Analyse Statique Comportementale des Langages de Programmation.

– Rapport d'Habilitation Université Paris XI –

Pierre Jouvelot

CRI
Ecole des Mines de Paris
35, rue Saint-Honoré
77305 Fontainebleau Cedex
E-mail: jouvelot@cri.ensmp.fr

29 Décembre 1993

Ce document est une introduction aux travaux présentés le 11 Janvier 1994 devant le jury composé de:

- Patrick Cousot (ENS), rapporteur
- Paul Feautrier (Paris VI), rapporteur
- David K. Gifford (MIT),
- Jean-Pierre Jouannaud (Paris XI),
- Phil Wadler (Glasgow U.), rapporteur

pour obtenir le diplôme d'Habilitation à Diriger des Recherches en Sciences de l'Université de Paris XI.

Pour complétement apprécier le contenu de ces recherches, le lecteur pourra se reporter avec profit au texte complet des articles donnés en annexe.

Remerciements

L'ensemble de ces travaux n'aurait pu être ce qu'il est sans le concours de nombreux chercheurs et amis qu'il m'est un plaisir de remercier ici :

- Au Centre de Recherche en Informatique (CRI) de l'Ecole des Mines de Paris, mon co-locataire de bureau François Irigoin, Rémi Triolet, Corinne Ancourt, Ronan Keryell, Michel Lenci, Robert Mahl, Jacqueline Altimira, Annie Pech-Ripaud, et mes étudiants, Vincent Dornic, Jean-Pierre Talpin, Yan-Mei Tang et Philippe Bazet.

- Au laboratoire Méthodologie et Architecture des Systèmes Informatiques (MASI) de Paris VI, mon directeur de thèse d'université le Professeur Feautrier et Claude Girault, ainsi que tous ceux qui m'ont aidé à l'obtenir.

- Au Programming Systems Research Group (PSRG) du Massachusetts Institute of Technology (MIT), le Professeur David K. Gifford, mon co-locataire de bureau transatlantique Mark A. Sheldon, Jim O'Toole, Franklyn Turbak, Michael "Ziggy" Blair, John Lucassen et Brian Reistad.

- Au Laboratoire de Recherche en Informatique (LRI) de Paris XI, les Professeurs Jean-Pierre Jouannaud qui m'a invité à présenter ma thèse d'habilitation à Paris XI et Jean-Pierre Sansonnet.

- Au Centre de Recherche Bull, mon co-auteur "industriel" Babak Dehbonei, Gérard Memmi et Nadia Tawbi.

- A l'Etablissement Technique Central de l'Armement (ETCA), Serge Petiton et Philippe Clermont, maintenant à HyParTech.

- Au Programme de Recherches Coordonnées C3, Patrice Quinton, pour son financement.

- A la Direction Recherche Et Technique (DRET), Philippe Sarrazin.

De plus, François Irigoin, Corinne Ancourt, Philippe Bazet, François Masdupuy, et Babak Dehbonei ont participé à la relecture de ce manuscript.

1 Introduction à l'analyse statique

Avancer l'état de l'art de l'analyse statique de langages de programmation, pour des applications, paradigmes et formalismes variés, représente le leitmotiv essentiel des travaux repertoriés dans ce rapport.

L'*analyse statique* d'un langage de programmation permet de déterminer, au moment de la compilation d'un programme ou d'un de ses modules, une approximation de sa sémantique standard. Le type d'information à déterminer, ainsi que la manière d'effectuer cette analyse, dépendent à la fois des objectifs ultimes de cette analyse et du paradigme de programmation du langage.

L'essentiel des travaux en revue se focalise sur l'étude du comportement dynamique des programmes (le *comment*) par opposition aux valeurs qu'ils calculent (le *quoi*). Ce comportement dynamique décrit les caractéristiques pratiques d'un processus d'évaluation en répondant aux questions comme "Y-a t'il des effets de bord durant le calcul?", "Combien de temps prend un tel calcul?" ou "Est-ce que cette évaluation peut s'effectuer en parallèle?". Ces questions ne concernent pas la valeur calculée par un programme, mais la manière dont celle-ci l'est.

Cette introduction présente un panorama des applications possibles des méthodes d'analyse des programmes, une discussion sur les impacts des différents styles de programmation sur les résultats et méthodes d'analyse et un parcours rapide des divers formalismes d'analyse généralement proposés. Une dernière partie montre comment ces différentes dialectiques structurent les travaux de recherche présentés dans la suite du document.

1.1 Motivations

Les raisons qui suggèrent de recourir à une analyse statique sont variées et se retrouvent dans toute l'étendue du cycle de vie d'un logiciel, que ce soit dans la phase de conception, programmation, exécution ou maintenance. A titre d'illustration, et pour fixer les idées, il peut être utile de signaler les applications suivantes :

- Vérification de conformité entre programme et spécification, elle-même probablement exprimée dans un autre langage de plus haut-niveau. L'archétype de ce type d'analyse est le *typage* d'un programme. La donnée du type d'une expression est une spécification restreignant les valeurs possibles de celle-ci à l'ensemble des valeurs représentant le type. La vérification ou reconstruction des types dans un programme permet d'en vérifier la consistance. Si cette consistance n'est que partielle (en particulier, les valeurs exactes des expressions ne sont souvent pas accessibles au moment du typage), elle permet néanmoins de déjà détecter certaines erreurs. A noter qu'il est intéressant de voir se profiler à l'horizon l'idée consistant à effectuer des analyses statiques sur de tels langages de spécification.

- Optimisation plus poussée d'un programme en fonction de son contexte d'utilisation et/ou de sa structure interne. Un exemple particulièrement marqué de cette préoccupation sera ici la *parallélisation automatique*, où l'on essaie de détecter dans un programme à priori séquentiel les commandes susceptibles d'être exécutées en parallèle sur une architecture parallèle donnée. La notion de type, vue précédemment, est également un exemple classique en optimisation car elle permet à la fois d'économiser

l'allocation en mémoire (absence de bits de type) et en temps (absence de test dynamique décidant entre les différentes sémantiques des opérations avec surcharge).

- Documentation automatique. En phases de mise au point ou d'évolution d'un logiciel, des analyses statiques peuvent être effectuées pour déterminer certaines propriétés dynamiques de modules, permettant au mainteneur d'obtenir rapidement une vue d'ensemble de la sémantique d'un module peu fréquemment étudié.

1.2 Paradigmes

Si les langages de programmation sont nombreux, les styles de programmation différents, eux, le sont moins. Programmations impérative, fonctionnelle, orientée-objet, logique et par contraintes forment l'essentiel des paradigmes de programmation. Si tous ont leurs avantages et inconvénients, il est indéniable que le paradigme de programmation, qui caractérise un langage pour lequel une analyse est définie, influe de manière marquante sur sa commodité d'utilisation.

A titre d'exemple, dans un langage impératif comme Fortran, le graphe de contrôle des appels de procédure est statique[1], tandis que dans un langage fonctionnel comme Standard ML, celui-ci est intrinsèquement dynamique, les objets fonctionnels y étant de première classe[2]. Si la détermination du graphe des fonctions appelées par une expression est relativement simple dans le premier cas, il en est tout autrement dans le second.

Les travaux ici présentés sont essentiellement focalisés sur les langages de programmation *impératifs*, rapidement étendus pour prendre en compte des aspects *fonctionnels* avec la gestion de valeurs de type fonctionnel. Cette alliance représente un compromis intéressant, permettant des styles de programmation variés, des spécifications fonctionnelles exécutables aux algorithmes classiques pour machines RAM. Un tel cadre permet ainsi de faire évoluer un projet logiciel de l'état de prototype à celui d'implémentation quasi-industriel; ceci est un gage de l'intérêt pratique de l'intégration de ces deux styles de programmation.

1.3 Formalismes

De nombreux formalismes existent pour spécifier des analyses statiques de programmes; le choix de l'un au détriment d'un autre est à la fois une question de goût pour le concepteur et d'adaption au paradigme du langage de programmation sous-jacent.

On préfère généralement utiliser une approche itérative, à base de graphes de contrôle, dans les langages impératifs classiques, langages dont la sémantique dynamique est difficile à spécifier dans les autres cadres; on pourra songer ainsi à la nécessité d'introduite des continuations pour prendre en compte les branchements. Cette approche, opérationnelle, permet de prendre en compte le caractère non-compositionnel de tels langages.

Les approches proposées ici, plus novatrices, s'appuient sur les techniques d'*interprétation abstraite* et de *sémantique naturelle*. Toutes deux sont compositionnelles et s'adaptent bien aux langages, relativement simplifiés par rapport à des langages de

[1] Pour simplifier la présentation, la possibilité de passer le nom d'une fonction en argument est ici passée sous silence.

[2] Une valeur est dite *de première classe* si elle est susceptible d'être passée en argument, stockée dans des structures de données ou retournée comme résultat d'une fonction.

programmation complets dont la complexité n'apporterait d'ailleurs rien sur le plan des résultats, qui ont été étudiés.

L'approche basée sur la sémantique dénotationnelle non-standard (interprétation abstraite) a été utilisée pour proposer de techniques nouvelles de parallélisation automatique, tandis que les systèmes de types (sémantique naturelle) ont permis de déterminer certaines propriétés comportementales en présence de fonctions d'ordre supérieur.

1.4 Plan

Sept publications internationales, comprenant cinq articles de conférence et deux publications dans des journaux, sont présentées dans le reste de ce document, chacune dans une section propre. Elles se découpent en deux thèmes essentiels : la parallélisation automatique et les systèmes d'effets. Une dernière section (section 9) présente l'ensemble de ces travaux dans une perspective historique et plus personnelle; le lecteur est invité à s'y reporter pour dégager une vision encore plus unifiée de l'ensemble de ces travaux et en consolider la logique.

Parallélisation automatique

Le premier thème explore les divers aspects d'une nouvelle approche pour la parallélisation automatique, appelée *parallélisation sémantique*.

Le cadre général est esquissé dans le premier article "Parallélisation Sémantique", écrit en collaboration avec le Professeur Paul Feautrier de l'Université Paris VI et publié dans la revue Informatique Théorique et Applications de l'AFCET [33]. Cet article fait la synthèse de deux papiers présentés à la conférence *European Symposium On Programming* tenu en 1986 [29] et à la conférence ACM *Principles Of Programming Languages* de 1987 [30].

La validation expérimentale des résultats présentés en 1986 fait l'objet du second article "A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions", publié dans les actes de la conférence ACM *International Conference on Supercomputing* de 1989 [32]. Babak Dehbonei, travaillant à l'époque au Corporate Research Center de Bull, en est co-auteur.

L'extension interprocédurale de ces recherches a été en partie intégrée dans le paralléliseur PIPS développé en collaboration avec le reste de l'équipe du Centre de Recherche en Informatique de l'Ecole des Mines de Paris. Ce système est décrit dans le troisième article de cette série, "Semantical Interprocedural Parallelization : An Overview of the PIPS Project", écrit conjointement avec François Irigoin et Rémi Triolet et publié dans les actes de la conférence ACM *International Conference on Supercomputing* de 1991 [25].

Systèmes d'effets

Si les techniques de parallélisation proposées précédemment peuvent s'adapter à divers langages impératifs, elles ne sont plus adaptées au traitement de langages autorisant la manipulation libre de valeurs fonctionnelles. L'effort de recherche décrit ici a consisté à développer la notion de *système d'effets* qui permet de poursuivre la parallélisation, et autres optimisations plus classiques, en présence de telles caractéristiques dans les langages typés. De la même manière qu'un type donne une approximation de la valeur calculée par une expression, un effet renseigne sur la manière selon laquelle cette valeur est calculée.

Ces effets peuvent être exprimés, tout naturellement, en termes d'opérations mémoire, mais aussi, de manière plus inattendue et montrant le caractère général de cette approche, de comportement non-fonctionnel ou de coût de calcul par exemple.

Si l'on peut faire remonter la mise au point d'un système d'effet explicite à 1986 [16], les travaux décrits dans "Algebraic Reconstruction of Types and Effects", présentés en 1991 à la conférence ACM *Principles Of Programming Languages* [37], montrent pour la première fois comment les informations nécessaires à la parallélisation peuvent être omises par le programmeur et reconstruites automatiquement par le compilateur. Ce travail a été effectué en collaboration avec le Professeur David K. Gifford du MIT.

Le résultat suivant, obtenu en collaboration avec Jean-Pierre Talpin, à l'Université Paris VI, affine cette avancée initiale en proposant une analyse plus fine des programmes. Cette publication, "Polymorphic Type, Region and Effect Inference", acceptée par le *Journal of Functional Programming* en 1992, doit paraitre dans le numéro 2 du volume 2 [54].

Si la motivation initiale du concept de système d'effets était de permettre la parallélisation de programmes en présence de fonctions d'ordre supérieur, les deux résultats suivants montrent sa généralisation à d'autres types d'analyse statique.

Un premier exemple a consisté à intégrer la notion d'effet de contrôle dans le modèle des effets. Ce résultat, décrit dans "Reasoning about Continuations with Control Effects", obtenu en collaboration avec David K. Gifford, est publié dans les actes de la conférence ACM *Programming Languages Design and Implementation* en 1989 [35].

Enfin, en collaboration avec Vincent Dornic, de l'Ecole des Mines de Paris, et toujours David K. Gifford, un système d'analyse de complexité en temps des programmes a été développé et implémenté. On trouvera le papier présentant cette approche, "Polymorphic Time Systems for Estimating Program Complexity", dans le journal ACM *Letters On Programming Languages And Systems* du volume 1, numéro 1 de 1992 [13].

2 Parallélisation Sémantique

L'article "Parallélisation Sémantique" [33] résumé ci-dessous présente les travaux de thèse décrits dans [31] et effectués sous la direction du Professeur Paul Feautrier au Laboratoire MASI de l'Université Paris VI. Il y est proposé une nouvelle approche de la parallélisation automatique de programmes séquentiels impératifs pour machines parallèles.

2.1 Introduction

Deux approches logicielles ont été, jusqu'à présent, proposées pour utiliser pleinement les nouvelles architectures parallèles de machines. La première consiste à concevoir de nouveaux langages de programmation dans lesquels le parallélisme est explicitement déclaré par le programmeur. La seconde approche du parallélisme essaye de compiler les langages séquentiels classiques en détectant automatiquement le parallélisme (implicite) présent dans les programmes.

Cette seconde approche est usuellement justifiée par, d'une part, l'existence d'un énorme stock de programmes séquentiels et, d'autre part, la rareté des spécialistes capables de concevoir directement leurs applications en parallèle. Elle est donc ordinairement vue comme une solution temporaire. Il est, au contraire, possible de penser que les méthodes développées pour la parallélisation automatique ont de fortes chances de se retrouver dans les compilateurs des futurs langages spécialisés. En effet, vue la grande variété des architectures parallèles, il parait illusoire de demander à un programmeur d'exhiber tout le parallélisme contenu dans son programme. Lui demander de faire un choix revient à lui faire intégrer dans son code les caractéristiques de la machine cible, ce qui va à l'encontre de toute l'évolution de la programmation moderne. Il parait plus raisonnable de laisser ce travail à la charge d'un compilateur, qui devra pour cela utiliser des méthodes similaires à celles qui sont présentées ici.

Pour atteindre un tel but, une perception très fine du programme source est nécessaire; le concept de *parallélisation sémantique* introduit ici est une voie possible pour atteindre ce but.

Le premier objectif est d'extraire automatiquement, au moment de la compilation, des *informations sémantiques* à partir du texte du programme. Elles seront utilisées pour implémenter des techniques sophistiquées de détection de constructions parallélisables à l'intérieur d'un programme séquentiel. Il sera alors possible de paralléliser des fragments de programmes qui n'étaient pas détectés comme parallélisables avec les méthodes plus classiques.

Pour exprimer ce processus complexe d'extraction d'informations (i.e., compréhension) de programmes, un cadre théorique puissant et riche est nécessaire. Le second objectif est d'utiliser des spécifications inspirées des techniques *dénotationnelles* [53]. La parallélisation de programmes est vue comme une interprétation non-standard du langage de programmation. A l'aide de la théorie des domaines et du formalisme de l'interprétation abstraite [8], on est alors à même de prouver la correction des méthodes de parallélisation par rapport à la sémantique standard du langage. A noter que cette notion de correction des transformations de programmes est généralement peu abordée par les chercheurs travaillant dans le domaine de la parallélisation; une méthode pour aborder ce point important est donc proposée ici.

Le dernier objectif de cette nouvelle approche est plus pragmatique. Il est bien connu qu'une spécification dénotationnelle d'un langage de programmation donne gratuitement un interprète *exécutable* de ce langage. En utilisant un langage fonctionnel d'ordre supérieur (en pratique, ML [27]), certaines de ces nouvelles techniques de parallélisation ont pu être implémentées dans un prototype de *parallèliseur sémantique* et ce directement à partir des spécifications [28].

2.2 Informations sémantiques

La parallélisation automatique est basée sur la possibilité de détecter au moment de la compilation la présence de conflits en mémoire entre différentes instructions. Dans les programmes scientifiques, ces instructions utilisent généralement des éléments de tableaux et détecter un conflit revient à décider de la satisfiabilité de l'égalité d'expressions d'indice. Améliorer la parallélisation revient à rendre plus puissant cet algorithme de décision.

Trois types d'informations différents ont été envisagés pour améliorer les processus de parallélisation classique : les *prédicats*, les *t-prédicats* et l'*évaluation symbolique*. Ceux-ci sont intégrés dans le même cadre formel, présenté ci-dessous.

- Un prédicat est vu ici comme un système d'inégalités linéaires sur les variables scalaires du programme [58]. Ces prédicats sont utiles pour décider de la présence ou de l'absence de conflit-mémoire entre deux instructions susceptibles d'être parallélisées, conflit qui est détecté grâce aux classiques conditions de Bernstein [3].

- Si les prédicats sont utiles pour traiter le problème de la parallélisation dans le cas où les indices de tableaux sont affines, ils ne sont plus suffisant dans le cas où des indirections sont utilisées. Ce cas, difficile et peu souvent traité, a été abordé en introduisant les t-prédicats qui sont une abstraction symbolique de la fonction que représente un tableau. Cette fonction est utilisée ultérieurement pour détecter les conflits.

- Enfin, certaines expressions qui ne sont pas parallélisables au sens de Bernstein [3] sont néanmoins susceptibles d'être exécutées de manière efficace sur certaines machines parallèles; il s'agit des réductions, comme la somme d'un vecteur, qui peuvent être parfois executées en temps $O(log\ N)$. L'évaluation symbolique est l'outil proposé ici pour détecter ce type de comportment.

2.3 Exemple

L'exemple suivant, obtenu sur le prototype de paralléliseur sémantique écrit en ML, montre l'application des techniques de prédicats et d'évaluation symbolique à un programme simple. Les t-prédicats n'ont pas été complètement intégrés au système, et ne sont donc pas illustrés ici. La fonction `parallel_pp` prend en entrée un texte de programme (entre guillemets) et retourne une chaine représentant sa version parallélisée. Le parallélisme est soit de type SIMD avec la boucle parallèle `for//`, soit MIMD avec le bloc parallèle `|[...]|` :

```
# parallel_pp "
    array t[100]; array u[100];
    var i; var n; var m;
```

```
    read n;
    s := 0;
    if n > 10 then
        n := n+1;
    else
        n := 10;
    for i to 10 do
        [
        t[i] := t[i+n];
        t[i+n+20] := 0 ;
        u[2*i+51] := u[2*i]+50;
        s := s+t[i+n];
        ]" ;;


‘   array t[100]; array u[100];
    var i; var n; var m;
    |[
        read n;
        s := 0;
    ]|
    if (n) > (10) then
        n := (n)+(1) ;
    else
        n := 10 ;
    |[
        s := for_red i to 10 eval (s+t[(i)+(n)]);
        [
        for// i to 10 do
            |[
                t[i] := t[(i)+(n)];
                t[((i)+(n))+(20)] := 0;
                u[((2)*(i))+(51)] := u[((2)*(i))+(50)]
            ]|
        i := 11;
        ]
    ]|
‘
: string
```

Pour paralléliser la boucle sur `i` en préservant la sémantique séquentielle du programme, il est nécessaire de se rendre compte que les valeurs possibles de `n` en entrée de boucle sont toujours supérieures à 10. En effet, dans ce cas, il n'y a pas de conflit possible entre les éléments du tableau `t` accédés en lecture et écriture par la première instruction de la boucle. Les paralléliseurs actuels ne sont pas à même de traiter ce type de programme; ceci nécessite de la part du paralléliseur une connaissance approfondie du programme à paralléliser. La

notion de prédicat permet de lever cette limitation. Sur cet exemple, le paralléliseur a obtenu cette information automatiquement par analyse du texte du programme précédant la boucle (par propagation de prédicats faisant intervenir `n`).

La propriété sur `n` n'est pas suffisante pour paralléliser la boucle dès que l'on prend en compte les autres instructions du corps de boucle. En effet, une autre raison pour laquelle la boucle sur `i` n'est pas parallélisée par les techniques actuelles provient du calcul de `s`. Pour se rendre compte que le calcul de `s` correspond à une réduction sur le tableau `t` (`s` est la somme d'un sous-tableau de `t`) et peut donc être sortie de la boucle, il est à la fois nécessaire de détecter la présence d'une opération associative sur `s` et de prouver que les éléments de `t` concernés ne sont pas modifiés par d'autres instructions de la boucle. Ceci peut être obtenu à l'aide d'une évaluation symbolique qui sera décrite par la suite.

2.4 Cadre d'analyse

Une *interprétation abstraite* définit une sémantique *non-standard* (i.e., différente de celle utilisée habituellement pour définir l'évaluation d'une commande par un ordinateur) d'un langage de programmation. Par exemple, on pourrait associer à chaque identificateur d'un programme une information approchée sur le signe (positif, négatif ou inconnu) de sa valeur en chaque point d'un programme; on parle alors d'une sémantique *attribuée*. On pourrait également obtenir, comme le fait [9], une précondition logique pour toute commande d'un programme; il s'agirait alors d'une sémantique *relationnelle*.

Une sémantique *directe* abstraite A d'un langage de programmation, définie sur le domaine `Com` de ses commandes, est une fonction de type :

$$A : \texttt{Com} \rightarrow D_a \rightarrow D_a$$

où D_a est appelé domaine *abstrait*. La sémantique standard C des commandes d'un langage est généralement vue comme un transformateur d'états mémoire $Store$. La relation qui lie A et la sémantique standard C du langage est donnée par une fonction dite de *concrétisation* γ qui applique D_a dans l'ensemble des parties $\mathcal{P}(Store)$; pour un élément abstrait donné d_a, $\gamma(d_a)$ représente l'ensemble des états-mémoire qui sont compatibles avec d_a.

Pour qu'une interprétation abstraite directe A soit correcte par rapport à la sémantique standard C , une condition doit être établie :

Condition 1 (Correction) *A est correcte par rapport à C si et seulement si, pour toute commande `C` et élément abstrait d_a,*

$$\{C[\![\texttt{C}]\!]s / s \in \gamma(d_a)\} \subseteq \gamma(A[\![\texttt{C}]\!]d_a)$$

La définition d'une interprétation abstraite pour un langage de programmation donné peut être faite de différentes manières : par exemple, [8] utilise une approche opérationnelle, tandis que [47] introduit une vue dénotationnelle. Cette seconde technique est utilisée ici pour obtenir un cadre général et unique de spécification de parallélisation.

En vue d'utiliser ces interprétations dénotationnelles abstraites comme des spécifications exécutables, il faut insister sur deux points. Premièrement, les domaines abstraits doivent être *syntaxiques* (au sens de [52]) pour être effectivement implémentables sur machine. Deuxièmement, il convient de calculer (i.e., approximer) les points fixes qui vont apparaître dans les spécifications, puisque le processus de parallélisation doit être fait au moment de la compilation.

2.5 Analyses sémantiques

Trois interprétations non-standard sont présentées dans [33] pour réaliser des analyses de parallélisation sémantique. La première voit une commande d'un programme comme un transformateur de prédicats; toute affectation linéaire de variables est utilisée pour enrichir le prédicat d'entrée avec l'information apportée par l'affectation. Ces prédicats sont alors associés à chaque instruction et utilisés dans la phase de parallélisation proprement dite (voir la section 4 pour une mise en pratique en vraie grandeur de ces idées).

La seconde interprétation voit les commandes comme des transformateurs d'environnement de t-prédicats. Un tel environnement associe chaque tableau à une fonction qui donne, pour tout indice, une valeur exacte de l'élément correspondant. La nécessité d'obtenir une information exacte est liée à l'utilisation ultérieure qui en est faite : elle intervient dans l'estimation de la valeur d'une expression d'indice dans une indirection comme `a[b[i]]`. Si l'approche utilisée est intéressante en théorie, elle est néanmoins limitée en pratique car elle nécessite d'appliquer, pour la représentation de ces fonctions, l'arithmétique de Presburger, dont la complexité est hyperexponentielle [48]. La dernière interprétation voit chaque commande comme un transformateur d'états symboliques; cette technique, étendue et appliquée en pratique, est décrite plus avant dans la section 3.

A coté de leur spécification formelle, toutes ces interprétations abstraites sont prouvées correctes par rapport à la sémantique standard; chaque théorème est une application du théorème général présenté ci-dessus. Sur un plan pratique, la langage ML [20] a été utilisé comme langage d'exécution de spécifications car il autorise la manipulation de fonctions d'ordre supérieur et les considère comme des "citoyens de première classe". Ce sont exactement les caractéristiques nécessaires à un langage pour qu'il soit aisément utilisable pour écrire des spécifications dénotationnelles. De plus, le système de typage automatique offert par ML est un avantage majeur quand il s'agit d'écrire des expressions complexes utilisant des fonctions largement currifiées : il assure une détection extrêmement efficace des erreurs.

3 Réductions généralisées

L'article "A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions" [32] résumé ci-dessous décrit une généralisation et implémentation complète d'une partie des travaux décrits dans [29] et [33]. Ce travail a été effectué en collaboration avec Babak Dehbonei, du Centre de Recherche Groupe de Bull.

3.1 Introduction

Si la possibilité d'utiliser l'analyse sémantique pour détecter des réductions comme le produit scalaire de vecteurs avait été introduite dans [29] et prototypée dans [31], il restait à montrer qu'une telle analyse était effectivement utile dans un compilateur plus industriel pour machines parallèles. C'etait le premier objectif des travaux répertoriés dans [32].

De plus, poursuivant une réflexion sur la nature profonde des réductions, il a été montré dans le même article que les expressions invariantes dans les boucles et les variables d'induction (i.e., les variables incrémentées par pas constant d'une itération à une autre) pouvaient être vues comme des cas particuliers de réductions. Ceci a conduit à introduire la notion de *réduction généralisée*, concept qui permet de mettre dans le même cadre théorique

et pratique les variables inductives, les invariants de boucle et les réductions.

L'approche proposée déjà dans [31] consiste à utiliser l'évaluation symbolique pour déterminer au moment de la compilation la fonction dénotée par le corps d'une boucle. Cette valeur symbolique est alors comparée par filtrage à une base de données de fonctions de réductions généralisées. S'il y a accord, le code parallèle associé est généré et le corps de boucle est purgé des instructions de calcul de la réduction. Dans le cas contraire, le doute subsiste et le corps de boucle est laissé tel quel.

L'approche proposée dans cet article préserve également les acquis présentés dans [31] : puissance et généralité de l'analyse, preuves de correction et prototypage rapide des spécifications. Cette technique, implémentée dans le vectoriseur Velour développé au Centre de Recherce Bull [11], permet de détecter des expressions parallélisables dans des benchmarks classiques là où les vectoriseurs plus traditionnels maintiennent une version séquentielle des programmes.

3.2 Evaluation Symbolique

L'approche proposée est basée sur une nouvelle méthode d'analyse appelée *évaluation symbolique globale*, qui formalise l'approche de [6]. Comme précédemment, l'évaluation symbolique y est vue comme une interprétation non-standard de la sémantique directe du langage.

La valeur symbolique d'une expression e est une liste d'*expressions gardées*. Une expression gardée consiste en une paire (b, e') de deux expressions du langage; la seconde expression e' représente la valeur de e quand la première expression booléene b, appelée *garde*, est vraie. La garde est introduite par la présence des instructions de test qui conditionnent les valeurs ultérieures des variables qui y sont modifiées. La valeur symbolique d'une expression est donc une liste comportant toutes ses valeurs possibles, fonctions des chemins parcourus par le contrôle lors de l'évaluation des conditionnelles. La sémantique d'une instruction est alors celle d'un transformateur de *mémoire symbolique*; une mémoire symbolique lie un identificateur à sa valeur symbolique.

Par rapport à la sémantique standard, les fonctions définissant les dénotations des expressions et instructions, outre qu'elles travaillent sur les domaines symboliques pendants des domaines standards, ont besoin d'un argument supplémentaire appelé *contexte*. Cette expression symbolique booléenne représente le chemin courant à travers les branches des instructions conditionnelles englobantes et est utilisée pour déterminer la garde des expressions symboliques.

A titre d'exemple de résultat obtenu par l'évaluateur, après évaluation symbolique de la suite d'instructions suivantes :

```
max = 0
s = t(i)
if (max < s)
        max = s
        incr = 1
else
        incr = 0
endif
incr = max+incr
```

à partir d'un état-mémoire initial qui lie chaque identificateur `i` à la valeur symbolique `[true,i]`, on obtient l'état suivant (après simplification) :

```
s    → [true,t(i)]
incr → [0 < t(i),1+t(i) ; 0 >= t(i),0]
max  → [0 < t(i),t(i) ; 0 >= t(i),0]
```

Le processus d'évaluation symbolique est linéaire en fonction du nombre d'instructions du programme et exponentiel par rapport au degré d'imbrication, généralement faible, des instructions conditionnelles.

3.3 Détection

Une fois effectuée l'évaluation symbolique d'un corps de boucle, la recherche des réductions généralisées se fait par appariement de la valeur symbolique de l'expression résultante des variables modifiées par la boucle avec une base de motifs représentant chacune des réductions effectivement parallélisables.

De manière formelle, une réduction généralisée sur une variable v est définie comme une fonction associative $f(v, e_l)$ où e_l est une expression indépendante de v. En pratique, et au vu de ce que nécessite la majorité des programmes scientifiques contenant des réductions, il est possible de se limiter aux réductions généralisées qui peuvent être définies par un quintuplet (v, i, g, e_l, f) où

- v est la variable calculant la réduction,
- i est la variable d'itération de boucle,
- g est une expression booléenne qui spécifie la garde associée à la réduction,
- e_l est une expression qui décrit la valeur, dépendant de l'indice de boucle, utilisée dans le calcul de la réduction,
- f est la fonction qui relie v à e_l quand g est vrai (v n'est pas modifié quand g est faux),

et où g, e_l et f peuvent utliser i. La valeur symbolique qui correspond à une telle réduction généralisée v est alors :

$$[g,f(v,e_l);\ not(g),v]$$

A titre d'exemple, le quintuplet correspondant à la recherche du maximum d'un tableau, opération qui est souvent implémentée efficacement sur les ordinateurs vectoriels ou parallèles et qui correspond à la boucle suivante :

```
for i to 100 do
    [
    if max > t[i] then
         max = t[i];
    ]
```

est (max,i,(max > t[i]),t[i],λ(x,y).y).

La base de schémas de réduction standards couple chaque quintuplet avec sa fonction de génération de code associée. C'est cette fonction qui est utilisée pour générer le code qui réalisera l'opération de réduction de manière efficace; il lui est passé comme argument le quintuplet instancié qui correspond au programme contenant la réduction. A titre d'exemple, dans le système Velour, le schéma correspondant à la recherche du dernier élément positif d'un tableau est défini de la manière suivante en LeLisp, langage d'implémentation de Velour [5] :

```
;; Quintuplet pour: if a[i] > 0.0 then s = a[i]
;;
(defconstant last-positive-pattern
  `(,(make-pattern
      ?1
      ?2
      (exp-binary '> (exp-array  ?3 (exp-scalar ?2))
                     (exp-constant 0.0))
      (exp-array ?3 (exp-scalar ?2))
      (lambda (x y) y))
    ,(lambda (v i g el f)
       (inst-assign (exp-scalar v)
                    (exp-call 'last-positive
                              (array-of-exp el) (loop-bounds i))))))
```

où `exp-binary`, `exp-array`, `exp-constant`, `exp-scalar`, `exp-call` et `inst-assign` sont des fonctions de la syntaxe abstraite du langage. La fonction `array-of-exp` permet de déterminer la nom du tableau référencé dans l'expression `el` et `loop-bounds` l'intervalle de valeurs parcouru par l'indice de boucle `i`. Le métacaractère `?` permet de dénoter les diverses variables de filtrage qui dénoteront respectivement la variable d'induction, l'indice de boucle et le tableau.

3.4 Résultats

Pour étudier les performances pratiques de l'approche proposée, une série de tests basée sur des suites de tests standards d'Argonne National Lab [4] et de Florida University [39].

Dans les tests d'Argonne, on s'est limité aux boucles contenant des définitions de variables d'induction et de réduction. Dans les résultats donnés dans la table suivante, "n", "p" et "v" indiquent respectivement une boucle non, partiellement ou totalement vectorisée; le nombre de boucles "v" est donné dans la dernière colonne :

Velour peut vectoriser la boucle s124 car il effectue une évaluation symbolique des instructions conditionnelles et est ainsi capable de détecter une variable d'induction mise à jour dans les deux branches alternatives. La boucle s332, appellée "search loop saving index", ne peut pas à l'heure actuelle être traitée par Velour car elle contient une instruction de branchement (qui pourrait être éliminée par un restructureur).

Dans les tests de l'Université de Floride, les 12 boucles décrites comme "Reduction Functions" ont été analysées. Il n'y a pas de boucles spécifiques des inductions dans les test de Floride. Parmi ces tests, toutes les boucles, sauf 10 et 12, ont été détectées comme étant des réductions par Velour.

	s121	s122	s123	s124	s125	s126	s127	#
Velour	v	v	n	v	v	v	v	6
Alliant FX/8	v	v	n	n	v	v	v	5
CDC Cyber 205	v	v	n	n	v	v	v	5
ETA-10	v	v	n	n	v	v	v	5
Hitachi S-810/820	v	v	n	n	v	v	v	5
Unisys ISP	v	v	p	p	v	v	v	5
Ardent Titan-1	v	n	n	n	v	v	v	4
CRAY X-MP CFT77	v	v	n	n	v	n	v	4
Convex C Series	v	v	n	n	v	n	v	4
Intel iPSC-VX	v	v	n	n	v	n	v	4
NEC SX/2	v	v	n	n	v	v	n	4
Amdahl 1200/1400	v	v	n	n	v	n	n	3
CRAY X-MP CFT	v	v	n	n	v	n	n	3
IBM 3090/VF	v	v	n	n	v	n	n	3
Stellar GS1000	v	v	n	n	v	n	n	3
CRAY-2	n	v	n	n	v	n	n	2
SCS 40	n	v	n	n	v	n	n	2
Gould NP1	v	n	n	n	n	n	n	1
CDC Cyber 990	n	n	n	n	n	n	n	0

Figure 1: Nombre de boucle d'Argonne vectorisées avec réductions (partie 1)

	s311	s312	s313	s314	s315	s316	s317	s318	s331	s332	#
Hitachi	v	v	v	v	v	v	v	v	v	v	10
Unisys	v	v	v	v	v	v	v	v	v	n	9
Velour	v	v	v	v	v	v	v	v	v	n	9
Alliant	v	v	v	v	v	v	n	v	v	n	8
Amdahl	v	p	v	v	v	v	n	v	v	v	8
Gould	v	v	v	v	v	v	n	v	v	n	8
NEC	v	v	v	v	v	v	v	v	n	n	8
X-MP CFT	v	v	v	v	n	v	v	v	n	v	8
Ardent	v	v	v	n	v	n	v	v	v	n	7
Cyber 205	v	v	v	v	v	v	n	v	n	n	7
Convex	v	v	v	n	v	n	v	v	v	n	7
CRAY-2	v	v	v	v	n	v	v	v	n	n	7
ETA-10	v	v	v	v	v	v	n	v	n	n	7
X-MP CFT77	v	v	v	n	n	n	v	v	v	v	7
Intel	v	v	v	n	v	n	n	v	v	n	6
SCS 40	v	v	v	n	n	n	v	v	n	n	5
Stellar	v	p	v	n	n	n	n	v	v	n	4
Cyber 990	v	p	v	n	v	n	n	n	n	n	3
IBM	v	p	v	n	n	n	n	v	n	n	3

Figure 2: Nombre de boucle d'Argonne vectorisées avec réductions (partie 2)

4 Parallélisation sémantique interprocédurale

L'article "Semantic Interprocedural Analysis : An Overview of the PIPS Project" [25] résumé ci-dessous décrit un paralléliseur pour Fortran développé au Centre de Recherche en Informatique de l'Ecole des Mines. PIPS correspond à une réflexion commune sur les besoins d'analyse interprocédurale pour obtenir une parallélisation efficace; les analyses présentées ci-dessus sont toutes intraprocédurales. Ce travail de recherche a été effectué en collaboration avec François Irigoin et Rémi Triolet du CRI.

4.1 Structure de PIPS

L'objectif principal du projet PIPS est d'étudier l'impact pratique des techniques interprocédurales pour l'analyse et la parallélisation de programmes scientifiques. Il est également conçu avec l'ambition de pouvoir traiter des programmes Fortran réels comme ceux disponibles dans le Perfect Club ou Linpack et non plus simplement des noyaux de petite taille caractéristiques de suites de test comme les boucles de Livermore ou d'Argonne.

Sa conception fait appel à des concepts relativement novateurs dans le domaine du génie logiciel, qui font de PIPS un produit un peu à part de la majorité des paralléliseurs existants. Étant un projet de recherche, PIPS est conçu autour d'une architecture modulaire et extensible, capable de s'adapter à des chercheurs variés, à l'interprocéduralité des analyses et à ses interfaces à la fois interactive et par lot. Chaque phase d'analyse d'un module manipule une représentation intermédiaire unique, soigneusement définie pour être facilement extensible à d'autres langages (la grande majorité des particularités liées à Fortran sont cachées) et minimale pour faciliter la tâche de programmation (même l'instruction d'affectation est vue comme un appel de procédure). Ces structures de données sont gérées par une pseudo-base de données chargée de masquer, dans la mesure du possible, au programmeur les transferts sur mémoire stable de ces données. Cette base est aussi responsable de l'enchainement des calculs, chaque phase y étant déclarativement définie en fonction de ses données d'entrée et de sortie.

La représentation intermédiaire des programmes utilisée dans PIPS est défini à l'aide de l'outil NewGen développé au CRI [36]. Celui-ci, à partir de descriptions de domaines, génére en C ou CommonLISP des fonctions de création, manipulation, modification, écriture et libération-mémoire adaptées à ces types de données. Cette approche déclarative et automatique permet d'abstraire la programmation des contraintes d'implémentation des structures de données et de normaliser l'écriture des logiciels. A titre d'illustration de la concision des descriptions NewGen et du choix de la représentation intermédiaire de PIPS, on trouvera ci-dessous la description du domaine des expressions Fortran :

```
expression = reference + range + call ;

reference = variable:entity x indices:expression* ;
range = lower:expression x upper:expression x increment:expression ;
call = function:entity x arguments:expression* ;
```

Le symbole `x` dénote le produit cartésien de domaines, la somme `+` l'union disjointe et l'itérateur `*` la liste. On voit ainsi qu'un noeud `call` de l'arbre abstrait, correspondant à un appel de fonction Fortran, contient l'entité appelée de type `function` ainsi que la liste des

expressions passées en `arguments`. Comme indiqué ci-dessus, une affectation est encodée d'une manière similaire, avec la fonction intrinsèque `=` et deux arguments correspondant aux membres gauche et droit de l'affectation. Ceci est possible en utilisant le mode de passage par référence, standard dans Fortran.

4.2 Analyses sémantiques

De part sa structure, PIPS est extensible avec un nombre a priori quelconque de phases d'analyse et de transformation (voir ci dessous 4.3). Ces phases d'analyse, rapidement esquissées par la suite, sont exprimées récursivement en terme du *graphe de contrôle structuré*.

Le graphe de contrôle structuré de PIPS est une structure de données nouvelle bien adaptée à la parallélisation de programmes. Les algorithmes de parallélisation existants, dans leur grande majorité, supposent que le programme initial est structuré, i.e., sans instruction de branchement. Cette hypothèse simplificatrice n'est pas applicable dans un produit qui se veut capable de traiter des programmes réels. Une première approche pour éliminer ce problème est de restructurer le programme, i.e. de remplacer les branchements par des structures de contrôle structurées ; cette phase n'a pas été implémentée car elle ne permet pas toujours d'éliminer les branchements de manière satisfaisante. La seconde approche, introduite pour la première fois dans le projet PIPS, consiste à construire un graphe de contrôle, mais en essayant de limiter l'étendue des branchements. En effet, si un corps de boucle contient un branchement qui lui est local, alors, vue de l'extérieur, cette boucle reste structurée et peut donc être parallélisée comme si elle l'était vraiment. On introduit donc une notion de hiérarchie dans la définition de ce graphe, hiérarchie qui mélange arbre de syntaxe abstrait et graphe orienté. Ceci se voit bien dans la définition, ici légérement simplifiée, du domaine des instructions utilisé dans PIPS :

```
instruction = block:instruction* + test + loop + call + unstructured ;

test = condition:expression x true:instruction x false:instruction ;
loop = index:entity x range x body:instruction x label:entity ;
unstructured = control x exit:control ;

control = instruction x predecessors:control* x successors:control* ;
```

Le graphe de contrôle structuré est défini par le domaine `unstructured` tandis que les instructions structurées sont soit une instruction élémentaire modifiant l'état-mémoire (`call`), une sequence (`block`), un test (`test`) ou une boucle DO (`loop`).

Toutes les phases d'analyse sémantique sont effectuées par induction sur la structure du domaine `instruction`. Parmi celles-ci, on trouve :

- *Effets.* Un effet est une approximation des opérations effectuées sur la mémoire par une ou plusieurs instructions. On distingue les effets propres qui sont locaux à une instruction, comme l'effet d'écriture sur le membre gauche d'une affectation, les effets cumulés qui prennent en compte les effets des sous-instructions et les effets résumés qui masquent les effets sur des variables locales. Contrairement aux effets propres qui font référence à l'adresse complète concernée, les effets cumulés ne sont définis que par la variable, scalaire ou tableau, qui a été modifiée ou lue. Puisque le système est

interprocédural, les effets cumulés d'une procédure appelée apparaissent comme des effets propres au niveau des sites d'appel.

- *Graphe use-def.* Les effets des instructions sont utilisés pour déterminer, à partir du graphe de contrôle structuré, les relations de dépendance *use-def*, utilisation vs. définition, entre instructions. Cette ébauche de graphe de dépendance sera raffiné à l'aide des informations déterminées par les analyses suivantes.

- *Prédicats.* Ils expriment, pour chaque instruction, des contraintes linéaires [59, 31] sur les variables scalaires entières qui peuvent être utilisées dans les expressions d'indice de tableau apparaissant dans les effets propres d'une instruction. Pour les déterminer, les instructions sont vues comme des *transformateurs* d'état-mémoire abstrait. Contrairement à l'approche présentée ci-dessus (section 2), ici ces états sont représentés sous forme de polyèdres convexes dans lesquels apparaissent à la fois les valeurs d'entrée et de sortie des variables.

- *Régions.* Ces prédicats raffinent les effets cumulés en abstrayant les expressions d'indices dans les accès aux tableaux. Les indices des éléments concernés par un effet sont approximés par un polyèdre dans lequel apparaissent des variables d'indexation virtuelles; ces variables sont contraintes par les prédicats se rapportant aux instructions. Ces informations sont utilisées dans la détermination de dépendance entre instructions.

L'aboutissement de ces diverses phases d'analyse est l'affinage des informations *use-def* en un graphe de dépendance sur lequel la méthode classique de parallélisation, adaptée à notre cas [1], est appliquée. Le test de dépendance utilise un algorithme de vérification de satisfiabilité de systèmes d'inéquations linéaires inspiré de celui de Fourier-Motzkin [59].

4.3 Transformations

PIPS propose un ensemble varié de transformations qui sont utilisées pour améliorer le processus de parallélisation. Le papier complet présente quatre des phases qui sont implémentées à l'heure actuelle :

- *Privatisation.* Les variables qui ne sont définies et manipulées que localement à une itération du corps de boucle peuvent être ignorées dans le processus de parallélisation. En utilisant le graphe *use-def*, cette phase détecte la boucle la plus englobante dans laquelle une variable peut être privatisée.

- *Réductions.* La méthode décrite dans la section 3 a été adaptée à PIPS. Puisque NewGen facilite la communication de données entre langages différents, cette adaptation a été relativement aisée, puisqu'il s'est agit d'un simple recodage en CommonLISP du code LeLisp utilisé dans Velour.

- *Distribution.* L'algorithme d'Allen et Kennedy [1] permet d'effectuer la distribution, permettant ainsi de séparer, dans une boucle importante, les parties parallélisables de celles qui doivent rester séquentielles pour préserver la sémantique originale du programme.

- *Réorganisation.* Si la structure initiale d'emboîtement d'un nid de boucles ne permet pas de le paralléliser, il est parfois possible de réorganiser l'espace d'itérations pour exhiber le parallélisme caché. La méthode hyperplane [41] a été implémentée dans PIPS, quoique non complètement intégrée.

De par la conception même de PIPS, de nouvelles phases de transformation (ou d'analyse) peuvent (et sont) facilement intégrées au fur et à mesure de l'évolution des recherches, par exemple pour prendre en compte d'autres modèles de mémoire telles que les architectures distribuées [2].

4.4 Résultats

PIPS, après deux ans et 6 homme-ans de conception, développement et test, tourne sur Sparc, offre une interface X11 et est capable de traiter des exemples réels de plusieurs milliers de lignes Fortran, comme ceux fournis par l'ONERA.

Il a déjà permis de tirer un certain nombre d'observations et de recommendations pratiques préliminaires pour améliorer le processus de parallélisation. La principale est d'ordre pragmatique : si l'analyse sémantique est en théorie utile pour améliorer la parallélisation sur des cas d'école, elle est souvent trop puissante dans les programmes réels, ceux-ci utilisant rarement les constructions complexes qui la rendrait nécessaire.

5 Reconstruction algébrique

L'article "Algebraic Reconstruction of Types and Effects" [37] résumé ci-dessous décrit une méthode d'analyse sémantique ayant permis [24], de paralléliser les programmes écrits dans des langages permettant de manipuler des fonctions d'ordre supérieur. Les travaux présentés dans les sections précédentes ne s'adaptent pas à ce type de paradigme de programmation sans l'introduction de nouveaux concepts comme ceux développés ici. Ce travail de recherche a été effectué en collaboration avec le Professeur David K. Gifford et son groupe de recherche, le Programmming Systems Research Group, du Massachusetts Institute of Technology aux Etats-Unis.

5.1 Cadre

L'analyse sémantique interprocédurale est aisée dans les langages impératifs comme Fortran car le graphe des appels de procédure y est facilement connu au moment de la compilation, si l'on excepte le cas des fonctions passées en paramètre, si peu usitées en pratique dans ce type de langages qu'une analyse minimale convient généralement. La situation change du tout au tout dans les langages plus fonctionnels comme CommonLISP ou Standard ML où la norme y est la manipulation de fonctions *first-class*. Ici, le graphe des appels est dynamique et toute fonction est une valeur composite, qui lie à un code de programme son environnement de définition. Le cas des langages à liaison dynamique ne sera pas évoqué ici, leur inadéquation en terme de méthodologie de programmation étant fréquemment jugée rédhibitoire à leur utilisation à grande échelle.

Pour déterminer une approximation du comportement dynamique d'une expression, l'approche proposée dans [16, 44] consiste à définir, en parallèle à un système de types : un *système d'effets*. De la même manière qu'un type renseigne sur la valeur calculée par

une expression, un effet indique comment cette valeur est calculée. Cette caractéristique comportementale peut être reliée à de nombreux aspects de l'évaluation d'un langage; on les verra ci-dessous exprimés en terme d'opérations mémoire. Cette approche est étendue, dans la section 7, aux exceptions de contrôle et enfin, dans la section 8, aux temps d'exécution. La donnée statique du type et de l'effet des expressions permet alors d'effectuer des analyses sémantiques comportementales variées d'un programme. La famille des langages FX a été conçue pour valider la pertinence de cette nouvelle approche.

Dans le langage FX-87 [17], les informations de type et d'effet doivent être fournies par l'utilisateur; le typage est dit *explicite*. Du fait de la puissance du système de type utilisé, une extension du typage polymorphique d'ordre quelconque [46], la découverte d'une méthode permettant d'inférer la majorité des informations de type et d'effet telle qu'elle est présentée ci-dessous permet d'envisager une utilisation plus réaliste de l'approche proposée par FX. Cette technique de reconstruction a été intégrée à la version actuelle du langage FX, appelée FX-91 [18].

Si Fortran ou un de ses dialectes simplifiés ont été les langages privilégiés pour étudier les analyses sémantiques dans le cadre de langages impératifs, un lambda-calcul étendu avec des constructions impératives est, très classiquement, utilisé ici pour définir ces analyses prenant en compte des valeurs fonctionnelles. La syntaxe de FX-91, compatible avec celle de Scheme [50], est utilisée par la suite.

5.2 Système d'effets

D'une manière similaire au typage, les effets des expressions d'un programme définissent une sémantique statique du langage. Celle-ci est généralement définie par un ensemble de règles exprimées selon le cadre de la sémantique naturelle [49]. Ce style de définition est généralement préféré à l'interprétation abstraite utilisée dans les sections précédentes, car il apparaît plus déclaratif et ainsi plus proche d'une spécification que devra connaître tout programmeur utilisant ce langage. Un séquent classique d'un système de typage est de la forme $A \vdash E \ : \ T$, indiquant que, dans un environnement de typage A liant chaque variable à son type, l'expression E est de type T.

L'introduction de la notion d'effet revient à introduire une nouvelle description F décrivant l'effet d'une expression. Les effets considérés ici sont, soit des identificateurs comme `pure` qui indique l'abscence d'effet, soit des effets composites créés grâce au combinateur `maxeff`. Les effets élémentaires sont introduits via les fonctions de manipulation mémoire définies dans l'environnement initial. Les règles de typage prennent alors la forme $A \vdash E \ : \ T \,!\, F$ et définissent, outre le type T de l'expression E, son effet F. Les règles statiques sont alors étendues pour prendre en compte cette nouvelle information; les typages de l'application et de l'abstraction deviennent alors :

$$(\rightarrow\text{IT})\frac{A[I:T] \ \vdash \ E_0 \ : \ T_0 \,!\, F_0}{A \ \vdash \ (\texttt{lambda}\ (I\ T)\ E_0) \ : (\texttt{subr}\ F_0\ (T)\ T_0) \,!\, \texttt{pure}}$$

$$(\rightarrow\text{E})\ \frac{\begin{array}{c} A \ \vdash \ E_0 \ : \ (\texttt{subr}\ F\ (T_1)\ T) \,!\, F_0 \\ A \ \vdash \ E_1 \ : \ T_1 \,!\, F_1 \end{array}}{A \ \vdash \ (E_0\ E_1) \ : \ T \,!\, (\texttt{maxeff}\ F_0\ (\texttt{maxeff}\ F_1\ F))}$$

Les deux règles fondamentales sont ici l'introduction de l'opérateur fonctionnel ($\rightarrow$IT) et son élimination ($\rightarrow$E). Pour l'abstraction, le type et l'effet du corps d'une fonction sont

déterminés dans un environnement augmenté du type de l'argument; le type résultat est utilisé pour définir le type fonctionnel, dénoté ici par le mot-clé `subr`. On notera le concept crucial d'*effet latent* : l'effet du corps d'une fonction est introduit dans le type de la fonction au moment de sa définition. Pour l'application, le type et l'effet résultat sont déterminés en spécifiant ceux de l'expression en position fonctionnelle et de son argument. Les types des arguments formels et réels doivent être les mêmes et le type résultat est celui de la valeur retournée par la fonction. Aux effets combinés des expressions est ajouté l'effet latent du type fonctionnel. On voit ici que l'effet latent est utilisé au moment de l'application pour transférer l'effet du corps du point de définition de la fonction au point d'utilisation. C'est cette propriété qui permet de traiter complètement les fonctions d'ordre supérieur.

Le langage FX-91 utilise les fonctions `new`, `get` et `set`, ayant pour effet latent respectif `init` pour indiquer l'allocation-mémoire mutable, `read` pour l'accès-mémoire en lecture et `write` en écriture. L'algèbre des effets de FX-91 est associative, commutative, idempotent et unitaire par rapport à l'opérateur de combinaison d'effet `maxeff` :

$$\begin{aligned}(\texttt{maxeff}\ F_0\ (\texttt{maxeff}\ F_1\ F_2)) &\sim (\texttt{maxeff}\ (\texttt{maxeff}\ F_0\ F_1)\ F_2)\\ (\texttt{maxeff}\ F_0\ F_1) &\sim (\texttt{maxeff}\ F_1\ F_0)\\ (\texttt{maxeff}\ F\ \texttt{pure}) &\sim F\\ (\texttt{maxeff}\ F\ F) &\sim F\end{aligned}$$

Cette spécification indique que ni l'ordre des effets-mémoire, ni leur répétition importent et que `pure` représente l'absence d'effet. Ce choix algébrique est fonction des analyses effectuées par la suite et de la sémantique du langage et varie quand on exprime d'autres propriétés (on en verra d'autres exemples par la suite).

Les descriptions de type sont alors classiquement définies par induction structurelle :

$$\begin{array}{rcll} T & ::= & I & \\ & & (\texttt{subr}\ F\ (T_0)\ T_1) & \textit{procédure}\end{array}$$

5.3 Reconstruction algébrique

La nouveauté introduite dans cet article consiste à autoriser l'omission de certains types d'arguments (et donc des effets qu'ils peuvent éventuellement contenir). On passe ainsi d'un système permettant la vérification de types et d'effets à un formalisme nécessitant une *reconstruction*. Ceci est exprimé simplement en ajoutant la règle suivante à la sémantique statique :

$$(\rightarrow\text{IM})\frac{A[I:M] \ \vdash\ E_0\ :\ T_0\ !\ F_0}{A\ \vdash\ (\texttt{lambda}\ (I)\ E_0)\ :(\texttt{subr}\ F_0\ (M)\ T_0)\ !\ \texttt{pure}}$$

Contrairement à la règle ($\rightarrow$IT), le type de l'argument n'est plus disponible dans le texte du programme; la sémantique statique indique ici qu'il sera reconstruit automatiquement. Le système statique sujet de cet article est complété[3] par la règle classique sur les variables :

$$(var)\frac{I:T\ \in\ A}{A\ \vdash\ I\ :\ T\ !\ \texttt{pure}}$$

[3]Le système décrit dans l'article publié contient également l'abstraction et la projection polymorphes d'ordre quelconque. Cet aspect est omis dans cette brève présentation.

La reconstruction des types structurels est un problème relativement bien maîtrisé depuis [45] quoique les travaux récents essaient toujours de prendre mieux en compte les effets de bord. Cet aspect est complètement traité, et de manière très générale, dans le formalisme présenté ici. Par contre, de part leur caractère algébrique riche, les effets posent des problèmes nouveaux. En effet, les techniques structurelles ne peuvent traiter que des algèbres libres et ce pour deux raisons : (1) l'algorithme d'unification de Robinson [51] qui les sous-tend ne peut imposer que l'équivalence syntaxique d'expressions et est impuissant en face d'axiomes algébriques complexes comme l'associativité ou la commutativité et (2) la reconstruction structurelle repose sur l'idée que chaque expression a un type le plus général unique qui peut être spécialisé par simple substitution syntaxique. Cette dernière raison est clairement illustrée par l'exemple suivant[4] :

```
(lambda (f)
  (lambda (g (subr write (bool) bool))
    (if #t
        g
        (lambda (x) (begin (f x) (g x))))))
```

Le type de `f` est omis, mais le fait qu'il est appliqué à `x` implique qu'il s'agit d'une procédure prenant un booléen en argument et retournant une valeur dont le type importe peu. Son effet latent e est contraint à vérifier l'équation `write` $\sim$ (`maxeff` e `write`) car les deux types fonctionnels de l'alternative doivent être de même type et, donc, de même effet latent. Les deux solutions possibles pour e, c'est-à-dire `write` et `pure`, ne peuvent pas être obtenues par simple substitution syntaxique; `f` ne peut donc pas avoir de type le plus général.

5.4 Algorithme de reconstruction

La reconstruction algébrique consiste à intégrer dans un même cadre la reconstruction syntaxique sur les composantes structurelles des types et la résolution de contraintes sur celles qui ne le sont pas, i.e. les effets. L'algorithme d'*unification algébrique* `U` étend celui de Robinson pour prendre en compte les contraintes algébriques d'effets : il calcule à la fois une substitution S sur les types et un ensemble de contraintes C sur les effets qui permettent d'unifier ses deux arguments. Dans la description ci-dessous, ϕ dénote l'ensemble vide, ν est une variable d'unification et [] la substitution identité.

```
U( T1, T2 ) =
T1 = ν =>
        if T1 ∉ T2 then ([T2/T1], φ) else fail
T2 = ν =>
        if T2 ∉ T1 then ([T1/T2], φ) else fail
T1 = I =>
        if T2 = I then ([], φ) else fail
T1 = (subr F1 (T11) T12) =>
        if T2 = (subr F2 (T21) T22) then
                let (S1, C1) = U( T11, T21 )
                let (S2, C2) = U( S1T12, S1T22 )
```

[4] `begin` et `if` sont les formes spéciales classiques représentant la séquence et l'alternative.

```
                (S_2 S_1 , C_1 ∪ C_2 ∪ {(F_1, F_2)})
        else fail
else fail
```

Deux expressions de type s'unifient si et seulement si l'ensemble de contraintes retourné par `U` est satisfiable. Un modèle m satisfait un ensemble de contraintes C, noté $m \models C$, si et seulement si pour toute contrainte (F, F') de C, on a $mF \sim mF'$. Cette satisfiabilité peut être vérifiée, par exemple en utilisant un algorithme d'unification ACUI [38]. Le papier présente une méthode plus efficace réduisant le problème à celui de la satisfiabilité de clauses de Horn en logique des propositions [14].

L'algorithme de reconstruction `R` utilise cette routine d'unification pour calculer le type T et l'effet F d'une expression E dans un environnement de typage A. Ces descriptions sont contraintes par un ensemble de contraintes C, tandis qu'une substitution S propage celles introduites par E dans l'environnement englobant. `R` retourne le quaduplet (T, F, C, S) :

```
R( A, E ) = case E in
I =>
        if [I : T] ∈ A then (T, pure, φ, []) else fail
(lambda (I T) E) =>
        let (T', F', C, S) = R( A[I : T], E )
        ((subr F' (T) T'), pure, C, S)
(lambda (I) E)
        let (T, F, C, S) = R( A[I : ν], E )
                where ν is fresh
        ((subr F (Sν) T), pure, C, S)
(E_0 E_1) =>
        let (T_0, F_0, C_0, S_0) = R( A, E_0 )
        let (T_1, F_1, C_1, S_1) = R( S_0 A, E_1 )
        let (S, C) = U( S_1 T_0, (subr ν_1 (T_1) ν_2) )
                where ν_i are fresh
        let F' = (maxeff F_0 (maxeff F_1 ν_1))
        (Sν_2, F', C_0 ∪ C_1 ∪ C, S S_1 S_0) else fail
```

L'algorithme `R` est *consistant* vis-à-vis de la sémantique statique : tout expression qui est typable selon `R` vérifie la sémantique statique pour toute instance de l'ensemble de contraintes.

Theorem 1 (Consistance) *Soit* $(T, F, C, S) = R(A, E)$ *et un modèle* m *:*

$$(m \models C) \Longrightarrow mSA \vdash E \ : \ mT \,!\, mF$$

L'algorithme `R` est *complet* vis-à-vis de la sémantique statique : toute expression typable est reconnue comme telle par `R` et le type et l'effet reconstruits sont équivalents à ceux donnés par la sémantique statique.

Theorem 2 (Complétude) *Si* $mSA \vdash E : T \,!\, F$, *il existe* (T', F', C', S'), *un modèle* m' *et une substitution* P' *tels que :*

$$\begin{cases} (T', F', C', S') = \mathtt{R}(A, E) \\ m'P'S'A = mSA, \ sur \ FV(E) \\ m' \models C' \\ T \sim m'P'T' \\ F \sim m'F' \end{cases}$$

Suivant en cela Standard ML, il est possible d'ajouter une certaine forme de *généricité* au système tel qu'il est présenté jusqu'à présent. Celle-ci étant réutilisée dans l'article suivant (voir section 6), sa présentation y est différée.

6 Inférence polymorphe de régions

L'article "Polymorphic Type, Region and Effect Inference" [54] résumé ci-dessous décrit une extension de la méthode de reconstruction algébrique permettant d'affiner la représentation des effets de bord. Jusqu'à présent, un simple identificateur comme `write` ou `read` indique la présence d'opérations non-fonctionnelles; l'extension décrite ici introduit la notion de *région* comme une abstraction de la mémoire, les effets étant alors indexés par régions. Ce travail de recherche a été effectué en collaboration avec Jean-Pierre Talpin, étudiant en thèse au Centre de Recherche en Informatique de l'Ecole des Mines de Paris.

6.1 Régions

Si les effets permettent de différencier les expressions fonctionnelles de celles qui sont impures, i.e. qui effectuent des opérations de mutation mémoire, les informations collectées telles que `read` ou `write` ne sont pas suffisantes pour réaliser des optimisations fines. Ainsi, si l'on considère l'expression :

```
(let ((a (make-vector 10 0))
      (b (make-vector 10 1)))
  (add-vector (inc-vector! a)
              (inc-vector! b)))
```

avec l'approche présentée précédemment, les deux opérations d'incrémentation de vecteurs ne peuvent être effectuées en parallèle; elles ont toutes deux un effet `write` qui indique la possibilité de conflit mémoire, alors qu'il est bien clair qu'elles sont indépendantes. Il est donc nécessaire de définir une abstraction plus fine des opérations de la sémantique dynamique et, en particulier, de préciser l'étendue de telles opérations. La notion de *région*, déjà introduite dans FX-87 [17], répond en partie à cet objectif; une région est une abstraction d'un ensemble de valeurs mutables comme les vecteurs ou les références. Deux valeurs sont allouées dans la même région si elles peuvent être en alias, i.e. qu'une mutation opérée sur l'une d'elles peut être observable via la seconde.

Des informations d'effet plus précises sont alors définies en les décorant avec la région sur laquelle ils s'opèrent. Il y a possibilité de conflit entre deux effets s'ils influencent la même région. Dans l'exemple précédent, si `a` est alloué dans la région `A` et `b` dans `B`,

les deux sous-expressions d'incrémentation, d'effet respectif `(write A)` et `(write B)`, sont indépendantes et peuvent donc être ordonnancées en parallèle. A noter néanmoins que cette approche ne permet toutefois pas de s'apercevoir que des écritures sur `A[2]` et `A[3]` sont indépendantes.

Utilisant une approche basée sur la reconstruction algébrique, l'article résumé ici montre comment associer automatiquement chaque valeur, plus précisément tout point d'allocation, d'un programme à une région dans un langage typé d'ordre supérieur. Outre cette nouvelle analyse statique, il y est montré, via la méthode d'induction de point fixe maximal [57], que cette sémantique statique est correcte par rapport à une sémantique dynamique instrumentée pour garder trace des effets de bord.

6.2 Sémantique dynamique

Le langage utilisé contient les primitives classiques du lambda-calcul typé, plus les primitives de gestion de références, à savoir allocation `new`, la lecture `get` et la mutation `set`. Cette sémantique dynamique $s_0, E \vdash e \rightarrow v, f, s$ voit classiquement une expression comme un transformateur d'état-mémoire s, liant adresse à valeur, qui retourne également une valeur v. L'addition majeure tient en ce qu'une *trace* f est également retournée; cette trace est un ensemble d'actions de la forme $init(l)$, $read(l)$ ou $write(l)$ représentant respectivement l'allocation, la lecture ou l'écriture d'une adresse l. On note f_x la fonction identique à f, sauf en x où elle est non définie.

$$(var) : \frac{\mathtt{x} \in Dom(E)}{s, E \vdash x \rightarrow E(\mathtt{x}), \emptyset, s}$$

$$(abs) : \frac{}{s, E \vdash (\mathtt{lambda}\ (x)\ e) \rightarrow \langle \mathtt{x}, \mathtt{e}, E_{\mathtt{x}} \rangle, \emptyset, s}$$

$$(app) : \frac{\begin{array}{l} s_0, E \vdash e \rightarrow \langle \mathtt{x}, \mathtt{e}'', E' \rangle, f, s \\ s, E \vdash e' \rightarrow v', f', s' \\ s', E' \cup \{\mathtt{x} \mapsto v'\} \vdash \mathtt{e}'' \rightarrow v'', f'', s'' \end{array}}{s_0, E \vdash (e\ \ e') \rightarrow v'', f \cup f' \cup f'', s''}$$

$$(new) : \frac{s_0, E \vdash e \rightarrow v, f, s \qquad l \notin Dom(s)}{s_0, E \vdash (\mathtt{new}\ e) \rightarrow l, f \cup \{init(l)\}, s \cup \{l \mapsto v\}}$$

$$(get) : \frac{s_0, E \vdash e \rightarrow l, f, s}{s_0, E \vdash (\mathtt{get}\ e) \rightarrow s(l), f \cup \{read(l)\}, s}$$

$$(set) : \frac{s_0, E \vdash e \rightarrow l, f, s \quad s, E \vdash e' \rightarrow v, f', s'}{s_0, E \vdash (\mathtt{set}\ e\ e') \rightarrow u, f \cup f' \cup \{write(l)\}, s'_l \cup \{l \mapsto v\}}$$

Sémantique Dynamique

L'article complet traite également la récursion via `letrec` et les définitions locales via

`let`. Le traitement du `let` est voisin de celui présenté dans la section précédente, une approche beaucoup plus puissante étant présentée dans l'article plus récent [55].

6.3 Sémantique statique

La sémantique statique associe chaque expression `e` à son type τ et effet σ. Chaque expression allouant une référence mutable le fait dans une région donnée selon la règle *(new)*; une valeur de type $ref_\rho(\tau)$ dénote ainsi une référence dans la région ρ vers un objet de type τ. Cette discipline revient à mettre en alias toutes les valeurs qui sont créées par la même expression textuelle d'une expression (voir les travaux de Luddy Harrison [22] sur les *procedure strings* pour une approche plus précise, quoique que beaucoup plus complexe) :

$$(var) : \frac{\mathtt{x} \mapsto \tau \in \mathcal{E}}{\mathcal{E} \vdash x : \tau, \emptyset} \qquad (abs) : \frac{\mathcal{E}_{\mathtt{x}} \cup \{\mathtt{x} \mapsto \tau\} \vdash e : \tau', \sigma}{\mathcal{E} \vdash (\mathtt{lambda}\ (x)\ e) : \tau \xrightarrow{\sigma} \tau', \emptyset}$$

$$(app) : \frac{\mathcal{E} \vdash e : \tau \xrightarrow{\sigma''} \tau', \sigma \qquad \mathcal{E} \vdash e' : \tau, \sigma'}{\mathcal{E} \vdash (e\ e') : \tau', \sigma \cup \sigma' \cup \sigma''} \qquad (new) : \frac{\mathcal{E} \vdash e : \tau, \sigma}{\mathcal{E} \vdash (\mathtt{new}\ e) : ref_\rho(\tau), \sigma \cup init(\rho)}$$

$$(get) : \frac{\mathcal{E} \vdash e : ref_\rho(\tau), \sigma}{\mathcal{E} \vdash (\mathtt{get}\ e) : \tau, \sigma \cup read(\rho)} \qquad (set) : \frac{\mathcal{E} \vdash e : ref_\rho(\tau), \sigma \qquad \mathcal{E} \vdash e' : \tau, \sigma'}{\mathcal{E} \vdash (\mathtt{set}\ e\ e') : unit, \sigma \cup \sigma' \cup write(\rho)}$$

Sémantique Statique

Les effets sont construits sur les régions, une région ρ étant une constante ou une union ($\cup$) de régions. Cette structure algébrique d'ensemble est munie d'une relation d'ordre par inclusion; cette ordre correspond, dans le domaine des effets, au sous-typage dans celui des types. La règle *(does)* autorise cette forme limitée de coercion :

$$(does) : \frac{\mathcal{E} \vdash e : \tau, \sigma \qquad \sigma' \sqsupseteq \sigma}{\mathcal{E} \vdash e : \tau, \sigma'}$$

Cet ajout par rapport au système de reconstruction algébrique classique a deux avantages : (1) en permettant d'être plus souple dans le choix des effets des expressions, il autorise certains programmes à être typés correctement alors qu'ils seraient rejetés par la méthode précédente et (2) il permet de retrouver une structure de typage comportant un type principal, comme dans ML.

Un algorithme de reconstruction a été proposé pour la reconstruction des types, régions et effets; il est une extension de la reconstruction algébrique et est omis ici. Grâce à l'introduction de la règle de coercion implicite *(does)* et à l'absence de typage et polymorphisme explicites, les contraintes algébriques sont des inégalités entre effets. Ces contraintes sont alors toujours satisfiables. Il ne peut pas y avoir d'erreur d'*effetage* comme il y a des erreurs de typage. Par ailleurs, autre avantage, le type reconstruit par l'algorithme est principal au sens des substitutions syntaxiques tandis que l'effet reconstruit est minimal au sens de l'inclusion des effets. On retrouve les caractéristiques des systèmes à la ML. Ceci est également du à ce que, en évitant la notion d'union de régions utilisée dans FX-87, une unification à la Robinson est suffisante pour réaliser l'inférence d'effet et de région.

6.4 Consistance sémantique

Un des résultats majeurs du papier présenté ici est la preuve de consistance entre la sémantique statique à base d'effet et la sémantique dynamique; ceci n'avait pas été fait complètement formellement jusqu'à présent, [43] donnant un premier jeu de justifications. On voit bien dans cette preuve comment, de la même manière qu'un type est une abstraction d'un ensemble de valeurs, un effet est une abstraction d'un ensemble de traces, i.e. de comportements.

Ici, cette notion d'abstraction est traduite par la notion de consistance entre trace f et effet σ, et type τ et valeur v. Un *modèle* $\mathcal{S}$ d'une mémoire s lie une adresse mémoire à ses type et région correspondants. On peut alors définir la consistance entre effet et trace d'une part, et type et valeur, d'autre part :

Definition 1 *Une trace f est* consistante *avec un effet σ dans un modèle $\mathcal{S}$, noté $\mathcal{S} \models f : \sigma$, si et seulement si :*

$$\forall\ init(l) \in f,\ \mathcal{S}(l) = (\rho, \tau)\ \wedge\ init(\rho) \in \sigma$$
$$\forall\ read(l) \in f,\ \mathcal{S}(l) = (\rho, \tau)\ \wedge\ read(\rho) \in \sigma$$
$$\forall\ write(l) \in f,\ \mathcal{S}(l) = (\rho, \tau)\ \wedge\ write(\rho) \in \sigma$$

Definition 2 *Une valeur v est* consistante *avec un type τ, noté $s : \mathcal{S} \models v : \tau$, si et seulement si l'une des trois propriétés suivantes est vérifiée :*

$s : \mathcal{S} \models u : unit$
$s : \mathcal{S} \models l : ref_\rho(\tau) \Leftrightarrow \mathcal{S}(l) = (\rho, \tau)$ *and* $s : \mathcal{S} \models s(l) : \tau$
$s : \mathcal{S} \models \langle \mathtt{x}, \mathtt{e}, E \rangle : \tau \Leftrightarrow$ *there exists* $\mathcal{E}$ *and* $s : \mathcal{S} \models E : \mathcal{E}$ *and* $\mathcal{E} \vdash (\mathtt{lambda}\ (x)\ e) : \tau, \emptyset$

On note $s : \mathcal{S} \models E : \mathcal{E}$ si et seulement si $Dom(E) = Dom(\mathcal{E})$ et $s : \mathcal{S} \models E(\mathtt{x}) : \mathcal{E}(\mathtt{x})$ pour tout $\mathtt{x} \in Dom(E)$.

Comme indiqué par [57], cette relation de consistance ne définit pas une relation unique et doit être vue comme représentant une équation au point fixe dont la valeur maximale est celle recherchée. Ce point fixe maximal représente le plus grand ensemble de valeurs consistantes avec leur type, qui peut ne pas être approché par itération classique du fait de la présence de structures cycliques qui peuvent être construites par effet de bord.

Le théorème de consistance entre sémantiques statique et dynamique, dont la preuve est un des résultats majeurs du papier, est alors le suivant :

Theorem 3 *Soient s un état-mémoire et $\mathcal{S}$ un modèle consistant. Soient E un environnement dynamique et $\mathcal{E}$ son environnement de type associé tels que $s : \mathcal{S} \models E : \mathcal{E}$. Si $\mathcal{E} \vdash e : \tau, \sigma$ et $s, E \vdash e \rightarrow v, f, s'$, alors il existe un nouveau modèle $\mathcal{S}'$ étendant $\mathcal{S}$ tel que $\mathcal{S}' \models f : \sigma$ et $s' : \mathcal{S}' \models v : \tau v$.*

7 Effets de contrôle

L'article "Reasoning about Continuations with Control Effects" [35] résumé ci-dessous décrit une nouvelle application du cadre théorique des types et effets rapidement décrit dans les sections précédentes. Si la notion d'effet a été initialement introduite pour prendre

en compte les modifications-mémoire introduites par les opérations à effet de bord, il s'avère que ce cadre est plus général et peut être utilement mis en oeuvre pour intégrer d'autres traits de programmation non-fonctionnels. Le coeur de cet article étudie l'ajout de continuations à un langage fonctionnel. Ce travail de recherche a été effectué en collaboration avec le Professeur David K. Gifford et son groupe de recherche, le Programmming Systems Research Group, du Massachusetts Institute of Technology aux Etats-Unis.

7.1 Continuations

Si l'addition d'opérations impératives détruit l'intégrité référentielle propre aux langages fonctionnelles, il existe d'autres constructions, utiles pour le développement intensif de logiciel comme la gestion des exceptions et erreurs, qui ont le même inconvénient. En effet, dans une expression comme :

```
(+ (raise 1) (raise 2))
```

une évaluation classique (de la gauche vers la droite) terminera avec la valeur 1, alors qu'une stratégie de parcours évaluant dans le sens opposé retournera 2. L'ordre d'évaluation n'a pas d'influence dans un langage complètement fonctionnel.

Ces comportements non fonctionnels peuvent être dans leur grande majorité ramenés à des manipulations explicites de la continuation de l'évaluation d'une expression. Une *continuation* représente, en un point donné de l'évaluation d'une expression, le reste du processus de calcul à effectuer avant d'obtenir le résultat définitif. Cette notion a été initialement introduite par Wadsworth [53] pour préciser formellement la sémantique dénotationnelle des langages permettant d'effectuer des branchements. Le langage Scheme autorise la manipulation explicite de la continuation d'une expression via la procédure `call-with-current-continutation`, ou `cwcc`, qui évalue son argument, fonction unaire dont l'argument formel est une procédure représentant (on dit souvent *réifiant*) la continuation englobante. Un exemple classique d'utilisation de continuation est donné ci-dessous :

```
(define (successively-divide x divisors)
  (call-with-current-continuation
    (lambda (k)
      (divide x divisors k))))

(define (divide res divisors k)
  (cond ((null? divisors) res)
        ((zero? (car divisors))
         (k 0))
       (else (divide (/ res (car divisors)) (cdr divisors) k))))
```

Ici, `x` est divisé successivement par tous les éléments de la liste `divisors`. Si un des éléments est nul, l'entier `0` est retourné. La procédure `k` réifie la continuation attachée au point de retour de `successively-divide`; elle permet de sortir des appels emboités à `divide` en présence de cas exceptionnel.

Ce comportement non fonctionnel rend caduque les techniques classiques d'analyse de flot de contrôle. L'article présenté ci-dessous introduit une nouvelle méthode statique d'analyse du flot de contrôle en présence de continuations, de fonctions d'ordre supérieur

et de compilation séparée. En utilisant la notion de masquage d'effet (voir section 7.3), cette approche permet de détecter quelles expressions sont non fonctionnelles et de limiter leur influence; ceci autorise l'utilisation locale de continuation, leur caractère non-fonctionnel n'étant pas observable de l'extérieur. Dans l'exemple précédent, bien que la fonction `successivily-divide` utilise de manière interne une continuation, elle est complètement fonctionnelle pour l'appelant.

7.2 `Goto` et `Comefrom`

Il est aisé de voir que ni la fonction `cwcc` ni l'appel d'une continuation ne peuvent être considérée comme ayant un effet latent pur. Le programme Scheme suivant montre qu'il n'est pas possible d'éliminer des sous-expressions communes utilisant `cwcc`, comme l'autoriserait un effet pur :

```
(let* ((f1 (cwcc (lambda (x) x)))
       (x (horrible-effects))
       (f2 (cwcc (lambda (x) x))))
  ...)
```

En effet, un appel à `f1` a pour conséquence d'effectuer les mutations présentes dans la fonction `horrible-effects`; ces effets ne sont pas effectués si `f2` est appelé. Les appels à `cwcc` ne peuvent donc être purs.

Le second exemple montre comment, de la même manière, un appel de continuation cette fois peut avoir des effets différents :

```
(let ((x (cwcc (lambda (f)
                 (cwcc (lambda (g) (f g)))
                 (h)
                  f))))
   (horrible-effects)
   ...
   (x 0)
   ...)
```

Dans cet exemple, entre les évaluations de `(f g)` et `(h)`, la fonction `horrible-effects` sera exécutée; il n'est dont pas possible de réordonner ces expressions, ce qui serait possible si l'effet latent d'une continutation était `pure`.

L'utilisation des continuations correspond donc à deux nouveaux constructeurs d'effets dit de *contrôle*, appellés `goto` et `comefrom`, qui décrivent les propriétés de flot de contrôle des expressions. Une expression qui ne possède pas d'effet `goto` appelle toujours sa continuation (implicite) de manière fonctionnelle; une expression qui n'a pas d'effet `comefrom` ne préserve pas sa continuation de retour (réifiée) en vue d'une utilisation ultérieure. Ce style de définitions en double négation est nécessaire pour exprimer le caractère conservatif des effets comme approximation du comportement dynamique des programmes. Le papier complet contient une preuve de consistance entre analyse statique et sémantique dynamique à base de continuation.

Pour introduire la notion de continuation dans un langage muni d'un système d'effet, il suffit de rajouter dans l'environnment de typage la variable `cwcc` dont le type est donné ci-dessous :

```
cwcc: (poly (r region)
         (poly (t type)
           (poly (e effect)
              (subr (maxeff (comefrom r) e)
                    ((subr e
                           ((subr (goto r) (t) void))
                           t))
                    t))))
```

Le type de la fonction `cwcc` est abstrait, via le constructeur de type polymorphe `poly`, sur la région `r` dans laquelle la continuation est allouée, le type `t` du résultat éventuel et l'effet latent `e` de la fonction passée en argument. L'effet latent de `cwcc` combine les effets d'appel de son argument et de capture de la continuation. L'argument de `cwcc` est une fonction qui prend pour argument la continuation capturée, vue comme une fonction d'effet latent `(goto r)`. Le type `void` indique que l'appel d'une continuation ne retourne jamais dans l'appelé.

Les effets de contrôle ont de multiples applications, qui recouvrent en partie celles des effets mémoire étudiés précédemment :

- Ils permettent au programmeur de spécifier, dans une forme qui est vérifiable au moment de la compilation, le comportement attendu à l'exécution d'un programme donné. Ceci améliore la documentation, la modularité et la maintenance des programmes. La notion d'effet de contrôle est également un nouveau cadre permettant d'analyser les langages offrant des continuations *first-class*. De plus, quand ces effets sont masqués (voir section 7.3), le programmeur est assuré que l'expression concernée se comporte de manière fonctionnelle.

- Les effets de contrôle permettent au concepteur de langages de programmation de limiter l'utilisation des continuations en vue de simplifier leur sémantique. Ainsi, par exemple, en insistant pour que les définitions de fonctions n'aient pas d'effet `comefrom` ou `goto`, le problème de la redéfinition d'une variable par retour dans une forme `define` est évité, clarifiant ainsi la sémantique de la boucle `top-level` d'un interprète.

- L'implémenteur d'un compilateur peut utiliser les effets de contrôle pour effectuer des optimisations en présence de continuations explicites. Par exemple, si une expression de capture de continuation a un effet de contrôle masqué, alors la continuation sera utilisée seulement dans les fonctions appelées et donc celle-ci peut être simplement allouée sur la pile au lieu du tas.

7.3 Masquage d'effet

La notion de *masquage d'effet*, déjà introduite dans FX-87 [17], permet de détecter et éliminer certains effets d'une expression comme les mutations de structures de données locales qui ne sont pas observables d'une fonction appelante. Les conditions qui permettent de masquer un effet d'une expression `E` sont les suivantes :

- Si aucune variable libre de `E` n'utilise la région `r` dans son type, alors les effets `read` et `write` de `E` sur `r` peuvent être éliminés.

- Si la région `r` n'apparait pas dans le type de `E`, alors les effets `alloc` sur `r` peuvent être masqués

En effet, si une expression lit ou modifie une zone mémoire qui n'apparait pas dans le type de son environnement (ou variables libres), les conséquences de ces opérations ne pourront pas être constatées par la suite. De plus, les allocations sur des régions qui n'apparaissent pas dans le type du résultat peuvent être négligées car elles sont perdues.

Cette technique de masquage d'effet peut être étendue pour éliminer les effets de contrôle. Un effet `(goto r)` ou `(comefrom r)` peut être masqué si, ni les variables libres de `E`, ni `E` n'ont un type qui contient `r`. En effet, on peut voir un appel à `cwcc` comme allouant une zone mémoire pour stocker la valeur courante de la pile d'activation tandis qu'un appel de continuation revient à lire (pour obtenir la nouvelle pile) et écrire (pour remplacer la pile). Cette analogie n'est néanmoins pas tout à fait exacte puisque l'effet `goto` demande des conditions plus strictes qu'une simple allocation. Ceci peut être vu sur l'exemple "dantesque" suivant :

```
(let ((x (cwcc (lambda (f)
                 (let ((y (cons f f)))
                     (cwcc (lambda (g)
                             (set-cdr! y g)
                             (f y)))
                     ((car y) y))))))
    (cwcc (lambda (h)
            (set-car! x h)
            ((cdr x) x))))
```

Ici, en appelant `f`, on lie d'abord `x` à la paire `y` qui contient `f` et la continuation `g` qui précède l'évaluation de `((car y) y)`. Le premier élément de `x` est modifié avec `h` qui correspond au reste du programme et l'expression correspondant à la liaison de `x` est évaluée à nouveau au moment où `((car y) y)` est évalué; cette évaluation est donc non fonctionnelle. Bien qu'il n'y ait pas de variables libres dans l'expression à laquelle `x` est lié, l'effet `goto` de l'appel `((car y) y)` ne peut être masqué; les continuations permettent d'exporter et modifier temporairement des structures locales, ce qui justifie la vérification supplémentaire sur les régions apparaissant dans le type du résultat. Le papier complet présente une preuve de la correction de cette optimisation par rapport à la sémantique dynamique.

En pratique, le masquage d'effet de contrôle permet d'optimiser l'implémentation des continuations. Il est possible de reconnaître quand un mécanisme d'allocation en pile suffit pour conserver une continuation, auquel cas `cwcc` peut être remplacé par un simple `catch/throw` plus efficace. Egalement, si les effets de contrôle sont accessibles au moment de l'évaluation, les seuls points d'activation qui doivent être sauvegardés dans le tas au moment de la capture d'une continuation dans une région `r` sont ceux qui possèdent un effet `(comefrom r)` et ce jusqu'au point où ces effets sont masqués; le reste peut rester en pile. Cette optimisation est particulièrement utile dans une implémentation parallèle car elle évite le partage de pile entre différents processus.

8 Complexité des programmes

L'article "Polymorphic Time Systems for Estimating Program Complexity" [13] résumé ci-dessous décrit une nouvelle application du cadre théorique des types et effets rapidement décrit dans les sections précédentes (sections 5 à 7). Si la notion d'effet a été définie pour intégrer les deux traits non-fonctionnels de programmation que sont les effets de bord et les continuations, elle est également capable de décrire d'autres types d'informations concernant le comportement d'un programme. Cet article montre comment l'analyse de complexité de programme peut être avantageusement effectuée avec un système d'effet. Ce travail de recherche a été effectué en collaboration avec Vincent Dornic, ancien étudiant en thèse au Centre de Recherche en Informatique de l'Ecole des Mines de Paris et à l'Université Paris VI, et le Professeur David K. Gifford et son groupe de recherche, le Programmming Systems Research Group, du Massachusetts Institute of Technology aux Etats-Unis.

8.1 Complexité

De nombreuses techniques d'analyse, telles que celles citées dans Knuth, existent pour déterminer les complexités temporelles ou spatiales des algorithmes dans le meilleur cas, le cas le pire ou le cas moyen [15]. Ces méthodes sophistiquées sont rarement automatiques quoique certains systèmes [42] aient été proposés pour obtenir, pour certaines classes de langages restreintes, ces informations au moment de la compilation.

Il est cependant utile de disposer d'approximations de complexité pour décider, par exemple, de l'optimisation de certaines parties d'un programme, les plus utilisées, permettant d'obtenir le meilleur rendement, ou de faire la distinction entre parallélisme maximal et parallélisme utile. Contrairement aux systèmes précédents comme [15], l'objectif visé dans cet article consiste à privilégier la prise en compte d'un langage de programmation riche et complet, permettant la programmation d'ordre supérieur et offrant des primitives d'effets de bord, quitte à simplifier les informations de complexité automatiquement obtenues.

Un *système de temps* est une nouvelle technique d'analyse statique des programmes qui permet de déterminer une estimation du temps d'exécution de toute expression d'un programme. Ce système de temps s'inscrit dans le cadre général des systèmes d'effets, présenté rapidement dans la section 5, et utilise une méthodologie de typage non-standard pour propager les informations de temps; le *temps latent* d'une fonction communique la complexité d'une fonction de son point de définition à ceux de son utilisation. Le temps d'une expression est soit une borne supérieure entière du nombre de tops d'horloge qu'une expression met à s'exécuter, soit la valeur distinguée `long` qui indique que l'expression peut contenir une boucle et peut alors prendre un temps arbitraire pour s'exécuter.

Bien que cette taxinomie puisse paraître simpliste, il a été montré que même une information fruste peut être utilisée en pratique dans le domaine du parallélisme, que ce soit MIMD (détection du code de grain suffisament gros [19, 21]) ou SIMD (les fonctions de temps constant sont plus intéressantes à appliquer sur un vecteur). Dans des compilateurs optimisants qui utilisent l'évaluation partielle pour spécialiser certaines modules d'un programme [7], il est impératif de connaitre les expressions dont la terminaison est assurée, sous peine de ne pas voir le processus de compilation terminer.

8.2 Système de temps

Le langage manipulé ici est un lambda calcul explicitement typé ([12] montre comment intégrer un algorithme de reconstruction à ce système plus simple, car explicite) et muni d'un système d'effets. L'article complet introduit également un polymorphisme de temps, lequel est nécessaire pour bien traiter les fonctionnelles d'ordre supérieure telles que `map`. La différence majeure avec les systèmes précédents concerne le domaine des effets, ici les temps :

$$
\begin{array}{lll}
m & \in & \text{Time} \\
m & ::= & \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots \mid \texttt{long} \\
 & & (\texttt{sumtime}\ m\ m) \qquad \textit{Accumulation des temps} \\
 & & i
\end{array}
$$

et leurs propriétés algébriques :

$$
\begin{array}{rcl}
(\texttt{sumtime}\ m_1\ (\texttt{sumtime}\ m_2\ m_3)) & \sim & (\texttt{sumtime}\ (\texttt{sumtime}\ m_1\ m_2)\ m_3) \\
(\texttt{sumtime}\ m_1\ m_2) & \sim & (\texttt{sumtime}\ m_2\ m_1) \\
(\texttt{sumtime}\ m\ \texttt{long}) & \sim & \texttt{long} \\
 & & \\
(\texttt{sumtime}\ m_1\ m_2) & \sim & m_1 + m_2\ \textit{ssi}\ m_i \neq \texttt{long}
\end{array}
$$

Un important aspect de cette algèbre est indiqué par l'équation $m = (\texttt{sumtime}\ m\ m_0)$ pour laquelle la seule solution pour m est `long` pour toute valeur de m_0. Une équation de cette forme est obtenue pour toute définition récursive de fonction. Cette mesure de temps `long` apparait ainsi comme une première approche, très conservative, vers des informations plus riches concernant les constructions de boucle; on peut espérer obtenir, à terme, des informations telles que *polynomial* ou *exponentiel* par rapport à la taille des arguments.

Les règles de typage sont simples, montrant ainsi la puissance du cadre des systèmes d'effets à intégrer d'autres types d'analyse. Ne sont données ci-dessous que les règles les plus intéressantes, concernant l'abstraction et l'application :

$$
\frac{\begin{array}{c}\text{TK}[i : t_1] \vdash e : t_0\ \$\ m \\ \text{TK} \vdash t_1 :: \texttt{type}\end{array}}{\text{TK} \vdash (\texttt{lambda}\ (i\ \ t_1)\ \ e) : (\texttt{subr}\ m\ (t_1)\ \ t_0)\ \$\ \texttt{1}}\ [\text{S.Lambda}]
$$

$$
\frac{\begin{array}{l}\text{TK} \vdash e_0 : (\texttt{subr}\ m_l\ \ (t_1)\ \ t_0)\ \$\ m_0 \\ \text{TK} \vdash e_1 : t_1\ \$\ m_1\end{array}}{\text{TK} \vdash (e_0\ \ e_1) : t_0\ \$\ (\texttt{sumtime}\ (\texttt{sumtime}\ m_0\ m_1)\ (\texttt{sumtime}\ m_l\ \texttt{1}))}\ [\text{S.Apply}]
$$

Règle de types et temps

Il est remarquable de constater qu'aucune primitive de récursion n'a besoin d'être définie. L'exemple donné ci-dessous montre comment définir l'opérateur de point-fixe `Y` directement.

8.3 Consistance

La preuve de consistance entre sémantique standard et statique nécessite d'instrumenter la dynamique du langage avec des informations de temps. Toute expression e s'évalue dans un état-mémoire St en une valeur v et un temps n qui représente le nombre d'étapes d'évaluation nécessaires pour obtenir cette valeur. Les règles sont simples et les trois plus importantes sont présentées ci-dessous.

$$\frac{[i \leftarrow v] \sqsubseteq \mathrm{St}}{\mathrm{St} \vdash i \rightarrow v \ , \ \mathtt{1}} \ [\text{D.Env}]$$

$$\mathrm{St} \vdash \mathtt{(lambda\ (}i\ \ t\mathtt{)}\ \ e\mathtt{)} \rightarrow \langle i, e, \mathrm{St} \rangle, \mathtt{1} \qquad [\text{D.Lambda}]$$

$$\frac{\begin{array}{c}\mathrm{St} \vdash e_0 \rightarrow \langle i, e', \mathrm{St}' \rangle \ , \ n_0 \\ \mathrm{St} \vdash e_1 \rightarrow v_1 \ , \ n_1 \\ \mathrm{St}'[i \leftarrow v_1] \vdash e' \rightarrow v \ , \ n\end{array}}{\mathrm{St} \vdash \mathtt{(}e_0\ \ e_1\mathtt{)} \rightarrow v \ , \ 1 + n + n_0 + n_1} \ [\text{D.Apply}]$$

Evaluation

A noter que le choix de la constante `1` pourrait être raffiné pour prendre en compte plus précisement une architecture donnée.

Le théorème de consistance permet de s'assurer que (1) types et valeurs produites sont en accord et que (2) le temps indiqué par la sémantique statique est effectivement une borne de l'évaluation. La consistance entre type et valeur est définie inductivement comme suit :

Definition 3 *La* consistance *entre une valeur v et un type t est définie par la relation ternaire suivante :*

$$\models v : t \iff$$
$$\textit{si} \quad t \sim \textit{bool} \quad \textit{alors} \quad v \in \textit{Bool}$$
$$\textit{si} \quad t \sim \texttt{(subr m (}t_1\texttt{) }t_0\texttt{)} \quad \textit{alors} \quad v = \langle i, e, St \rangle \ \textit{et}$$
$$\exists\, TK \ \textit{s.t.} \begin{cases} St \ : TK \\ TK[i : t_1] \ \vdash \ e : t_0 \ \$ \ m \end{cases}$$

Le théorème de consistance, dont la preuve se trouve dans [12], s'applique uniquement aux expressions qui terminent.

Theorem 4 (Consistance)

$$\left.\begin{array}{l} St \vdash e \rightarrow v \ , \ n \\ TK \vdash e : t \ \$ \ m \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} v : t \\ n \leq m \end{array}\right.$$

L'article complet présente également un algorithme de vérification de type, dont la correction par rapport à la sémantique statique est également prouvée. Du fait qu'il ne s'agit pas ici de traiter le cas de la reconstruction des types et temps (voir [12]), il n'est pas nécessaire d'introduire d'algorithme d'unification équationnel. Un simple algorithme de normalisation permet de comparer les expressions de temps entre elles.

8.4 L'opérateur de point fixe

L'approche évoquée ci-dessus permet de représenter directement les fonctions récursives via l'opérateur de point fixe `Y`, qui peut être écrit dans le langage. L'article complet montre, en effet, comment ajouter la notion de type récursif nécessaire à ce style de fonction. Quand l'opérateur `Y` est appliqué à une fonction potentiellement récursive, il retourne une fonction de temps latent `long`. Par contre, si la fonction initiale n'est pas récursive, `Y` retourne une fonction de même temps latent. L'intégration de `Y` se fait sans perte de précision. Dans un langage non-typé avec appel par valeur, `Y` est défini comme :

$$\texttt{Y} \equiv \lambda f.(\lambda x.(f\ \lambda\ ().(x\ x))\ \lambda x.(f\ \lambda\ ().(x\ x)))$$

Il est alors représentable dans un langage avec système de temps comme :

```
Y ≡ (plambda ((t type)(m1 time)(m2 time))
       (lambda ((f (subr m1
                         ((subr (sumtime m1 6) () (subr m2 (t) t)))
                         (subr m2 (t) t))))
          ((lambda ((x (drec tx
                              (subr (sumtime m1 3) (tx) (subr m2 (t) t)))))
              (f (lambda () (x x))))
           (lambda ((x (drec tx
                              (subr (sumtime m1 3) (tx) (subr m2 (t) t)))))
              (f (lambda () (x x)))))))
```

où l'on peut noter que le type de `x` est récursif (puisqu'il est auto-appliqué). Après auto-application, le type récursif disparait. Cette forme générale utilise le polymorphisme explicite (`plambda`) qui est présenté dans le papier complet.

9 Perspectives, en guise de conclusion

Cette dernière section reprend l'ensemble des travaux évoqués ci-dessus dans une perspective historique, et plus personnelle, et esquisse certains développements futurs.

9.1 De la compilation à la parallélisation

Avant mon travail de thèse, je m'étais intéressé aux diverses analyses statiques des programmes et, plus précisément, aux questions de compilation. Les aspects à la fois théoriques, comme les définitions sémantiques ou les techniques d'analyse syntaxique, et pratiques, vu à travers la génération de code assembleur ou l'écriture de compilateurs, ont toujours intéressé le physicien que j'étais alors.

Mon stage de DEA à Paris VI a consisté en une adaptation du Portable C Compiler de Steve Johnson, des Bell Labs, pour un co-processeur flottant développé simultanément par un autre étudiant du DEA. La connaissance profonde d'un tel outil m'a été fort utile quand on m'a demandé de participer au projet Isis de développement d'un superordinateur vectoriel au sein du groupe Bull. J'ai été chargé de mettre au point le premier compilateur C; celui-ci fut utilisé avec succès tout au long des phases de développement du logiciel système de cette machine.

Alors que je poursuivais, indépendamment, mes investigations sur les sujets informatiques qui m'étaient alors moins familiers, en l'occurrence les diverses méthodes de spécification des sémantiques de langages de programmation, ce fut avec intérêt que j'acceptais d'intégrer le groupe naissant du professeur Paul Feautrier à Paris VI. Il me suggérait en effet d'étudier les relations possibles entre les techniques de parallélisation automatique de programmes et les méthodes dénotationnelles.

Mon travail de thèse (voir [31] et [33]) a consisté à montrer que l'approche dénotationnelle non-standard, telle que développée par Fleming Nielson à partir des travaux de Patrick Cousot, pouvait être adaptée à la parallélisation de programmes impératifs et permettait, dans un cadre théorique propre et puissant, de développer des méthodes sophistiquées de parallélisation. J'ai étudié trois méthodes de parallélisation intégrant des techniques d'analyse sémantique : l'utilisation d'assertions pour affiner le calcul du graphe de dépendances, la détection des réductions et l'analyse des programmes comportant des indirections sur tableaux. Un aspect incident de mon travail, mais qui se révèlera crucial par la suite, a été l'implémentation d'un prototype de *parallèliseur sémantique* en ML.

Mon travail de thèse a motivé, après ma soutenance, mon détachement de la Météorologie Nationale, où je préparais ma thèse à temps partiel, vers l'Ecole des Mines de Paris. J'y ai rejoint Rémi Triolet et Francois Irigoin pour travailler sur le paralléliseur PIPS [25]. Ce travail s'est fait en concomittance avec mon activité de consultant dans le Centre de Recherche Bull où j'ai pu poursuivre certains des travaux entamés lors de ma thèse, Bull s'efforçant alors de développer son propre paralléliseur. Cette collaboration été également la source de travaux de recherches importants [32]. Cette activité de recherche se poursuit activement encore aujourd'hui, en particulier sur certains aspects théoriques des analyses *dataflow* utilisées dans PIPS ou la mise au point de méthodes d'analyse symbolique encore plus sophistiquées (voir [10]).

9.2 De la parallélisation aux systèmes de typage

Avant de rejoindre l'Ecole des Mines, je souhaitais effectuer un séjour postdoctoral au MIT pour donner à ma recherche une coloration internationale qui me semblait cruciale. Après maintes prises de contacts, je rejoignais le groupe du professeur David K. Gifford qui travaillait sur les problèmes de parallélisation de langages ayant des composantes à la fois impérative et fonctionnelle (Lisp est né au MIT). Si la parallélisation des programmes impératifs est un sujet difficile, y rajouter la prise en compte de fonctions d'ordre supérieur le rend encore plus ardu. L'approche qui a été utilisée consiste à encoder à l'intérieur des types à la ML du langage des informations sur le comportement dynamique des programmes permettant, à la compilation et en présence de telles fonctions, de détecter le parallélisme. J'ai participé, lors de mon postdoc, à la définition du langage FX qui intègre un tel formalisme dit *système d'effets* et ai proposé de poursuivre cette collaboration fructeuse, ce qui fut accepté aussi bien à l'Ecole des Mines qu'au MIT. Cet échange se situe également dans le cadre de la Convention MIT-Grandes Ecoles.

A mon retour à l'Ecole des Mines, en parallèle avec mon travail sur la parallélisation impérative, j'ai démarré une activité de recherche ciblée sur l'analyse des langages intégrant des spécificités fonctionnelles. J'ai décidé de cibler cette recherche sur l'inférence de types et d'effets pour des langages de la famille de Standard ML, sous-classe du langage FX. Mon groupe a été à l'origine d'un certain nombre de résultats significatifs au cours des dernières années [12, 13, 54, 55, 56] et ce, à partir d'un résultat majeur présenté dans [37]. Dans cet article, je montre comment les techniques d'unification algébrique peuvent être utilisées pour réaliser l'inférence des effets dans les langages fonctionnels polymorphes. Par la suite, des aspects aussi bien théoriques que pratiques ont été abordés, en particulier via un contrat qui lie le MIT et l'Ecole des Mines pour l'étude de la génération de code pour machines massivement parallèles du type Connection Machine 2. Ce contrat s'est achevé officiellement durant l'été 1992 et une publication est en cours de préparation.

Les travaux se poursuivent en essayant toujours d'intégrer ces résultats dans le cadre des travaux sur Standard ML et d'accélérer leur adoption dans les langages de programmation et les compilateurs optimisants.

9.3 Des systèmes de typage au génie logiciel

De manière assez inattendue, mes travaux sur la parallélisation des langages impératifs et le typage des langages fonctionnels se rejoignent sur un nouveau point en cours de maturation. Le développement d'un logiciel aussi complexe que PIPS a demandé la mise au point d'un outil logiciel permettant de faciliter la manipulation de structures de données codées en C ou CommonLISP. A partir de déclarations de types de données concrets, cet outil, NewGen [36], génère automatiquement des bibliothèques de programmes adaptés à la manipulation de valeurs de ces types. J'envisage, avec un étudiant, d'étendre cette approche pour prendre en compte d'autres genres d'informations, comme par exemple des contraintes sémantiques sur les types, en vue d'étendre les champs d'application de cet outil.

References

[1] Allen, R. and Kennedy K. Automatic Translation of FORTRAN Programs to Vector Form. *ACM TOPLAS,* (October 1987).

[2] Ancourt, C. *Génération Automatique de Codes de Transfert pour Multiprocesseurs à Mémoires Locales Thèse de l'Université Paris VI,* (1991).

[3] Bernstein, A. J. Analysis of Programs for Parallel Processing. *IEEE Trans. on Elec. Comp., vol. 15,* (Octobre 1966), 757-763.

[4] Callahan, D., Dongarra, J. and Levine, D. Vectorizing Compilers : A Test Suite and Results. *IEEE Supercomputing'88, Florida,* (Novembre 1988).

[5] Chailloux, J. & al. *Le_lisp Version 15.2. INRIA Tech. Rep.* (Mai 1986).

[6] Clarke, A. L. and Richardson, D. J. Symbolic Evaluation Methods : Implementations and Applications *In Computer Program Testing, North-Holland,* (1981).

[7] Consel, C. Binding time analysis for higher-order untyped functional languages. *ACM LFP'90, Nice,* (June 1990), 264-273.

[8] Cousot, P. and Cousot, R. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction of Approximations of Fixpoints. *ACM POPL'77,* (Janvier 1977), pp 238-252.

[9] Cousot, P. and Halbwachs, N. Automatic Discovery of Linear Restraints among Variables of a Program. *ACM POPL'78,* (Janvier 1978), 84-96.

[10] Dehbonei, B. and Jouvelot, P. Semantical Interprocedural Analysis by Partial Symbolic Evaluation. *ACM PEPM'92, San Francisco,* (Juin 1992).

[11] Dehbonei, B. and Memmi, G. Velour : A New Vectorizing Compiler Prototype. *ACM ICS'88, Boston,* (Mai 1988).

[12] Dornic, V. Analyse de complexité des algorithmes : vérification et inférence. *PhD thesis, Ecole des Mines de Paris,* (Juin 1992).

[13] Dornic, V., Gifford, D. K. and Jouvelot, P. Polymorphic Time Systems for Estimating Program Complexity. *ACM LOPLAS, vol. 1, no. 1,* (1992).

[14] Dowling, W. F., and Gallier, J. H. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Logic Programming (3),* (1984).

[15] Flajolet, P. and Vitter, J. S. Average-case analysis of algorithms and data structures. *Research report INRIA 718,* (August 1987).

[16] Gifford, D. K. and Lucassen, J. M. Integrating Functional and Imperative Programming. *ACM LFP'86, Cambridge,* (Août 1986).

[17] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. *The FX-87 reference manual. Research Report MIT/LCS/TR-407,* (1987).

[18] Gifford, D. K., Jouvelot, P., Sheldon, M. A. and O'Toole, J. W. *Report on the FX-91 Programming Language. Research Report MIT/LCS/TR-531,* (Février 1992).

[19] Goldberg, F. B. *Multiprocessor execution of functional programs. Research Report, Yale, DCS/RR-618,* (April 1988).

[20] Gordon, M. J. C. and Milner, R. *Edinburgh LCF.* Lect. Note in Comp. Sci. 78, Springer Verlag, (1979).

[21] Gray, S. L. Using futures to exploit parallelism in Lisp. *MIT S.B. Master Thesis,* (1983).

[22] Harrison, W. L., III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symb. Comp., 2,* 3/4, (Octobre 1989), 179-396.

[23] Hickey, T. and Cohen, J. Automating program analysis. *JACM 35,* 1 (Janvier 1988), 185-220.

[24] Hammel, R. T. and Gifford, D. K. *FX-87 Performance Measurements : Dataflow Implementation. Massachusetts Institute of Technology, LCS/TR-421,* (1988).

[25] Irigoin, F., Jouvelot, P. and Triolet, R. Semantical Interprocedural Parallelization : An Overview of the PIPS Project. *ACM Sigarch ICS'91, Cologne,* (Juin 1991).

[26] Irigoin, F. and Triolet, R. Supernode Partitioning. *ACM POPL'88, San-Diego,* (Janvier 1988).

[27] Jouvelot, P. ML : Un Langage de Maquettage. *Journées d'étude "Nouveaux Langages pour le Génie Logiciel", AFCET,* (1985).

[28] Jouvelot, P. Designing New Languages and New Language Manipulation Systems using ML. *ACM SIGPLAN Notices, vol. 21,* (Août 1986), 40-52.

[29] Jouvelot, P. Parallelization by Semantic Detection of Reductions. *ESOP86, Lect. Note in Comp. Sci. 213, Springer Verlag,* (Mars 1986), 223-236.

[30] Jouvelot, P. Semantic Parallelization : A Practical Exercise in Abstract Interpretation *ACM POPL'87, Munich,* (Janvier 1987).

[31] Jouvelot P. *Parallélisation Sémantique : Une Approche Dénotationnelle Non-Standard pour la Parallélisation de Programmes Séquentiels* Thèse de l'Université Paris VI, Rapport MASI 174, (Février 1987).

[32] Jouvelot, P. and Dehbonei, B. A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions. *ACM Sigarch ICS'89, Crete,* (Juin 1989).

[33] Jouvelot, P. and Feautrier, P. Parallélisation Sémantique. *ITA, vol. 24, no. 2,* (1990).

[34] Jouvelot, P. and Gifford, D. K. The FX-87 Interpreter. *IEEE ICCL'88, Miami Beach,* (October 1988).

[35] Jouvelot, P. and Gifford, D. K. Reasoning about Continuations with Control Effects. *ACM PLDI'89, Portland,* (Juin 1989).

[36] Jouvelot, P. and Triolet, R. *NewGen : A Language Independent Program Generator. Rapport Interne CAII 191,* (1989).

[37] Jouvelot, P. and Gifford, D. K. Algebraic reconstruction of types and effects. *ACM POPL'91, Orlando,* (January 1991).

[38] Kapur, D., and Narendran, P. NP-Completeness of the Set Unification and Matching Problems. *8th Inter. Conference on Automated Deduction, LNCS 230, Springer-Verlag,* (1986).

[39] Keech, M. S. *Test Loops for Fortran 200 Vectorize. FSUCC Tech. Rep. 2,* (Avril 1988).

[40] Kozen, D. Semantics of probabilistic programs. *J. Comput. Syst. Sci. 22* (1981), 328-350.

[41] Lamport, L. The Parallel Execution of DO Loops. *Communications of the ACM,* (1974).

[42] Le Métayer, D. ACE : An automatic complexity evaluator. *ACM TOPLAS 10,* 2 (April 1988), 248-266.

[43] Lucassen, J. M. Types and effects. towards the integration of functional and imperative programming. *PhD dissertation, MIT-LCS,* (September 1987).

[44] Lucassen, J. M. and Gifford, D. K. Polymorphic effect systems. *Principles on Programming Languages, ACM PoPL'88 proceedings, San Diego,* (January 1988).

[45] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences, vol. 17,* (1978), 349-375.

[46] McCracken, N. J. *An Investigation of a Programming Language with a Polymorphic Type Structure. PhD Thesis, Syracuse University,* (Juin 1979).

[47] Nielson, F. *Program Transformations in a Denotational Setting* ACM TOPLAS, vol. 7, (Juillet 1985), 359-379.

[48] Oppen, D. C. A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetics. *JCSS, vol. 16,* (1978), 323-332.

[49] Plotkin, G. *A structural approach to operational semantics. Technical report DAIMI-FN-19, Aarhus University,* (1981).

[50] Rees, J. *et al. Revised*^{3} *Report on the Algorithmic Language Scheme. AI Memo 848a, MIT Artificial Intelligence Laboratory,* (Septembre 1986).

[51] Robinson, J. A. A machine-oriented logic based on the resolution principle. *JACM, vol. 12, no. 1,* (1965), 23-41.

[52] Scott, D. The Lattice of Flow Diagrams. *Symp. on Semantics of Algorithmic Lang., Springer Verlag,* (1972), 311-366.

[53] Stoy J. E. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory.* MIT Press, (1977).

[54] Talpin, J.-P. and Jouvelot, P. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming, vol. 2, no. 2,* (1992).

[55] Talpin, J. P. and Jouvelot, P. The Type and Effect Discipline. *IEEE LICS'92, Santa Cruz,* (Juin 1992).

[56] Tang, Y-M., and Jouvelot, P. *Control-Flow Effects for Escape Analysis. Workshop on Static Analysis, Bordeaux,* (Octobre 1992).

[57] Tofte, M. Operational semantics and polymorphic type inference. *Univ. of Edinburgh, thesis CST-52-88,* (1988).

[58] Triolet, R. *Contribution à la parallélisation automatique de programmes FORTRAN comportant des appels de procédures.* Thèse de Docteur-Ingénieur, Université Paris VI, (Décembre 1984).

[59] Triolet, R., Irigoin, F., and Feautrier, P. Direct Parallelization of Call Statements. *ACM Symposium on Compiler Construction,* (1986).

Contents