubseteq & \langle i, \perp_{-1}\rangle \\
\perp_n & \sqsubseteq & \langle i, v\rangle \\
\perp_n & \sqsubseteq & \langle i, \perp_{-1}\rangle \\
\perp_n & \sqsubseteq & \perp_n
\end{array}$$

où :

$$\begin{array}{rcl} v & \in & \mathcal{V} - \{\bot_{-1}\} \\ i & \in & \mathcal{I} \\ a & \in & \mathcal{A}_n - (\{\bot_n\} \cup \{\langle i, \bot_{-1}\rangle / i \in \mathcal{I}\}) \end{array}$$

4.3 Recherche d'une approximation

Nous avons vu plus haut que la transformation T_b associée à une boucle n'est pas calculable. En effet, d'une part le nombre de tours de la boucle ne nous est pas connu, et d'autre part le temps pris pour calculer les θ compositions de la transformation T_c serait très largement supérieur au temps pris par l'exécution directe de la boucle. Nous avons conclu à la nécessité de chercher une approximation de T_b.

Qu'appelons-nous une "approximation" ?

4.3.1 Définition d'une approximation

Approximation d'une transformation

Comment peut-on définir une approximation d'une transformation ? C'est facile : on sait moins de choses dans une approximation que dans la transformation originale. Nous allons donc faire rentrer une certaine part d'incertitude dans nos transformations, matérialisée sous forme de nos nouvelles valeurs regroupées sous le symbole $\bot$. On peut alors définir ce qu'est une approximation :

Définition 4.2 *Une transformation T' est une approximation d'une transformation T si et seulement si :*

$$\forall \langle a, M\rangle \in \mathcal{A} \times \mathcal{M}, (T'(M))(a) \sqsubseteq (T(M))(a)$$

Certaines approximations sont "meilleures" que d'autres, c'est à dire sont plus précises. Introduisons une relation d'ordre (partielle) nous permettant de formaliser cette notion :

Définition 4.3 *La transformation T' est plus précise que T'' si et seulement si T'' est une approximation de T'.*

Notons pour l'anecdote que toute transformation possède une approximation. En effet, quelque soit le programme, la transformation :

$$T_\perp : \mathcal{M} \longrightarrow \mathcal{M}$$
$$M \longmapsto T_\perp(M) = \left(\begin{array}{ll} \mathcal{A} & \longrightarrow \mathcal{A} + \mathcal{V} + \top \\ a & \longmapsto \perp \end{array} \right) \tag{4.2}$$

est une approximation de sa transformation. En réalité, cette fonction est écrite sous forme simplifiée : en toute rigueur, il aurait fallu distinguer suivant les adresses a pour renvoyer le $\perp$ correspondant à l'espace $\mathcal{A}_n$ adéquat.

Cette transformation $T_\perp$ étant l'approximation de toute transformation, il s'agit de la *moins* précise. C'est une propriété intéressante, car l'espace des transformations doté de la relation d'ordre partielle "est moins précis que" possède une borne inférieure. Nous sommes assurés de toujours trouver au moins une approximation valide, même si celle-ci ne nous apporte pas beaucoup de renseignements.

Approximation d'une transformation associée à θ tours de boucle

Avec la définition 4.2, nous serons incapables de prouver, dans le cas général, qu'une transformation T' est une approximation de la transformation T_b associée à une boucle, pour la simple raison que nous ne pouvons calculer T_b !

Il faut donc trouver une autre formulation pour vérifier qu'une transformation est une approximation de la transformation T_c^θ associée à θ tours de boucle (nous supposerons dans tout le reste de ce sous-chapitre que $\theta \geq 1$).

Pour cela, il suffit de constater que l'approximation doit satisfaire deux conditions, issues d'un raisonnement par récurrence : ce doit être une approximation pour le corps de la boucle, et cette approximation doit être stable par le corps. Écrivons ces deux conditions :

Propriété 4.1 *Une transformation T' (fonction du nombre θ de tours de boucle parcourus) est une approximation de la transformation T_c^θ associée à l'exécution de θ tours de la boucle de corps c si :*

$$\left\{ \begin{array}{l} \forall M \in \mathcal{M}, T'(M,1) \textit{ est une approximation de } T_c(M) \\ \qquad \textit{et} \\ \forall M \in \mathcal{M}, T_c(T'(M,\theta)) = T'(M,\theta+1) \end{array} \right. \tag{4.3}$$

Nous avons utilisé dans cet énoncé le nombre θ de tours de boucle parcourus. Nous ne saurons le calculer que dans un faible nombre de cas, ceux où nous avons reconnu une boucle du type *for*. Il existe de très (trop) nombreux autres cas où la transformation approchée ne dépendra pas de θ. Cette condition s'écrira alors sous forme dégénérée :

$$\left\{\begin{array}{l}\forall M \in \mathcal{M}, T'(M) \text{ est une approximation de } T_c(M) \\ \qquad \text{et} \\ \forall M \in \mathcal{M}, T_c(T'(M)) = T'(M)\end{array}\right. \tag{4.4}$$

Comment trouver une approximation ?

Il n'existe pas, à notre connaissance, de méthode générale de recherche d'une approximation. En mathématique, certains calculs (recherche de primitives, recherche de point fixe) nécessitent une part d'empirisme. La solution que nous proposons relève de la même démarche. Nous proposerons une méthode qui donne de bons résultats pour le but recherché, mais qui ne peut prétendre à la généralité pour d'autres problèmes que le nôtre. Nous suivrons un certain nombre de modifications en vérifiant que le résultat est bien conforme à la propriété 4.1. Nous ne garantirons pas la "meilleure" approximation, mais nous en garantirons une.

En quelques mots, la démarche que nous allons suivre consiste à partir de la transformation T_c et à en trouver une approximation qui satisfasse la propriété 4.1, et en particulier dont tous les constituants (fonctions f_i, g_i et h_i) soient stables par T_c.

Exemple

Prenons un exemple (voir figure 4.1) pour illustrer les différentes modifications que nous allons faire subir aux transformations dans ce chapitre. Nous verrons ainsi concrètement comment trouver une approximation satisfaisant la propriété 4.1.

```
while (i < n)
{
      if (i % 2 == 0)
            *p = *p + 1 ;
      *q = *q + 1 ;
      p = p + 1 ;
      q = q + 1 ;
      i = i + 1 ;
}
```

Figure 4.1: Exemple de programme

La transformation (après simplifications triviales) associée au corps de la boucle est :

$$
\begin{cases}
\text{si } a = M(\langle \texttt{\_q}, 0\rangle) \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(M(\langle \texttt{\_q}, 0\rangle)) + 2 \\
\text{sinon si } a = M(\langle \texttt{\_q}, 0\rangle) \text{ alors} & M(M(\langle \texttt{\_q}, 0\rangle)) + 1 \\
\text{sinon si } a = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(M(\langle \texttt{\_p}, 0\rangle)) + 1 \\
\text{sinon si } a = \langle \texttt{\_p}, 0\rangle \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\
\text{sinon si } a = \langle \texttt{\_q}, 0\rangle \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\
\text{sinon si } a = \langle \texttt{\_i}, 0\rangle \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1 \\
\text{sinon} & M(a)
\end{cases}
\tag{4.5}
$$

Si nous entamons l'opération de concrétisation jusqu'à l'équation 3.9, nous obtenons (toujours après simplifications) :

$$
\begin{cases}
\text{si } a = M(\langle \texttt{\_q}, 0\rangle) \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(M(\langle \texttt{\_q}, 0\rangle)) + 2 \\
\text{si } a = M(\langle \texttt{\_q}, 0\rangle) \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(M(\langle \texttt{\_q}, 0\rangle)) + 1 \\
\text{si } a = M(\langle \texttt{\_q}, 0\rangle) \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(M(\langle \texttt{\_q}, 0\rangle)) + 1 \\
\text{si } a = M(\langle \texttt{\_q}, 0\rangle) \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(M(\langle \texttt{\_q}, 0\rangle)) + 1 \\
\text{si } a = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(M(\langle \texttt{\_p}, 0\rangle)) + 1 \\
\text{si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 = 0 \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge M(\langle \texttt{\_i}, 0\rangle)\%2 \neq 0 \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1
\end{cases}
\tag{4.6}
$$

Ce qui permet d'écrire la forme concrète :

$$\tau_c = \left| \begin{array}{l} \{\texttt{p == q}, \texttt{i \% 2 == 0}\} \;/\; [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{i} \leftarrow \texttt{i+1}, \texttt{*q} \leftarrow \texttt{*q+2}] \\ \{\texttt{p == q}, \texttt{i \% 2 != 0}\} \;/\; [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{i} \leftarrow \texttt{i+1}, \texttt{*q} \leftarrow \texttt{*q+1}] \\ \{\texttt{p != q}, \texttt{i \% 2 == 0}\} \;/\; [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{i} \leftarrow \texttt{i+1}, \texttt{*q} \leftarrow \texttt{*q+1}, \texttt{*p} \leftarrow \texttt{*p+1}] \\ \{\texttt{p == q}, \texttt{i \% 2 != 0}\} \;/\; [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{i} \leftarrow \texttt{i+1}, \texttt{*q} \leftarrow \texttt{*q+1}] \end{array} \right. \tag{4.7}$$

Nous allons voir maintenant les étapes nécessaires pour, partant d'une transformation telle que celle-ci, arriver à une approximation de la transformation associée à la boucle.

4.3.2 Prise en compte du test

Lorsque nous avons défini la notion d'approximation de la transformation associée à la boucle, nous n'avions mentionné que son corps et non son test. Un de nos soucis est de limiter l'explosion du nombre de termes dans une transformation. Pour éviter de conserver trop de gardes caduques, une modification simple consiste à retirer tous les termes contraires au test.

```
while (p < q)
{
      *p = *p + 1 ;
      *q = *q + 2 ;
      p = p + 1 ;
      q = q - 1 ;
}
```

Figure 4.2: Exemple de programme

Prenons un exemple (voir figure 4.2). La transformation associée est (sous forme concrète) :

$$\tau = \left| \begin{array}{l} \{\texttt{p == q}\} \;/\; [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q-1}, \texttt{*p} \leftarrow \texttt{*p+3}] \\ \{\texttt{p != q}\} \;/\; [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q-1}, \texttt{*p} \leftarrow \texttt{*p+1}, \texttt{*q} \leftarrow \texttt{*q+2}] \end{array} \right.$$

Or, le test qui gouverne la boucle nous apprend que, dans le corps de celle-ci, le pointeur p ne peut égaler le pointeur q. La deuxième ligne n'est donc jamais sélectionnée, la garde est toujours fausse (quel que soit le tour de boucle). Nous pouvons donc retirer le terme et notre transformation est donc considérablement simplifiée !

4.3.3 Retrait des gardes non stables

La première approximation que nous allons faire sur la transformation T_c consiste à retirer les gardes qui dépendent d'un tour de boucle donné.

Notion de stabilité

Définition 4.4 *Une expression $e(M)$ est dite* stable *par le corps de la boucle si et seulement si :*

$$\forall M \in \mathcal{M}, e(M) = e(T_c(M))$$

En particulier, pour les gardes qui sont formées d'expressions de la forme $e_1(M)$ op $e_2(M)$, où *op* est l'opérateur d'égalité (= ou $\neq$), la stabilité signifie que :

$$\forall M \in \mathcal{M}, (e_1(M) \text{ op } e_2(M)) = (e_1(T_c(M)) \text{ op } e_2(T_c(M)))$$

```
while (i < n)
{
      *r = *r + 1 ;
      *p = *p + 1 ;
      *q = *q + 1 ;
      p = p + 1 ;
      q = q + 1 ;
}
```

Figure 4.3: Exemple de programme

Expliquons cette définition sur un exemple (voir figure 4.3). La transformation associée au corps de la boucle est (représentée ici sous forme concrète pour abréger l'écriture) :

$$\tau = \left| \begin{array}{l}
\{\texttt{p == r}, \texttt{q == r}, \texttt{p == q}\} \ / \ [\texttt{*r} \leftarrow \texttt{*r+3}, \texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}] \\
\{\texttt{p == r}, \texttt{q != r}, \texttt{p != q}\} \ / \ [\texttt{*r} \leftarrow \texttt{*r+2}, \texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{*q} \leftarrow \texttt{*q+1}] \\
\{\texttt{p != r}, \texttt{q == r}, \texttt{p != q}\} \ / \ [\texttt{*r} \leftarrow \texttt{*r+2}, \texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{*p} \leftarrow \texttt{*p+1}] \\
\{\texttt{p != r}, \texttt{q != r}, \texttt{p == q}\} \ / \ [\texttt{*r} \leftarrow \texttt{*r+1}, \texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{*p} \leftarrow \texttt{*p+2}] \\
\{\texttt{p != r}, \texttt{q != r}, \texttt{p != q}\} \ / \ [\texttt{*r} \leftarrow \texttt{*r+1}, \texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{*p} \leftarrow \texttt{*p+1}, \texttt{*q} \leftarrow \texttt{*q+1}]
\end{array} \right.$$

Isolons deux gardes particulières :

- $M(\langle \_p, 0\rangle) = M(\langle \_q, 0\rangle)$ et sa négation

  Cette expression est stable par le corps de la boucle, puisque si nous la considérons après un tour, nous avons la nouvelle expression : $M(\langle \_p, 0\rangle) + 1 = M(\langle \_q, 0\rangle) + 1$, ce qui revient évidemment au même ;

- $M(\langle \_p, 0\rangle) = M(\langle \_r, 0\rangle)$ et sa négation

  Cette expression n'est pas stable par le corps de la boucle, puisque après une itération, nous avons : $M(\langle \_p, 0\rangle) + 1 = M(\langle \_r, 0\rangle)$, ce qui est tout à fait différent.

Notons que le test gouvernant la boucle ne doit pas être stable par le corps de la boucle, sinon la boucle n'est jamais exécutée ou c'est une boucle qui ne s'arrête pas.

Retrait des gardes

Seules les gardes stables par le corps de la boucle peuvent rester dans une approximation de la transformation associée au corps de la boucle. Nous allons donc chercher à éliminer les gardes instables de notre transformation. Pour cela, sans pousser jusqu'à la concrétisation, mettons T_c sous la forme de l'équation 3.9, exposant ainsi toutes les gardes sous forme d'expressions $e_1(M) = e_2(M)$ où $e_1(M) \neq e_2(M)$.

Comment éliminer ces gardes ? À partir de T_c mise sous sa nouvelle forme, éliminons toutes les paires (e_1, e_2) instables par le corps de la boucle. Nous introduisons de ce fait une ambiguïté : Si nous avons les deux termes :

$$\begin{cases} \text{si } a = f(M) \wedge g(M) \wedge e_1(M) = e_2(M) \text{ alors } h_i(M) \\ \text{si } a = f(M) \wedge g(M) \wedge e_1(M) \neq e_2(M) \text{ alors } h_j(M) \end{cases} \tag{4.8}$$

et si nous retirons la paire (e_1, e_2), que vaudra $(T_c(M))(f(M))$ lorsque la garde $g(M)$ sera vraie ?

Pour lever l'ambiguïté, introduisons l'incertitude. Ou plus exactement, la certitude que nous ne connaissons pas la valeur exacte :

$$\text{si } a = f(M) \wedge g(M) \text{ alors } v \qquad\qquad \text{où } v = \perp$$

Si nous ne savons pas quelle est la valeur exacte, nous pouvons en revanche affirmer que cette case mémoire a été modifiée par la transformation. En ce sens, les valeurs $\perp$ sont différentes de $\top$ qui indique une absence d'information pour la case mémoire concernée, c'est à dire une non-affectation de cette case.

Notons que cette modification de la mémoire est inverse de celle que nous avions faite dans la section 3.4.3, qui visait à introduire de nouveaux termes. Nous allions dans le sens d'une plus grande précision. Ici, c'est l'inverse puisque nous cherchons une approximation.

Laquelle des valeurs $\perp$ mettons-nous à la place de $h_i(M)$ et $h_j(M)$?

- tout d'abord, si $\forall M \in \mathcal{M}, h_i(M) = h_j(M)$, il n'y a pas besoin d'introduire d'imprécision. Il suffit de conserver $v = h_i(M)$;
- ensuite, si $h_i(M) \in \mathcal{A}$ (ce qui entraîne par conséquent $h_j(M) \in \mathcal{A}$), et si la différence $h_i(M) - h_j(M)$ est définie (c'est à dire si $h_i(M)$ et $h_j(M)$ pointent à l'intérieur du même objet), alors nous pouvons transformer v en $h_i(M) + \perp_{-1}$ (notons que $h_i(M) + \perp_{-1} = h_j(M) + \perp_{-1}$) ;
- sinon, nous sommes obligés de conclure à l'incertitude certaine : $v = \perp_n$ si $h_i(M) \in \mathcal{A}_n$, $v = \perp_{-1}$ si $h_i(M) \in \mathcal{V}$.

Cette opération va se renouveler fréquemment dans la suite de l'exposé. Nous proposons de l'appeler *opération de rencontre*, et nous noterons $h_i(M) \otimes h_j(M)$ la valeur que nous avons définie ci-dessus.

Ces trois possibilités vont dans le sens d'une moins grande précision. Il n'est pas forcément facile de réaliser ces tests. En cas de doute, il faut avoir une valeur moins précise et non une valeur fausse.

Retrait des ambiguïtés

Dans l'équation 4.8, nous nous sommes placés dans un cas favorable : les adresses $f(M)$ sont deux fonctions égales. Dans le cas général, cependant, cette égalité n'existe pas. Reprenons l'exemple de la figure 4.3. Si nous retirons les gardes composées des paires $(M(\langle\_\mathtt{p}, 0\rangle), M(\langle\_\mathtt{r}, 0\rangle))$ et $(M(\langle\_\mathtt{q}, 0\rangle), M(\langle\_\mathtt{r}, 0\rangle))$, il va falloir regrouper les termes suivants (dans le cas où $M(\langle\_\mathtt{p}, 0\rangle) = M(\langle\_\mathtt{q}, 0\rangle)$) :

$$\left\{\begin{array}{ll}
\text{sinon si } a = M(\langle_\mathtt{r}, 0\rangle) \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(M(\langle_\mathtt{r}, 0\rangle)) + 3 \\
\text{sinon si } a = \langle_\mathtt{p}, 0\rangle \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(\langle_\mathtt{p}, 0\rangle)) + 1 \\
\text{sinon si } a = \langle_\mathtt{q}, 0\rangle \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(\langle_\mathtt{q}, 0\rangle)) + 1 \\
\text{sinon si } a = M(\langle_\mathtt{r}, 0\rangle) \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(M(\langle_\mathtt{r}, 0\rangle)) + 2 \\
\text{sinon si } a = \langle_\mathtt{p}, 0\rangle \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(\langle_\mathtt{p}, 0\rangle)) + 1 \\
\text{sinon si } a = \langle_\mathtt{q}, 0\rangle \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(\langle_\mathtt{q}, 0\rangle)) + 1 \\
\text{sinon si } a = M(\langle_\mathtt{q}, 0\rangle) \wedge M(\langle_\mathtt{p}, 0\rangle) = M(\langle_\mathtt{q}, 0\rangle) \text{ alors} & M(M(\langle_\mathtt{q}, 0\rangle)) + 1
\end{array}\right.$$

Autrement dit, nous cherchons à regrouper les deux lignes :

$$\left|\begin{array}{l}
\{\mathtt{p == r}\} \ / \ [\mathtt{*r} \leftarrow \mathtt{*r+3}, \mathtt{p} \leftarrow \mathtt{p+1}, \mathtt{q} \leftarrow \mathtt{q+1}] \\
\{\mathtt{p == r}\} \ / \ [\mathtt{*r} \leftarrow \mathtt{*r+2}, \mathtt{p} \leftarrow \mathtt{p+1}, \mathtt{q} \leftarrow \mathtt{q+1}, \mathtt{*q} \leftarrow \mathtt{*q+1}]
\end{array}\right.$$

Que constatons-nous ? Par le raisonnement exposé ci-dessus, avec l'aide de l'opérateur $\otimes$, nous pouvons prédire que la valeur affectée à l'adresse $M(\langle \texttt{\_r}, 0\rangle)$ sera $\bot_{-1}$, la valeur affectée à l'adresse $\langle \texttt{\_p}, 0\rangle$ sera $M(\langle \texttt{\_p}, 0\rangle)) + 1$, la valeur affectée à l'adresse $\langle \texttt{\_q}, 0\rangle$ sera $M(\langle \texttt{\_q}, 0\rangle)) + 1$. Mais qu'en est-il de la valeur située à l'adresse $M(\langle \texttt{\_q}, 0\rangle)$? Il n'y a aucun conflit, donc on pourrait conclure que cette valeur doit être $M(M(\langle \texttt{\_q}, 0\rangle)) + 1$, ce qui est faux.

Comme nous l'avons mentionné plus haut, le problème vient de ce que nous avons considéré, dans l'équation 4.8, des fonctions $f(M)$ égales. Ceci nous amène à reformuler le cas général :

$$\begin{cases} \text{si } a = f_i(M) \wedge g(M) \wedge e_1(M) = / \neq e_2(M) \text{ alors } h_i(M) \\ \text{si } a = f_j(M) \wedge g(M) \wedge e_1(M) = / \neq e_2(M) \text{ alors } h_j(M) \end{cases} \tag{4.9}$$

Dans ces conditions, le regroupement des deux lignes ne peut s'effectuer que si les fonctions $f_i(M)$ et $f_j(M)$ peuvent être égales. Il y a deux cas à considérer :

- $\forall M \in \mathcal{M}, f_i(M) = f_j(M) \wedge g(M)$
  C'est le cas précédent, Dans ce cas, les deux lignes doivent être remplacées par :
  $$\left\{ \text{si } a = f_i(M) \wedge g(M) \text{ alors } h_i(M) \otimes h_j(M) \right. \tag{4.10}$$
- $\exists M \in \mathcal{M}, f_i(M) = f_j(M) \wedge g(M)$
  C'est le nouveau cas. Les deux lignes doivent alors être remplacées par :
  $$\begin{cases} \text{si } a = f_i(M) \wedge g(M) \text{ alors } h_i(M) \otimes h_j(M) \\ \text{si } a = f_j(M) \wedge g(M) \text{ alors } h_i(M) \otimes h_j(M) \end{cases} \tag{4.11}$$
  Notons que la nouvelle transformation que nous obtenons est valide, car bien qu'ayant une ambiguïté sur la sélection de l'adresse, le contenu de la case mémoire est en fait le même.

D'autre part, des ambiguïtés peuvent surgir de la ligne *sinon M(a)*. En effet, considérons l'exemple de la figure 4.1 qui illustre particulièrement bien ce problème : l'affectation à `*p` est régie par le test du `if`. Ce test, qui devient une paire d'expressions $(M(\langle \texttt{\_i}, 0\rangle)\%2, 0)$ dans une garde est instable par le corps de la boucle. Si on retire purement et simplement ce test, on déduit que l'affectation à `*p` est accomplie à chaque tour de boucle, ce qui est absolument faux ! En réalité, si le test n'est pas vérifié, la ligne *sinon M(a)* est sélectionnée. L'ambiguïté est ici implicite.

Définition 4.5 *Il y a ambiguïté* implicite *sur la ligne i de la transformation si on peut exhiber une mémoire M telle que :*

$$\forall j, f_i(M) \neq f_j(M) \; ou \; g_j(M) faux$$

Autrement exprimé, cela signifie que l'on peut trouver une configuration mémoire telle que nous sommes obligés de considérer la ligne *sinon M(a)*.

Dans ce cas, nous pouvons écrire la nouvelle ligne :

$$\left\{ \text{ si } a = f_i(M) \wedge g(M) \text{ alors } h_i(M) \otimes M(f_i(M)) \right. \tag{4.12}$$

Ainsi, notre ligne apparemment sans conflit en recelait quand même un, implicite, que nous avons explicité, puis résorbé.

Exactitude des modifications

Plusieurs problèmes se posent quant à l'exactitude des modifications que nous apportons à T_c pour en trouver une approximation.

De manière générale, que faire lorsque nous ne sommes pas sûrs d'un test ? La réponse est qu'il faut adopter une attitude pessimiste. Le regroupement de termes conduit à une approximation, c'est à dire une perte de précision : nous ne savons plus quelle valeur contient une case mémoire. Il vaut mieux faire trop d'approximations que d'affirmer à tort qu'une case mémoire contient une valeur exacte.

Quels sont les points sujets à problème ? Récapitulons ce que nous avons vu dans cette section :

- nous commençons par détecter les paires d'expressions non stables par le corps de la boucle. Cette détection ne peut être faite exactement par un procédé automatique. Dans certains cas, nous serons sûrs de la stabilité, car nous connaissons quelques règles de l'arithmétique, comme par exemple $(x = y) = (x + n = y + n)$. Dans tous les autres cas, nous conclurons à la non stabilité, donc il y aura approximation ;
- nous testons ensuite les adresses similaires $(f_i(M) = f_j(M))$. Cette détection est, elle aussi, impossible à réaliser automatiquement et exactement. Si nous sommes pas sûrs de l'inégalité, nous devrons procéder au regroupement, donc à la rencontre de $h_i(M)$ et $h_j(M)$;
- nous réalisons enfin l'opération de rencontre $(h_i(M) \otimes h_j(M))$. Cette opération est régie par trois règles triées par ordre de précision. Si nous ne sommes pas sûrs d'un niveau de précision, il faudra utiliser le niveau inférieur.

Nous adoptons donc une démarche très pessimiste, mais rigoureuse. À défaut d'une certitude, nous concluons à la non-connaissance. Nous voyons donc que la qualité de l'approximation dépendra directement de la qualité des méthodes de calcul symbolique et de vérification de tests que nous emploierons. Le pire qui puisse nous arriver est de trouver que nous ne savons rien (c'est à dire d'arriver à $T_\perp$), et donc de déclarer forfait pour la parallélisation. Mais nous ne trouverons pas de résultat faux.

Exemple

Reprenons l'exemple de la figure 4.1. Il n'y a qu'une paire d'expressions non stable par le corps de la boucle, il s'agit de $(M(\langle \texttt{\_i}, 0\rangle)\%2, 0)$. La paire $(M(\langle \texttt{\_q}, 0\rangle), M(\langle \texttt{\_p}, 0\rangle))$ ne l'est pas puisque, après un tour de boucle, elle devient $(M(\langle \texttt{\_q}, 0\rangle) + 1, M(\langle \texttt{\_p}, 0\rangle) + 1)$, ce qui revient au même.

Il faut donc faire disparaître toute utilisation de la paire $(M(\langle \texttt{\_i}, 0\rangle)\%2, 0)$. Repartons de l'équation 4.6, retirons ces paires instables, et réordonnons les lignes suivant les ambiguïtés qui sont nées de ce retrait (notons que l'ordre des lignes dans l'équation 4.6 est le même, c'est le hasard) :

$$\begin{cases}
\text{si } a = M(\langle \texttt{_q}, 0\rangle) \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(M(\langle \texttt{_q}, 0\rangle)) + 2 \\
\text{si } a = M(\langle \texttt{_q}, 0\rangle) \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(M(\langle \texttt{_q}, 0\rangle)) + 1 \\
\text{si } a = M(\langle \texttt{_q}, 0\rangle) \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(M(\langle \texttt{_q}, 0\rangle)) + 1 \\
\text{si } a = M(\langle \texttt{_q}, 0\rangle) \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(M(\langle \texttt{_q}, 0\rangle)) + 1 \\
\text{si } a = M(\langle \texttt{_p}, 0\rangle) \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(M(\langle \texttt{_p}, 0\rangle)) + 1 \\
\text{si } a = \langle \texttt{_p}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_p}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_p}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_p}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_p}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_q}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_q}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_q}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_q}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_q}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_i}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_i}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_i}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) = M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_i}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_i}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_i}, 0\rangle) + 1 \\
\text{si } a = \langle \texttt{_i}, 0\rangle \wedge M(\langle \texttt{_q}, 0\rangle) \neq M(\langle \texttt{_p}, 0\rangle) \text{ alors} & M(\langle \texttt{_i}, 0\rangle) + 1
\end{cases} \tag{4.13}$$

Les première et deuxième lignes sont ambiguës, les troisième et quatrième aussi. La cinquième correspond au cas d'ambiguïté implicite que nous avons évoqué plus haut. Ensuite, les lignes 6 et 7 sont ambiguës, 8 et 9, etc.

Regroupons maintenant les lignes suivant la relation "est ambiguë avec", pour donner :

$$\begin{cases} \text{si } a = M(\langle \_\mathtt{q}, 0\rangle) \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(M(\langle \_\mathtt{q}, 0\rangle)) + \perp_{-1} \\ \text{si } a = M(\langle \_\mathtt{q}, 0\rangle) \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(M(\langle \_\mathtt{q}, 0\rangle)) + 1 \\ \text{si } a = M(\langle \_\mathtt{p}, 0\rangle) \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(M(\langle \_\mathtt{p}, 0\rangle)) + \perp_{-1} \\ \text{si } a = \langle \_\mathtt{p}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{p}, 0\rangle) + 1 \\ \text{si } a = \langle \_\mathtt{p}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{p}, 0\rangle) + 1 \\ \text{si } a = \langle \_\mathtt{q}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{q}, 0\rangle) + 1 \\ \text{si } a = \langle \_\mathtt{q}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{q}, 0\rangle) + 1 \\ \text{si } a = \langle \_\mathtt{i}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{i}, 0\rangle) + 1 \\ \text{si } a = \langle \_\mathtt{i}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{i}, 0\rangle) + 1 \end{cases} \tag{4.14}$$

La forme concrète, toujours plus facile à lire, est :

$$\left| \begin{array}{l} \{\mathtt{p\ ==\ q}\} \ / \ [\mathtt{p} \leftarrow \mathtt{p+1}, \mathtt{q} \leftarrow \mathtt{q+1}, \mathtt{i} \leftarrow \mathtt{i+1}, \mathtt{*q} \leftarrow \mathtt{*q+}\perp_{-1}] \\ \{\mathtt{p\ !=\ q}\} \ / \ [\mathtt{p} \leftarrow \mathtt{p+1}, \mathtt{q} \leftarrow \mathtt{q+1}, \mathtt{i} \leftarrow \mathtt{i+1}, \mathtt{*q} \leftarrow \mathtt{*q+1}, \mathtt{*p} \leftarrow \mathtt{*p+}\perp_{-1}] \end{array} \right. \tag{4.15}$$

Nos gardes sont maintenant stables par le corps de la boucle, et nous avons trouvé une approximation de T_c. Nous avons donc partiellement vérifié les conditions de la propriété 4.1.

Conclusion

Récapitulons brièvement les modifications que nous avons fait subir à la transformation T_c dans cette section :

1. nous avons identifié et retiré les gardes non stables par le corps de la boucle ;
2. puis nous avons détecté des ensembles d'adresses compatibles, c'est à dire qui peuvent être égales ;
3. ensuite, nous avons partitionné la transformation suivant des ensembles de lignes de gardes égales et d'adresses compatibles ;
4. et enfin, nous avons regroupé ces ensembles en un ou plusieurs termes avec l'opération $\otimes$ de *rencontre* des valeurs.

Nous avons alors obtenu une nouvelle transformation, approximation de T_c.

4.3.4 Approximation des adresses non stables

Cette nouvelle transformation, approximation de T_c, ne comporte donc plus de gardes non stables. Mais les adresses, ainsi que les valeurs qui s'y trouvent (fonctions f_i et h_i) peuvent encore contenir des expressions non stables par le corps de la boucle.

Nous allons, dans cette section, détecter les fonctions f_i instables, et en fournir des approximations. Tout d'abord, nous allons identifier, les adresses utilisées, et les classer en "variables scalaires" et en "tableaux". Ensuite, nous approximerons les adresses de ces variables, et nous lèverons les ambiguïtés que nous aurons éventuellement fait surgir.

Reconnaissance des tableaux et des scalaires

Le critère de stabilité par le corps de la boucle (voir propriété 4.1) implique que nous devons trouver deux familles de fonctions (f'_i et h'_i) stables par T_c. Pour ce faire, composons f_i par T_c (sachant la garde g_i), comme nous l'avions fait pour le retrait des gardes non stables, et comparons $f_i(M)$ à $f_i(T_c(M))$:

- $f_i(M) - f_i(T_c(M))$ est non définie

  Cela ne peut arriver que lorsque $f_i(M) \in \mathcal{A}_n$, lorsque l'on fait la différence de deux adresses dans deux objets différents. Par exemple :

  ```
  p = & t1 [0] ;
  while (e)
  {
         *p = ··· ;
         p = & t2 [0] ;
  }
  ```

  C'est le cas le plus défavorable : un pointeur "fou". Nous ne pouvons pas cerner précisément la case mémoire initialisée au moyen du pointeur p[1].

- $f_i(M) - f_i(T_c(M))$ est définie et nulle

  Dans ce cas, la case mémoire référencée par $f_i(M)$ est une "variable scalaire". Par exemple :

  ```
  p = & tab [5] ;
  *p = 0 ;
  while (*p < n)
  {
         ...
         *p = *p + 1 ;
  }
  ```

[1]Toutefois, dans ce cas précis, nous pourrions dire que c'est soit le tableau `t1`, soit le tableau `t2`.

La case pointée par le pointeur p, bien qu'à l'intérieur d'un tableau, est utilisée comme une variable scalaire classique. Son adresse ne change pas entre deux tours de boucle.

- $f_i(M) - f_i(T_c(M))$ est définie et non nulle

  C'est l'utilisation typique d'un tableau, ou d'un pointeur utilisé pour parcourir un tableau.

Approximation

Nous avons répertorié trois sortes de comportements de variable. Voyons maintenant comment nous pouvons les approximer en une nouvelle fonction $f'_i(M)$ de façon que la $f'_i(T_c(M)) = f'_i(M)$.

- Pointeur fou

  Que faut-il en conclure ? Nous sommes dans le cas le plus défavorable. En effet, la seule chose que nous puissions affirmer en toute sécurité est que toute case mémoire de l'ensemble $\mathcal{A}_n$ est atteinte potentiellement par cette adresse. Nous écrirons donc :

$$\text{si } a = \perp_n \wedge g_i(M) \text{ alors } \perp_{n-1} \tag{4.16}$$

  Notons que $\perp_n$ n'est pas fonction d'une mémoire, donc est forcément stable par $T_c(M)$.

- Variable scalaire

  L'adresse d'une variable scalaire ne change pas entre deux tours de boucle, elle est donc stable par le corps de la boucle. Nous gardons la ligne intacte :

$$\text{si } a = f_i(M) \text{ alors } h_i(M) \tag{4.17}$$

  Puisque $f_i(T_c(M)) - f_i(M) = 0$ (par définition des variables scalaires), f_i est stable par $T_c(M)$.

- Tableau

  Nous allons faire une approximation simple : toutes les cases du tableau vont être déclarées inconnues. Ceci va donc nous donner la ligne (en supposant que $f_i(M) \in \mathcal{A}_n$) :

$$\text{si } a = f_i(M) + \perp_{-1} \text{ alors } \perp_{n-1} \tag{4.18}$$

  La stabilité de $f'_i(M) = f_i(M) + \perp_{-1}$ par $T_c(M)$ est assurée si l'on considère que :

  - $f_i(M) = \perp_n$, auquel cas $f'_i(M) = \perp_n$, stable par $T_c(M)$ comme nous l'avons vu plus haut,
  - $f_i(M) = \langle i, v \rangle$, auquel cas $f'_i(M) = \langle i, \perp_{-1} \rangle$.

Retrait des ambiguïtés

Nous avons donc remplacé chaque ligne de notre transformation par les lignes 4.16, 4.17 ou 4.18 suivant le cas. Mais, ce faisant, nous avons introduit des ambiguïtés. En effet, si la ligne i est remplacée par la ligne 4.16, et si nous avons une ligne j qui correspond à :

$$\text{si } a = f_j(M) \wedge g_i(M) \ldots$$

avec $f_j(M) \in \mathcal{A}_n$, nous avons introduit une ambiguïté bien que $f_i(M) \neq f_j(M)$ initialement.

Comment résorber ces ambiguïtés ? C'est simple, nous l'avons déjà fait en retirant les gardes non stables : nous allons fusionner toutes les lignes exposant une ambiguïté. Prenons les deux lignes suivantes :

$$\left\{ \begin{array}{ll} \text{si } a = f_i(M) \wedge g_i(M) \text{ alors} & h_i(M) \\ \text{si } a = f_j(M) \wedge g_i(M) \text{ alors} & h_j(M) \end{array} \right. \tag{4.19}$$

Il y a ambiguïté si les gardes sont égales (il n'y a pas d'ambiguïté possible des gardes) et si $f_i(M) = f_j(M)$. Il suffit de remplacer des deux lignes par (en supposant que $f_i(M) \in \mathcal{A}_n$) :

$$\left\{ \text{si } a = f_i(M) \otimes f_j(M) \wedge g_i(M) \text{ alors } \perp_{n-1} \right. \tag{4.20}$$

Nous arrivons ainsi à une nouvelle transformation dans laquelle toutes les variables scalaires sont inchangées (sauf celles qui pourraient éventuellement être dans des zones considérées comme des tableaux), et toutes les variables parcourant un tableau ont été changées pour considérer à la place le tableau dans sa totalité.

Exemple

Reprenons l'exemple de la figure 4.1, et de la transformation de l'équation 4.14, que nous avons obtenue après les modifications de la section 4.3.3.

Classons les variables :

- pointeurs fous

  Il n'y a aucun pointeur fou dans cet exemple, comme dans beaucoup de programmes réels corrects (heureusement) ;

- variables scalaires

  Les adresses qui ne changent pas d'un tour de boucle à l'autre sont : $\langle \texttt{\_p}, 0\rangle$, $\langle \texttt{\_q}, 0\rangle$ et $\langle \texttt{\_i}, 0\rangle$. La détermination est particulièrement simple ici, puisque ces adresses sont constantes.

- tableaux

  Les adresses qui changent entre deux tours de boucle, mais qui restent dans le même objet que l'objet initial sont $M(\langle \texttt{\_q}, 0\rangle)$ et $M(\langle \texttt{\_p}, 0\rangle)$. En effet, la différence $M(\langle \texttt{\_q}, 0\rangle) - (T_c(M))(\langle \texttt{\_q}, 0\rangle)$ est définie et vaut 1 (idem pour p).

Ce classement fait, nous pouvons procéder aux approximations :

$$\begin{cases} \text{si } a = M(\langle \texttt{\_q}, 0\rangle) + \bot_{-1} \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & \bot_{-1} \\ \text{si } a = M(\langle \texttt{\_q}, 0\rangle) + \bot_{-1} \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & \bot_{-1} \\ \text{si } a = M(\langle \texttt{\_p}, 0\rangle) + \bot_{-1} \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & \bot_{-1} \\ \text{si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\ \text{si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & M(\langle \texttt{\_p}, 0\rangle) + 1 \\ \text{si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\ \text{si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & M(\langle \texttt{\_q}, 0\rangle) + 1 \\ \text{si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1 \\ \text{si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \text{ alors} & M(\langle \texttt{\_i}, 0\rangle) + 1 \end{cases} \tag{4.21}$$

Ce qui nous donne la forme concrète :

$$\left| \begin{array}{l} \{\texttt{p == q}\} \,/\, [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{i} \leftarrow \texttt{i+1}, \texttt{*q} \leftarrow \bot_{-1}] \\ \{\texttt{p != q}\} \,/\, [\texttt{p} \leftarrow \texttt{p+1}, \texttt{q} \leftarrow \texttt{q+1}, \texttt{i} \leftarrow \texttt{i+1}, \texttt{*q} \leftarrow \bot_{-1}, \texttt{*p} \leftarrow \bot_{-1}] \end{array} \right. \tag{4.22}$$

Conclusion

Faisons le point sur les actions qui nous ont mené à cette nouvelle approximation :

1. nous avons classé les adresses en trois catégories :
   - pointeurs fous, c'est à dire les adresses qui varient hors de l'objet initial après un tour de boucle,
   - variables scalaires, c'est à dire les adresses qui ne changent pas entre deux tours,
   - tableaux, c'est à dire les adresses qui varient à l'intérieur du même objet ;

2. nous avons ensuite remplacé chaque adresse par une approximation illustrant l'intervalle de variation de ces adresses ;
3. cette opération ayant introduit des ambiguïtés, nous les avons retirées par opération de rencontre sur les lignes en conflit.

Nous avons dorénavant une nouvelle transformation, approximation de la précédente, elle-même approximation de T_c. Par transitivité, cette nouvelle transformation est elle-même une approximation de T_c. Le premier critère de la propriété 4.1 est donc respecté. De plus, chaque adresse étant stable par le corps T_c, le deuxième critère est partiellement respecté (il l'est pour les f_i, mais pas encore pour la transformation complète).

4.3.5 Approximation des contenus de la mémoire

La transformation obtenue dans la section précédente n'est que partiellement stable. Seules les adresses et les gardes sont stables par le corps de la boucle. En revanche, les valeurs lues dans la mémoire (c'est à dire les fonctions $h_i(M)$) ne sont pas encore stables.

De plus, nous n'avons pas pris en compte le nombre de tours de boucle jusqu'ici. Ce qui veut dire que la propriété 4.1 ne peut être vérifiée que dans le cas dégénéré de l'équation 4.4.

Approximation

Nous cherchons donc à approximer les contenus de la mémoire, c'est à dire les fonctions $h_i(M)$. Plus précisément, nous cherchons à approximer les fonctions $h_i(M)$ qui ont échappé aux modifications précédentes.

Elles ne sont pas nombreuses : reconsidérons la catégorisation des adresses que nous avons faite dans la section précédente. Seules les variables scalaires (ou adresses utilisées comme telles) sont restées intactes (équation 4.17), toutes les autres ont été approximées par $\perp_{n-1}$ (équations 4.16 et 4.18). Les approximations de cette section ne pourront donc concerner que les variables scalaires.

Notre variable scalaire va prendre une succession de valeurs $(u_n)_{0 \leq n \leq \theta}$. Quelle sera la valeur de cette variable en fin de boucle, c'est à dire quelle sera la valeur de u_θ ?

Face à une suite, nous pouvons envisager de reconnaître un certain nombre de cas. Par exemple, le cas d'une suite arithmétique, le cas d'une suite géométrique, etc. pour lesquels le calcul du θ-ème terme est facile. On peut alors remplacer la fonction $h_i(M)$ par une expression fonction de θ et éventuellement de la valeur initiale à l'entrée de la boucle, $M(f_i(M))$. Tous les autres cas, ou les cas non reconnus peuvent être remplacés, avec perte de précision mais pas à tort, par $\perp_{n-1}$ (si $f_i(M) \in \mathcal{A}_n$).

Le nombre de cas que nous allons reconnaître est plus réduit. Il est basé sur un test simple, la différence $h_i(M) - M(f_i(M))$:

- $h_i(M) - M(f_i(M))$ est définie et constante (non $\perp$)

  Dans ce cas, il s'agit d'une variable inductive (suite arithmétique). Si nous nommons δ cette différence, nous pouvons exprimer la valeur de la case référencée par $f_i(M)$ en fin de boucle par l'expression :

  $$h'_i(M) = M(f_i(M)) + \delta\theta \tag{4.23}$$

  où θ est le nombre de tours de boucle parcourus et $M(f_i(M))$ est la valeur de cette case mémoire en début de boucle.

  Cette forme, la seule à dépendre du nombre de tours de boucle parcourus, satisfait la propriété 4.1. En effet :

  – pour $\theta = 1$, $h'_i(M) = h_i(M)$, donc h'_i est une approximation de h_i (c'est même la meilleure approximation que l'on puisse trouver),

  – δ est constant, donc stable par T_c. Nous introduisons la quantité θ, donc h'_i est fonction de M et θ. $h'_i(T_c(M), \theta) = (T_c(M))(f_i(T_c(M))) + \delta\theta$. Or f_i est stable par T_c, donc $h'_i(T_c(M), \theta) = (T_c(M))(f_i(M)) + \delta\theta$. $(T_c(M))(f_i(M))$ est justement la définition de $h_i(M)$, donc $h'_i(T_c(M), \theta) = h_i(M) + \delta\theta$. Par définition de δ ($\delta = h_i(M) - M(f_i(M))$), nous avons donc $h'_i(T_c(M), \theta) = M(f_i(M)) + \delta + \delta\theta$, donc $h'_i(T_c(M), \theta) = h'_i(M, \theta+1)$;

  Les deux conditions de la propriété 4.1 sont donc vérifiées ;

- $h_i(M) - M(f_i(M))$ est définie et non constante

  Dans ce cas, il s'agit bien d'une somme, mais que nous sommes incapables (étant donnés les moyens dont nous nous munissons) d'identifier. La seule chose dont nous sommes sûrs, c'est que si $h_i(M)$ pointe dans un objet, $h_i(T_c(M))$ pointera dans le même objet. Ceci nous autorise donc à écrire la valeur (approchée) en fin de boucle :

  $$h'_i(M) = M(f_i(M)) + \perp_{-1} \tag{4.24}$$

  Dans le cas d'une variable simple (c'est à dire dont l'adresse est dans $\mathcal{A}_0$), le résultat sera donc $\perp_{-1}$. Sinon, il s'agit d'un pointeur, le résultat sera *quelque part* dans l'objet pointé initialement (résultat de la forme $\langle i, \perp_{-1} \rangle$).

  Cette forme est stable par T_c ;

- $h_i(T_c(M)) - h_i(M)$ n'est pas définie

  Ce cas, le plus défavorable, correspond à un pointeur fou. Nous ne pouvons rien conclure, si ce n'est que la valeur finale est élément de $\mathcal{A}_{n-1}$ si $f_i(M)$ appartient à $\mathcal{A}_n$. La valeur finale est donc :

  $$h'_i(M) = \perp_{n-1} \tag{4.25}$$

  Cette forme est également stable par T_c ;

Exemple

Toujours sur l'exemple de la figure 4.1, utilisons comme base la transformation de l'équation 4.21

Nous avons détecté trois variables scalaires : $\langle \_\mathtt{p}, 0\rangle$, $\langle \_\mathtt{q}, 0\rangle$ et $\langle \_\mathtt{i}, 0\rangle$. Ces trois variables sont inductives (le pas de variation est 1). Nous pouvons donc réécrire leur valeur finale en fonction de θ :

$$\begin{cases} \text{si } a = M(\langle \_\mathtt{q}, 0\rangle) + \bot_{-1} \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & \bot_{-1} \\ \text{si } a = M(\langle \_\mathtt{q}, 0\rangle) + \bot_{-1} \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & \bot_{-1} \\ \text{si } a = M(\langle \_\mathtt{p}, 0\rangle) + \bot_{-1} \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & \bot_{-1} \\ \text{si } a = \langle \_\mathtt{p}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{p}, 0\rangle) + \theta \\ \text{si } a = \langle \_\mathtt{p}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{p}, 0\rangle) + \theta \\ \text{si } a = \langle \_\mathtt{q}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{q}, 0\rangle) + \theta \\ \text{si } a = \langle \_\mathtt{q}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{q}, 0\rangle) + \theta \\ \text{si } a = \langle \_\mathtt{i}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) = M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{i}, 0\rangle) + \theta \\ \text{si } a = \langle \_\mathtt{i}, 0\rangle \wedge M(\langle \_\mathtt{q}, 0\rangle) \neq M(\langle \_\mathtt{p}, 0\rangle) \text{ alors} & M(\langle \_\mathtt{i}, 0\rangle) + \theta \end{cases} \tag{4.26}$$

Ce qui nous donne la forme concrète suivante :

$$\left| \begin{array}{l} \{\mathtt{p\ ==\ q}\} \ / \ [\mathtt{p} \leftarrow \mathtt{p+}\theta, \mathtt{q} \leftarrow \mathtt{q+}\theta, \mathtt{i} \leftarrow \mathtt{i+}\theta, \mathtt{*q} \leftarrow \bot_{-1}] \\ \{\mathtt{p\ !=\ q}\} \ / \ [\mathtt{p} \leftarrow \mathtt{p+}\theta, \mathtt{q} \leftarrow \mathtt{q+}\theta, \mathtt{i} \leftarrow \mathtt{i+}\theta, \mathtt{*q} \leftarrow \bot_{-1}, \mathtt{*p} \leftarrow \bot_{-1}] \end{array} \right. \tag{4.27}$$

Conclusion

À la suite des précédentes modifications, il ne restait plus qu'un seul élément non stable par la transformation T_c associée au corps de la boucle : les contenus des cases mémoire dans le cas des variables scalaires ou utilisées comme telles.

Nous avons donc approximé leur valeur en fin de boucle : lorsque nous avons reconnu des variables inductives, nous avons remplacé leur valeur par la valeur exacte, fonction de θ, en vérifiant la stabilité par T_c. Cette vérification est différente des précédentes, puisque nous devons ici tenir compte de θ et ne plus simplement tester l'identité de deux expressions.

Lorsqu'une variable scalaire ne pouvait être ramenée à une variable inductive, nous avons remplacé sa valeur par $M(f_i(M)) + \bot_{-1}$ ou $\bot_{n-1}$ suivant les cas.

4.3.6 Conclusion

Qu'avons nous démontré dans ce sous-chapitre ?

Nous avons défini ce qu'est une approximation de la transformation associé à θ tours de boucle. Cette définition étant insuffisante pour calculer une approximation, nous avons trouvé un critère simple. En quelques mots, il s'agit de trouver une approximation de la transformation associée au corps, et qui soit stable par ce même corps.

Ceci nous a permis de procéder par approximations successives de T_c :

- nous avons d'abord retiré les gardes qui n'étaient pas stables ;
- puis nous avons arrangé les adresses pour qu'elles soient toutes stables ;
- et enfin, nous avons approximé les valeurs des variables scalaires afin d'en donner une valeur stable (éventuellement fonction du nombre de tours de boucle).

Le résultat final est une transformation où tous les composants sont stables, c'est à dire une transformation stable qui est en même temps une approximation de T_c. Le résultat est donc une approximation de T_c^θ.

4.4 Transformation associée à la boucle

Nous sommes arrivés, dans le sous-chapitre précédent, à donner une approximation de T_c^θ. Mais nous n'avons pas calculé l'approximation de la transformation T_b (voir équation 4.1) associée à la boucle complète.

D'autre part, rappelons-nous que notre objectif est de reconstituer des boucles du même type que les boucles *for* de Pascal ou *do* de Fortran. Peut-on le faire, et à quelles conditions ?

Nous allons, après avoir retiré un cas particulier, chercher à calculer le nombre de tours de boucle parcourus. Si cela est possible, nous pourrons obtenir une boucle *for* correspondant à notre boucle, ainsi que la transformation qui lui est associée. Si cela n'est pas possible, nous serons obligé de procéder à une ultime approximation pour obtenir la transformation associée à la boucle.

Notons que les étapes que nous allons proposer doivent être éventuellement séparées pour chaque groupe de gardes. Une configuration mémoire (sélectionnée par quelques gardes particulières) peut en effet donner une précision suffisante pour permettre le calcul du nombre de boucles parcourus. Nous distinguerons alors les résultats finaux suivant les gardes.

4.4.1 Boucle jamais parcourue

Reprenons la formulation de T_b, et éclatons-la en deux lignes pour distinguer le cas de la boucle jamais parcourue :

$$T_b(M) = \begin{cases} \text{si } L(e,M) = 0 & \text{alors } Id_{\mathcal{M}} \\ \text{sinon si } \left(\bigwedge_{0\leq i<\theta}(L(e,T_c^i(M)) \neq 0)\right) \wedge L(e,T_c^\theta(M)) = 0 & \text{alors } T'(M,\theta) \end{cases} \tag{4.28}$$

Cette approche n'apporte rien de nouveau, si ce n'est le retrait d'un cas particulier fastidieux. Rappelons que nous avons supposé dans tout le début de ce chapitre que $\theta \geq 1$. Cette formulation garantit cette supposition.

4.4.2 Nombre de tours de boucle parcourus

L'approximation T' de T_c^θ que nous avons obtenue est fonction du nombre θ de tours de boucle parcourus si elle comporte des variables inductives. Pour obtenir la transformation associée à la boucle elle-même, il faut s'affranchir de θ, soit en en donnant la valeur en fonction de M, la mémoire initiale, soit en faisant une dernière approximation pour supprimer toute référence au nombre de tours de boucle.

D'autre part, une boucle de type *do* de Fortran nécessite la connaissance de la borne supérieure du compte-tours de boucle. Cette borne est $\theta - 1$ si nous prenons un compte-tours variant par pas de 1 à partir de la valeur initiale 0. Il faut donc calculer la valeur de θ en fonction de M.

Résolution du système

Reprenant l'équation 4.1, la valeur de θ est la solution de l'équation :

$$\left(\bigwedge_{0\leq i<\theta}(L(e,T_c^i(M)) \neq 0)\right) \wedge L(e,T_c^\theta(M)) = 0 \tag{4.29}$$

Il n'y a évidemment pas de méthode générale pour obtenir la valeur de θ.

La première difficulté est que nous ne connaissons pas la forme de T_c^i, mais nous en connaissons une approximation $T'(M,i)$. Or, cette approximation ne nous permet pas de calculer θ, car nous perdons de la précision, ce qui implique que le système ci-dessus est faussé.

Le seul cas où le système n'est pas faussé est lorsque :

$$\forall \langle i, M \rangle \in \mathcal{E} \times \mathcal{M}, L(e, T_c^i(M)) = L(e, T'(M, i)) \tag{4.30}$$

Cette formulation signifie que l'expression e qui gouverne la boucle n'est fonction que des variables inductives (ou constantes) que nous avons identifiées en 4.3.5[2]. Si c'est le cas, nous pouvons donc réécrire le système :

$$\left(\bigwedge_{0 \leq i < \theta} (L(e, T'(M, i)) \neq 0) \right) \wedge L(e, T'(M, \theta + 1)) = 0 \tag{4.31}$$

Malgré cette constatation importante, on ne peut toujours pas résoudre ce système dans le cas général. Peut-on trouver des cas particuliers où nous saurons résoudre ?

Isolons l'inconnue du système : supposons que nous ayons des variables d'inductions d'adresses a_j (avec $1 \leq j \leq m$). Chacune de ces variables a un pas (incrément entre chaque tour de boucle) constant égal à δ_j, et sa valeur au i-ème tour de boucle est donc de la forme $M(a_j) + i\delta_j$. Dans ces conditions, $L(e, T'(M, i))$ n'est fonction que de i (posons $L(e, T'(M, i)) = \alpha(i)$). Le système devient alors :

$$\left(\bigwedge_{0 \leq i < \theta} (\alpha(i) \neq 0) \right) \wedge \alpha(\theta + 1) = 0 \tag{4.32}$$

Il n'y a pas de méthode générale pour résoudre le système. Si nous pouvons le manipuler pour le mettre sous la forme :

$$\theta = \beta \tag{4.33}$$

où β est une "constante", c'est à dire une expression ne mettant en œuvre que les variables dont la valeur ne change pas entre deux tours de boucle, les $M(a_j)$ et les δ_j, alors nous pouvons calculer la valeur de θ.

Quand peut-on obtenir facilement la valeur de θ ? Nous avons pris le parti de ne reconnaître que peu de cas, c'est à dire les fonctions affines d'une (et une seule) variable d'induction.

Exemple

Toujours avec l'exemple de la figure 4.1, illustrons le calcul de θ : l'expression e est ici `i < n`. Le pas de la variable d'adresse $\langle \texttt{\_i}, 0 \rangle$ est 1, la variable d'adresse $\langle \texttt{\_n}, 0 \rangle$ ne change pas de valeur

[2]Cette condition revient à dire que $L(e, T'(M, i))$ ne contient pas $\perp$, puisque toute variable non inductive ou non constante a été approximée.

entre deux tours de boucle, c'est donc une variable constante. Notons que les pointeurs p et q ne peuvent altérer n : puisqu'ils sont incrémentés, ils ne peuvent donc pas pointer dans une variable simple ($d_m(n) = 1$). D'autre part, aucune garde ne nous apporte de précision particulière. Le calcul que nous faisons pour la garde $M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle)$ est inchangé pour la garde inverse.

Il faut donc trouver une valeur θ telle que :

$$\left(\bigwedge_{0 \leq x < \theta} (L(\mathtt{i} < \mathtt{n}, T'(M, x)) \neq 0) \right) \wedge L(\mathtt{i} < \mathtt{n}, T'(M, \theta + 1)) = 0 \tag{4.34}$$

C'est à dire :

$$\left(\bigwedge_{0 \leq x < \theta} ((M(\langle \texttt{\_i}, 0\rangle) + x < M(\langle \texttt{\_n}, 0\rangle)) \neq 0) \right) \wedge (M(\langle \texttt{\_i}, 0\rangle) + \theta < M(\langle \texttt{\_n}, 0\rangle) \neq 0) = 0 \tag{4.35}$$

Si nous simplifions les opérateurs logiques, et si nous isolons x et θ, nous obtenons le nouveau système :

$$\left(\bigwedge_{0 \leq x < \theta} (x < M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle)) \right) \wedge \theta \geq M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \tag{4.36}$$

Ce qui nous donne finalement l'équation :

$$\theta = M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \tag{4.37}$$

Nous savons donc comment calculer le nombre de tours de boucle.

Conclusion

Le calcul du nombre θ de tours de boucles parcouru n'est possible que dans un nombre réduit de cas :

- l'expression qui gouverne la boucle n'est fonction que des variables inductives et de variables constantes, et
- si on peut isoler le compte-tours dans cette expression.

Nous ne pouvons envisager le calcul automatique de θ dans le cas général, aussi serons-nous amenés, dans la réalisation, à ne considérer que quelques cas particuliers d'expressions.

Dans tous les autres cas, nous devons admettre notre impuissance et renoncer à calculer θ.

4.4.3 Transformation associée à la boucle

Sachant maintenant si nous pouvons calculer le nombre de tours de boucle, nous sommes à même de donner enfin une approximation de la transformation associée à la boucle.

Nombre de tours de boucle connu

Lorsque le nombre de tours de boucle est connu, cela signifie que la transformation approchée de T_c^θ est fonction de θ, puisqu'elle contient des variables inductives. Rappelons que θ est fonction :

- de l'adresse des variables inductives et constantes,
- de la valeur de ces cases mémoire,
- des pas d'induction des variables inductives, eux-mêmes fonctions éventuellement de variables ;

Le calcul de θ n'était possible que grâce à la connaissance de l'expression e. Nous pouvons donc remplacer dans T' toute référence à θ par sa valeur.

Nous obtenons ainsi une nouvelle transformation ne dépendant plus que de l'état initial de la mémoire.

Comme effet secondaire, nous pouvons modifier la boucle pour en faire une vraie boucle *do*. Dans le texte du programme, il suffit de :

- remplacer la ligne `while` (e) par le texte `do` (α ; `0` ; θ`-1`) en utilisant une syntaxe hybride entre Fortran et C (nous avons utilisé le `do` de Fortran pour ne pas risquer la confusion avec le `for` de C) ;
- si une variable inductive x n'est modifiée qu'une seule fois :
  - remplacer toute référence à cette variable avant sa modification par $x + \alpha\delta_x$,
  - remplacer toute référence à cette variable après sa modification par $x + (\alpha + 1)\delta_x$,
  - supprimer la modification de cette variable,
  - ajouter la ligne $x = x + \theta\delta_x$ après la boucle afin de préserver cette valeur pour la suite du programme ;

- si une variable inductive est modifiée plus d'une fois :
  - découper le programme en segments correspondant aux valeurs successives de la variable,
  - calculer la valeur de la variable en fonction de α et δ_x en se servant des transformations calculées au fur et à mesure pour chaque instruction de la boucle,
  - remplacer toute référence à la variable par sa valeur,
  - retirer les affectations à cette variable.

Nombre de tours de boucle inconnu

Si nous n'avons pas réussi à calculer le nombre de tours de boucle, il faut faire disparaître toute référence à θ dans T', la transformation approchée de T_c^θ.

Les seuls éléments restant dépendant de θ sont les valeurs des variables inductives ($h'_i(M) = M(f_i(M)) + \delta_i\theta$). Il suffit de remplacer leur valeur par leur valeur à un tour de boucle inconnu, c'est à dire : $h'_i(M) = M(f_i(M)) + \perp_{-1}$.

Après cette ultime approximation, nous obtenons une nouvelle transformation ne faisant absolument plus référence à θ.

Exemple

Dans l'exemple de la figure 4.1, nous savons calculer le nombre de tours de boucle, nous pouvons donc transformer notre boucle en une boucle *do* et exprimer toutes les variables inductives en fonction du compte-tours de boucle :

```
do (tmp ; 0 ; n - i - 1)
{
      if ((i + tmp) % 2 == 0)
            *(p + tmp) = *(p + tmp) + 1 ;
      *(q + tmp) = *(q + tmp) + 1 ;
}
i = i + (n - i) ;
p = p + (n - i) ;
q = q + (n - i) ;
```

La transformation associée est (en tenant compte du cas ou la boucle n'est jamais exécutée) :

$$
\begin{cases}
\text{si } a = M(\langle \texttt{\_q}, 0\rangle) + \bot_{-1} \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } \bot_{-1} \\
\text{sinon si } a = M(\langle \texttt{\_q}, 0\rangle) + \bot_{-1} \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } \bot_{-1} \\
\text{sinon si } a = M(\langle \texttt{\_p}, 0\rangle) + \bot_{-1} \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } \bot_{-1} \\
\text{sinon si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } M(\langle \texttt{\_p}, 0\rangle) + M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \\
\text{sinon si } a = \langle \texttt{\_p}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } M(\langle \texttt{\_p}, 0\rangle) + M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \\
\text{sinon si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } M(\langle \texttt{\_q}, 0\rangle) + M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \\
\text{sinon si } a = \langle \texttt{\_q}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } M(\langle \texttt{\_q}, 0\rangle) + M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \\
\text{sinon si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) = M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } M(\langle \texttt{\_i}, 0\rangle) + M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \\
\text{sinon si } a = \langle \texttt{\_i}, 0\rangle \wedge M(\langle \texttt{\_q}, 0\rangle) \neq M(\langle \texttt{\_p}, 0\rangle) \wedge (M(\langle \texttt{\_i}, 0\rangle) < M(\langle \texttt{\_n}, 0\rangle)) \neq 0 \\
\qquad\qquad \text{alors } M(\langle \texttt{\_i}, 0\rangle) + M(\langle \texttt{\_n}, 0\rangle) - M(\langle \texttt{\_i}, 0\rangle) \\
\text{sinon } M(a)
\end{cases}
\tag{4.38}
$$

Ce qui nous donne la forme concrète suivante :

$$
\left|
\begin{array}{l}
\{\texttt{p == q}, \texttt{i < n}\} \;/\; [\texttt{p} \leftarrow \texttt{p+n-i}, \texttt{q} \leftarrow \texttt{q+n-i}, \texttt{i} \leftarrow \texttt{n}, \texttt{*q} \leftarrow \bot_{-1}] \\
\{\texttt{p != q}, \texttt{i < n}\} \;/\; [\texttt{p} \leftarrow \texttt{p+n-i}, \texttt{q} \leftarrow \texttt{q+n-i}, \texttt{i} \leftarrow \texttt{n}, \texttt{*q} \leftarrow \bot_{-1}, \texttt{*p} \leftarrow \bot_{-1}]
\end{array}
\right.
\tag{4.39}
$$

4.5 Réalisation et exemples

Nous décrivons dans cette partie l'état actuel de la réalisation, puis nous montrons ce qui a été implémenté sur des exemples en terminant leur étude manuellement pour montrer ce qui reste à faire. Enfin, nous donnons un exemple de nid de boucles traité manuellement pour montrer le fonctionnement de notre méthode sur un exemple complexe.

Les premières étapes de la méthode décrite dans ce chapitre sont implémentées. Les étapes suivantes, et surtout l'intégration de la composition des transformations face à des approximations, nécessitent une adaptation de la réalisation du chapitre précédent que nous n'avons pas pu accomplir faute de temps.

4.5.1 Représentation d'une approximation

Les transformations sont, comme dans le chapitre précédent, traitées sous forme concrète. Mais l'introduction de la valeur $\perp$ pose un problème de représentation et de traitement. Face à ce problème, deux solutions sont possibles :

1. choisir un nouvel élément syntaxique pour représenter les $\perp_i$ (avec $i \geq -1$) dans les expressions ;
2. choisir une représentation spécialisée compte-tenu des propriétés de ces nouvelles valeurs.

La première solution est sans conteste la plus générale, mais elle est également la plus délicate à implémenter du fait de la propagation de la complexité due à l'introduction de $\perp$ dans les expressions : la détection d'une approximation, mais aussi les calculs et les simplifications doivent être modifiés en conséquence.

La deuxième solution nécessite la compréhension des situations dans lesquelles nous pouvons être amenés à rencontrer la valeur $\perp$. Nous avons fait ce bilan, et nous avons conclu aux possibilités suivantes :

Signification	Conditions	Représentation
$\perp_n \leftarrow \perp_{n-1}$	$n \geq 0$	`(bottom` *type*`)`
$a + \perp_{-1} \leftarrow \perp_{n-1}$	$a \in \mathcal{A}_n, n \geq 0$	`(tabmod` *type a*`)`
$a \leftarrow a + \perp_{-1}$	$a \in \mathcal{A}_n, n \geq 0$	`(plussomething` *type a*`)`
$a \leftarrow \perp_{n-1}$	$a \in \mathcal{A}_n, n \geq 0$	`(unknown` *type a*`)`
$a \leftarrow v$	$a \in \mathcal{A}_n, n \geq 0, v \in \mathcal{A} \times \mathcal{V}$	`(store` *type a v*`)`

La dernière possibilité correspond, bien sûr, aux substitutions du chapitre précédent.

Mais si les substitutions sont modifiées, les termes des gardes sont également perturbées par l'adjonction des nouveaux éléments, et il faut introduire deux nouveaux opérateurs :

Signification	Conditions	Représentation
$\forall \langle v_1, v_2 \rangle \in \mathcal{V}^2, a_1 + v_1 \neq a_2 + v_2$	$\langle a_1, a_2 \rangle \in \mathcal{A}_n^2, n \geq 0$	`(difftab` *type* a_1 a_2`)`
$\exists \langle v_1, v_2 \rangle \in \mathcal{V}^2, a_1 + v_1 = a_2 + v_2$	$\langle a_1, a_2 \rangle \in \mathcal{A}_n^2, n \geq 0$	`(sametab` *type* a_1 a_2`)`

La première ligne correspond à la possibilité, pour deux adresses a_1 et a_2, de pointer dans le même tableau, alors que la deuxième ligne correspond à l'impossibilité de pointer dans le même tableau.

4.5.2 Implémentation

Les points suivants ont été réalisés :

- prise en compte du test (voir 4.3.2) ;
- retrait des gardes non stables (voir 4.3.3) ;
- approximation des adresses non stables (voir 4.3.4).

Ceci représente environ 1 500 lignes de Common Lisp fortement commentées.

Le module le plus important est le module de retrait des gardes non stables, qui suppose une fusion des transformations suivant des gardes compatibles (c'est a dire qui sont ou peuvent être vraies) : la méthode utilisée consiste à construire un graphe dans lequel les sommets sont les gardes et les arcs expriment la relation de compatibilité, puis à extraire toutes les composantes fortement connexes et les réunir en une seule garde et substitution.

Il reste donc à réaliser l'approximation des variables inductives, ce qui passe par la détection d'un compte-tours de boucle, qui avait été réalisée dans une précédente implémentation non décrite dans cette étude. Un travail important reste la fiabilisation[3] de l'implémentation des boucles.

Mais le point le plus délicat est certainement la prise en compte de ces approximations dans la composition des transformations. Cela nécessite une adaptation car les mécanismes utilisés sont fortement spécialisés et font des suppositions qui ne sont plus vérifiées en face de $\perp$.

La réalisation n'est pas encore suffisamment avancée et fiabilisée pour avancer un calcul de complexité. Cependant, l'exposé de la méthode nous a permis de constater qu'une seule passe était faite sur la transformation associée à la boucle. De plus, nous ne générons pas plus de précision, nous en retirons au contraire. Ceci nous porte à penser, intuitivement, que la complexité est linéaire en fonction du nombre de lignes et de termes dans la transformation associée au corps de la boucle.

4.5.3 Exemples

Nous citons ici quelques exemples issus de notre implémentation dans son état actuel. Pour chaque exemple, nous donnons la fonction en C, la transformation associée au corps de la boucle

[3]Nous ne sommes pas encore en mesure de faire des tests, en confiance, sur des fonctions de 200 lignes comme au chapitre précédent.

(τ_c), puis la transformation partiellement approximée (τ_p). Enfin, nous complétons par la transformation complètement approximée (τ_a) par nos soins (c'est à dire manuellement) pour illustrer la méthode.

Les variables utilisées sont les mêmes que dans les exemples du chapitre précédent (voir page 111).

Exemple

```
void tr01 (void)
{
	while (i1 < 10)
	{
		t2 [i1] = t1 [i1] ;
		i1 = i1 + 1 ;
	}
}
```

$$\tau_c = \left| \; \{\text{vrai}\}/[\texttt{*(t2 + i1)} \leftarrow \texttt{*(t1 + i1) + 1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right.$$

$$\tau_p = \left| \; \{\text{vrai}\}/[\texttt{*(t2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right.$$

Commentaire : une simple approximation des adresses a eû lieu, ce qui conduit à conclure que le tableau `t2` a été modifié, donc on ne peut plus rien dire sur ses éléments. D'autre part, la reconnaissance des variables d'induction n'étant pas terminée, la variable `i1` est laissé inchangée dans l'état actuel de la réalisation.

$$\tau_a = \left| \begin{array}{l} \{\texttt{i1 < 10}\}\ /[\texttt{*(t2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{i1} \leftarrow \texttt{10}] \\ \{\texttt{i1 >= 10}\}/[] \end{array} \right.$$

Exemple

```
void tr02 (void)
{
	while (i1 < 10)
	{
		p1 [i1] = p1 [i1] + 1 ;
		i1 = i1 + 1 ;
	}
}
```

$$\tau_c = \left| \ \{\text{vrai}\}/[\texttt{*(p1 + i1)} \leftarrow \texttt{*(p1 + i1) + 1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right.$$

$$\tau_p = \left| \ \{\text{vrai}\}/[\texttt{*(p1 + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right.$$

Commentaire : ce cas diffère du précédent par la détermination du tableau modifié. On ne peut conclure avec précision sur le tableau modifié, mais on sait que c'est le tableau pointé par `p1`.

$$\tau_a = \left| \begin{array}{l} \{\texttt{i1 < 10}\}\ /[\texttt{*(p1 + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{i1} \leftarrow \texttt{10}] \\ \{\texttt{i1 >= 10}\}/[] \end{array} \right.$$

Exemple

```
void tr03 (void)
{
      while (p1 < p2)
      {
            *p1 = *p1 + 1 ;
            p1 = p1 + 1 ;
      }
}
```

$$\tau_c = \left| \ \{\text{vrai}\}/[\texttt{*p1} \leftarrow \texttt{*p1 + 1}, \texttt{p1} \leftarrow \texttt{p1 + 1},] \right.$$

$$\tau_p = \left| \ \{\text{vrai}\}/[\texttt{*(p1 + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{p1} \leftarrow \texttt{p1 + 1}] \right.$$

Commentaire : cette fonction est une version plus "agressivement C" des deux fonctions précédentes : le parcours du tableau est assuré par un pointeur.

$$\tau_a = \left| \begin{array}{l} \{\texttt{p1 < p2}\}\ /[\texttt{*(p1 + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{p1} \leftarrow \texttt{p1 + (p2 - p1)}] \\ \{\texttt{p1 >= p2}\}/[] \end{array} \right.$$

Exemple

```
void tr04 (void)
{
```

```
        while (i1 < a)
        {
              *p1 = 5 ;
              p1 = p1 + 1 ;
              i1 = i1 + 1 ;
        }
}
```

$$\tau_c = \left| \; \{\texttt{p1 != \&i1}\}/[\texttt{*p1} \leftarrow 5, \texttt{p1} \leftarrow \texttt{p1 + 1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right.$$

$$\tau_p = \left| \; \{\text{vrai}\}/[\texttt{*(p1} + \perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{p1} \leftarrow \texttt{p1 + 1}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right.$$

Commentaire : le parcours du tableau est assuré par un pointeur, mais la case mémoire ne peut être la variable `a` qui sert de limite de boucle.

$$\tau_a = \left| \begin{array}{l} \{\texttt{i1 < a}\}\,/[\texttt{*(p1} + \perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{p1} \leftarrow \texttt{p1 + (a - i1)}, \texttt{i1} \leftarrow \texttt{a}] \\ \{\texttt{i1 >= a}\}/[] \end{array} \right.$$

Exemple

```
void tr05 (void)
{
        while (*p1 < a)
        {
              *p2 = 5 ;
              p2 = p2 + 1 ;
              *p1 = *p1 + 1 ;
        }
}
```

$$\tau_c = \left| \begin{array}{l} \{\texttt{p1 == p2}\}/[\texttt{p2} \leftarrow \texttt{p2 + 1}, \texttt{*p1} \leftarrow 5] \\ \{\texttt{p1 != p2}\}/[\texttt{p2} \leftarrow \texttt{p2 + 1}, \texttt{*p2} \leftarrow 5, \texttt{*p1} \leftarrow \texttt{*p1 + 1}] \end{array} \right.$$

$$\tau_p = \left| \begin{array}{l} \{\texttt{p1 sametab p2}\}/[\texttt{*(p2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{*p1} \leftarrow \perp_{-1}, \texttt{p2} \leftarrow \texttt{p2 + 1}] \\ \{\texttt{p1 difftab p2}\}/[\texttt{*(p2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{*p1} \leftarrow \perp_{-1}, \texttt{p2} \leftarrow \texttt{p2 + 1}] \end{array} \right.$$

Commentaire : la variable `*p1` est une variable d'induction, mais nous ne pouvons malheureusement pas nous en rendre compte. En effet, le retrait des gardes non stables a provoqué le remplacement de la transformation τ_c par la transformation $\{\texttt{vrai}\}/[\texttt{p2} \leftarrow \texttt{p2 + 1}, \texttt{*p1} \leftarrow \perp_{-1}, \texttt{*p2} \leftarrow \perp_{-1}]$, ce qui a anéanti l'induction de `*p1`.

$$\tau_a = \left| \begin{array}{l} \{\texttt{p1 sametab p2}\}/[\texttt{*(p2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{*p1} \leftarrow \perp_{-1}, \texttt{p2} \leftarrow \texttt{p2 + }\perp_{-1}] \\ \{\texttt{p1 difftab p2}\}/[\texttt{*(p2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{*p1} \leftarrow \perp_{-1}, \texttt{p2} \leftarrow \texttt{p2 + }\perp_{-1}] \end{array} \right.$$

Exemple

```
void tr06 (void)
{
      while (p1 < p2)
      {
            *p1 = 5 ;
            *p2 = 3 ;
            p1 = p1 + 1 ;
            p2 = p2 - 1 ;
      }
}
```

$$\tau_c = \left| \begin{array}{l} \{\texttt{p1 == p2}\}/[\texttt{*p2} \leftarrow \texttt{3}, \texttt{p1} \leftarrow \texttt{p1 + 1}, \texttt{p2} \leftarrow \texttt{p2 - 1}] \\ \{\texttt{p1 != p2}\}/[\texttt{*p2} \leftarrow \texttt{3}, \texttt{p1} \leftarrow \texttt{p1 + 1}, \texttt{p2} \leftarrow \texttt{p2 - 1}, \texttt{*p1} \leftarrow \texttt{5}] \end{array} \right.$$

$$\tau_p = \left| \; \{\text{vrai}\}/[\texttt{*(p1 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{*(p2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{p1} \leftarrow \texttt{p1 + 1}, \texttt{p2} \leftarrow \texttt{p2 - 1}] \right.$$

Commentaire : la prise en compte du test fait disparaître le cas `p1 == p2`, ce qui simplifie considérablement la détection des variables d'induction. En revanche, le test que nous proposons de faire pour évaluer les bornes de la boucle est un peu trop simple pour ce cas, ce qui nous conduit à τ_a pessimiste.

$$\tau_a = \left| \; \{\text{vrai}\}/[\texttt{*(p1 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{*(p2 + }\perp_{-1}\texttt{)} \leftarrow \perp_{-1}, \texttt{p1} \leftarrow \texttt{p1 + }\perp_{-1}, \texttt{p2} \leftarrow \texttt{p2 + }\perp_{-1}] \right.$$

Exemple

Cet exemple n'a pas été traité par notre analyseur, mais nous l'avons fait manuellement pour montrer le comportement de la méthode exposée dans ce chapitre dans le cas d'un nid de boucles.

```
void tr07 (void)
{
      int tt [20][10], t2 [10] ;

      i1 = 0 ;
      while (i1 < 20)
      {
            i2 = 0 ;
            while (i2 < 10)
            {
                  tt [i1][i2] = t2 [i2] + i1 ;
                  i2 = i2 + 1 ;
            }
            i1 = i1 + 1 ;
      }
}
```

La transformation associée au corps de la boucle interne est (voir exemple tr01) :

$$\left| \{\text{vrai}\}/[\texttt{*(tt + i1 * 20 + i2)} \leftarrow \texttt{*(t2 + i2) + i1}, \texttt{i2} \leftarrow \texttt{i2 + 1}] \right. \quad (4.40)$$

L'approximation de la transformation associée à la boucle interne dans sa totalité est :

$$\left| \begin{array}{l} \{\texttt{i2 < 10}\}/[\texttt{*(tt + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{i2} \leftarrow \texttt{10}] \\ \{\texttt{i2 >= 10}\}/[] \end{array} \right. \quad (4.41)$$

Nous avons détecté que la variable i2 est une variable d'induction. Le test qui gouverne la boucle étant simple, nous pouvons transformer cette boucle interne en boucle *do* :

```
do (tmp1 ; i2 ; 9)
{
      tt [i1][tmp1] = t2 [tmp1] + i1 ;
}
i2 = i2 + 10 ;
```

La composition avec la ligne i2 = 0 retire la deuxième ligne de transformation de l'équation 4.41, et la garde de la première ligne devient toujours vraie.

Après composition avec la ligne i1 = i1 + 1, nous obtenons la transformation associée au corps de la boucle externe :

$$\left| \; \{\text{vrai}\}/[\texttt{*(tt + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{i2} \leftarrow \texttt{10}, \texttt{i1} \leftarrow \texttt{i1 + 1}] \right. \tag{4.42}$$

L'approximation de la transformation associée au corps de la boucle externe apporte peu d'informations : le tableau `tt` est déjà modifié, seule la variable `i1` est inductive. Ceci nous amène à la transformation associée à la boucle externe :

$$\left| \begin{array}{l} \{\texttt{i1 < 20}\}\,/[\texttt{*(tt + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{i2} \leftarrow \texttt{10}, \texttt{i1} \leftarrow \texttt{20}] \\ \{\texttt{i1 >= 20}\}/[] \end{array} \right. \tag{4.43}$$

Là encore, la variable `i1` est inductive. La composition avec la ligne `i1 = 0` nous amène, comme précédemment, à retirer la ligne `i1 >= 20` dans l'équation 4.43, et à remplacer la garde `i1 < 20` par une garde toujours vraie. Ceci nous donne la transformation finale :

$$\left| \; \{\text{vrai}\}/[\texttt{*(tt + }\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{i2} \leftarrow \texttt{10}, \texttt{i1} \leftarrow \texttt{20}] \right. \tag{4.44}$$

et la fonction `tr07` devient, après substitution des boucles :

```
void tr07 (void)
{
      int tt [20][10], t2 [10] ;

      i1 = 0 ;
      do (tmp2 ; i1 ; 19)
      {
            i2 = 0 ;
            do (tmp1 ; i2 ; 9)
            {
                  tt [tmp2][tmp1] = t2 [tmp1] + tmp2 ;
            }
            i2 = i2 + 10 ;
      }
      i1 = i1 + 20 ; }
```

La version parallélisée de ce cette fonction est donc, après utilisation des techniques classiques que nous avons évoquées au chapitre 1 :

```
void paratr07 (void)
{
      int tt [20][10], t2 [10] ;
```

```
        dopar (tmp2 ; 0 ; 19)
        {
              dopar (tmp1 ; 0 ; 9)
              {
                    tt [tmp2][tmp1] = t2 [tmp1] + tmp2 ;
              }
        }
  }
```

où `dopar` est la version parallèle de la boucle `do`.

Exemple

Ce dernier exemple est une version modifiée du précédent. Il ne s'agit plus de montrer les étapes de notre méthode sur un exemple complexe, mais de prendre un exemple plus "agressivement C" : nous reprenons la même fonction que précédemment, mais réécrite avec des parcours de tableau via des pointeurs.

```
  void tr08 (void)
  {
        int tt [20][10], t2 [10] ;
        int *p1, *p2, *p3, i1 ;

        p1 = tt [0] ;
        i1 = 0 ;
        while (p1 < tt [20])
        {
              p2 = p1 ;
              p3 = t2 ;
              while (p2 < p1 + 10)
              {
                    *p2 = *p3 + i1 ;
                    p2 = p2 + 1 ;
                    p3 = p3 + 1 ;
              }
              p1 = p1 + 10 ;
              i1 = i1 + 1 ;
        }
  }
```

La transformation associée au corps de la boucle interne est :

$$\Big|\ \{\text{vrai}\}/[\texttt{*p2} \leftarrow \texttt{*p3 + i1}, \texttt{p2} \leftarrow \texttt{p2 + 1}, \texttt{p3} \leftarrow \texttt{p3 + 1}]$$

L'approximation de la transformation associée à la boucle interne dans sa totalité est :

$$\left|\begin{array}{l}\{\texttt{p2 < p1+10}\}\ /[\texttt{*(p2+}\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{p2} \leftarrow \texttt{p2+(p1+10-p2)}, \texttt{p3} \leftarrow \texttt{p3+(p1+10-p2)}] \\ \{\texttt{p2 >= p1+10}\}/[]\end{array}\right.$$

Les variables inductives sont cette fois-ci `p2` et `p3`. La variable `p1` étant constante, le test est encore une fois simple et nous pouvons transformer la boucle interne en une boucle *do* :

```
do (tmp1 ; 0 ; p1 + 10 - p2 - 1)
{
      p2 [tmp1] = p3 [tmp1] + i1 ;
}
p3 = p3 + (p1 + 10 - p2) ;
p2 = p2 + (p1 + 10 - p2) ;
```

Continuons la progression. La transformation associée au corps de la boucle externe est, après les compositions intermédiaires :

$$\left|\ \{\text{vrai}\}/[\texttt{*(p1+}\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{p2} \leftarrow \texttt{p1+10}, \texttt{p3} \leftarrow \texttt{t2+(p1+10-p1)}, \texttt{p1} \leftarrow \texttt{p1+10}, \texttt{i1} \leftarrow \texttt{i1+1}]\right.$$

Les variables inductives étant `p1` et `i1`, cela donne, après approximation et en posant $\theta =$ `(&tt[20][0]-p1)/10` :

$$\left|\begin{array}{l}\{\texttt{p1<tt+20}\}\ /[\texttt{*(p1+}\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{p2} \leftarrow \bot_{-1}, \texttt{p3} \leftarrow \texttt{t2+10}, \texttt{p1} \leftarrow \texttt{p1+}\theta, \texttt{i1} \leftarrow \texttt{i1+}\theta] \\ \{\texttt{p1>=tt+20}\}/[]\end{array}\right.$$

La transformation finale est donc :

$$\left|\ \{\text{vrai}\}/[\texttt{*(tt+}\bot_{-1}\texttt{)} \leftarrow \bot_{-1}, \texttt{p2} \leftarrow \bot_{-1}, \texttt{p3} \leftarrow \texttt{t2+10}, \texttt{p1} \leftarrow \texttt{\&tt[20][0]}, \texttt{i1} \leftarrow \texttt{20}]\right.$$

et la fonction `tr08` devient, après substitution des boucles :

```
void tr08 (void)
{
      int tt [20][10], t2 [10] ;
      int *p1, *p2, *p3, i1 ;
```

```
        p1 = & tt [0][0] ;
        i1 = 0 ;
        do (tmp2 ; 0 ; 19)
        {
              p2 = p1 + tmp2 * 10 ;
              p3 = t2 ;
              do (tmp1 ; 0 ; p1 + tmp2*10 + 10 - p2 - 1)
              {
                    p2 [tmp1] = p3 [tmp1] + i1 + tmp2 ;
              }
              p3 = p3 + (p1 + tmp2*10 + 10 - p2) ;
              p2 = p2 + (p1 + tmp2*10 + 10 - p2) ;
        }
        i1 = 20 ;
        p1 = p1 + 20 * 10 ;
  }
```

Une simple propagation de constantes nous permet de réécrire cette fonction en :

```
  void tr08 (void)
  {
        int tt [20][10], t2 [10] ;
        int *p1, *p2, *p3, i1 ;

        p1 = & tt [0][0] ;
        i1 = 0 ;
        do (tmp2 ; 0 ; 19)
        {
              p2 = & tt [0][0] + tmp2*10 ;
              p3 = t2 ;
              do (tmp1 ; 0 ; p1 + 10 - p2 - 1)
              {
                    *(&tt[0][0] + tmp2*10 + tmp1) = t2 [tmp1] + i1 + tmp2 ;
              }
              p2 = p2 + 10 ;
              p3 = p3 + 10 ;
        }
        i1 = 20 ;
        p1 = & tt [0][0] + 20 * 10 ;
  }
```

L'affectation centrale au tableau `tt` peut être réécrite en utilisant des indices :

```
  tt [tmp2][tmp1] = t2 [tmp1] + i1 + tmp2 ;
```

Et nous pouvons alors paralléliser comme précédemment. Notre objectif de "désoptimisation" est atteint : un nid de boucles est parallélisable, même en présence de pointeurs.

4.6 Conclusion

Nous avons, dans ce chapitre, exposé une méthode pour élaborer une analyse sémantique des boucles en présence de pointeurs, en vue de la parallélisation.

Notre démarche a été de définir l'objectif, c'est à dire la transformation associée à la boucle. Cet objectif étant, bien sûr, non calculable, nous avons cherché à en obtenir une approximation.

Nous avons alors défini ce qu'est une approximation : nous avons introduit de nouveaux éléments, nommés *bottom* (un par $\mathcal{A}_n$, plus un pour $\mathcal{V}$), et nous avons défini les règles qui les régissent. Ces éléments nous ont permis de définir le sens d'une approximation de valeur ou adresse, puis d'une approximation de transformation.

Fort de ces nouveaux éléments, nous avons entrepris d'approximer notre objectif, c'est à dire la transformation associée à la boucle. Nous avons commencé par approximer n itérations du corps, en trois phases successives :

1. nous avons retiré les gardes (fonctions g_i) non stables par le corps de la boucle ;
2. puis nous avons approximé les adresses (fonctions f_i) non stables ;
3. et enfin nous avons approximé les contenus de la mémoire (fonctions h_i).

Le troisième point nous a amené à identifier les variables inductives, ce qui nous a permis, dans la suite, de chercher un compte-tours et des bornes, puis de remplacer les références à ces variables inductives par des références au compte-tours de boucle. Dans le cas où nous ne pouvions détecter de compte-tours, nous avons pris un parti pessimiste.

La dernière partie de ce chapitre était consacrée à la réalisation. Nous avons décrit son état actuel, déjà avancé, puis nous avons évoqué ce qui restait à écrire. Pour illustrer nos propos, ainsi que la méthode décrite dans ce chapitre, nous avons donné quelques exemples, en montrant la sortie de la réalisation actuelle, et en terminant l'analyse manuellement. Les deux derniers exemples, au delà des cas d'école, nous ont permis de confronter des exemples complets et complexes à notre méthode, et nous ont permis de constater que l'objectif de cette étude, la parallélisation en présence de pointeurs, est désormais réalisable, et proche d'être réalisé.

Conclusion

Nous avons donc développé ce que nous pourrions appeler un “frontal de paralléliseur” (voir figure 1.2) pour un sous-ensemble du langage C. Afin d'utiliser des techniques maintenant bien connues et largement répandues de détection automatique de parallélisme en Fortran, nous avons procédé en trois étapes :

- nous avons défini un sous-ensemble du langage C nous permettant d'aborder l'analyse, mais nous avons tenu à conserver les deux éléments essentiels du langage, à savoir les expressions avec effets de bord et les pointeurs et l'arithmétique qu'ils induisent ;
- nous avons ensuite retiré tous les effets de bord que le programmeur avait introduit dans le texte source, en utilisant au besoin des variables auxiliaires, ce qui va dans le sens d'une *désoptimisation* du programme ;
- nous avons ensuite calculé les valeurs de chaque case de la mémoire après exécution du programme, afin de détecter les pointeurs utilisés à la place d'indices dans des tableaux. Lors de l'analyse, nous avons pu identifier les boucles de type *do*, et exprimer toutes les variables possibles en fonction de l'indice de boucle entier que nous avons introduit.

Toute l'analyse s'est déroulée en présence de pointeurs et d'arithmétique sur ces pointeurs. Afin que cette analyse soit possible, nous avons explicité une structuration de la mémoire en types et niveaux d'indirection dans les types. Cette structuration a évité la génération de gardes superflues, et a surtout évité que nous prenions des attitudes beaucoup trop pessimistes.

Les gardes que nous obtenons conditionnent l'état de la mémoire, ce qui signifie que les optimisations dépendent des paramètres d'exécution ou des arguments de la fonction. Dans le cas où les conditions sont remplies pour la transformation d'une boucle en boucle de type *do*, nous pouvons appliquer les techniques de parallélisation maintenant classiques.

La méthode que nous avons définie et partiellement implémentée est exploitable puisqu'elle conduit a des temps d'analyse très courts pour la plupart des fonctions. Dans les cas les plus défavorables, la complexité est exponentielle. Dans la plupart des cas, cependant, elle est sans doute polynomiale, mais malgré des des fonctions très importantes, les temps mesurés sont trop faibles pour l'affirmer avec certitude.

La réalisation se compose d'un analyseur syntaxique (15 000 lignes de C), d'un normaliseur (800 lignes de Common Lisp), d'un analyseur de séquences linéaires (3 000 lignes de Common Lisp) et d'un analyseur de boucles (1 500 lignes de Common Lisp). On peut estimer que la finition de la réalisation nous amènera à environ 3 000 lignes pour l'analyseur de boucles et l'adaptation des séquences linéaires pour la gestion des approximations nécessitera 4 000 lignes.

La qualité de l'analyse peut être amélioré grandement en raffinant certains constituants du programme, tels que le calculateur symbolique, le vérificateur d'expressions booléennes, ou le résolveur d'équations. Cependant, cette qualité se paye par des temps d'analyse plus longs, et l'expérience nous indiquera si l'équilibre que nous avons trouvé est le bon.

Nous pouvons tirer de ce travail trois principales extensions :

- la vérification de la validité de programmes, bien que non liée à la parallélisation, peut bénéficier de l'approche que nous avons adoptée. Nous avons en effet supposé que les programmes étaient valides, et en particulier que la mémoire était correctement structurée et que les programmes respectaient cette structuration. Nous pourrions suivre la même démarche que celle décrite dans ces pages et vérifier que cette structuration est respectée, que le texte de la norme est respecté ;
- le calcul des transformations est très bien adapté à la parallélisation inter-procédurale. Une transformation est en fait la synthèse de l'action du programme, et peut être utilisée pour représenter une fonction, permettant ainsi la simplification de certaines gardes sans perte de précision ;
- la méthode que nous avons développée peut être facilement étendue pour calculer directement le graphe de dépendances, ce qui permet de s'intégrer plus facilement dans un paralléliseur existant, tel que *PAF*, en remplaçant cette phase.

Nous avons présenté des exemples simples destinés à présenter certains points de la méthode développée ici. Il manque à notre étude des exemples complets et réels, voire des *benchmarks* permettant de comparer notre méthode par rapport aux méthodes pour Fortran.

Nous nous sommes concentrés sur les pointeurs utilisés pour parcourir des tableaux. Si cette étape est indispensable pour l'étude du langage C, il y a beaucoup d'autres utilisations de pointeurs, notamment dans le cadre des structures de données évoluées qui font la richesse des langages de programmation modernes.

Enfin, l'hypothèse que nous avons utilisée pour obtenir une bonne finesse dans l'analyse, à savoir la structuration de la mémoire, est en contradiction avec l'utilisation des *casts* et des *unions*.

La grande majorité de ces utilisations peut certainement être supprimée en allant dans le sens d'une *désoptimisation* encore plus poussée. Pouvons-nous détecter ces utilisations et les modifier pour continuer à bénéficier de la structuration de la mémoire ?

On pourrait continuer la liste des axes de recherches dans le droit prolongement de ce que nous avons étudié ici. On le voit, le sujet est riche de problèmes originaux et motivants qui ne demandent qu'à être explorés !

Annexe A

Représentation intermédiaire

Pour référence, voici la syntaxe exacte de la représentation intermédiaire utilisée dans cette étude.

programme	::= declaration*
declaration	::= sdecl
	| udecl
	| vdecl
	| fdecl
sdecl	::= (**sdecl** ident champ*)
udecl	::= (**udecl** ident champ*)
champ	::= (ident type)
type	::= (constructeur-ou-qualif* type-de-base)
constructeur-ou-qualif	::= qualificateur
	| constructeur
qualificateur	::= (**const**)
	| (**volatile**)

constructeur ::= (**pdecl**)
| (**adecl** **nil**)
| (**adecl** constante-entiere)

type-de-base ::= (**struct** ident)
| (**union** ident)
| (**simple-type** simple-type)
| (**fun** type declaration-parametres)
| (**void**)

simple-type ::= **char**
| **signed char**
| **unsigned char**
| **short**
| **signed short**
| **short int**
| **signed short int**
| **unsigned short**
| **unsigned short int**
| **int**
| **signed**
| **signed int**
| **unsigned**
| **unsigned int**
| **long**
| **signed long**
| **long int**
| **signed long int**
| **unsigned long**
| **unsigned long int**
| **float**
| **double**
| **long double**

declaration-parametres ::= (**proto** liste-vdecl-proto)
| (**noproto** liste-vdecl-noproto)
| (**noproto**)

liste-vdecl-proto ::= **void**
| vdecl-abstrait* **ellipsis**
| vdecl-concret* **ellipsis**

vdecl-abstrait ::= (**vdecl** **nil** type **auto** **nil**)

vdecl-concret	::= (**vdecl** ident type **auto** **nil**)
liste-vdecl-noproto	::= **void**
	| vdecl-concret*
vdecl	::= (**vdecl** ident type allocateur-global valeur-initiale)
allocateur-global	::= **static**
	| **global**
	| **extern**
valeur-initiale	::= **nil**
	| expression
fdecl	::= (**fdecl** ident type allocateur-global corps)
corps	::= **nil**
	| (**prog** vdecl-local* instruction*)
vdecl-local	::= (**vdecl** ident allocateur-local valeur-initiale)
allocateur-local	::= **auto**
	| **register**
	| **static**
bloc	::= (**prog** instruction*)
instruction	::= while
	| dowhile
	| for
	| if
	| selectq
	| goto
	| label
	| break
	| return
	| store
	| fcall

```
while          ::= ( while expression bloc )

dowhile        ::= ( dowhile expression bloc )

for            ::= ( do ( ident expression expression expression ) bloc )

if             ::= ( if expression bloc )
                 | ( if expression bloc bloc )

selectq        ::= ( selectq expression )
                 | ( selectq expression clause* )

clause         ::= ( t bloc )
                 | ( ( constante+ ) bloc )

goto           ::= ( goto ident )

label          ::= ( label ident )

break          ::= ( break )

return         ::= ( return )
                 | ( return expression )

store          ::= ( store type expression expression )

fcall          ::= ( fcall type ( expression* ) )

expression     ::= constante
                 | ident
                 | ( op1 expression )
                 | ( op2 expression expression )
                 | ( fetch type expression )
                 | ( cast type expression )
                 | ( sfield ident ident )
                 | ( sizeof type )
```

op1	::=	**neg**
	|	**bitnot**
	|	**lognot**

op2	::=	**add**
	|	**mul**
	|	**div**
	|	**sub**
	|	**mod**
	|	**bitand**
	|	**bitior**
	|	**bitxor**
	|	**logand**
	|	**logior**
	|	**shr**
	|	**shl**
	|	**eq**
	|	**ne**
	|	**gt**
	|	**lt**
	|	**ge**
	|	**le**

Bibliographie

[AH90] Ammarguellat Z., Harrison III W., *Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation.* Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, pp 283–295, June 1990.

[AHU83] Aho A., Hopcroft J., Ullman J., *Data Structures and Algorithms.* Addison Wesley, 1983.

[AJ88] Allen R., Johnson S., *Compiling C for Vectorization, Parallelization, and Inline Expansion.* Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp 241–249, June 1988.

[AK84] Allen J., Kennedy K., *Automatic Loop Interchange.* Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction, 1984.

[AK85] Allen J., Kennedy K., *PFC: A Program to Convert Fortran to Parallel Form.* Supercomputers: Design and Applications, IEEE Computer Society Press, 1985.

[AK87] Allen J., Kennedy K., *Automatic Translation of Fortran Programs to Vector Form.* ACM Transactions on Programming Languages and Systems, Vol. 9, No 4, pp 491–542, October 1987.

[Amma86] Ammarguellat Z., *Analyse syntaxique de programmes Fortran en vue de leur parallélisation.* Rapport MASI RT2, Juin 1986.

[Amma88] Ammarguellat Z., *Restructuration des programmes Fortran en vue de leur parallélisation.* Thèse de l'Université Paris 6, Rapport MASI 90.3, 1988.

[AN88] Aiken A., Nicolau A., *Optimal Loop Parallelization.* Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp 308–317, June 1988.

[Anco91] Ancourt M., *Génération automatiques de codes de transfert pour multiprocesseurs à mémoires locales.* Thèse de l'Université Paris 6, 1991.

[ANSI88] American National Standard Institute, *Draft Proposed American National Standard for Information Systems – Programminng Langage C.* Document No X3J11/88-002., January 1988.

[Arna91] Arnaud R., *Les vectoriseurs du commerce.* Support de présentation, 1991.

[AS85] Abselon, Sussman, *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

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

[Bach86] Bach M., *The Design of the Unix Operating System.* Prentice Hall, 1986.

[Back78] Backus J., *Can Programming be Liberated from the von Neuman Style? A Functional Style and its Algebra of Programs.* Communications of the ACM, Vol. 21, No 8, pp 613–641, August 1978.

[Baer73] Baer J., *A Survey of Some Theoretical Aspects of Multiprocessing.* ACM Computing Surveys, Vol. 5, No 1, pp 31–80, March 1973.

[Bana91] Banâtre J.P., *La programmation parallèle.* Eyrolles, 1991.

[Bane76] Banerjee U., *Data Dependence in Ordinary Programs.* Report 79-989, Dept of Computer Science, University of Illinois, 1979.

[Bann79] Banning J., *An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 29–41, 1979.

[BC88] Bryant B., Chao S., *Denotational Semantics for Program Analysis.* SIGPLAN Notices, Vol. 23, No 1, pp 83–91, January 1988.

[Bern66] Bernstein A., *Analysis of Programs for Parallel Processing.* IEEE Transactions on Computers, Vol. 15 No 15, pp 757–763, October 1966.

[Bodi90] Bodin F., *Data Structure Analysis in C Programs.* International Workshop on Compilers for Parallel Computers, Paris, Décembre 1990.

[CCKT86] Callahan D., Cooper K., Kennedy K., Torczon L., *Interprocedural Constant Propagation.* Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices Vol. 21, No. 7, pp 152–161, July 1986.

[CDH84] Chailloux J., Devin M., Hulot J.M., *Le_Lisp, A Portable and Efficient Lisp System.* INRIA Res. Rep. 319, Juillet 1984.

[Chai87] Chailloux J., *Le_Lisp, Manuel de Référence, Version 15.21.* INRIA Tech. Rep., Décembre 1987.

[CK88] Cooper K., Kennedy K., *Interprocedural Side-Effect Analysis in Linear Time.* Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp 57–66, June 1988.

[CK89] Cooper K., Kennedy K., *Fast Interprocedural Alias Analysis.* Proceedings of the ACM Conference on Principles of Programming Languages (POPL), pp 29–41, 1989.

[Clar81] Clarke L., "Symbolic Evaluation Methods: Implementations and Applications". Dans *Computer Program Testing*, North Holland, pp 65–102, 1981.

[Cous77] Cousot P., Cousot R., *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 269–282, January 1977.

[Cous81] Cousot P., "Semantic Foundations of Program Analysis". Dans *Program Flow Analysis*, Prentice Hall, pp 303–342, 1981.

[Cout86] Coutant D., *Retargetable High Level Alias Analysis.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 110–118, 1986.

[CWZ90] Chase D., Wegman M., Zadeck K., *Analysis of Pointers and Structures.* Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, pp 296–310, June 1990.

[Davi87] David P., *Analyse et mesure d'une étape du paralléliseur PAF.* Rapport de stage de DEA, 1987.

[Davi89] David P., *Une représentation intermédiaire pour la compilation du langage C.* Rapport MASI 300, Octobre 1989.

[DE82] Dannenberg R., Ernst G., *Formal Program Verification Using Symbolic Execution.* IEEE Transactions on Software Engineering, Vol. 8, No 1, pp 43–52, January 1982.

[Dehb90] Dehbonei B., *Étude de la génération de code et de l'analyse interprocédurale au sein d'un environnement de programmation parallèle.* Thèse de l'Université Paris 6, 1990.

[ES90] Ellis M., Stroustrup B., *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Feau84] Feautrier P., "Outil de calcul symbolique". Dans *International Symposium on Programming, Toulouse*, Springer Verlag, pp 113–124, Avril 1984.

[Feau88a] Feautrier P., *Array Expansion.* International Conference on Supercomputing, Saint Malo, July 1988.

[Feau88b] Feautrier P., *Parametric Integer Programming.* RAIRO Recherche Opérationnelle, Vol 22, No 3, Septembre 1988.

[Feau88c] Feautrier P., *Semantical Analysis and Mathematical Programming ; application to parallelization and Vectorization.* Workshop on Parallel and Distributed Algorithms, Bonas, October 1988.

[Feau88d] Feautrier P., *Parallélisation et vectorisation automatique : état de l'Art et recherches récentes.* Actes des Journées Firtech Systèmes et Télématique, Novembre 1988.

[Flyn66] Flynn M., *Very High-Speed Computing Systems.* Proc. IEEE, Vol. 54, pp 1901–1909, 1966.

[FOW87] Ferrante J., Ottenstein K., Warren J., *The Program Dependance Graph and Its Use in Optimization.* ACM Transactions on Programming Languages and Systems, Vol. 9, No 3, pp 319–349, July 1987.

[FU76] Fong A., Ullman J., *Induction Variables in Very High Level Languages.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 104–111, 1976.

[Goub88] Goubault J., *CSTE: un utilitaire de propagation interprocédurale de constantes pour le paralléliseur PIPS.* Rapport de stage de DEA, Ecole des Mines, Septembre 1988.

[Grob88] Grob L., *Automatic Exploitation of Concurrency in C: Is It Really So Hard?.* Ultracomputer Note 100, NYU, June 1988.

[HB85] Hwang K, Briggs F., *Computer Architecture and Parallel Processing.* McGraw Hill, 1985.

[HO82] Hoffmann C., O'Donnell M., *Pattern Matching in Trees.* Journal of the ACM, Vol. 29, pp 68–95, January 1982.

[Hoar78] Hoare C.A.R., *Communicating Sequential Processes.* Communications of the ACM, Vol. 21, No 8, pp 666–677, August 1978.

[HPR89] Horwitz S., Pfeiffer P., Reps T., *Dependence Analysis for Pointer Variables.* SIGPLAN'89 Conference on Programming Language Design and Implementation, pp 28–40, 1989.

[HS87] Harbison S., Steele G. Jr, *C, A Reference Manual, second edition.* Prentice Hall, 1987.

[IN90] Imprimerie nationale, *Lexique des règles typographiques en usage à l'Imprimerie nationale.* Imprimerie nationale, 1990.

[Inmo82] Inmos, *The Occam Programming Manual.* Inmos, 1982.

[JD89] Jouvelot P., Dehbonei B., *Recursive Pattern Matching on Concrete Data Types.* SIGPLAN Notices, Vol. 24, No 11, pp 84–93, 1989.

[John75] Johnson S., *Yacc – Yet Another Compiler Compiler.* Computing Science Technical Report 32, AT&T Bell Laboratories, 1975.

[John78] Johnson S., *A Portable Compiler: Theory and Practice.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 14–19, 1978.

[John79] Johnson S., *A Tour Through the Portable C Compiler.* AT&T Bell Laboratories, 1979.

[Jouv87a] Jouvelot P., *Parallélisation sémantique : une approche dénotationnelle non-standard pour la parallélisation de programmes impératifs séquentiels.* Thèse de l'Université Paris 6, 1987.

[Jouv87b] Jouvelot P., *Semantic Parallelization: A Practical Exercise in Abstract Interpretation.* Proceedings of the ACM Conference on Principles of Programming Languages (POPL), pp 39–48, 1987.

[JR81] Johnson S., Ritchie D., *The C Language Calling Sequence.* Computing Science Technical Report 102, AT&T Bell Laboratories, 1981.

[Kill73] Killdal G., *A Unified Approach to Global Program Optimization.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 194–206, October 1973.

[KKLW80] Kuck D., Kuhn R., Leasure B., Wolfe M., *The Structure of an Advanced Vectorizer for pipelined Processors.* Fourth International Computer Software and Applications Conference IEEE, October 1980.

[Knut68] Knuth D., *The Art of Computer Programming.* Addison Wesley, Vol 1 : Fundamentals Algorithms, 1968.

[KP81] Kernighan B., Plauger P., *Software Tools in Pascal.* Addison Wesley, 1981.

[KR78] Kernighan B., Ritchie D., *The C Programming Langage.* Prentice Hall, 1978.

[KR88] Kernighan B., Ritchie D., *The C Programming Langage, second edition.* Prentice Hall, 1988.

[KU77] Kam J., Ullman J., *Monotone Data Flow Analysis Frameworks.* Acta Informatica, Vol. 7, pp 305–317, 1977.

[Lamp74] Lamport L., *The Parallel Execution of DO Loops.* Communications of the ACM, Vol 17, No 2, pp 83–96, February 1974.

[Laur90] Laurent-Verleye C., *Vectorisation automatique des boucles comportant des instructions de branchement.* Thèse de l'Université Paris 6, 1990.

[Lesk75] Lesk M., *Lex – A Lexical Analyzer Generator.* Computing Science Technical Report 39, AT&T Bell Laboratories, 1975.

[Li86] Li K., *A Note on the Vector C Language.* Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices Vol. 21, No 1, pp 49–57, January 1986.

[LMKQ89] Leffler S., McKusick M.K., Karels M., Quarterman J., *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison Wesley, 1989.

[Love76] Loveman D., *Program Improvements by Source to Source Transformations.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 140–152, 1976.

[LT85] Lichnewsky A., Thomasset F., *Techniques de base pour l'exploitation automatique du parallélisme dans les programmes.* Rapport INRIA 460, 1985.

[LT88] Lichnewsky A., Thomasset F., *Introducing Symbolic Problem Solving Techniques in the Dependence Testing phases of a Vectorizer.* Proceedings of the 1988 International Conference on Supercomputing, July 1988.

[Mann74] Manna Z., *Mathematical Theory of Computation.* McGraw Hill, 1974.

[May83] May D., *Occam.* SIGPLAN Notices, Vol. 18, No 4, April 1983.

[MB84] Meyer B., Baudoin C., *Méthodes de programmation.* Eyrolles, Collection de la Direction des Études et Recherches d'Électricité de France, 1984.

[PGHL89] Polychronopoulos C., Girkar M., Haghighat M., Ling Lee C., Leung B., Schouten D., *Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors.* Proceedings of the 1989 International Conference on Parallel Processing, 1989.

[PH90] Patterson D., Hennessy J., *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[PKL80] Padua D., Kuck D., Lawrie D., *High Speed Multiprocessors and Compilation Techniques.* IEEE Transactions on Computers, C29, September 1980.

[Poly88] Polychronopoulos C., *Compiler Optimizations for Enhancing Parallelism and their Impact on Architecture Design.* IEEE Transactions on Computers, Vol. 37, No 8, pp 991–1004, August 1988.

[PW86] Padua D., Wolfe M., *Advanced Compiler Optimizations for Supercomputers.* Communications of the ACM, Vol. 29, No 12, pp 1184–1201, December 1986.

[Quin87] Quinn M., *Designing Efficient Algorithms for Parallel Computers.* McGraw Hill, 1987.

[Riff86] Rifflet J.M., *La programmation sous Unix.* McGraw Hill, 1986.

[Riff90] Rifflet J.M., *La communication sous Unix.* McGraw Hill, 1990.

[RT81] Reif J., Tarjan R., *Symbolic Program Analysis in Almost Linear Time.* SIAM Journal of Computing, Vol. 11, pp 81–93, February 1981.

[Seni91] Seniak N., *Théorie et pratique de Sqil, un langage intermédiaire pour la compilation de langages fonctionnels.* Thèse de l'Université Paris 6 (à soutenir), 1991.

[Seth89] Sethi R., *Programming Languages, Concepts and Constructs.* Addison Wesley, 1989.

[SF85] Schreiner A., Friedman G. Jr, *Introduction to Compiler Construction with Unix.* Prentice Hall, 1985.

[SGN88] Scheifler R., Gettys J., Newman R., *X Window System, C Library and Protocol Reference.* Digital Press, 1988.

[Stal88] Stallman R., *Internals of GNU CC.* Free Software Foundation, 1988.

[Stee84] Steele G. Jr, *Common Lisp, The Language.* Digital Press, 1984.

[Stre85] Strevens C., *The Transputer.* Proceedings of the International Symposium on Computer Architecture, 1985.

[Stro86] Stroustrup B., *The C++ Programming Language.* Addison Wesley, 1986.

[Taba90] Tabak D., *Multiprocessors.* Prentice Hall, 1990.

[Tawb91] Tawbi N., *Parallélisation automatique : estimation des durées d'exécution et allocation statique de processeurs.* Thèse de l'Université Paris 6, 1991.

[TDF87] Tawbi N., Dumay A., Feautrier P., *PAF : un paralléliseur pour Fortran.* Rapport MASI 185, Mai 1987.

[Thor86] Thornburg J., *Number Crunching in C.* SIGPLAN Notices, Vol. 21, No 6, pp 59–60, June 1986.

[TIF86] Triolet R., Irigoin F., Feautrier P., *Direct Parallelization of Call Statements.* Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices Vol. 21, No 7, pp 176–185, July 1986.

[TMC87] Thinking Machines, *Connection Machine Model CM-2 Technical Summary.* Thinking Machines Technical Report HA87-4, April 1987.

[Trio84] Triolet R., *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédure.* Thèse de l'Université Paris 6, Décembre 1984.

[Weih80] Weihl E., *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 83–94, 1980.

[WZ85] Wegman M., Zadeck K., *Constant Propagation with Conditional Branches.* Proceedings of ACM Symposium on Principles of Programming Languages (POPL), pp 291–299, January 1985.

[ZC91] Zima H., Chapman B., *Supercompilers for Parallel and Vector Computers.* ACM Press, Addison Wesley, 1991.