Rapport INRIA 1994 — Programme 2

Langages déclaratifs

Projet LANDE

3 mai 1995

Projet LANDE

Langages déclaratifs

Localisation : *Rennes*

Mots-clés : analyse de programme (1), compilation (1), environnement de programmation (1), évaluation partielle (1), génie logiciel (1), gestion de mémoire (1), Lambda-Prolog (1), LANDE (1), langage fonctionnel (1), programmation logique (1), ramasse-miettes (1), transformation de programme (1), typage (1).

Lande est un projet commun Inria/CNRS (URA 227).

1 Composition de l'équipe

Responsable scientifique

Daniel Le Métayer, DR Inria

Secrétaire

Marie-Noëlle Georgeault, SAR Inria

Personnel Inria

Pascal Brisset, CR, jusqu'en octobre 1994
Pascal Fradet, CR
Florimond Ployette, IR, à temps partagé avec l'atelier
Olivier Ridoux, CR

Personnel Ura 227

Yves Bekkers, professeur, université de Rennes 1
Charles Consel, professeur, université de Rennes 1
Mireille Ducassé, professeur associé, Insa de Rennes

Lucien Ungaro, maître de conférences, université de Rennes 1

Chercheurs doctorants

Christophe Bonnet, bourse MESR
Rémi Douence, bourse MESR
Ronan Gaugne, bourse Inria-Région, à partir d'octobre 1994
Valérie Gouranton, bourse MESR
Luke Hornof, bourse NSF
Pascale Louvet, bourse MESR
Julien Mallet, bourse MESR, à partir d'octobre 1994
Barbara Moura, bourse MESR
Jacques Noyé
Igor Stéphan, bourse MESR
Lionel Van Aertryck, bourse Cifre
Eugen-Nicolae Volanschi, bourse Inria, à partir d'octobre 1994

Collaborateurs extérieurs

Patrice Boizumault, professeur, IMA Angers, ENM Nantes depuis octobre 1994

Chercheurs invités

David Burke, USA, bourse NSF
Flemming Damm, Danemark, bourse HCM jusqu'en juin 1994
Paul Tarau, Canada, juin-juillet 1994
Tommy Thorn, Danemark, à partir de juillet 1994

2 Présentation générale et objectifs

2.1 Genèse du projet

Le projet Lande existe officiellement depuis février 1994. Il réunit les membres de l'ancien projet Mali et la partie langages de haut niveau du projet LSP (maintenant Solidor). Il attire de plus un certain nombre de nouveaux chercheurs en provenance d'Oregon Graduate Institute (Charles Consel et quelques étudiants) et de l'ECRC (Mireille Ducassé et Jacques Noyé). La spécialité de la composante Mali est la mise en œuvre des langages de programmation logique, avec un accent particulier sur la gestion de mémoire. Cet axe est renforcé par l'arrivée des chercheurs de l'ECRC. Les langages de haut niveau étudiés par le groupe issu de LSP

sont essentiellement les langages fonctionnels et le formalisme Gamma qui a vu le jour dans ce projet. Charles Consel est un spécialiste de l'évaluation partielle, une technique de transformation automatique qui s'applique à différents modèles de programmation.

Il nous a paru naturel d'unir nos forces pour créer un projet commun ayant pour thème les langages déclaratifs. En effet, les modèles logique et fonctionnel, même s'ils se sont développés par le passé dans des communautés disjointes, impliquent un point de vue particulier sur la programmation et partagent bon nombre de problématiques et de techniques. Par ailleurs, l'évaluation partielle s'est surtout développée récemment dans le contexte des langages déclaratifs (aussi bien fonctionnels que logiques). Cette proximité thématique entre les différents composants du projet se double d'une similitude dans la démarche, chacun s'efforçant de concilier approche formelle et développements réalistes. Ces raisons nous font penser que le projet nouvellement créé est en mesure d'apporter, par sa nature pluri-culturelle, une valeur ajoutée capable de donner de nouvelles perspectives aux travaux sur les langages de haut niveau à l'Irisa. Le défi qui se présente à nous est de faire en sorte que le projet soit plus que la réunion de personnes continuant à poursuivre des objectifs de recherche indépendants.

2.2 Thèmes de recherche

Les langages déclaratifs offrent au programmeur un pouvoir d'expression très appréciable tout en reposant sur des bases mathématiques bien établies. C'est cette situation originale aux confins de la théorie et de la pratique qui nous intéresse dans le projet Lande. Le thème général de nos recherches est le développement de techniques et d'outils fondés formellement.

Nous avons notamment exploré différentes techniques de transformation de programmes. On peut citer à cet égard deux activités majeures:

- La description de la compilation des langages fonctionnels comme une succession de transformations de programmes. Ce travail s'est concrétisé dans le compilateur Tabac qui produit du code efficace et dont la correction est prouvée. Il sert maintenant de plate-forme pour expérimenter de nouvelles techniques d'analyse et d'optimisation de programmes.

- L'étude des principes et des applications de l'évaluation partielle. L'évaluation partielle est une technique qui permet de dériver automatiquement des versions spécialisées et efficaces de programmes génériques. Ces recherches ont mené au développement du système d'évaluation partielle Schism qui a été utilisé avec succès dans diverses applications, comme la conception d'environnements de programmation et l'optimisation de programmes.

L'analyse de programmes est également un thème d'activité primordial dans le projet : plusieurs techniques ont été proposées et intégrées aux outils cités plus haut. Elles reposent sur une forme d'inférence de types. Dans cette activité comme dans les précédentes nous avons cherché à allier les aspects formels (preuves de correction des techniques et des optimisations qui les utilisent) et pratiques (conceptions d'analyseurs d'une efficacité raisonnable et intégrés au compilateur).

Nos travaux sur l'implantation des langages de programmation logique se distinguent par la prépondérance accordée aux problèmes de gestion de mémoire. Nous avons ainsi proposé une méthode de gestion de mémoire adaptée aux langages indéterministes (Mali) dont différentes versions (logicielles et matérielles) ont été implantées. Mali a été utilisée pour la mise en œuvre de plusieurs langages de programmation logique et nos efforts récents ont porté sur l'implantation du langage d'ordre supérieur λProlog.

Nous poursuivons aussi une activité inspirée du formalisme Gamma que nous avons proposé il y a quelques années et qui a depuis servi de base à différentes recherches. Gamma permet une description abstraite des programmes, sans contraintes d'ordonnancement inutiles, ce qui facilite le raisonnement sur les programmes et leur implantation sur des machines parallèles.

3 Actions de recherche

Nous distinguons dans nos activités les travaux centrés sur les langages déclaratifs (langages fonctionnels, programmation logique, Gamma) et ceux qui concernent des techniques (analyse de programmes, évaluation partielle). Ces dernières ont surtout été étudiées dans le cadre des langages déclaratifs dans le passé mais nous considérons maintenant leur généralisation et l'application à d'autres types de langages.

3.1 Langages fonctionnels

Notre centre d'intérêt premier sur ce thème est la conception de techniques fondées pour la mise en œuvre efficace des langages fonctionnels.

3.1.1 Réduction forte

Participants : Pascal Fradet

Le compilateur Tabac repose sur une transformation de programmes qui fait apparaître des fonctions, dites de continuation, permettant de décrire le contrôle de manière fonctionnelle. Mais Tabac, comme tous les compilateurs de langages fonctionnels, met en œuvre la réduction faible : il ne réduit pas les expressions situées dans le corps d'une fonction. Il est des applications pour lesquelles on aimerait lever cette restriction (évaluation symbolique, compilation de langages logiques d'ordre supérieur comme λProlog). Nous avons montré que la transformation en continuations peut être utilisée pour compiler ce style de réduction forte [12]. Une expression transformée est appliquée à une continuation particulière et sa forme normale (forte ou de tête) est évaluée par l'habituelle réduction faible. Ainsi n'avons pas à concevoir de machine abstraite spécialisée et cette technique permet à un compilateur standard de simplifier des fonctions.

3.1.2 Glaneur de cellules étendu

Participants : Pascal Fradet

Un des problèmes majeurs des langages fonctionnels est leur consommation prohibitive de mémoire. En particulier, de nombreux programmes fonctionnels comportent des fuites de mémoire : des références sont conservées sur des structures devenues inutiles. Les glaneurs de cellules (GC) classiques ne peuvent récupérer ces structures et de tels programmes nécessitent énormément d'espace ou épuisent la mémoire avant de pouvoir rendre leur résultat.

Nous avons proposé une méthode pour récupérer plus de mémoire que les GC traditionnels [11]. Cette technique est basée sur une propriété vérifiée par les langages polymorphes : le théorème de paramétricité. Cette propriété permet de déduire du type polymorphe d'une fonction une information sur l'utilité de ses arguments. Par exemple, la paramétricité

formalise le fait qu'une fonction de type $List\ \alpha \rightarrow int$ n'a pas besoin des éléments de la liste argument pour produire son résultat.

Notre GC utilise ces informations pour détecter et récupérer des (parties de) structures toujours référencées. Nous avons également proposé deux extensions simples du typage classique qui permettent d'inférer des types donnant une information d'utilité plus précise. Cette technique résoud plusieurs types de fuites de mémoire très communes. De plus, elle s'applique à tout type de langage fonctionnel fortement typé (d'ordre supérieur, strict ou paresseux, pur ou à effets de bord) et aux techniques de GC stop&copy ou mark&sweep .

Ce GC étendu a été mis en œuvre et intégré dans le compilateur Tabac. Nous avons rencontré de nombreux programmes classiques pour lesquels notre GC réduit la taille mémoire minimale nécessaire de 20% à 40%. Bien sûr, il est aussi possible d'exhiber des programmes pour lesquels on améliore l'ordre de grandeur de la complexité mémoire ou, à l'inverse, pour lesquels on ne peut rien récupérer de plus qu'un GC ordinaire. Si notre méthode n'impose aucun surcoût lors de l'exécution normale du programme, le GC lui même est plus complexe qu'un GC standard. Le procédé nécessite d'unifier des informations de type et impose deux parcours de la structure accessible pour résoudre le problème des objets partagés. Notre GC est typiquement trois fois plus lent pour des programmes stricts et de quatre à six fois plus lent pour des programmes paresseux. Notre politique consiste à utiliser un GC standard la plupart du temps ; l'extension est déclenchée à chaque fois qu'il ne reste plus d'espace disponible et, de temps en temps, suivant des critères d'occupation mémoire et de gain escompté. En récupérant plus de mémoire, notre GC étendu allège la charge des GC exécutés par la suite, ce qui rentabilise le surcoût. Cette stratégie fonctionne particulièrement bien lorsque l'espace mémoire disponible devient faible (le temps de gestion de mémoire global est raccourci de façon significative). Lorsque l'espace disponible est confortable, l'extension se déclenche plus rarement et le temps passé dans les GC est inchangé.

3.1.3 Taxonomie des techniques de mise en œuvre des langages fonctionnels

Participants : Rémi Douence, Pascal Fradet

La mise en œuvre des langages fonctionnels offre différents choix concernant la stratégie d'évaluation, la gestion des variables ou la représentation des structures de données en mémoire. Par exemple, la mise en œuvre peut être basée sur la réduction de graphe ou sur des environnements, utiliser l'appel par valeur ou par nécessité, etc ... Il s'ensuit une pléthore de propositions de mise en œuvre (e.g. la CAM , la G-Machine, TIM, ...).

Nous avons exploité le principe de compilation par transformations de programmes pour exprimer dans un même cadre les différentes techniques de mise en œuvre. Cette démarche a permis de comparer et d'identifier des points de choix fondamentaux dans les diverses machines abstraites. Notre objectif est à la fois d'obtenir une taxonomie générale des implantations existantes et de concevoir des stratégies hybrides adaptées au programme à compiler.

3.2 Les langages de programmation logique

Nous nous intéressons aussi bien aux aspects linguistiques de la programmation logique (typage de λProlog, reconstruction pragmatique du langage) qu'aux techniques de mise en œuvre de langages de programmation logiques étendus (λProlog, clauses généralisées).

3.2.1 Typage de λProlog

Participants : Pascale Louvet, Olivier Ridoux

L'unification des λ-termes n'est pas bien définie pour les λ-termes non typés. Par ailleurs, nous considérons que le typage est un plus pour la méthodologie de programmation en Prolog. Le langage de types de λProlog est celui des types simples avec variables (ML^-). Cependant le système de types des programmes de λProlog n'est pas complètement spécifié. Chaque clause d'un programme peut être typée séparément en utilisant les règles d'inférence du λ-calcul simplement typé avec variables de type, mais le typage des programmes (ensembles de clauses) est plus difficile. En particulier la définition du langage ne précise pas si toutes

les occurrences en tête de clause d'un même symbole prédicatif ont le même type ou non. Le cas où elles ont le même type est connu sous le nom de *principe de généricité des définitions*. C'est celui que nous étudions. Mais nous ne cherchons pas seulement à compléter la spécification du système de types actuel, nous voulons définir un système capable de mieux prendre en compte certaines pratiques de programmation. Le système actuel est à la fois trop peu puissant et trop rigide. trois extensions sont étudiées :

- La possibilité d'exprimer des types en fonction d'autres types ou de termes. Ceci permet, par exemple, de spécifier des listes de longueur bornée par un entier, ou encore de définir un schéma de type complexe.

- Le passage des types en arguments dans les prédicats. On introduit ainsi une forme de polymorphisme *ad hoc* en λProlog.

- La possibilité de spécifier des contraintes sur un type. Cette idée s'inspire des types qualifiés définis dans le cadre des langages fonctionnels.

3.2.2 Pragmatique de λProlog

Participants : Pascal Brisset, Olivier Ridoux

λProlog est un langage très riche qui peut déconcerter un programmeur novice. On peut aussi se demander si les extensions apportées dans le domaine des formules comme dans celui des termes sont nécessaires simultanément et si des langages intermédiaires intéressants ne pourraient pas être définis, au moins dans un but pédagogique. Nous avons étudié cette question en montrant que des liens de nécessité conduisent à adopter toutes les extensions à partir du moment où le langage des termes de Prolog est étendu à celui des λ-termes simplement typés et où l'$\alpha\beta$-équivalence est intégrée à la théorie de l'égalité. De cette reconstruction découle une heuristique de programmation par induction sur les types qui peut servir de guide pour la programmation en λProlog [23].

3.2.3 La mise en œuvre de λProlog

Participants : Yves Bekkers, Pascal Brisset, Pascale Louvet, Florimond Ployette, Olivier Ridoux

Nous avons travaillé à la mise en œuvre de λProlog en utilisant Mali comme technique de représentation des données. Toute la généralité de Mali a été utilisée, et aucune addition n'a été nécessaire [24].

Nous traduisons les programmes λProlog en des programmes impératifs (écrits en C) qui utilisent Mali comme mémoire pour la résolution. Chaque prédicat est traduit en une fonction qui fait évoluer quatre continuations (échec, succès, programme et signature). Cette méthode s'est montrée efficace. Par exemple, une application de démonstration automatique (du projet Croap, de l'Inria Sophia) est en moyenne vingt fois plus rapide sur notre système qu'avec eLP (l'implantation réalisée à Carnegie Mellon). En outre, ce démonstrateur bénéficie de la gestion de mémoire de Mali et accepte donc des applications beaucoup plus importantes.

Nous poursuivons actuellement notre effort de mise en œuvre dans deux directions essentielles :

- Une refonte du compilateur qui intègrera un lecteur de termes universel et prendra en compte le système de types mentionné plus haut.
- La mise au point d'un interprèteur pour le langage. Ce dernier permettra de pallier la relative lourdeur d'utilisation du compilateur, élargissant ainsi le cercle des utilisations possibles de λProlog.

3.2.4 Mise en œuvre de Prolog avec clauses généralisées

Participants : Yves Bekkers, Igor Stéphan

La programmation logique standard (Prolog) doit sa popularité à ce que sa théorie, la logique des clauses de Horn, est bien comprise et produit simplement une procédure d'interprétation *naturelle*. L'étude des extensions de Prolog, en particulier celles qui visent à élargir la classe de logique considérée, est un domaine de recherche actif ces dernières années.

Après une analyse et une comparaison de trois extensions de Prolog, *Near-Horn Prolog* de Loveland, SPRF de Plaisted et *N-Prolog* de Gabbay

et Reyle, qui ont toutes comme point commun d'utiliser l'analyse de cas comme mécanisme de raisonnement pour les clauses étendues, nous avons mis au point une stratégie efficace d'interprétation des clauses disjonctives. Les clauses disjonctives sont classiquement interprétées par la règle d'éclatement (*splitting*). À cause de ses effets combinatoires (duplication du but et du programme), on souhaite appliquer cette règle dans une preuve le plus tard possible, sinon jamais. Nous proposons une stratégie dans laquelle la nécessité d'appliquer la règle de *splitting* est détectée *a posteriori* [20] ; quand elle est détectée, la preuve est *retouchée* itérativement jusqu'à trouver une sous-preuve minimale où il est nécessaire d'appliquer la règle. Nous étudions la programmation de cette stratégie dans le contexte de la technologie Prolog .

3.2.5 Compilation générique de la programmation logique par contraintes

Participants : Christophe Bonnet

La programmation logique par contraintes apparaît aujourd'hui comme une manière particulièrement intéressante d'exploiter les atouts de la programmation logique en dépassant ses limitations. Une partie importante de l'effort de recherche entrepris dans ce cadre a été consacrée à la réalisation de résolveurs efficaces pour certains types de problèmes (booléens, systèmes linéaires, domaines finis ...). Mais, indépendamment de la recherche du bon algorithme , la réalisation d'un système de programmation logique par contraintes ou l'adjonction de nouvelles capacités à un système existant restent des entreprises délicates.

C'est pour répondre à ce problème que nous cherchons à élaborer des modèles généraux de résolution et de représentation des contraintes, afin d'aboutir à un sytème générique de compilation. L'objectif est d'obtenir, à partir de modules décrivant des résolveurs, un compilateur pour un langage de programmation logique utilisant ces résolveurs. Dans ce but, nous étudions actuellement l'*élimination gaussienne généralisée*, dont l'unification des termes du premier ordre et la résolution d'équations linéaires sont trivialement des cas particuliers.

3.2.6 Étude approfondie de la machine abstraite de Warren

Participants : Jacques Noyé

La machine abstraite de Warren (WAM) est le modèle le plus utilisé pour décrire les implantations de langages de programmation logique. Il n'existe pourtant pas, à notre connaissance, d'étude systématique des forces et des faiblesses de ce modèle. Nous avons examiné en profondeur quatre aspects importants de la mise en œuvre de Prolog sur la WAM [2]. Les trois premiers aspects, le contrôle avant, le retour arrière, et la coupure couvrent la majeure partie du contrôle standard de Prolog. Leur étude éclaircit les similarités entre contrôle avant et contrôle arrière et montre les limites de la politique de gestion de mémoire adoptée par la WAM. Le dernier aspect traite des techniques de modification réversible. Ces techniques jouent un rôle majeur dans la mise en œuvre de nombreuses extensions et fournissent un bon exemple de la manière d'étendre les possibilités d'un système Prolog à partir de modifications mineures du moteur Prolog de base.

3.3 Le formalisme Gamma

Participants : Daniel Le Métayer

Gamma est un formalisme que nous avons proposé il y a quelques années et qui a depuis servi de base à différentes recherches. Gamma permet une description abstraite des programmes, sans contraintes d'ordonnancement inutiles. On peut illustrer le comportement des programmes Gamma à l'aide de la métaphore de la réaction chimique : l'exécution est une succession de réactions chimiques consommant les éléments d'un multi-ensemble pour produire de nouveaux éléments selon certaines règles.

Le modèle Gamma suggère une approche nouvelle de la conception de programmes : nous avons montré comment un grand nombre de problèmes, dans des domaines d'application variés, trouvent en Gamma des solutions élégantes et dont la correction est facile à prouver. Nous avons proposé récemment une version de Gamma d'ordre supérieur qui permet d'avoir une vision unifiée des programmes et des données [16]. On peut ainsi insérer à l'intérieur d'un multi-ensemble un programme actif (réagissant). Nous avons défini la sémantique de ce langage et avons montré l'intérêt de ce surcroît de puissance d'expression. Il permet entre autres

de décrire naturellement les machines abstraites chimiques de Berry et Boudol et de définir des structures de contrôles adaptées à différentes architectures. Nous avons en particulier dérivé dans le langage lui-même une version de programme représentant l'exécution selon le modèle à vecteur adapté à la Connection Machine . Nous poursuivons cet effort dans le cadre du projet BRA *Coordination* avec un objectif plus pragmatique : il s'agit d'intégrer dans le langage un moyen de définir des données plus structurées. Ceci serait bénéfique à deux point de vue :

- le programmeur n'aurait plus, pour structurer ses données, à introduire un codage inélégant et source d'erreurs ;
- on pourrait envisager une mise en œuvre plus efficace de Gamma car les informations structurelles contenues dans le programmes pourraient être exploitées à la compilation.

La difficulté est qu'on ne peut recourir à la méthode habituelle pour définir des structures de données (types récursifs) car ceci induirait une technique de programmation récursive incompatible avec les principes de Gamma. Nous cherchons plutôt une description des structures de données de nature topologique , par exemple sous forme de relations.

3.4 Analyse de programmes

L'analyse statique est un élément clef pour la mise en œuvre efficace des langages déclaratifs. Citons notamment l'analyse de nécessité pour les langages fonctionnels paresseux, l'analyse de temps de liaison pour l'évaluation partielle et l'analyse de mode pour les langages logiques. Un des soucis actuels de la communauté en matière d'analyse de programmes est la conception d'analyseurs efficaces et correctement intégrés dans les compilateurs. Nous nous sommes attaqués à ces problèmes en considérant l'analyse sous l'angle de l'inférence de types non standards. Nous travaillons également à l'extension des champs d'application de l'analyse de programmes, notamment pour traiter des propriétés ayant trait à la protection de l'information.

3.4.1 Inférence de types paresseux

Participants : Ronan Gaugne, Valérie Gouranton, Daniel Le Métayer

Deux techniques principales ont été étudiées pour l'analyse de programmes déclaratifs : l'analyse sémantique (interprétation abstraite) et

l'analyse syntaxique (synthèse de types). Ces techniques étaient jusqu'à récemment incomparables : l'analyse sémantique était plus précise mais réputée moins efficace que l'analyse syntaxique. Il a été proposé en 1991 un système d'inférence de types équivalent à l'interprétation abstraite pour l'analyse de nécessité (c'est à dire aussi précis). Les types intersection (ou conjonction) jouent à cet égard un rôle essentiel. Il restait cependant à associer un algorithme au système de types en question. Nous avons poursuivi ce travail (en collaboration avec C. Hankin d'Imperial College) en montrant qu'il est possible de dériver de manière systématique les techniques traditionnelles d'analyse par interprétation abstraite à partir de la spécification sous forme de système de types, dressant ainsi un pont entre les deux techniques [7, 13].

Ce premier résultat nous a ensuite conduit à proposer un nouvel algorithme de synthèse de types complet et efficace pour le système d'inférence considéré. Il procède par inférence de types paresseux, calculant les informations de types de sous-expressions à la demande, ce qui lui confère un avantage majeur sur l'interprétation abstraite traditionnelle. Celle-ci peut être vue comme un moyen de calculer systématiquement le type complet d'une expression (conjonction de tous les types possédés par l'expression). Il s'avère que le type complet d'une sous-expression est rarement nécessaire. L'algorithme d'inférence de types paresseux a été implanté et s'avère en effet plus efficace que les autres analyseurs connus. Nous avons montré que l'idée de base s'étendait au traitement des listes et à d'autres types d'analyse et nous nous intéressons maintenant à la conception d'un système générique pour l'analyse de programmes qui permettrait de paramétrer la propriété recherchée [14, 15]. L'outil ainsi conçu devrait trouver plusieurs applications dans le projet et au delà.

3.4.2 Correction d'optimisations reposant sur des analyses globales

Participants : Daniel Le Métayer

La puissance et la correction des techniques d'analyse ont été largement étudiées ; cependant les optimisations de compilateur mettant à profit les résultats d'analyse sont généralement présentées informellement et leur correction n'est pas démontrée. En fait, les choix d'optimisation reposant sur les informations de nécessité sont loin d'être simples, notamment quand on considère des listes partiellement évaluées, et leur correction

mérite d'être étudiée sérieusement. Nous avons exprimé formellement ces optimisations dans le cadre de la compilation par transformation de programmes décrite plus haut et nous avons montré leur correction [4]. Cette formalisation a le mérite de mettre au jour les choix d'optimisation possibles.

3.4.3 Inférence de types par résolution de contraintes

Participants : Flemming Damm

L'accueil de Flemming Damm comme boursier HCM nous a permis d'aborder l'analyse de programmes par inférence de types sous l'angle de la résolution de contraintes ensemblistes. Nous avons montré comment un langage de types assez riche (comprenant union, intersection et récursivité) peut être représenté par des expressions régulières et nous avons proposé une procédure de décision complète pour l'inclusion de types [10, 27].

3.4.4 Preuve automatique de propriétés de sécurité de programmes

Participants : Daniel Le Métayer

L'analyse de programmes est généralement considérée dans le cadre de la compilation optimisante. Il est plus rare de voir ces techniques appliquées à la preuve de propriétés ayant trait à la correction des logiciels. Nous pensons qu'il s'agit pourtant d'un débouché important pour ce genre de travaux et nous avons entamé des recherches dans cette direction en considérant des propriétés de sécurité des logiciels. Nous prenons ici le mot sécurité au sens de protection de l'information. Cette notion a été formalisée en terme de flux d'informations définis à partir de la sémantique opérationnelle d'un langage impératif simple. Nous avons proposé une axiomatisation des propriétés de sécurité et nous en avons dérivé un algorithme d'inférence correct et complet [22, 8]. Cet algorithme conduit à un analyseur efficace, capable de prouver mécaniquement des propriétés sur les flux d'informations. Ce travail a été mené en coopération avec le projet *Solidor*. Nous entendons poursuivre cet effort, notamment en vue de traiter un langage plus riche incluant des primitives de communication.

3.4.5 Analyse dynamique

Participants : Mireille Ducassé

L'arrivée dans le projet de Mireille Ducassé permet de compléter nos travaux sur l'*analyse statique* de programmes par une compétence en matière d'*analyse dynamique* [6, 28]. L'analyse dynamique, qui consiste à étudier le comportement effectif des programmes, a été relativement moins étudiée que l'analyse statique. Elle peut cependant devenir un complément significatif de l'analyse statique (cas du prototypage notamment). Notre objectif dans ce domaine est de bâtir des outils plus homogènes et plus puissants d'aide à la mise au point de programmes déclaratifs.

3.5 Évaluation partielle

L'évaluation partielle a pour but de spécialiser un programme en fonction de certaines de ses données d'entrées [1]. Cette transformation de programmes préserve la sémantique initiale dans la mesure où le *programme spécialisé*, appliqué aux données manquantes, produit le même résultat que le programme original appliqué à toutes les données. En pratique, un évaluateur partiel consiste en un ensemble réduit de règles de transformation de programmes visant à évaluer les expressions dépendantes des données disponibles. L'évaluation partielle a été utilisée pour des applications aussi variées que la génération de compilateurs à partir d'interprèteurs, l'optimisation de programmes numériques, les bases de données et le graphisme.

Dans ce domaine, nos travaux s'articulent autour des axes de recherche suivants :

- Techniques d'évaluation partielle pour langages applicatifs.
- Développement du système d'évaluation partielle Schism.
- Applications de l'évaluation partielle.
- Spécialisation incrémentale.

3.5.1 Techniques d'évaluation partielle pour langages applicatifs

Participants : Charles Consel, Barbara Moura, Tommy Thorn

Nous avons conçu diverses analyses statiques et transformations de programmes permettant d'améliorer le processus d'évaluation partielle en temps d'exécution, qualité des programmes spécialisés et type de programmes traités. Nous avons en particulier travaillé sur l'analyse de temps de liaison qui permet de compiler une partie du processus d'évaluation partielle [3]. Pour cette analyse, nous avons proposé une nouvelle approche basée sur une technique de *calcul symbolique de point fixe* permettant d'analyser une fonction indépendamment de ses contextes d'utilisation. Cette technique s'applique aux structures de données et aux fonctions d'ordre supérieur. Nous envisageons d'explorer d'autres problèmes de flots de données où une telle technique pourrait être utilisée.

Toujours dans le domaine de l'analyse de temps de liaison, nous avons développé des techniques permettant d'accélérer la convergence du processus d'itération de point fixe. Ces techniques s'appuient sur une approximation du graphe de contrôle des programmes analysés. Ces programmes étant d'ordre supérieur, la difficulté est donc de construire incrémentalement leur graphe de contrôle. Ces techniques permettent d'analyser des programmes jusqu'à dix fois plus vite que l'analyse non-optimisée.

Les techniques d'évaluation partielle que nous avons étudiées jusqu'à récemment étaient limitées aux langages ne comportant pas d'effets de bord. L'utilisation de ces techniques pour des langages applicatifs comportant des traits impératifs nous a conduit à étudier une approche basée sur la transformation de programmes. Nous développons des techniques qui visent à transformer un programme impératif en une représentation fonctionnelle. Cette représentation permet toutefois de préserver suffisamment la structure des programmes transformés pour que leur spécialisation soit comparable à celle obtenue par un évaluateur partiel impératif.

3.5.2 Le système d'évaluation partielle Schism

Participants : Charles Consel, Luke Hornof, Barbara Moura, Tommy Thorn

Du point de vue pratique ces travaux ont donné lieu à la conception d'un système d'évaluation partielle pour programmes fonctionnels d'ordre supérieur : Schism. Schism est évaluateur partiel générique qui permet de traiter une variété de langages applicatifs comme Scheme et ML. L'évaluation partielle des appels de fonctions est guidée par des annotations qui assurent la terminaison de ce traitement. Ces annotations sont produites automatiquement. Schism repose sur une analyse de temps de liaison *polyvariante*. Cette analyse permet de collecter plusieurs descriptions abstraites de chaque fonction d'un programme pour tenir compte des différents contextes dans lesquels cette fonction est appelée.

Le système Schism est installé dans différentes universités ; il a été utilisé pour diverses applications dont le filtrage, la génération de compilateurs, le calcul incrémental et l'instrumentation de programmes.

3.5.3 Applications de l'évaluation partielle

Participants : David Burke, Charles Consel, Barbara Moura

Schism est le système que nous utilisons pour explorer une variété d'applications de l'évaluation partielle. Récemment, nous avons porté notre effort sur son utilisation pour la génération de compilateurs à partir d'interprèteurs. Étant donné le langage traité par Schism, les interprèteurs utilisés peuvent être vus comme des spécifications dénotationelles exécutables. L'intérêt est alors d'éliminer la couche d'interprétation par évaluation partielle. L'objectif est que le compilateur ainsi obtenu puisse effectuer les opérations correspondant à la sémantique statique du langage. D'autre part, le compilateur obtenu est correct par construction.

Traditionnellement, les compilateurs générés par évaluation partielle produisent un code de très haut niveau. De fait, certains aspects de la compilation ne sont pas traités. Notre approche vise au contraire à étudier toutes les couches de la compilation, de façon uniforme en utilisant l'évaluation partielle. Nous excluons toutefois de cette étude la génération de code machine pour laquelle des générateurs de générateurs de code efficaces existent (par exemple BURG).

La description des couches de compilation est basée sur la notion de machine abstraite. Chaque machine abstraite est décrite par une machine abstraite de plus bas niveau qui exhibe plus de détails opérationnels. Ultimement, la machine abstraite correspond au processeur cible. Ces différents niveaux de machines abstraites permettent d'effectuer une description par couches d'un langage.

Nous avons étudié la génération de compilateurs à partir d'interprèteurs Pascal (un sous-ensemble) et Scheme. Pour ces deux langages les résultats sont très encourageants : pour Pascal, par exemple, nous avons obtenu du code compilé seulement 20% plus lent que celui obtenu par un compilateur de production. Pour Scheme, nous avons pu partir de la norme de ce langage, décrite par une sémantique dénotationnelle ; ceci garantit que le compilateur généré est fidèle à la spécification du langage.

3.5.4 Spécialisation incrémentale

Participants : Charles Consel, Luke Hornof, Eugen-Nicolae Volanschi

Nous développons une nouvelle forme d'évaluation partielle que nous appelons *spécialisation incrémentale*. C'est une méthode uniforme qui vise à instancier des opérations en fonction d'invariants qui deviennent valides à certaines étapes de l'exécution d'un programme. La validité d'un invariant pouvant être limitée dans le temps, des *gardes* sont associées aux invariants. De fait, l'invariant rend explicite les postulats sur lesquels une optimisation est basée, alors qu'une garde permet de détecter quand ces postulats sont incorrects. Ainsi, dès que certains invariants sont valides les procédures dépendant de ces invariants sont spécialisées ; elles sont mises à jour dès qu'ils deviennent invalides.

L'efficacité de cette technique d'optimisation repose sur l'efficacité de la phase de spécialisation. Pour obtenir cette efficacité, la phase de spécialisation est compilée *avant* l'exécution. Ce processus de compilation produit des fragments de binaires non-instanciés (*code templates*). À l'exécution, ces fragments de binaires non-instanciés sont complétés par des valeurs et assemblés. Une implémentation de cette approche est en cours pour la spécialisation incrémentale de programmes écrits en langage C.

Dans le cadre d'un projet avec le Cnet, nous étudions l'application de la spécialisation incrémentale aux systèmes d'exploitation. On s'aper-

çoit en effet que les réseaux de machines hétérogènes, les périphériques à haut débit et les tâches massivement distribuées font naître un besoin pressant d'adaptabilité des systèmes opératoires. Tout en restant adaptable et donc généraux, ces systèmes opératoires doivent toutefois être performants. En soi, ces deux impératifs semblent contradictoires. Nous proposons de réconcilier généralité et performance en utilisant la spécialisation incrémentale pour spécialiser dynamiquement des programmes en fonction de certains invariants. Ce projet vise plus particulièrement à optimiser le système de communication par messages de Chorus.

4 Actions industrielles

Nos travaux sur la spécialisation incrémentale se font en contact étroit avec la société Hewlett Packard et l'Oregon Graduate Institute. Nous avons également répondu avec succès à un appel d'offre du Cnet qui se traduit par un contrat de 3 ans pour étudier l'application de la spécialisation incrémentale aux systèmes d'exploitation.

Pour ce qui concerne les applications de l'analyse de programmes, nous entamons une collaboration avec la société rennaise AQL qui se concrétise par une bourse Cifre. Le but de cette action est d'appliquer des méthodes formelles mécanisées pour procurer une aide à la phase de test des programmes.

5 Actions nationales et internationales

5.1 Programmes nationaux

Le projet Lande est impliqué dans le GDR programmation (pôles Langages fonctionnels et Programmation logique) et Yves Bekkers fait partie de son équipe de direction. Nous sommes également engagés dans une action soutenue par la Dred et qui regroupe des projets de l'Inria Rocquencourt (Cristal), de l'ENS, de l'École des Mines et du Laas sur le thème des Méthodes formelles et langages de haut niveau . Par ailleurs, M. Ducassé est membre du groupe Programmation Logique de l'Afcet.

5.2 Projets européens

Nous participons au projet Esprit BRA, *Coordination* avec l'ECRC, l'université de Bologne, Uninova (Portugal), Imperial College, Diku, l'université de Genève et le centre de recherches de Xerox de Grenoble.

5.3 Relations internationales

Nous collaborons avec D. Schmidt de l'université du Kansas dans le cadre de la convention Inria-NSF. Certains de nos travaux sur les langages fonctionnels et sur Gamma sont menés conjointement avec C. Hankin d'Imperial College (Londres) et D. Sands de Diku (Copenhague).

Une visite de deux mois de Paul Tarau nous a permis d'initier une collaboration avec l'université de Moncton (Canada) sur l'implantation des langages de programmation logique. Nous avons également déposé un dossier dans le cadre de la convention Inria-NSF pour un partenariat avec l'université de Pennsylvanie (D. Miller) et l'université de Carnegie Mellon (P. Lee et F. Pfennig). Nous travaillons aussi avec d'autres universités sur l'évaluation partielle (Oregon Graduate Institute, Aarhus, notamment). Ces collaborations se traduisent par des publications communes et des séjours de courte durée de chercheurs et d'étudiants. Pour conclure, mentionnons que nous participons aux réunions du groupe de travail bilatéral (Europe-États-Unis) Atlantique (sur la sémantique, l'analyse et la transformation de programmes).

5.4 Comités de programmes, conférences

C. Consel a été membre du comité de programme de la conférence PEPM (Partial Evaluation and semantics-based Program Manipulation) et membre du comité de selection de NSF Research Initiation Award . Il a co-organisé une école d'été sur l'évaluation partielle à l'université de Carnegie Mellon. Par ailleurs, il a été invité à donner un séminaire à l'université de Montréal.

M. Ducassé a fait partie des comités de programme des conférences JFPL (Journées Francophones sur la Programmation en Logique) et ERGO-IA'94 (Ergonomie et Informatique Avancée). Elle a été invitée à présenter ses travaux à l'université de Saarbrück et au laboratoire Dec de Paris.

D. Le Métayer a été membre des comités de programme des conférence SAS (Static Analysis Symposium) et JFLA (Journées Francophones

des langages applicatifs). Il a présenté les travaux du projet au centre de recherche Bull (Clayes-Sous-Bois), à AQL (Rennes) et Cap Gemini Innovation (Paris).

O. Ridoux a fait partie du comité de programme de la conférence ILPS (International Logic Programming Symposium).

6 Diffusion des résultats

6.1 Prototypes

La version 6 de Mali et la version 1 de Prolog/Mali sont déposées à l'agence pour la protection des programmes (APP). Ces deux systèmes sont installés sur le serveur *ftp* de l'Irisa. Tous deux sont accompagnés d'une documentation.

L'évaluateur partiel Schism est installé dans les universités d'Aarhus, Carnegie Mellon University, Indiana University, Kansas State University, Oregon Graduate Institute, Stanford University, et Yale University. Il est utilisé pour l'enseignement et la recherche. Il sera distribué sur le réseau en 1995.

6.2 Actions d'enseignement

En plus des enseignements assurés au titre des services réguliers de nos enseignants-chercheurs, nous avons encadré 4 stages de DEA et sommes intervenus dans les occasions suivantes :

- Y. Bekkers a assuré une formation à λProlog dans le cadre de l'école Jeunes Chercheurs du GDR Programmation. Il a donné un cours de programmation logique à l'École des Mines et à l'ISIA (Sophia Antipolis) et un cours de programmation logique d'ordre supérieur à l'ENS.
- R. Douence et P. Louvet sont moniteurs à l'université de Rennes 1.
- M. Ducassé donne un cours sur le langage de spécification formelle **B** au Celar (Rennes).
- D. Le Métayer intervient en DEA d'informatique sur le thème *programmation fonctionnelle*.

- O. Ridoux a donné un cours de programmation logique avancée en DESS/ISA et un cours de programmation par contraintes à l'école de printemps du LITP.

7 Bibliographie du projet

Livres et monographies

[1] C. CONSEL, O. DANVY, *Partial evaluation of procedural languages*, MIT Press, 1995.

Thèses

[2] J. NOYÉ, *Élagage de contexte, retour arrière superficiel, coupure et autres : une étude approfondie de la WAM*, thèse, université de Rennes 1, novembre 1994.

Articles et chapitres de livre

[3] J. M. ASHLEY, C. CONSEL, «Fixpoint computation for polyvariant static analyses», ACM *Transactions on Programming Languages and Systems*, 1995.

[4] G. BURN, D. LE MÉTAYER, «Proving the correctness of compiler optimisations based on strictness analysis», *Journal of Functional Programming*, 1995.

[5] C. CONSEL, S. C. KHOO, «On-line & off-line partial evaluation: semantic specifications and correctness proofs», *Journal of Functional Programming*, 1995.

[6] M. DUCASSÉ, J. NOYÉ, «Logic programming environments: dynamic program analysis and debugging», *Journal of Logic Programming 19/20*, 1994, p. 351–384.

[7] C. L. HANKIN, D. LE MÉTAYER, «Lazy types and program analysis», *Science of Computer Programming*, 1995.

Communications à des congrès, colloques, etc.

[8] J.-P. BANÂTRE, C. BRYCE, D. L. MÉTAYER, «Compile-time detection of information flow in sequential programs», *in : proc. European Symposium on Research in Computer Security*, Springer Verlag, LNCS, 1994.

[9] C. CONSEL, «Fast strictness analysis via symbolic fixpoint iteration», *in : proc. Static Analysis Symposium*, LNCS, 864, Springer Verlag, p. 423–431, septembre 1994.

[10] F. DAMM, «Subtyping with union types, intersection types and recursive types», *in : proc. int. Symposium on Theoretical Aspects of Computer Software*, LNCS, 789, Springer Verlag, p. 687–706, avril 1994.

[11] P. FRADET, «Collecting more garbage», *in : proc.* ACM *conf. on Lisp and Functional Programming*, ACM, p. 24–33, juin 1994.

[12] P. FRADET, «Compilation of head and strong reduction», *in : proc. 5th European Symposium on Programming*, LNCS, 788, Springer Verlag, p. 211–224, avril 1994.

[13] C. L. HANKIN, D. LE MÉTAYER, «Deriving algorithms from type inference systems: Application to strictness analysis», *in : proc.* ACM *Symposium on Principles of Programming Languages*, p. 202–212, janvier 1994.

[14] C. L. HANKIN, D. LE MÉTAYER, «Lazy type inference for the strictness analysis of lists», *in : proc. 5th European Symposium on Programming*, LNCS, 788, Springer Verlag, p. 211–224, avril 1994.

[15] C. L. HANKIN, D. LE MÉTAYER, «A type-based framework for program analysis», *in : proc. Static Analysis Symposium*, LNCS, 864, Springer Verlag, p. 380–394, septembre 1994.

[16] D. LE MÉTAYER, «Higher-order multiset programming», *in : proc. DIMACS workshop on Specification of Parallel Algorithms*, American Mathematical Society, 1994.

[17] J. NOYÉ, «An Overview of the Knowledge Crunching Machine», *in : Emerging Trends in Database and Knowledge-based Machines*, IEEE Computer Society Press, 1994.

[18] J. NOYÉ, «To Trim or not to Trim?», *in : Implementations of logic programming systems*, Kluwer Academic Publishers, 1994.

[19] O. RIDOUX, «Imagining CLP(Λ,$\equiv_{\alpha,\beta}$)», *in : Actes de l'école de printemps du LITP*, Springer Verlag, LNCS, 1994.

[20] I. STÉPHAN, «Du *ou* dans les arbres», *in : Journées Francophones sur la Programmation Logique*, Bordeaux, 1994. dans *Journées Francophones sur la Programmation Logique*, Bordeaux.

[21] I. STÉPHAN, «Du *ou* dans les arbres», *in : II$^{\text{ièmes}}$ Rencontres Nationales des Jeunes Chercheurs en I.A.*, Marseille, 1994.

Rapports de recherche et publications internes

[22] J.-P. Banâtre, C. Bryce, D. Le Métayer, «Mechanical proof of security properties», *rapport de recherche*, Inria, mai 1994.

[23] C. Belleannée, P. Brisset, O. Ridoux, «A pragmatic reconstruction of λProlog», *rapport de recherche*, Inria, 1994.

[24] P. Brisset, O. Ridoux, «The architecture of an implementation of λProlog: Prolog/Mali», *rapport de recherche*, Inria, 1994, (Workshop on the implementation of logic programming, ILPS'94).

[25] S. Coupet-Grimal, O. Ridoux, «On the use of advanced logic programming languages in computational linguistics», *rapport de recherche*, Inria, 1994.

[26] F. Damm, «Subtyping with union types, intersection types and recursive types II», *rapport de recherche*, Inria, mars 1994.

[27] F. Damm, «Type inference with set theoretic type operators», *rapport de recherche*, Inria, juin 1994.

[28] M. Ducassé, «Automated debugging with trace analysis: the case of Opium, in "Proc. KI'94 workshop on Development, Test and Maintenance of Declarative AI-Programs», *rapport de recherche n° 238*, gmd, septembre 1994.

8 Abstract

Declarative languages (which include functional and logic programming languages) are based on well-established mathematical foundations and provide very expressive programming tools. The main goal of the Lande project is to explore this unique position, at the crossroads of theory and practice, to provide formally-based software engineering tools.

The project studies and develops a range of techniques for a more efficient implementation of declarative languages through program analysis, program transformation and sophisticated memory management methods. We have put forward a method for memory management especially adapted to non-deterministic languages (Mali). Several versions (software and hardware) of Mali have been realised and used for the implementation of various logical programming languages (including the higher-order λProlog).

Our work on program transformation results in the Tabac compiler (which produces efficient and provably correct code) and the Schism partial evaluation tool (for deriving specialised, efficient versions of generic

programs). Schism has been used successfully in various applications such as designing programming environments, optimising programs and deriving compilers from the semantics of a language.

Static program analysis is also a major research topic of the Lande project. Several program analysis methods have been proposed in the project and integrated into the above tools. Most of them are based on a non-standard type inference system.

Scientific Context :

- Compilation
- Program analysis
- Program transformation
- Partial evaluation
- Specialisation
- Memory management, garbage collection
- Typing
- Logic programming languages, λProlog
- Functional programming
- Software engineering

Developments :

- Schism (partial evaluation tool)
- lambda-Prolog (compiler)

Table des matières