INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Équipe MIRÓ

Systèmes à Objets, Types et Prototypes : Sémantique et Validation

Lorraine - Sophia Antipolis

THÈME 2A

Table des matières

1. Composition de l'équipe

Miró est un projet commun à l'INRIA Lorraine et l'INRIA Sophia Antipolis, et l'Université Nancy 2.

Responsable scientifique
Luigi Liquori [CR INRIA Lorraine]

Responsables permanents
Dominique Colnet [Professeur Nancy 2]
Joëlle Despeyroux [CR INRIA Sophia-Antipolis]

Personnel Inria
Olivier Zendra [CR INRIA Lorraine]

Ingénieurs experts
Philippe Ribet [Ingénieur expert INRIA, 6/2001-6/2003]
Jérôme Boutet [Ingénieur associé INRIA, 9/2002-9/2003]

Chercheurs doctorants
Alberto Ciaffaglione [Co-tutelle INPL-Univ. Udine, 3/2001-04/2003]
Benoît Sonntag [Doctorant INRIA, -2003]

Stagiaires
Cyryl Proch [DEA janvier à juillet 2002]
Karthik Narayanaswamy [Stagiaire IIT, Inde, 5-6/2002]

Collaborateur extérieur
Furio Honsell [*Ad Honorem*, Président des Universités de Udine, Italie]

Assistantes de projet
Nathalie Bellesso [INRIA Sophia Antipolis, à temps partiel]
Laurence Benini [INRIA Lorraine, à temps partiel]

2. Présentation et objectifs généraux

2.1. Pourquoi Miró

Johachim Miró [142] est, à notre humble avis, un grand peintre *à objets* ; en fait, ses tableaux de la période 1940-1960 sont basés sur l'utilisation de nombreux objets géométriques : points, points colorés, carrés, cercles, fenêtres, lignes, courbes, etc. Tous ces éléments sont très chers aux amateurs de la programmation par objets. On peut même aller jusqu'à imaginer que ce peintre, qui s'est installé en France à Montmartre au début des années 50, a certainement inspiré les concepteurs de Simula, de SmallTalk, et la communauté d'informaticiens qui ont étudié les aspects théoriques du paradigme à objets.

2.2. Objectifs généraux de Miró

Les langages à objets ont acquis une importance prépondérante dans les applications informatiques à grande échelle. Cette utilisation a rendu nécessaire l'étude formelle de ces langages pour à la fois mieux en cerner les caractéristiques fondamentales et aussi pour pouvoir définir de nouveaux langages à objets et concurrents, capables de combiner une plus grande expressivité avec une sécurité et efficacité d'utilisation.

L'action Miró explore la possibilité de « concilier » la programmation à objets et la programmation fonctionnelle, tout en gardant l'esprit de l'une et l'élégance mathématique de l'autre, et s'intéresse à la certification des outils développés autour de ces langages (interprètes, compilateurs, ...), avec comme assistant à la preuve privilégié le système Coq.

En complément de ces axes de recherche principaux, nous étudions des théories typées, dans la recherche de nouveaux systèmes améliorant l'activité de la preuve formelle, et dans la compilation efficace des langages de programmation à objets.

Notre programme de recherche se focalise essentiellement sur les axes suivants :

1. l'étude, la définition et l'implantation certifiée d'un langage de programmation à classes (et de son compilateur) , appelé *SmallTalk2K*, et d'un langage à prototypes (*c.à.d.* à objets purs), appelé *FunTalk* (et d'un interprète et d'un compilateur), langage intermédiaire du compilateur de SmallTalk2K ;
2. l'étude de l'efficacité et de la sûreté des langages à objets et notamment d'*Eiffel* (et ses évolutions) et de son compilateur *SmallEiffel* ;
3. l'étude de systèmes de types pour les langages à objets et pour les assistants de preuves ; la réécriture et les calculs formels ($\lambda, \varsigma, \rho, \pi, \ldots$) comme base des langages de programmation à objets, fonctionnels et concurrents ;
4. l'étude du système d'exploitation *Isaac* et de son langage propriétaire à prototypes *Lisaac*.

2.3. Fusion avec l'action Certilab

Suite à l'expertise de J. Despeyroux en tant que relectrice de l'action Miró, l'action Certilab de l'INRIA Sophia Antipolis a fusionné avec Miró le 14 janvier 2002. Miró est donc devenu une action bi-localisée entre Nancy et Sophia Antipolis.

2.4. Présentation des membres de l'équipe

2.4.1. Certification de logiciel et théories typées

Le domaine de recherche de J. Despeyroux concerne la certification de logiciel. J. Despeyroux s'est intéressé notamment à la spécification des langages de programmation et aux preuves de propriétés de langages (dont deux exemples typiques sont la preuve de conservation de types d'un langage lors de son exécution, et la preuve de correction d'un compilateur). Elle travaille dans ce domaine à différents niveaux (théorie et pratique) et en suivant différentes approches. Après une thèse (1982) dans le domaine des types abstraits algébriques, J. Despeyroux a participé, entre 1983 et 1987, à l'élaboration de la Sémantique Naturelle (issue de la la Sémantique Opérationnelle Structurelle de G. Plotkin) en étudiant les preuves de traduction dans cette approche [8]. À cette époque, elle faisait des preuves « sur papier ». De 1988 à 1991, elle a développé son propre système d'assistant à la preuve (nommé Théo). Depuis, elle préfère utiliser des systèmes d'assistant à la preuve développés par d'autres.

En 1992, J. Despeyroux a découvert le grand intérêt des spécifications dites « d'ordre supérieur », qui utilisent une méthode proposée par, notamment, G. Plotkin et F. Pfenning. Aussi surprenant que cela puisse paraître, il n'y a toujours pas de consensus sur le choix de la représentation des variables dans un langage fonctionnel. Les spécifications utilisant les indices de N. de Bruijn (technique la plus connue) sont souvent illisibles et les preuves selon cette approche arbitrairement compliquées. À l'opposé, les méthodes dites « d'ordre supérieur » consistent à utiliser directement les fonctions du système choisi (le Calcul des Constructions et le système Coq par exemple) pour formaliser les notions de variables liées et de substitution du langage à décrire.

Cette approche à l'ordre supérieur permet des spécifications et des preuves concises, claires et élégantes (dans lesquelles les termes dépendants de variables sont représentés comme des fonctions de ces variables). Malheureusement, les principes d'induction et de récursion connus à l'époque ne s'appliquaient pas aux structures (termes de type fonctionnel) utilisées dans cette approche. Pour attaquer ce problème, ouvert depuis 1986 et réputé difficile, J. Despeyroux a travaillé dans deux directions complémentaires.

D'une part, en collaboration avec A. Hirschowitz [79] puis avec A. Felty (U. Ottawa) [78] J. Despeyroux a proposé deux nouvelles méthodes de spécification d'ordre supérieur, permettant d'utiliser l'induction classique proposée, par exemple, dans le système Coq. L'idée maîtresse de ces méthodes, qui a été reprise dans la définition de nouvelles théories typées, est de considérer un terme ouvert, dépendant d'une liste de variables, comme une fonction de cette liste de variables. Les termes « valides » sont définis par un prédicat. Dans la

première méthode (la plus intéressante), les sémantiques sont données sur les termes valides, qui sont des termes fonctionnels.

D'autre part, dans un premier pas vers la définition d'un système de type qui incorpore la méthodologie du système LF (*Logical Framework* développé à Edimbourg) dans des systèmes comme Coq et en collaboration avec F. Pfenning (CMU) et C. Schürmann (Yale) [82] [17] puis avec un doctorant, P. Leleu [80] [9][10], J. Despeyroux a proposé récemment plusieurs systèmes permettant la récursion sur des termes de type fonctionnel. Ces systèmes sont des λ-calculs modaux étendus avec des opérateurs de raisonnement par cas et d'itération préservant l'adéquation des représentations fonctionnelles (*i.e.* représentations utilisant la syntaxe abstraite d'ordre supérieur).

Ces systèmes ont toutes les propriétés souhaitées : préservation du typage par la réduction, confluence, normalisation forte de la réduction et extension conservative par rapport au λ-calcul simplement typé. L'un de ces systèmes a été étendu aux types dépendants, sans l'élimination forte [81].

J. Despeyroux a réalisé l'étude de nombreux exemples sur de petits langages impératifs (Imp de Winskel) [77] ou applicatifs classiques (Mini-ML) [68][8], sur papier en 1986, en machine depuis, principalement dans le système Coq. Ces exemples sont repris dans des notes de cours rédigées pour le DEA MDFI de Marseille [77]. Depuis 1997, elle participe à la formalisation de différents calculs concurrents ou/et à objets, en suivant différentes approches (d'ordre supérieur ou non). Ces études ont été réalisées dans le cadre de stages d'été et d'une thèse de doctorat (G. Gillard) [90]. J. Despeyroux a proposé récemment une spécification originale du π-calcul [76].

2.4.2. Techniques de compilation efficace

Le compilateur SmallEiffel [104][69][132] [4][19], démarré en 1994 par D. Colnet, a donné lieu en 1995 à la première diffusion dans le domaine public d'un prototype de compilateur Eiffel gratuit. Depuis 1995, ce compilateur est utilisé par les étudiants de la licence d'informatique de l'U.H.P. Nancy 1 ainsi que par les élèves de l'ESIAL. Depuis sa création, SmallEiffel en est à sa 26^{e} version et ne cesse de s'améliorer et de se diffuser au plan international.

La distribution actuelle de SmallEiffel comporte : un traducteur Eiffel vers ANSI-C, un traducteur Eiffel vers la machine virtuelle Java (*JVM bytecode*), un indenteur de programme, un extracteur d'interfaces paramétrable (HTML, TEX, etc.), une bibliothèque de base richement garnie ainsi qu'un bon nombre d'outils supplémentaires comme par exemple un dé-compilateur de *bytecode* Java.

Grâce à la technique originale d'inférence de types développée, plus de 80% des appels des méthodes sont couramment liés statiquement (la vitesse d'un exécutable Eiffel est comparable à celle d'un programme écrit en C++ ou en C).

De par sa qualité reconnue au plan international, SmallEiffel a été promu logiciel GNU par R. Stallman, auteur d' `Emacs` et `gcc`, président fondateur de la FSF (*Free Software Foundation*), qui nous a fait l'honneur de venir nous rencontrer au LORIA afin de concrétiser le label GNU. Ainsi, depuis janvier 1998, SmallEiffel est officiellement *SmallEiffel, The GNU Eiffel Compiler*.

Le résultat principal des recherches effectuées dans le cadre de la thèse de doctorat d'O. Zendra [135] a été le développement et la diffusion du compilateur SmallEiffel. Intégrant les techniques d'optimisation développées dans la thèse d'O. Zendra, SmallEiffel produit un code d'excellente qualité, dont les performances sont en général supérieures (voire très supérieures) à celles des autres compilateurs Eiffel disponibles dans le commerce.

2.4.3. Qualité et optimisation du code dans les programmes Java

Récemment, O. Zendra a mené des recherches consistant à étudier les interactions entre logiciels et matériel afin d'optimiser au mieux des programmes écrits dans des langages à objets. Ces recherches se focalisent plus particulièrement sur l'optimisation des programmes Java et les interactions entre JVM et processeur [134] [25][29][27]. Le langage Java s'impose depuis plusieurs années avec une très grande rapidité, dans toutes les couches de l'industrie. Il rencontre cependant certains problèmes, notamment de lenteur. Cette voie de recherche a pour but d'étudier et d'améliorer les optimisations de code effectuées par les compilateurs, afin

de permettre aux développeurs informatiques de produire des programmes (traitements de texte, tableurs, navigateurs Internet, outils multimédia, contrôleurs embarqués, etc.) plus rapides, moins gourmands en mémoire et plus sûrs.

2.4.4. Lambda-calcul, calculs à objets et types

Les compétences de L. Liquori qui pourront être utiles à l'action Miró sont une (petite) connaissance du lambda-calcul, ses systèmes de types (dépendants, polymorphe et d'ordre supérieur) et ses logiques sous-jacentes via l'hysomorphisme de Curry-Howard, les calculs à objets purs et leurs systèmes de types, les systèmes de réécriture et en particulier le Rho-calcul ainsi que quelques connaissances dans le domaine des télécommunications grâce à son expérience chez Telecom Italia Labs (97-98) et a ses cours à l'École des Mines de Nancy (99-02).

2.5. Guide de lecture

La structure du document est la suivante : la section 2 présente les acquis de l'action. La section 3 présente les fondements scientifiques. La section 4 passe en revue les différents domaines d'application. La section 5 présente les logiciels développés ou en cours de développement. La section 6 présente le programme de recherche détaillé, ainsi que les résultats nouveaux. La section 7 présente les contrats industriels. La section 8 présente les actions régionales, nationales et internationales. La section 9 présente la diffusion des résultats au sein de la communauté scientifique. La lecture du présent document ne nécessite aucun pré-requis.

3. Fondements scientifiques

3.1. Introduction

Cette section présente les fondements scientifiques de Miró, le langage à objets Eiffel et son compilateur GNU SmallEiffel, les calculs à objets purs, dits *à prototypes*, les systèmes d'exploitations (à objets), les calculs de réécriture, et notamment le ρ-calcul, l'interaction entre les langages à objets et les langages fonctionnels, la théorie des types, les preuves formelles, ainsi que les assistants à la preuve.

3.2. Les langages et calculs à objets

3.2.1. Le langage Eiffel et le compilateur GNU SmallEiffel

Eiffel concrétise une certaine idée du développement de logiciels, qui repose sur le fait qu'il peut être effectué comme un ouvrage d'ingénieur, ayant pour but d'atteindre un certain niveau de qualité logicielle, au terme d'un processus rigoureux de production et d'enrichissement de composants réutilisables, scientifiquement spécifiés, paramétrables, et communiquant sur la base de contrats clairement définis et organisés selon des classifications multi-critères systématiques.

Si la notion de classe en programmation à objets trouve sa raison d'être dans une implantation des types de données abstraits, il ne suffit pas qu'une classe soit connue au travers des seules opérations qui lui sont applicables, mais aussi au travers les propriétés formelles de ces opérations. En Eiffel, le rôle des *assertions* est multiple : elles aident à produire du code correct et robuste, favorisent une documentation de haut niveau, fournissent un support de mise au point et servent de base au traitement des exceptions. L'action Miró focalise sa recherche sur Eiffel, dans le cadre du développement du compilateur *SmallEiffel*.

3.2.2. Le principe d'analyse globale et spécialisation des méthodes vivantes

Glossaire

Analyse globale Le fait de considérer le programme dans son ensemble (*whole system analysis*) par opposition à la compilation séparée.

Afin de *spécialiser* l'ensemble du programme, *c.à.d.* générer du code adapté autant que possible au contexte spécifique au programme compilé [64], l'équipe Miró s'intéresse souvent (dans le cadre du compilateur SmallEiffel) à une *approche globale* au système. Dans ce cas on peux s'appuyer sur des algorithmes d'analyse globale, de prédiction de type et d'expansion en ligne (*inlining*) puissants [70][133] [4][19].

L'*analyse globale* consiste à rassembler des informations sur l'ensemble du système compilé. Ceci va à l'encontre des pratiques actuellement les plus courantes pour la compilation des langages à objets, qui reposent sur la compilation séparée de fichiers sources indépendants, dont le principal inconvénient est qu'elle rend l'optimisation globale plus difficile. En effet, la compilation séparée est souvent considérée comme la seule technique permettant des (re)compilations rapides, grâce à son incrémentalité intrinsèque. Elle est de plus bien adaptée à l'utilisation de bibliothèques logicielles du marché fournies sans leur code source et intégrées lors de l'édition de liens. En revanche, son principal inconvénient est qu'elle ne permet d'optimiser que *localement*, à l'échelle du fichier source compilé. Elle ne permet ainsi pas de prendre en compte dans l'optimisation les fichiers et bibliothèques déjà compilés. L'analyse globale se focalise sur l'optimisation du système dans son ensemble. Elle a pour but de permettre au compilateur de préciser le *contexte* dans lequel chaque élément du code source (expression, instruction, variable, méthode, etc.) va être compilé. Ceci permet de passer de la génération d'un code le plus général possible à la production de code plus spécialisé, et donc potentiellement plus efficace. Afin d'avoir une analyse globale aussi rapide que possible, il est important de ne pas compiler tout le code source accessible (y compris dans les bibliothèques) sans discrimination, mais seulement le code indispensable au système, *c.à.d.* le code *vivant* de ce système.

1
```
deferred class FRUIT
feature
  stone_fruit : BOOLEAN is deferred end;
  display is
    do
      if stone_fruit then
        io.put_string("It is a stone fruit.%N");
      else
        io.put_string("It may be an apple.%N");
      end;
    end;
end
```

2
```
class PEACH
inherit FRUIT;
feature
  stone_fruit : BOOLEAN is true;
end
```

3
```
class APPLE
inherit FRUIT;
feature
  stone_fruit : BOOLEAN is false;
end
```

4
```
PEACHdisplay(PEACH* Current) {
  printf("It is a stone fruit.\n"); }
```

5
```
APPLEdisplay(APPLE* Current) {
  printf("It may be an apple.\n"); }
```

Figure 1. Spécialisation du code généré pour la méthode `display` *définie dans la classe* FRUIT. *Les cadres* ***1****,* ***2*** *et* ***3*** *correspondent respectivement au source Eiffel des classes* FRUIT, PEACH *et* APPLE. *Les cadres* ***4*** *et* ***5*** *montrent le code C généré pour les classes* PEACH *et* APPLE.

En Eiffel, le point de départ d'un programme, appelé la *racine* de l'application, est spécifié par une paire composée d'une classe initiale et de l'une de ses procédures de création. La première étape du processus

d'analyse statique globale calcule quelles parties du code source Eiffel sont *vivantes* (accessibles depuis la racine) ou *mortes* (inaccessibles). Ceci peux s'effectuer pour (comme *e.g.* en SmallEiffel) de façon totalement statique, *c.à.d.* sans analyse de flot. Le résultat de ce premier calcul est donc indépendant de l'ordre des instructions dans le programme.

Le processus de calcul du code vivant ainsi que de l'ensemble des types vivants est à la fois récursif et itératif. La récursivité est utilisée pour suivre le graphe d'appel de l'application. Le processus est répété itérativement en partant des méthodes vivantes de chaque type vivant jusqu'à être capable d'effectuer une itération complète sans ajouter ni nouveau type vivant ni nouvelle méthode vivante.

Par exemple en SmallEiffel, l'algorithme actuel de calcul du code vivant, même s'il donne des résultats plus qu'acceptables, n'est pas parfait. Par exemple, pour un appel `foo.bar`, lorsque le type statique `foo` de `foo` ne fait pas partie des types vivants, on ne devrait pas considérer la définition de `bar` dans ce type comme vivante car, dans certains cas, ce code peut amener l'algorithme à considérer à tort qu'un autre type est vivant. Cette constatation fait apparaître un besoin criant de pouvoir *garantir* que le calcul du code vivant est correct (et décidable avec une complexité acceptable). Une veine de recherche importante dans ce domaine est de formaliser le processus de calcul du code vivant dans le but de converger vers un implantation approchant plus précisément la partie réellement vivante de l'application. La *spécialisation automatique du code* en fonction du type du receveur est un point essentiel dans la stratégie de compilation de SmallEiffel [71]. Cette spécialisation consiste en quelque sorte à essayer de dérouler avant l'exécution le mécanisme de liaison dynamique, ce qui peut être considéré comme une forme d'évaluation partielle.

La figure 1 montre comment la méthode `display`, définie dans la classe FRUIT, héritée dans les classes PEACH et APPLE, est automatiquement spécialisée lors de la traduction en code C. En effet, comme le prédicat `stone_fruit` prend la valeur vrai, dans la classe PEACH et la valeur faux dans la classe APPLE, l'expression conditionnelle est toujours vraie dans la classe PEACH et toujours fausse dans la classe APPLE. Ainsi, il ne subsiste aucune trace de l'expression conditionnelle au niveau du code généré. En outre, dans le véritable code généré par SmallEiffel, l'argument `Current` (`self` de SmallTalk ou `this` de Java) des fonctions `PEACHdisplay` et `APPLEdisplay`, destiné à référencer le receveur, est supprimé car non utilisé dans le code spécialisé.

Ainsi, toutes les méthodes vivantes d'un type vivant sont également spécialisées. En raison de la duplication-spécialisation du code dans tous les types vivants, la pseudo variable `Current` a toujours un seul et unique type dynamique, le type vivant dans lequel cette méthode est spécialisée. Tous les appels concernant `Current` sont donc des appels directs ne nécessitant *jamais* de code de liaison dynamique. Le receveur étant en général l'objet le plus utilisé dans une méthode, cette optimisation concerne une très grande partie des envois de messages. Pour cette raison, les scores d'inférence de type atteignent des niveaux très élevés ($\sim 90\%$).

Contrairement à ce que l'on pourrait craindre, cette technique de duplication-spécialisation des méthodes n'entraîne pas d'augmentation importante de la taille des exécutables. En fait, les exécutables produits par SmallEiffel sont souvent plus petits que ceux produits par les compilateurs du commerce, voire plus petits que les exécutables obtenus à partir de classes C++ (cf. par exemple [4]). Ceci est dû en partie au fait que la spécialisation des méthodes tend à élaguer du code, comme dans l'exemple de la figure 1.

De plus, l'implantation de la liaison dynamique à l'aide de code de sélection dichotomique (*Binary Tree Dispatch*) permet le remplacement direct (*inlining*) de nombreuses méthodes. En particulier, toutes les fonctions d'écriture et de lecture d'attributs ne sont jamais ni appelées, ni même définies [19].

La technique de duplication-spécialisation est particulièrement bien adaptée à un contexte d'analyse globale où l'on ne considère que le code vivant. En particulier, le fait de travailler sur un système fermé permet de spécialiser le code de toutes les *classes feuilles*, les classes sans sous-classes dans le système fermé. Plus généralement, on appelle un *type feuille* un type vivant sans sous-type vivant dans le système fermé. Du point de vue spécialisation de code, le fait que le type d'une expression soit un type feuille n'est pas sans importance : comme dans le cas de `Current`, un envoi de message appliqué sur une expression dont le type est un type feuille correspond à un appel direct, sans liaison dynamique. La seule différence entre `Current` et une expression de type feuille consiste à envisager également le cas où l'expression ne référence aucun objet lors de l'exécution (`Void`).

3.2.3. Implantation de la liaison dynamique par sélection dichotomique

La connaissance de l'ensemble des types vivants offre d'importantes possibilités d'optimisations du code généré. Une des plus remarquables optimisations réalisée dans le cadre du compilateur SmallEiffel a consisté à implanter la liaison dynamique en générant du code de sélection dichotomique. Tous les types vivants étant connus, on associe statiquement à chacun de ces types un identifiant entier qui sert d'indicateur à l'exécution. La détermination du type dynamique de l'objet receveur est réalisée par du code de sélection dichotomique comme présenté à la figure 2.

```
if ( typeId < 3 )
   then
      if ( typeId < 2 ) then specialized code for type #1
                        else specialized code for type #2
      endif
   else
      if ( typeId < 4 ) then specialized code for type #3
      else specialized code for type #4
      endif
endif
```

Figure 2. Le code généré pour implanter la liaison dynamique avec sélection dichotomique.

Le principal avantage de cette technique est d'éviter l'utilisation de tableaux de pointeurs de fonctions, *Virtual Function Tables* (VFT), comme en C++ par exemple. En outre, il est souvent possible de remplacer un appel de fonction par le code qui lui correspond (*inlining*) au cœur même du code de sélection dichotomique (voir [19]).

3.2.4. Les calculs à prototypes et leurs systèmes de types

Parmi les langages à objets, les langages à prototypes [110] (ou langages à délégation) constituent un axe de recherche important de l'action Miró ; en particulier, on s'intéresse souvent aux aspects théoriques de ces langages, notamment aux *calculs à objets*, calculs où la création des nouveaux objets est déléguée aux objets eux-mêmes (calculs à délégation ou *delegation-based*). Ces calculs sont très importants pour fournir des sémantiques opérationnelles, dénotationnelles, et des systèmes de types sophistiqués pour les langages à objets.

Le *Lambda Calcul des Objets (LCO)* défini par Fisher, Honsell et Mitchell [11] à l'université de Stanford, et le *Calcul à Objets (OC)* défini par Abadi et Cardelli [1] au Centre de Recherche Digital de Palo Alto en sont les principaux représentants. En particulier, les aspects de typage et d'implantation des dits calculs ont été développés dans [14][15][12][26]. [46][98][99][43][96][44][57]

Dans les calculs à objets purs, les objets sont définis directement à partir d'autres objets, en utilisant ces derniers comme prototypes. Les seules opérations sur les objets sont l'extension d'un objet avec une variable ou une méthode (*object extension*), et la surcharge d'une variable ou d'une méthode (*object overriding*). L'objet modifié hérite de toutes les propriétés du prototype. Plusieurs modèles fonctionnels et impératifs avec différents systèmes de typage ont été présentés dans les dernières années [106][114][113][83][55][45][123][122].

Les calculs LCO et OC conduisent à divers systèmes de types qui empêchent statiquement l'erreur `message-not-found` à l'exécution, qui apparaît quand un objet reçoit un message qui n'est pas présent dans son interface. Les systèmes de types pour LCO et OC sont très puissants : en particulier ils permettent la *mytype method specialization*, *c.à.d.* la possibilité de spécialiser les types des méthodes héritées (une telle possibilité est déjà offerte par le langage Eiffel, avec le type `like Current`, qui malheureusement a un système de types

qui ne prévient pas statiquement l'erreur `message-not-found`). Avec le calcul à prototypes on peut modéliser des langages comme Self [18], Obliq [59], Kevo [128], Emerald [120], Cecil [63], et Omega [42].

Les langages à prototypes (et ses calculs sous-jacents) peuvent aussi être utilisés en tant que « langages cibles » pour implanter et étudier des propriétés formelles des langages à classes, puisque les classes peuvent être vues comme des objets capables de recevoir le message `new` de création d'un objet.

3.2.5. *Fonctions vs. objets : un conflit sans cause*

Mots clés : *Self*, *O-Haskell*, *Claire*, *Moby*, *Obliq*, *OCaml*, *Eiffel*.

Glossaire

Self Langage à objets où tout est objet (pas de classe, rien que des objets) [18].

O-Haskell Version à objets de Haskell [117][111], qui est un langage fonctionnel pur, *c.à.d.* sans la possibilité d'avoir accès à la représentation interne des structures en mémoire (pas d'effet de bord).

Claire Langage fonctionnel, logique et à objets développé chez Bouygues Telecom par Y. Caseau et F. Laburthe [62].

Moby Langage de programmation à prototypes en cours de développement chez AT & T [84][86][85].

Obliq Langage de programmation à prototype développé chez Digital/Compaq. Ce langage est conçu pour la mobilité du code [59].

OCaml Langage fonctionnel et à objets développé dans le projet INRIA Cristal [73].

Eiffel Langage à base de classes statiquement typé intégrant depuis peu certains paradigmes habituellement réservés aux langages purement fonctionnels (fermeture et évaluation partielle).

Les langages à objets sont quasi universellement considérés comme intrinsèquement impératifs : en fait ils ont une notion d'état, *c.à.d.* de variables propres à chaque instance d'objet. Néanmoins :

- une grande partie de la littérature scientifique sur la théorie des langages à objets a été développée au travers de petits langages à objets fonctionnels purs (voir par exemple, à ce sujet, le papier sur *Featherweight Java* de A. Igarashi, B. Pierce et P. Wadler [95]) ;
- des concepts comme l'héritage, la liaison dynamique, l'envoi de message, la surcharge de méthodes et l'extension dynamique des objets ne sont pas du tout en contradiction avec le paradigme fonctionnel lui même.

L'un des objectifs de Miró consiste à explorer la possibilité de « concilier » le monde objet et le monde fonctionnel, tout en gardant l'esprit de l'un et l'élégance mathématique de l'autre.

3.3. La construction de logiciel certifié

Une partie de nos recherches concerne la certification de logiciel. Nous nous intéressons notamment à la spécification des langages de programmation et aux preuves de propriétés de langages (dont deux exemples typiques sont la preuve de conservation des types d'un langage lors de son exécution, et la preuve de correction d'un compilateur). Nous travaillons dans ce domaine à différents niveaux (théorie et pratique) et en suivant différentes approches.

3.3.1. *Formalisation des objets à travers la réécriture d'ordre supérieur*

Mots clés : *Lambda-Calcul*, *Rho-calcul*, *sémantique*, *théorie de types*.

Glossaire

ρ-calcul Calcul de réécriture d'ordre supérieur [65] [21][22].

Le ρ-calcul peut être utilisé pour formaliser et intégrer le paradigme à objets et la réécriture. Le ρ-calcul intègre les propriétés complémentaires de la réécriture du premier ordre et du λ-calcul ainsi que des caractéristiques permettant d'exprimer le non-déterminisme. Ce calcul est suffisamment puissant pour décrire non seulement la réécriture avec des règles conditionnelles mais aussi leur contrôle. Ainsi, le ρ-calcul permet la description des langages basés sur la réécriture. La syntaxe du ρ-calcul est très simple :

$$\begin{array}{rcll} t & ::= & a \mid X \mid t \longrightarrow t \mid t^{\bullet} t \mid & termes \quad simples \\ & & null \mid t, t & sequences \end{array}$$

où a dénote une constante, X dénote une variable, $t \longrightarrow t$ une fonction, $t^{\bullet} t$ une application, $null$ dénote une séquence vide, et t, t une séquence. Dans le ρ-calcul, l'application est une opération décrite au même niveau du calcul que l'abstraction, *c.à.d.* les règles de réécriture. On obtient ainsi un calcul similaire au λ-calcul avec motifs de S. P. Jones [118] où l'abstraction est faite non seulement par rapport à une variable mais en considérant aussi le contexte de la variable.

Le ρ-calcul peut être employé pour donner une sémantique à l'application des règles du langage Elan [47], un environnement de programmation et de prototypage reposant sur la logique de réécriture associée à la notion de stratégie.

$$\begin{array}{rcl} Point & \triangleq & val \to S \to v(1\ 1), \\ & & get \to S \to S.val, \\ & & set \to S \to v(X\ Y) \to (S.val := S' \to v(X\ Y)) \\ PClass & \triangleq & new \to S \to (val \to S' \to (S.preval)^{\bullet} S', \\ & & \qquad get \to S' \to (S.preget)^{\bullet} S', \\ & & \qquad set \to S' \to (S.preset)^{\bullet} S'), \\ & & preval \to S \to S' \to v(1\ 1), \\ & & preget \to S \to S' \to S'.val, \\ & & preset \to S \to S' \to v(X\ Y) \to (S'.val := S'' \to v(X\ Y)) \\ & \text{où} & \\ obj.m & \triangleq & obj^{\bullet} m^{\bullet} obj \quad \text{Kamin } \textit{self-application} \\ (a.m := b) & \triangleq & (m \to b, a) \ \textit{update} \text{ fonctionnel} \\ (a.m := b) & \triangleq & (m \to b, kill_m{}^{\bullet} a) \ \textit{update} \text{ impératif grâce à } kill_m \\ kill_m & \triangleq & (X, m \to Z, Y) \to X, Y \\ & \text{et} & \\ PClass.new & \mapsto & Point \end{array}$$

Figure 3. Codage d'un objet $Point$ et d'une classe $PointClass$ en ρ-calcul [1].

Le ρ-calcul est également un excellent candidat pour étudier l'expression du paradigme à objets dans un calcul de réécriture ; en fait, l'interprétation dénotationnelle de l'appel d'une méthode m sur un objet obj dans la programmation à objet (Kamin [13]) peut être simplement représentée en ρ-calcul en terme de filtrage et application. La figure 3 présente, par exemple, le codage en ρ-calcul d'un simple objet $Point$ et d'une simple classe $PointClass$ [1] (on présente soit un codage fonctionnel, soit un codage impératif, grâce au ρ-term $kill_m$, qui sélectionne une méthode dans l'objet et l'élimine).

3.3.2. La théorie de types

Mots clés : *Théories typées*, *Calcul des Constructions (co-)Inductives*.

Glossaire

Calcul des Constructions (CC) Système de types dépendants, polymorphes et d'ordre supérieur pour le λ-calcul, introduit par T. Coquand et G. Huet [72]. Ce système, étendu avec les types (co-)inductifs (ICC) [131], constitue le noyau du système Coq.

Les théories typées, grâce à l'isomorphisme de Curry-Howard, [93] fournissent une base théorique aux assistants à la preuve. Suivant cet isomorphisme, une propriété à démontrer est spécifiée par un type et une démonstration de cette propriété devient un λ-terme (*c.à.d.* un programme fonctionnel) ayant ce type. On voit que la puissance d'expression de la théorie choisie a une grande influence sur le niveau d'abstraction avec lequel l'utilisateur pourra exprimer la propriété à prouver, comme sur la facilité (ou la difficulté !) qu'il aura à faire sa preuve. Ces dernières années ont vu différentes tentatives pour enrichir les théories typées connues, par l'ajout de modules, de sous-typage, de types inductifs, de types quotients et même de types objets. L'un des axes de recherche courants est la recherche de nouvelles théories typées permettant la récursion sur des termes de type fonctionnel.

3.3.3. Le logiciel certifié

Mots clés : *Preuves formelles*, *logiciel certifié*.

> *Les logiciels sont de plus en plus complexes. Hier encore, un logiciel représentait quelques milliers de lignes de code, tandis qu'aujourd'hui de façon courante ces produits en nécessitent plusieurs millions, ce qui entraîne une quantité plus importante d'erreurs de programmation. Une étude menée par le National Institute of Standards and Technology évalue à 60 milliards de dollars par an le coût des bogues informatiques, dont 22 milliards de dollars pourraient être évités si l'industrie informatique améliorait ses équipements de test des logiciels. Dans ce secteur qui emploie 700 000 ingénieurs et dont le chiffre d'affaires atteint 180 milliards de dollars, les développeurs consacrent environ 80% de leurs dépenses à identifier et à corriger des erreurs de programmation, mais cela ne suffit pas à empêcher la multiplication des bogues [141].*

Dans le domaine du logiciel certifié, on distingue en général la validation d'outils généraux pour les langages de programmation (tels que des interpréteurs, ou des compilateurs) de la validation des programmes eux-mêmes.

Dans le premier cas, on parle de *preuve de propriétés de langages*. Deux exemples typiques de ce domaine sont la preuve de conservation de types d'un langage lors de son exécution, et la preuve de correction d'un compilateur. La première propriété assure une certaine cohérence entre le système de vérification de type et les règles d'évaluation d'un langage. Ce type de preuve, essentiel pour les concepteurs du langage, donne par ailleurs au programmeur une certaine confiance dans les outils qu'il utilise.

La *preuve de correction de programmes* consiste généralement à prouver certaines propriétés d'un programme, par exemple la validité d'un invariant. Cependant, dans le cas idéal où l'on peut extraire un programme à partir de sa spécification, la preuve de correction de programmes devient *programmation certifiée*. À l'intersection des deux domaines, on trouve la certification d'un interpréteur, ou d'un compilateur.

Plus généralement, les compétences sur les preuves formelles intéressent potentiellement l'ensemble des domaines industriels où la présence d'erreurs même minimes dans les programmes peut avoir des conséquences graves : danger pour la vie humaine comme dans l'énergie nucléaire ou les transports (avions, automobiles), la chirurgie assistée par ordinateur, ou simplement coût prohibitif comme dans le commerce électronique et les réseaux de télécommunication, l'aérospatial, les *SoC, i.e. Systems on Chips*, les cartes à puces ou bien dans les domaines où les enjeux financiers sont importants comme le domaine bancaire. Tous ces domaines ont besoin de *logiciels certifiés*.

De nombreuses équipes de recherche mettent au point des techniques qui permettent de développer des logiciels validés vis-à-vis de spécifications, ce qui permettrait « théoriquement » de réduire le risque d'erreur à zéro (en « pratique » des erreurs peuvent également se glisser dans les spécifications). Ces techniques sont plus ou moins coûteuses et plus ou moins générales. Avec le logiciel Coq, fondé sur une théorie typée du Calcul des Constructions (co-)Inductives, on se trouve à un bout du spectre : on peut aborder des problèmes variés et complexes mais avec un coût de développement plus important que d'habitude[1]. Il est donc intéressant de diminuer ce coût, d'une part par la recherche de théories typées et de méthodes de formalisation permettant des spécifications et des preuves plus concises, d'autre part par le développement de bibliothèques d'exemples.

3.3.4. Les assistants à la preuve

Mots clés : *Coq*, *PCoq*, *PVS*, *Isabelle*, *HOL*, *Twelf*.

Glossaire

Coq Assistant à la preuve fondé sur le Calcul des Constructions (co-)Inductives [94].

PCoq Interfaces graphiques pour Coq permettant le *proof by pointing c.à.d.* utiliser la structure des formules logiques pour aider l'utilisateur à effectuer les étapes du raisonnement en les désignant à la souris [41][97].

PVS Assistant à la preuve disposant de stratégies de développement de preuve automatiques pour le model-checking [125].

Isabelle Assistant de preuves fondé sur la logique d'ordre supérieur [115][116] permettant des preuves plus courtes que le système Coq grâce à la réécriture et à l'unification d'ordre supérieur. Puisque la meta-logique d'Isabelle est nettement moins expressive que celle de Coq, les preuves Coq les plus abstraites ne peuvent pas s'exprimer dans Isabelle.

HOL Assistant de preuves fondé aussi sur la logique d'ordre supérieur [112].

Twelf Assistant de preuves qui utilise la syntaxe abstraite d'ordre supérieur et le principe d'induction [139].

Les assistants à la preuve sont très utilisés ces dernières années pour essayer de démontrer des propriétés très difficiles comme, par exemple, la normalisation forte d'un calcul ou la conservation de types pour un langage de programmation. Depuis peu, on s'intéresse aussi à la production de *logiciels sûrs*. Dans ce domaine, les systèmes basés sur une théorie typée vérifiant l'isomorphisme de Curry-Howard sont non seulement *a priori* les plus fiables (puisque leurs fondements sont connus), mais aussi les plus puissants, en terme d'expressivité. Dans le cas où l'on peut extraire un programme à partir de sa spécification, la preuve de correction de programme devient *programmation certifiée*. La spécification d'un langage de programmation se prête bien à être formalisée à l'aide de propriétés mathématiques directement transposables dans les démonstrateurs de théorèmes. La preuve de correction d'un compilateur (qui est lui-même un logiciel) représente un exemple typique de l'utilité des démonstrateurs de théorèmes.

Parmi les assistants à la preuve, l'un des plus prometteurs semble être le système Coq : avec son système de types basé sur les types (co-)inductifs, il permet la spécification des langages (exprimés à l'aide des grammaires inductives) et de leurs systèmes de types. Avec Coq, ont été développées récemment beaucoup de bibliothèques : la preuve des propriétés des programmes impératifs, la modélisation de la *scope extrusion* dans le π-calcul, ainsi que la normalisation forte d'une restriction de Coq lui-même (voir le papier « *Coq in Coq* » de B. Barras [39], une sorte de mini-bootstrap du système lui même), etc.

Le système PCoq, interface homme machine dédiée au système Coq, développée dans le projet Lemme permet le *proof by pointing*, *c.à.d.* que l'environment utilise la structure des formules logiques pour aider systématiquement l'utilisateur à effectuer les étapes de base du raisonnement en les désignant à la souris. Une autre interface pour Coq est proofGeneral [36] développée au LFCS d'Edinburgh.

Pour résumer, notre intérêt sera focalisé :

[1] Les techniques de *Model Checking* sont à l'autre bout du spectre.

- sur les aspects théoriques, voire fondateurs, de ces outils [9][17][21][22][20] ;
- sur les aspects d'utilisation de ces outils, car l'un des objectifs principaux de Miró est la construction d'un interprète et d'un compilateur sûr et certifié, pour un langage à objets.

3.3.5. Méthodes de spécification dans les assistants à la preuve

Aussi surprenant que cela puisse paraître, il n'y a toujours pas de consensus sur le choix de la représentation des variables dans un langage fonctionnel. On dispose à l'heure actuelle de plusieurs familles de méthodes pour décrire les variables et leur manipulation, *c.à.d.* essentiellement la substitution, à α-conversion près (un terme est défini modulo le renommage de ses variables). L'action Miró expérimente toutes ces méthodes sur différents langages dans le système Coq.

La technique la plus connue est celle des indices de N. de Bruijn. Les variables sont implémentées par des entiers, représentant leur profondeur dans le terme, le nombre de liaisons entre leur occurrence et la racine du terme. Par exemple, les termes $\lambda x.x$ et $\lambda x.\lambda y.x$ seront respectivement représentés par les termes $(lam \quad 0)$ et $(lam \quad (lam \quad 1))$. Les termes sont bien définis à α-conversion près, mais ces termes étant eux-mêmes déjà illisibles, les spécifications le sont naturellement aussi, d'autant plus qu'un tel terme doit être complètement récrit lorsqu'il est plongé dans un autre terme (les entiers doivent tous être décalés) ! Ce problème rend les preuves arbitrairement compliquées. Dans la plupart des cas, les trois quarts du développement concernent la manipulation des indices de N. de Bruijn, et non la sémantique elle-même.

La preuve de la propriété de conservation de types (simples) pour un π-calcul polyadique semble déjà déraisonnable, sinon impossible, dans cette approche. Cette technique, encore très utilisée, paraît donc hautement inadaptée à la formalisation des langages de programmation. Cependant, il existe un cas où l'utilisation des indices de N. de Bruijn est tout à fait indiquée : le cas du *bootstrap* d'un langage implémenté en utilisant ces indices. C'est le cas de la formalisation de « *Coq in Coq* », réalisée par B. Barras.

G. Gillard [89][90] a exploré une technique due à A. Gordon [91]. Cette méthode, à l'origine implémentée dans le système HOL, consiste à spécifier les termes à α-conversion près, sur une syntaxe utilisant des indices de N. de Bruijn (qui disparaissent dans la syntaxe finale). La méthode permet des spécifications relativement proches de ce que l'on écrit sur le papier, une fois prouvé un ensemble de lemmes sur les spécifications contenant des indices de N. de Bruijn.

J. McKinna et R. Pollack [105][129] utilisent pour spécifier des règles de typage une technique de premier ordre intéressante qui distingue les variables libres (formalisées par des *paramètres*) des variables liées (formalisées par des *variables*). Les descriptions sémantiques renomment judicieusement les variables d'un terme en paramètres avant tout traitement des sous-termes, pour effectuer la substitution inverse à la fin du traitement, évitant ainsi le problème de « capture de variable ». Leur approche nous semble être aujourd'hui l'une des meilleures.

Ainsi, il ne peut pas y avoir de phénomène de capture de variable lors d'une substitution (l'introduction d'une variable déjà liée dans un terme nécessite normalement son *renommage* préalable pour éviter toute capture de variable libre). J. McKinna et R. Pollack ont ainsi formalisé "*Lego in Lego* » [102][119] (Lego et Coq sont des systèmes très voisins).

Enfin, il existe une méthode qui permet d'utiliser directement les fonctions du système choisi (Coq par exemple) pour formaliser les notions de variables liées et de substitution du langage à décrire. On appelle ce type de description une spécification d'ordre supérieur. L'idée de base de la méthode est de représenter les variables du langage spécifié (le langage dit « objet ») par les variables du langage utilisé pour spécifier (le langage dit « méta »). Ainsi, un constructeur d'un langage L sera défini par des déclarations du type : $lam : (L \Longrightarrow L) \Longrightarrow L$. Le fils d'un nœud lam est ici une fonction (méta). Cette technique permet souvent les descriptions et les preuves les plus concises et les plus élégantes.

Cette technique permet également de formaliser les *side conditions* (variables libres dans une hypothèse) et les changements de contexte (déchargement d'une hypothèse en déduction naturelle) dans les règles d'inférence. Elle fournit souvent la description la plus concise et la plus élégante d'un langage. De plus, par voie de conséquence, les preuves sont aussi grandement facilitées. Dans cette approche, on économise

même la preuve du lemme de substitution, cœur techniquement difficile, commun à toutes les preuves de propriétés d'un langage fonctionnel. L'utilisateur peut se concentrer sur les difficultés de son langage, sans devoir ré-implémenter les fonctions que le système a déjà implémentées pour lui.

Le problème est que les schémas de récurrence et les principes d'induction usuels ne s'appliquent pas aux termes définis en suivant cette approche. La solution à long terme passe par la conception d'une nouvelle théorie typée. La solution à court terme consiste à inventer de nouvelles méthodes de description des langages dans les systèmes existants. J. Despeyroux a proposé différentes solutions, en suivant ces deux voies (voir section 2.4).

Pour conclure sur ces différentes méthodes de description des liaisons dans les langages de programmation, la méthode due à J. McKinna et R. Pollack nous semble aujourd'hui la meilleure pour décrire un langage fonctionnel en l'absence d'un système permettant l'utilisation conjointe de la syntaxe abstraite d'ordre supérieur et de l'induction. La technique de spécification d'ordre supérieur est la plus abstraite, la plus élégante ; c'est celle que nous préférerons toujours chaque fois que ce sera possible. Cette technique ne nous semble pas encore avoir fait ses preuves pour les langages impératifs, bien qu'il existe plusieurs expériences dans ces langages. Cependant la recherche est encore active sur ce sujet, que nous avons inclus dans nos objectifs de recherche. Ceci dit, cette méthode ne sera vraiment fiable que lorsque l'on disposera d'un système de type dans lequel les principes de récursion et d'induction sur des termes de type fonctionnel (termes usuels dans les spécifications à l'ordre supérieur) pourront être formalisés (et prouvés corrects). Enfin, A. Pitts [88] a introduit récemment une méthode du premier ordre, consistant à formaliser les variables à α-conversion près, qui semble très intéressante.

4. Domaines d'application

4.1. Introduction

Mots clés : *PDA*, *PDA-OS*, *téléphones mobiles*, *Mobile-OS*, *programmation sûre*, *logiciel certifié*, *code mobile*, *internet*, *télécommunications*.

Les domaines concernés sont : les ordinateurs de poche et leurs systèmes d'exploitation, les téléphones mobiles, les télécommunications, les systèmes d'exploitation en général, la génération de code pour les micro-contrôleurs synthétisables, l'électronique embarquée, les *SoC, i.e. Systems on Chips*), ou bien dans des domaines où les enjeux financiers sont importants (comme l'aérospatial, les télécommunications ou le domaine bancaire) et comme dit précédemment, tous les domaines touchant aux logiciels critiques.

4.2. Électronique embarquée

4.2.1. PDA et PDA-OS

Participants : Jérôme Boutet, Dominique Colnet, Benoît Sonntag.

L'évolution des technologies et du matériel permet aujourd'hui d'intégrer des outils informatiques performants dans des appareils divers et de petites tailles. Nous nous intéressons particulièrement à l'expansion massive des agendas électroniques.

Actuellement nous observons un phénomène de diversité comparable à celui du début des ordinateurs personnels. Cette diversité se situe à tous les niveaux de cette nouvelle informatique. Nous constatons une variété au niveau des processeurs, des systèmes d'exploitation et de l'interface utilisateur qui reflètent les différentes possibilités du matériel : commande par clavier, écran tactile ou reconnaissance vocale.

Dans ce domaine, deux grandes tendances se dégagent aujourd'hui : les applications écrites en langage C pour les performances, ou dans un langage interprété pour la portabilité. La puissance qu'offrent les *palms* est certes impressionnante, mais reste un sujet de préoccupation pour le développeur. Il nous paraît donc indispensable de lier performance du code compilé spécifique à un processeur avec portabilité et standardisation de l'interface de programmation. La sûreté de ce genre d'application paraît être une priorité.

Nous n'avons pas encore de solution et de direction très claire dans cette type d'applications, mais nous pensons que le langage Eiffel et, peut être, le langage FunTalk ainsi que le système Isaac pourront nous être utiles sur ce type de plate-formes.

4.2.2. Systèmes d'exploitation

Mots clés : *Isaac*, *Self*, *prototype*, *Architecture à Blackboards*.

Participants : Dominique Colnet, Benoît Sonntag.

Glossaire

Isaac Système d'exploitation entièrement à objets dynamiques et prototypes [23],[127].
Lisaac Langage à prototypes, conçu pour l'implantation de l'OS à objets Isaac [24].

Dans l'action Miró, nous explorons la possibilité d'intégration de concepts objets au cœur même des systèmes d'exploitation. Cette branche de recherche fait l'objet de la thèse de B. Sonntag. L'évolution des technologies et du matériel nous amène à revoir la conception du noyau des systèmes d'exploitation actuels. Nous pensons que l'introduction de concepts objets au cœur même des systèmes d'exploitation simplifierait et optimiserait l'utilisation des divers composants physiques d'un ordinateur. De manière générale, notre démarche consiste à repenser la définition d'un noyau système en intégrant des concepts objets, voire multi-agents.

Nous sommes aujourd'hui en mesure d'affirmer que ce type de démarche apporte un réel progrès aussi bien dans l'adaptation dynamique du système que dans la simplification et le pouvoir expressif du code. De plus, notre approche a des répercussions notoires pour l'utilisateur de ce type de système. L'ultime attente réside en sa capacité de s'adapter, et de s'auto-gérer pour concevoir automatiquement des liens entres les objets (liens de clientèle, mais aussi d'héritage), qui permettront de construire une application dynamiquement selon les actions et les types de données traitées par l'utilisateur à un instant donné.

Pour mener à bien nos objectifs, sans délaisser les performances globales du système, nous avons commencé par étudier les architectures de processeurs (notamment Intel x86). Cette étude avait deux objectifs : permettre une portabilité rapide du système et une utilisation approfondie des capacités du processeur pour la mise en place des mécanismes objets. Parmi les différents concepts des langages à objets existants, nous avons choisi de privilégier les prototypes, et de construire un langage à prototypes, Lisaac, permettant l'élaboration du système.

Parallèlement à ce développement, B. Sonntag a conçu et réalisé avec ce langage un système d'exploitation expérimental appelé *Isaac*, *entièrement objet* et accédant directement le matériel sans aucune autre couche intermédiaire. Ainsi, il n'utilise le BIOS d'un PC qu'au moment du boot. Isaac possède aussi une interface graphique de haut niveau et pourra supporter l'ensemble des produits GNU via une simple compilation. Il est suffisamment avancé et compact pour servir de base à l'incorporation rapide de nouveau concepts et à l'étude de nombreux problèmes liés à l'interaction entre les objets et les systèmes d'exploitation. De plus, les premières réalisations menées par B. Sonntag semblent montrer qu'un tel système d'exploitation est extrêmement compact et se prête particulièrement bien aux systèmes embarqués.

Enfin, une formalisation du langage propriétaire à la base du système Isaac est en cours d'étude. Nous pensons que ce langage pourrait être formalisé avec une sémantique opérationnelle relativement simple car le langage a une taille raisonnable (une vingtaine de constructeurs).

4.3. Télécommunications

Participants : Claudio Casetti [Polytechnique de Turin], Luigi Liquori.

Parmi les problématiques plus intéressantes dans le cadre des applications écrites dans un langage à prototypes, on rappelle la problématique du « logiciel reconfigurable » (ou de l'*hot swapping*), *c.à.d.* la possibilité d'une application de se réconfigurer à *run-time*, par exemple, à travers la modification du corps d'une méthode suite au soulèvement d'une exception.

En outre, nous pensons que la communauté des télécommunications est prête à introduire les méthodes formelles dans le cycle de développement des logiciels. Comme exemple, nous citons (en anglais) une

contribution de C. Casetti et L. Liquori à une expression d'intérêt (EoI) envoyée à la Communauté Européenne pour la formation d'un réseau d'excellence (NoE) nommé *Design and dimensioning of the 3rd Generation Internet* (appel à proposition prévu début 2003).

Proposed contribution to the NoE By C. Casetti (Politecnico di Torino) and L. Liquori (INRIA)

> « It is well known that models/standard/protocols in telecommunications are written using not sophisticated type systems or languages, such as *e.g.* ASN.1 or GDMO : as a simple example, the semantics between Managers and Agents is defined in terms of Managed Objects, specified according to the GDMO standard language ; for each management domain, the proper information model is defined : information between Manager and Agent, exchanged in the form of operations, is given in *plain english* ; this "way of life" is error-prone and leads to many and dangereous errors that usually can be discovered by the engineers whose task is to *implement* protocols/behaviours/objects into suitables software objects or Object Oriented Simulation Environment. In the best of scenarios, ambiguities and criticism will come out in this simulation stage.
>
> We think that the community of researchers involved in developing standards, protocols and applications for telecommunication is now mature to use formal methods (based on logics, mathematics, lambda calculi, type systems, rewriting, statics analysis, ...) and proof assistants (such as Coq, Isabelle, PVS, B, ...) to design / check / validate / certify / develops protocols / advanced-services / applications / emulators in the 3rd generation internet. The major cost in the design phase (it is well understood that a fairly concrete mathematical model costs more in terms of human resources than a UML chart or an ITU reccommendation) will benefit from a major "solidity » of the specification of protocols and related software ; in case of some proof assistant the proved specifications can be turned directly into executable code (C in case of a B-development, OCaml in case of Coq-development).
>
> We are of course far from a "democratization » of such proof assistants (which can actually be used only by experts) but their recent use in the setting of formal certification of critical applications such as smartcard, embedded software, compilers, protocols for cryptography, Java language ... shows that in the near future their use will be as widespread as the use of C++ libraries or design patterns or object orientation.
>
> Our (modest) contribution to the NoE will be to foster the idea that 3rd Generation Internet should put special emphasis on formal methods and formal certification, by showing their application to the development of advanced protocols in the TCP/IP stack »

5. Logiciels

5.1. SmallEiffel : The GNU Eiffel Compiler

Participants : Dominique Colnet, Olivier Zendra, Philippe Ribet.

Les buts éminemment pratiques de nos travaux sur SmallEiffel nous ont dès le début amenés à nous poser la question de la validation expérimentale de nos idées. Il nous est fort naturellement apparu qu'une des meilleures façons de le faire consistait à les implanter dans un compilateur créé *ex nihilo*, dont les deux objectifs principaux étaient qu'il soit capable de compiler vite, tout en générant du code efficace. Afin de mettre nos travaux à disposition du plus grand nombre, nous avons choisi de diffuser SmallEiffel comme logiciel libre (http://SmallEiffel.loria.fr).

SmallEiffel nous a ainsi servi d'outil expérimental dans lequel nous avons pu implanter nos idées et valider nos hypothèses, ainsi que de vecteur de diffusion auprès de la communauté scientifique et du public. Mais il a aussi constitué un sujet d'observation, un banc d'essai significatif, extrêmement intéressant et plein d'enseignements, facilitant ainsi l'émergence de nouvelles idées. En effet, rappelons que SmallEiffel est complètement auto-compilé (*bootstrapped*) en Eiffel et représente dans sa version `-0.74` environ 110 000 lignes d'Eiffel pour environ 300 classes vivantes, ce à quoi il convient d'ajouter l'ensemble de la série de programmes utilisés pour valider le compilateur, soit environ 2500 classes représentant 250 000 lignes d'Eiffel.

Intégrant les techniques d'optimisation développées dans la thèse d'O. Zendra, SmallEiffel produit un code d'excellente qualité, dont les performances sont en général supérieures (voire très supérieures) à celles des

autres compilateurs Eiffel disponibles dans le commerce. Il se compare également très favorablement aux compilateurs C++.

La très bonne qualité du code généré par SmallEiffel permet l'obtention d'exécutables de très petite taille, grâce notamment à la *compilation sur nécessité* et la *spécialisation des primitives*. De même, cette spécialisation est indéniablement à l'origine de la rapidité des exécutables produits. SmallEiffel étant écrit en Eiffel, ses exécutables bénéficient eux aussi des optimisations appliquées aux programmes compilés. Ses propres performances, en terme de mémoire et surtout de temps de compilation sont elles aussi excellentes, comparées aux autres compilateurs Eiffel ou à des compilateurs C++.

Ces qualités montrent donc l'efficacité des techniques sur lesquelles nous avons focalisé notre travail. Elles montrent également que les objectifs initiaux (processus de compilation plus efficace, code généré plus efficace, portabilité) ont été atteints de façon fort satisfaisante.

Depuis juillet 2002 le logiciel SmallEiffel s'appelle *SmartEiffel* (http://SmartEiffel.loria.fr).

5.2. FunTalk / SmallTalk2K

Participants : Alberto Ciaffaglione, Luigi Liquori, Jöelle Despeyroux, Olivier Zendra.

Nous avons commencé la définition du langage SmallTalk2k/FunTalk. Un simple interprète non typé (réalisé en Eiffel) pour FunTalk a déjà été implanté. Le système de types sera prochainement intégré dans l'interprète [74][124]. A. Ciaffaglione a terminé la formalisation d'un sous-ensemble de FunTalk, essentiellement **imp**ς de Abadi et Cardelli, en Coq : on a formalisé la sémantique opérationnelle et un système de type du premier ordre avec sous-typage.

5.3. Isaac / Lisaac

Participants : Jérôme Boutet, Dominique Colnet, Benoît Sonntag, Olivier Zendra.

Le *challenge* actuel du système Isaac est celui d'établir une passerelle fiable entre les outils présents sous UNIX et le système lui même. En effet, pour que notre système d'exploitation ne reste pas un système « jouet » et expérimental, il est indispensable d'établir des passerelles et de l'ouvrir par compatibilité avec d'autres systèmes déjà présents. Une première documentation du système Isaac se trouve sur la page web http://www.loria.fr/~bsonntag. L'étudiant C. Proch [31] a sensiblement amélioré la syntaxe et la sémantique opérationnelle de Lisaac. Un interprète avec debogueur a été implanté.

5.3.1. Portage de Isaac sur PDA.

La commission nationale INRIA des postes d'accueil d'ingénieur associé 2002, a décidé de retenir notre proposition du portage du système d'exploitation Isaac sur une architecture différente de la gamme des processeurs Intel (architecture x86). En particulier on focalisera notre intérêt sur les architectures utilisées dans les assistants personnels (PDAs). J. Boutet travaillera, à partir de septembre 2002, sur le portage de la `GlibC` sur Isaac (pour augmenter rapidement et avec un effort limité le nombre d'applications disponibles sur Isaac), et sur le portage d'Isaac lui même sur les architectures StrongARM, Motorola MC68K et DragonBall.

6. Résultats nouveaux

6.1. Introduction

N.B. Avant d'énumérer les résultats nouveaux obtenus en 2002, je (L.L.) pense profitable de décliner également notre programme de recherche, qui se focalise essentiellement sur les axes suivants :

1. l'étude, la définition et l'implantation certifiée d'un langage de programmation à classes (et de son compilateur) , appelé *SmallTalk2K*, et d'un langage à prototypes (*c.à.d.* à objets purs), appelé *FunTalk* (et d'un interprète et d'un compilateur), langage intermédiaire du compilateur de SmallTalk2K ;

2. l'étude de l'efficacité et de la sûreté des langages à objets et notamment d'*Eiffel* (et ses évolutions) et de son compilateur *SmallEiffel* ;
3. l'étude de systèmes de types pour les langages à objets et pour les assistants de preuves ; la réécriture et les calculs formels ($\lambda, \varsigma, \rho, \pi, \ldots$) comme base des langages de programmation à objets, fonctionnels et concurrents ;
4. l'étude du système d'exploitation *Isaac* et de son langage propriétaire à prototypes *Lisaac*.

6.2. FunTalk et SmallTalk2K

Mots clés : *SmallTalk*, *FunTalk*, *Eiffel*, *Self*, *LCO*, *OC*.

Participants : Alberto Ciaffaglione, Joëlle Despeyroux, Luigi Liquori, Olivier Zendra.

Nous avons pour objectif de définir un nouveau langage à objets inspiré à la fois du langage Eiffel pour ce qui concerne les aspects génie logiciel, et des langages à prototypes pour ce qui concerne les aspects formels. En outre, les aspects dynamiques du langage SmallTalk nous semblent appropriés pour augmenter l'expressivité du langage.

Il est bien connu que le langage SmallTalk a été le premier espace de travail ou *système* complètement orienté vers les objets. SmallTalk était plus qu'un langage puisqu'il avait déjà un environnement composé d'un *Browser de classes* (pour « naviguer » dans les bibliothèques), un *Workspace* (pour tester les applications), et un *Debugger*, eux-mêmes écrits en SmallTalk !

Beaucoup de concepts de SmallTalk ont été repris dans d'autres langages comme Java, C++ et Eiffel, par exemple. Malheureusement et heureusement, cela dépend du point de vue suivant lequel on se place, le langage SmallTalk est un langage interprété, donc lent, mais aussi très expressif, car il n'est pas typé. Il est largement utilisé dans les projets où la vitesse d'exécution n'est pas cruciale comme, par exemple, dans le prototypage d'applications.

Depuis plus d'une dizaine d'années, plusieurs équipes ont essayé de trouver un système de typage sûr pour SmallTalk sans trop restreindre l'expressivité du langage. Dans la théorie de types, on connaît très bien les équations :

- langages typés : pas expressifs mais sûrs ;
- langages non typés : très expressifs mais pas sûrs.

6.2.1. Les caractéristiques du langage FunTalk/SmallTalk2K

L'absence de typage statique confère une très grande expressivité au langage SmallTalk : il permet la création dynamique de classes et la définition réflexive du système. On souhaite conserver une grande partie de l'expressivité de SmallTalk dans le langage SmallTalk2K. Bien sûr, le système de types que nous allons définir a pour objectif de rejeter statiquement certains *mauvais* programmes (voir *e.g.* celui de la figure 4, écrit en Eiffel). Notre objectif consiste à conserver les acquis des langages Eiffel et SmallTalk avec un système de typage sûr.

Puisque le système de types sera fortement inspiré du travail de recherche sur les langages à prototypes, on pourra par exemple ajouter la modification dynamique des instances comme dans les langages à délégations (comme par exemple le langage Beta [103] qui est une sorte de langage hybride à classes et à objets). Cette possibilité permettra, par exemple, d'écrire en SmallTalk2K l'affectation suivante (on utilise une syntaxe à la Java et on suppose l'existence d'une simple classe `POINT`) :

```
point = new POINT(2,4);
point.set(int dx,int dy) = {new body pour set};
point.set(3,5);
```

Cela correspond à changer dynamiquement le comportement d'une seule instance de la classe `POINT` qui, après l'affectation de `set`, aura un comportement pour la méthode `set` *différent* de toutes les autres instances de `POINT`. Une telle caractéristique est déjà présente dans des langages à prototypes comme Self par exemple, et a été récemment explorée dans le cadre du langage Java [75].

Nos objectifs ici sont de concevoir la sémantique et un compilateur d'un langage *intermédiaire* à prototypes, fortement typé, appelé FunTalk. La définition de SmallTalk2K pourra être exprimée à travers ce langage. Un programme écrit en SmallTalk2K pourrait être pré-compilé vers un programme en FunTalk équivalent en utilisant, par exemple, les transformations en *continuation passing style (CPS)* [121][138] ou une capture « brutale » de la pile.

Le langage SmallTalk2K pouvant être vu comme une couche à classes au dessus du langage bas niveau à prototypes FunTalk, nous envisageons la possibilité de rendre disponible dans SmallTalk2K quelques primitives propres au langage FunTalk (par exemple la modification ou l'extension dynamique des méthodes dans un objet, *c.à.d.* dans une instance de classe). La sémantique statique et dynamique de FunTalk pourra être inspirée des calculs à prototypes OC et LCO en version impérative. Dans ce cas, l'efficacité d'exécution du code compilé n'est pas un problème, mais la définition sémantique du langage devient beaucoup plus compliquée. On formalisera la mémoire avec des techniques standard.

6.2.2. Phase formelle de conception de FunTalk/SmallTalk2K

La phase formelle de la construction d'un langage (avec son interprète et compilateur) sûr et certifié est un objectif très ambitieux.

On prévoit les étapes suivantes :

- étude des langages OCaml, Haskell, Obliq, Self, Beta, Kevo, Emerald, Cecil, Moby, ainsi que des calculs O_1, O_2, O_3 basées sur OC [1], et LCO [11] (à la base de Moby) ;
- description d'une syntaxe, d'une sémantique opérationnelle (*small-step*) et d'une sémantique naturelle (*big step*) avec stratégie d'évaluation pour le langage FunTalk ;
- description d'un système de types pour FunTalk inspiré des systèmes de types pour les calculs LCO et OC. Ce système devra prévoir la *mytype method specialisation*, *c.à.d.* le `like Current` d'Eiffel, une solide notion de *subtyping* et/ou *matching* [52][49][32] et la possibilité de typer statiquement les méthodes binaires ;
- démonstration du théorème de préservation des types (*subject reduction*) et de complétude (*type soundness*), *c.à.d.* respecter le slogan de Milner *« well-typed programs cannot go wrong »*, pour FunTalk ; l'expérience acquise avec le Lambda-calcul, le ρ-calcul, les calculs LCO et OC et leurs systèmes de types sera très utile dans cette phase ;
- description d'une syntaxe pour le langage SmallTalk2K et sa traduction en langage intermédiaire FunTalk. Deux techniques de compilations seront considérées (en particulier pour la gestion des exceptions) : intuitivement, la première favorise l'élégance et la seconde les performances du code compilé :
  - les transformations en *continuation passing style* (ou à la *stackless closure-passing style*) à la SMLNJ [138][35] ;
  - la capture « brutale » du point d'exécution, *c.à.d.* une sorte d'*execution stack capture* à la OCaml, Bigloo ;
- description d'un interprète pour FunTalk ou d'une phase de compilation de FunTalk ; deux choix sont à présent possibles, notamment une traduction vers JVM bytecode de Sun ou vers la nouvelle plate-forme .NET de Microsoft. Le premier choix permettra de factoriser la recherche des dernières années en France (à l'INRIA en particulier) et à l'étranger autour de la validation de la JVM et du langage Java (entre autres [140][109]).

Pendant le développement théorique du langage FunTalk, on étudiera la possibilité de formaliser et de prouver les théorèmes fondamentaux, *e.g.* conservation des types et complétude, avec l'assistant de preuves Coq. La vérification ainsi qu'une possible extraction de programmes à partir de preuves en Coq devrait nous aider à la production d'un premier interprète / compilateur pour FunTalk certifié. L'extraction de programmes en Coq est sans doute un sujet de recherche très intriguant qui nous intéresse beaucoup ; une particulière

attention sera dévouée au choix d'une méthode de codage (premier ordre ou ordre supérieur) qui permettra au moins la certification du logiciel visé (interprète ou compilateur).

6.2.3. Phase de conception et d'implantation

Nous prévoyons les étapes suivantes pour la construction des interprètes, compilateurs et environnements pour SmallTalk2K et FunTalk :

- construction (non certifiée) d'un premier interprète pour FunTalk `ft` ;
- ajout d'un module de type checking à l'interprète (non certifié) de FunTalk ;
- certification (validation) de l'interprète et du module de contrôle de types en Coq (*type soundness*) ;
- étude d'extraction via Coq de l'interprète typé pour FunTalk `coqft` ;
- étude de possibilité d'utilisation de FunTalk comme un langage de script, dans le style de Ruby [137], Python [136], ... ;
- construction (non certifiée) d'un compilateur FunTalk vers JVM de Sun ou .NET de Microsoft `ft2jvm`, `ft2net` ;
- étude d'extraction via Coq du compilateur pour FunTalk `coqft2jvm`, `coqft2net` ;
- auto-compilation (*i.e. bootstrapping*) du compilateur pour FunTalk. À partir d'un compilateur pour FunTalk (écrit dans un langage quelconque ou extrait par un assistant de preuve type Coq, pas forcément le plus efficace mais, on l'espère, certifié « sûr » [40][92]) on construira un compilateur optimisée pour FunTalk ;
- construction d'un premier compilateur (non certifié) du langage SmallTalk2K en FunTalk `st2ft` ;
- étude d'extraction via Coq du compilateur `coqst2ft` ;
- construction de l'environnement (*c.à.d.* quelques bibliothèques pour SmallTalk2K en SmallTalk2K ou en FunTalk (en quasi totale similitude avec SmallTalk) ;
- évaluation du langage et de son environnement sur des programmes non triviaux.

6.3. Eiffel et SmallEiffel

Mots clés : *Eiffel*, *typage sûr*, *covariance*, *liaison dynamique*, *analyse globale*, *prédiction de type*, *auto-compilation*.

Participants : Dominique Colnet, Luigi Liquori, Philippe Ribet, Olivier Zendra.

Glossaire

Typage sûr Dans ce contexte, le fait que les indications de types écrites dans le programme soient respectées lors de l'exécution.

Auto-compilation Compilation du compilateur en utilisant le texte source du compilateur lui-même (*bootstrap*).

6.3.1. Match-O, un dialecte d'Eiffel probablement sûr

Malheureusement, Eiffel qui est le langage qui se rapproche le plus de SmallTalk du point de vue de son expressivité, possède un système de types qui n'est pas sûr, *c.à.d.* qu'un programme bien typé selon les règles d'Eiffel peut se révéler incorrect à l'exécution (pendant l'exécution il soulèvera un exception concernant une erreur de type). Dans un langage à objets, une telle erreur se produit lors de l'exécution quand un objet reçoit un message qui n'est pas présent dans son interface.

Si l'on fait abstraction du *principe de cohérence globale* de B. Meyer [16], qui n'est implanté dans aucun compilateur Eiffel, on peut dire que le langage Eiffel n'est pas un langage avec un système de types sûr. Considérons par exemple le programme Eiffel de la figure 4. Dans cet exemple on définit deux classes `POINT` et `PIXEL` qui possèdent une méthode `is_equal` spécialisée dans `PIXEL`. La classe `MAIN` crée deux instances de `POINT` et une de `PIXEL`. L'affectation `p :=r` est acceptée par le compilateur puisque `r` qui est de type `PIXEL` peut être aussi considérée comme une expression de type `POINT`. Malheureusement, l'appel `p.is_equal(q)`

produit une erreur lors de l'exécution car c'est la version spécialisée de `is_equal`, définie dans `PIXEL`, que l'on exécute (liaison dynamique oblige). En effet, cette redéfinition suppose que l'argument `q` possède la méthode `status`. Comme le type dynamique de `q` est `POINT`, l'invocation de `status` avec une instance de `POINT` provoque une erreur. Le même problème, induit en fait par la redéfinition covariante de `is_equal`, se produit également à l'issue d'un passage d'argument (fonction `breakit`) (voir aussi le papier « *On Binary Methods* » [54]).

```
class POINT inherit ANY redefine is_equal end;
creation set
feature
   x: REAL; y: REAL;
   set(a: real, b: real) is
      do
         x := a; y := b;
      end;
   is_equal(other: like Current): BOOLEAN is
      do
         Result := (x = other.x) and (y = other.y);
      end
end

class PIXEL inherit POINT redefine is_equal end;
creation set
feature
   status: BOOLEAN;
   set_status(s: BOOLEAN) is
      do
         status := s;
      end;
   is_equal(other: like Current): BOOLEAN is
      do
         Result := (x = other.x) and (y = other.y)
                   and (status = other.status);
      end;
end
```

```
class MAIN creation main
feature
   p: POINT; q: POINT; r: PIXEL;
   main is
      do
         !!p.set(1,1); -- New POINT in variable p
         !!q.set(2,2); -- New POINT in variable q
         !!r.set(3,3); -- New PIXEL in variable r
         p := r; -- Accepted because of subtyping.
         if p.is_equal(q)   -- CRASH! (1)
         then ... else ... end;
         breakit(r,q) -- Accepted because of subtyping.
      end;
   breakit(p1, p2: POINT) is
      do
         if p1.is_equal(p2) -- CRASH! (2)
         then ... else ... end;
      end;
end
```

Figure 4. Un programme Eiffel qui laisse apparaître un problème de type lors de l'exécution.

Dans [6], D. Colnet et L. Liquori ont défini un dialecte d'Eiffel, appelé *Match-O* en raison de l'ajout d'un nouveau type appelé « type `match` ». Match-O n'a pas encore une sémantique formelle, mais son nouveau système de types, inspiré des travaux de K. Bruce *et al.* [56][53][51][54][58][50][52] *rejette* le programme (erroné) de la figure 4 lors de la compilation. Ainsi, dans Match-O, on remplace systématiquement le type `like Current` par le type `match Current`. Intuitivement, le type `match Current`, permet d'exprimer le fait qu'une entité est exactement du même type que le receveur au regard du principe de liaison dynamique. Cette nouvelle notation de type, que nous proposons d'ajouter dans Eiffel, permet à la fois de mieux décrire le type des entités manipulées mais aussi d'obtenir des programmes plus sûrs. Une étude complète de l'intégration du type `match` par rapport à la définition de classes génériques est en cours. Nous avons écrit un compilateur qui implante ce dialecte, accessible à http://www.loria.fr/~colnet/Match-O/macho.tgz.

L'expérience du dialecte Match-O s'est révélée très positive, car une partie du code de la maquette (notamment relative à la comparaison des objets) a déjà été incluse dans la nouvelle distribution de SmallEiffel.

Comme tous les langages de taille réelle, le langage Eiffel est en cours d'évolution : en particulier, la communauté qui développe Eiffel (autour de B. Meyer) s'intéresse à établir une passerelle vers le monde .NET de Microsoft ; Eiffel# [126] est un dialecte de Eiffel qui « compile » de manière naturelle vers la plate-forme .NET ; le système de types d'Eiffel# bloque la covariance (en similitude avec Match-O) et l'héritage multiple.

6.3.2. Amélioration de l'analyse globale

Bien qu'il donne actuellement d'excellents résultats, l'algorithme d'analyse globale et de prédiction de type peut encore être amélioré. Des informations supplémentaires peuvent être rassemblées en augmentant la précision de l'analyse statique. Une voie qui se présente naturellement à l'esprit consiste à prendre en compte le flot d'exécution, de façon à quasiment éliminer les parties du code qui sont considérées vivantes par notre algorithme actuel alors qu'il s'agit en fait de code mort. Ceci permettrait également d'affiner encore la précision de la prédiction de type, d'où une amélioration potentielle des performances de la liaison dynamique. Cependant, il est important d'évaluer le coût à la compilation d'une telle analyse *flow sensitive*, qui pourrait s'avérer incompatible avec la compilation rapide requise pour des environnements de développement incrémental. Il est également possible d'augmenter encore la spécialisation du code généré en l'adaptant non seulement au type du receveur, mais aussi aux types dynamiques des paramètres des méthodes. C'est ce que fait l'algorithme du produit cartésien (*Cartesian Product Algorithm*) [34] proposé par O. Agesen. Ceci impose cependant la génération de très nombreuses versions d'une même méthode, qui devraient être plus efficaces car plus spécifiques, mais qui risquent de provoquer un accroissement important de la taille du code généré et du temps de compilation. Il semble donc intéressant d'étudier plus en détail les différents degrés de spécialisation possibles et l'adéquation des compromis qu'ils représentent.

6.3.3. Couplage avec des techniques d'analyse dynamique

Les travaux sur SmallEiffel se sont pour l'instant focalisés sur les bénéfices de l'analyse et l'optimisation statiques, faites lors de la compilation. Un domaine de recherche connexe et très actif existe, qui porte sur les techniques d'analyse et d'optimisations dynamiques, *c.à.d.* effectuées pendant l'exécution du programme compilé. Il nous semblerait judicieux d'aborder aussi ce domaine très actuel, qui génère une littérature importante. Nous souhaitons étudier ces techniques dynamiques et leurs apports au processus de compilation, notamment au niveau du couplage possible entre l'analyse globale statique et l'analyse dynamique. Nous pensons en effet qu'il est possible de bénéficier de synergies importantes entre ces deux mondes et que les informations statistiques, de profils, rassemblées au cours de l'analyse dynamique devraient permettre d'améliorer encore les performances des programmes compilés à l'aide de notre système actuel. Ces techniques dynamiques appliquées à la liaison dynamique (basées généralement sur divers types de caches à l'exécution), par opposition aux techniques statiques (basées sur des informations et structures définies à la compilation) comme celles sur lesquelles nous avons travaillé jusqu'à présent, nous apparaissent comme particulièrement intéressantes. Elles présentent à notre avis un fort potentiel, notamment en complément de techniques statiquement calculées. Comme nous l'avons déjà souligné, la liaison dynamique et son implantation optimale sont en effet pour nous des clés importantes vers des programmes à objets performants.

6.3.4. Généralisation à des systèmes non figés

Pour l'instant, nos travaux ont porté sur des systèmes complets et figés, sans chargement ni création dynamique de classe. Dans l'optique de rendre les techniques que nous avons étudiées encore plus largement utilisables, il nous semble souhaitable d'étendre leurs capacités aux cas de systèmes incomplets. Des extensions relativement simples sont possibles pour utiliser ces techniques dans de tels contextes, comme la compilation séparée d'applications ou de bibliothèques, mais elles semblent de nature à dégrader fortement les performances du code compilé dans ces conditions. Il nous paraît donc indispensable d'étudier des solutions permettant d'éviter cette dégradation. Ceci pourrait impliquer de fortes modifications des algorithmes actuellement utilisés dans le compilateur SmallEiffel. Dans le même ordre d'idée, nous souhaitons vivement étendre nos recherches de façon à y inclure la compilation de langages plus dynamiques, tout particulièrement Java ou même Self. Ce type de langages offre des possibilités différentes de celles des langages de classes plus classiques, statiques, mais impose des contraintes plus importantes au compilateur. Ceci devrait nous amener très rapidement à nous intéresser à la compilation dynamique (en ligne) proprement dite, qui, elle, permet de s'adapter à un environnement et un source changeants.

6.3.5. L'évolution du langage Eiffel

La définition du langage Eiffel est encore en cours d'évolution. L'action Miró s'intéresse à cette évolution aussi bien au niveau des décisions de modification qu'au niveau de l'implantabilité de ces choix. La nouvelle définition prévoit l'ajout d'un nouveau type de base, type *Tuples*, qui donne la possibilité de manipuler des N-uplets typés de valeurs. Avec le type *Tuples*, il sera possible de typer plus strictement les fonctions ayant plusieurs résultats ainsi que les fonctions à nombre variable d'arguments. La méthode d'analyse globale devrait permettre de spécialiser chaque sorte de *Tuples* vivant. Dans la nouvelle définition d'Eiffel, un *agent* est, en quelque sorte, comparable à une fermeture, *closure*, d'un langage fonctionnel. L'objectif de cet ajout consiste essentiellement à pouvoir augmenter l'expressivité dans les assertions. Les agents permettent également une certaine forme d'évaluation partielle, comme dans les langages fonctionnels. De nouvelles possibilités d'introspection sont également en projet. Ces dernières n'étant que consultatives (*e.g.* la réflexivité, *i.e.* combien de méthodes dans une classe, type des arguments d'une méthode, etc.), cela ne devrait pas remettre en cause l'hypothèse du système fermé qui est un point clef dans la stratégie de compilation de SmallEiffel.

6.4. Réécriture, types et preuves

Participants : Gilles Barthe [Projet Lemme], Horatiu Cirstea [Projet Prothéo], Jöelle Despeyroux, Claude Kirchner [Projet Prothéo], Luigi Liquori.

6.4.1. Le Rho-calcul typé

En 2002, la collaboration avec des éléments du projet Prothéo est devenue plus étroite. Nous avons publié deux articles à des conférences internationales ; dans [66], nous avons montré comment le ρ-calcul peut être utilisé en tant que *lingua franca* pour capturer différents paradigmes de programmation, comme, par exemple, la programmation à objets. En particulier, nous avons présenté un codage du LCO [11] et du OC [1] en ρ-calcul. La possibilité de codifier d'une façon naturelle le comportement de `self` (`this` de Java) dans le ρ-calcul permet une représentation des objets comme de simples ensembles, et des méthodes comme de simples règles de réécriture. Dans un certain sens, tous les formalismes et techniques propres aux calculs à prototypes, type LCO et AC, peuvent être reproduits dans le ρ-calcul.

En outre, dans [67], nous avons proposé le ρ-calcul dans un cadre de typage à la Church, en présentant huit systèmes de types placés dans un cube à la Barendregt (voir figure 5). Le plus puissant de ces systèmes s'inspire essentiellement de l'*Extended Calculus of Constructions* de Z. Luo (ECC) [101].

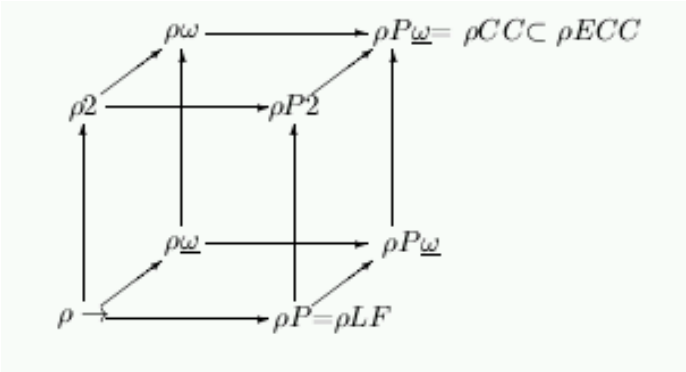

Figure 5. Le Rho-cube

On pense que l'étude du ρ-calcul (typé ou non) est fondamentale et prometteuse pour explorer différentes sémantiques des langages à objets, et étendre le Calcul de Constructions (et peut-être aussi Coq) avec une solide notion de réécriture (voir aussi [37]). Les recherches les plus fondamentales dans Miró seront menées dans ce sens.

6.4.2. Les systèmes de types purs avec motifs

En 2002, la collaboration avec G. Barthe du projet Lemme s'est intensifiée. Une papier sur le systèmes de types purs avec motifs, directement inspiré au rho-calcul à été accepté a POPL-03 [20] et nous sommes en train de rédiger une version longue pour un journal ; dans notre travail nous définissons une extension des *Pure Type Systems,* **PTS** [38] avec pattern matching (*Pure Patterns Type Systems,* $\mathbf{P^2TS}$). Cette extension est fortement inspirée du papier *Lambda Calculus with Patterns* de V. van Oostrom [130]. Avec J. Despeyroux on s'intéresse également à l'intégration dans les $\mathbf{P^2TS}$ d'algorithmes (décidables) d'unification d'ordre supérieur (*e.g.* à la D. Miller [107]).

La syntaxe des $\mathbf{P^2TS}$ est très simple :

$$\begin{array}{lcll} \Gamma & ::= & \varnothing \mid \Gamma, a : t \mid \Gamma, X : t & \quad contextes \\ t & ::= & a \mid X \mid \lambda t : \Gamma.t \mid \Pi t : \Gamma.t \mid t\ t \mid [t \ll_\Gamma t].t \mid t, t & \quad termes \end{array}$$

où Γ dénote un contexte, a dénote une constante, X dénote une variable, $\lambda t : \Gamma.t$ une fonction avec motifs, $\Pi t : \Gamma.t$ un type produit avec motifs, $t\ t$ une application, $[t_1 \ll_\Gamma t_3].t_2$ une *delayed matching abstraction* où $[t_1 \ll_\Gamma t_3]$ représente une équation de filtrage pas encore résolue, et t, t une séquence. Les règles d'évaluation ont la forme suivante :

$$\begin{array}{llcl} (\rho) & (\lambda t_1 : \Gamma.t_2)t_3 & \longrightarrow & [t_1 \ll_\Gamma t_3].t_2 \\ (\sigma) & [t_1 \ll_\Gamma t_3].t_2 & \longrightarrow & \sigma_{(t_1 \ll_\Gamma t_3)}(t_2) \\ (\delta) & (t_1, t_2)\ t_3 & \longrightarrow & t_1\ t_3, t_1\ t_3 \end{array}$$

où $\sigma_{(t_1 \ll_\Gamma t_3)}$ est la solution d'une équation de filtrage. En complète similitude avec le λ-cube de Barendregt, on peut envisager (cela est actuellement en cours d'étude) un « cube » des systèmes de types avec motifs à la Barendregt (montré en figure 6), qui pourrait être à la base d'un futur noyau d'un assistant à la preuve dans l'esprit du système Coq.

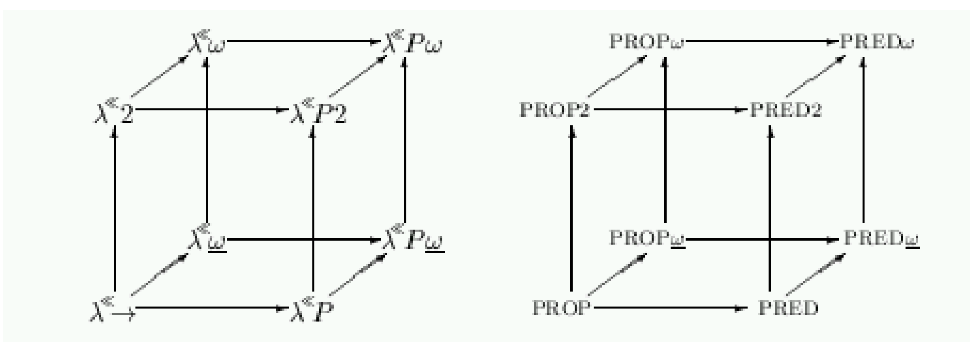

Figure 6. Le $\mathbf{P^2TS}$*-cube et le LOGIC-cube*

6.4.3. Les théories typées et les calculs formels

Participants : Alberto Ciaffaglione, Joëlle Despeyroux, Luigi Liquori.

Nous avons mentionné précédemment (section 2.4) le travail pionnier réalisé par J. Despeyroux, en collaboration avec F. Pfenning, C. Schürmann et P. Leleu, dans la recherche de nouvelles théories typées permettant la récursion sur des termes de type fonctionnel.

L'extension de ces systèmes aux produits dépendants, dans toute sa généralité, apparaît difficile. F. Pfenning et C. Schürmann ont préféré, comme D. Miller, introduire un système à deux niveaux : un niveau pour les expressions, et un niveau pour la récursion. C'est une approche raisonnable. Ceci dit, le but à long terme reste

la définition d'un système intégrant la récursion et les types inductifs aux expressions et prédicats, au même niveau, comme dans le Calcul des Constructions Inductives [131].

Comme d'autres projets de l'INRIA (*e.g.* Mimosa, Moscova, ...), l'action Miró s'intéresse aux calculs (et à ses systèmes de types) qui modélisent la mobilité, le réseau et ses problèmes de sécurité. Récemment, dans la littérature scientifique, on a assisté à la naissance d'une *plethora* de calculs comme, par exemple, les Ambients de Cardelli et Gordon [61][60], le Join-calcul de Fournet, Gonthier, Lévy, Maranget et Rémy [87], le Blue-calcul de Boudol [48], le Spi-calcul de Abadi et Gordon [33], et le Pi-calcul de Milner [108]. En outre, des travaux récents ont montré comment le paradigme à objets et ses concepts de base peuvent facilement s'intégrer dans ces nouveaux calculs.

L. Liquori avait proposé une extension du Blue Calcul [48] : le *Deep Blue Calcul* [100]. Le Deep Blue Calcul est un descendant direct du Blue Calcul. Ce calcul pourrait être vu comme une extension du Blue Calcul et du calcul des objets de Abadi et Cardelli. La recherche visait à chercher des transformations en CPS qui traduisaient le Deep Blue Calcul dans le Blue Calcul. L'idée était de démontrer que le Blue Calcul pouvait être vu comme un langage cible pour le Deep Blue et celui-ci pouvait être considéré comme point de départ pour la définition de langages à objets concurrents d'ordre supérieur.

On cherchera à être « à la page » par rapport à ces sujets de recherche et à ce que, éventuellement, les concepts de mobilité, réseaux et sécurité puissent s'intégrer d'une façon naturelle dans les calculs et/ou langages auxquels on s'intéresse.

6.4.4. Les preuves formelles

Nous nous intéresserons ici principalement à la certification des outils développés dans l'action autour des langages et calculs à objets. Cependant, partant du principe qu'un projet de recherche a pour vocation d'appréhender une problématique dans son ensemble, plutôt que dans des cas trop particuliers, nous nous attacherons à étudier les moyens d'améliorer le confort, et en fait la faisabilité, des preuves formelles en général. Pour ce faire, nous poursuivrons les recherches entreprises par J. Despeyroux, A. Hirschowitz, P. Leleu et G. Gillard : étude de petits exemples de formalisation de langages (principalement les langages et calculs concurrents et à objets), développement de méthodes de formalisation de ces langages, et étude de nouvelles théories typées améliorant l'activité de preuve (voir sections 3.3.2, 6.4.3).

Les preuves demandées sont encore très peu automatisées. L'action Miró veut investir dans ce domaine de recherche stratégique. Les éléments logiciels présents dans les appareils modernes ont une importance cruciale pour le bon fonctionnement de ces appareils. Dans de nombreux cas, il n'est plus possible de recourir uniquement à des campagnes de tests pour assurer leur correction et des techniques de programmation certifiée doivent être développées et mises à la portée des ingénieurs.

6.5. Le système d'exploitation Isaac

Participants : Dominique Colnet, Cyril Proch, Benoît Sonntag, Olivier Zendra.

6.5.1. Le système d'exploitation Isaac

L'évolution des technologies et du matériel nous amène à revoir la conception du noyau des systèmes d'exploitation actuels. Nous pensons que l'introduction de concepts objets de haut niveau au cœur même des systèmes d'exploitation simplifie et optimise l'utilisation des divers composants physiques d'un ordinateur. Ce type de démarche apporte un réel progrès aussi bien dans l'adaptation dynamique du système, que dans la simplification et le pouvoir expressif du code. Isaac est avant tout un nouveau système d'exploitation objet. Il allie modularité, flexibilité, adaptabilité, évolutivité et rapidité. D'un genre nouveau, il ne possède aucun noyau, ni micro noyau et fonctionne directement sur l'architecture de la machine. Aucune machine virtuelle n'est nécessaire et aucune autre entité que l'objet n'est présent. Actuellement, les objets qui constituent Isaac tirent pleinement partie de sa flexibilité pour modéliser de manière élégante et répondre aux attentes d'un système traditionnel du noyau jusqu'à l'interface utilisateur. La plus grande particularité d'Isaac est de garder suffisamment de sémantique dans le binaire pour que les objets conservent toutes leurs expressivités objets

durant l'exécution. Le système est capable d'avoir une vision suffisamment fine des objets pour les manipuler, les inter-connecter, les cloner, ou changer les liens de multi-héritages durant l'exécution même.

Isaac [24] est constitué de son propre modèle objet à base de prototypes, le langage Lisaac qui a permis de le réaliser ressemble conceptuellement à Self avec des atouts directement liés à la conception de système portable et proche de la machine. Son compilateur produit du code binaire efficace grâce à des optimisations de haut niveau : produit cartésien, prédiction de type, suppression de la liaison dynamique, suppression de la récursivité, spécialisation du code, ... Par son modèle simple et homogène, nous sommes en droit de penser qu'il devient possible de construire et de faire évoluer les applications par simple ajout d'objet ou par mutation des prototypes existants via le clonage et l'héritage dynamique.

Son spectre d'application est varié : sa forte modularité et sa légèreté (moins d'un mega indispensable) lui permettent de s'adapter naturellement à des architectures exotiques comme les assistants personnels (palm) ou l'informatique embarquée. En effet, il devient facile d'assembler son système d'exploitation à la carte pour répondre au mieux à des contraintes industrielles et matérielles.

Sans une baisse des performances, Isaac possède des mécanismes très particuliers de protection directement gérés par le matériel qui permet de garantir une sécurité inviolable sur les objets essentiels du système. Sa forte modularité provenant des concepts objets lui permet d'aguerrir une certaine stabilité naturelle : un objet endommagé n'affecte pas (ou peu) l'intégrité du système.

6.5.2. Le langage Lisaac

Le langage à prototypes *Lisaac*, proche de Self, apporte un modèle uniforme pour la réalisation de nos concepts. Il se démarque par sa capacité à communiquer entres objets déjà compilés connus ou partiellement connus. Il possède les qualités de flexibilité de Self avec des outils directement liés à la mise en place de systèmes d'exploitation ou plus généralement à la programmation de bas niveau avec un langage de haut niveau. Nous avons développé un compilateur pour un langage spécialisé à base de prototypes, nommé Lisaac, qui nous sert à implanter notre système d'exploitation Isaac. Le compilateur Lisaac utilise des méthodes de compilation avancées comme l'analyse de flot pour développer l'algorithme du produit cartésien (CPA) de prédiction de type. Les premiers résultats sont encourageants et ont donné lieu à une première publication internationale présentant notre langage [23]. Le stage de DEA de C. Proch [31] a considérablement fait évoluer la syntaxe du langage ainsi que la compréhension de la sémantique du langage lui même. Les nouvelles possibilités qu'offre ce langage au niveau de la programmation de bas niveau permettent également d'uniformiser le modèle de notre système Isaac.

En 2001, nous avons spécifié un nouveau format d'exécutable pour Isaac, lui permettant une meilleure flexibilité et une meilleure évolutivité. Ce format permettra aussi dans un futur proche l'intégration des produits GNU de manière transparente pour le système. Il est en cours d'implantation. Le gestionnaire du système de fichier d'Isaac a fait l'objet d'une ré-formalisation complète permettant une modularité et une bonne abstraction de la notion de fichier sous forme d'objet. L'interface utilisateur d'Isaac a également nécessité l'étude d'une forte abstraction des besoins en terme d'ergonomie. Notre modèle d'interface peut ainsi se décliner très simplement vers une représentation concrète diverse et appropriée à l'utilisateur. Sa représentation peut être graphique, vocale ou autre. Elle passe d'une représentation à une autre par simple modification d'héritage dynamique de l'interface abstraite.

Enfin, une formalisation du langage Lisaac est en cours d'étude. Nous pensons que ce langage pourrait être formalisé avec une sémantique opérationnelle relativement simple car le langage a une taille raisonnable (une vingtaine de constructeurs). En conclusion :

- Isaac est un système d'exploitation entièrement conçu avec des objets ;
- Lisaac est un langage simple de programmation à prototype dans la veine de Self, avec lequel le système Isaac est décrit.

Nous estimons que le langage Lisaac est, peut-être, une forme sophistiquée et pas encore formalisée d'un langage à la Self avec un contrôle statique des types ; cette possibilité est actuellement étudiée dans l'action Miró ; Lisaac pourrait être compilé vers le langage intermédiaire FunTalk ; dans le cas affirmatif, on pourrait envisager un système d'exploitation complètement à objets - et à notre connaissance le premier, basé sur Isaac, et, potentiellement, certifié sûr.

6.6. Gestion mémoire pour la bibliothèque des ATerms

Participants : Pierre-Etienne Moreau [Projet Prothéo], Olivier Zendra.

La bibliothèque des ATerms est bien connue dans le domaine de la réécriture de termes. Elles est utilisée intensivement par diverses équipes de recherche, et des centaines d'outils sont basés sur elle, notamment l'environnement ASF+SDF, l'environnement ELAN, Stratego, JJForester, etc. Il s'agit d'un outil développé au CWI d'Amsterdam qui offre une API pratique pour manipuler des termes, un système gérant automatiquement l'aliasing maximal (ou partage maximal) des termes, ainsi qu'un système de gestion mémoire automatique.

Notre travail, en collaboration avec le projet Prothéo et le CWI d'Amsterdam, consiste en améliorer les performances du ramasse-miettes intégré à la bibliothèque. Il semble en effet que, du fait de la pression mémoire extrême exercée par certains programmes manipulant des termes, une grande partie du temps d'exécution soit passée à gérer la mémoire.

Nous avons souhaité concevoir un nouvel algorithme de gestion mémoire pour les ATerms qui prenne réellement en compte les spécificités de cette bibliothèque fonctionnelle, afin d'offrir les meilleures performances possibles. La principale caractéristiques que nous avons considérée est le fait que dans cette bibliothèque, un sous-terme n'est jamais modifié après la création du terme englobant. Ceci permet d'envisager une stratégie de gestion mémoire améliorée basée sur le principe des générations. La mémoire y est divisée en deux (ou plus) sous-espaces correspondant chacun à une génération ("jeunes" ou "vieux" objets). Ceci permet de tirer avantage du principe générationnel faible d'Ungar qui indique que la plupart des objets meurent jeunes, et que seuls quelques-uns ont une longue durée de vie. En effet, en ne collectant qu'une partie de la mémoire (typiquement, en collectant plus souvent la jeune génération que la vieille), il devient possible de diminuer les pauses dues au ramasse-miettes. Cela permet de plus d'améliorer le taux de récupération d'objets et de diminuer le temps total passé dans le ramasse-miettes. Cependant, en pratique, la plupart des systèmes générationnels ont un coût de gestion des sous-espaces (le plus souvent par recopie) qui est assez élevé, et diminue le gain de temps total.

Dans le cadre de nos travaux [28], démarrés en février 2002, nous avons décidé de créer un nouvel algorithme générationnel non basé sur la copie, contrairement à la plupart des algorithmes générationnels existants. Nous avons ainsi conçu GC2, un algorithme générationnel à marquage-balayage, qui évite le surcoût souvent imputables aux générations. Nous avons implanté une première version de GC2 dans la bibliothèque des ATerms, et obtenons des résultats extrêmement positifs, avec de bons gains en mémoire et surtout en temps (environ 20%). Cette amélioration est immédiatement applicable, de façon totalement transparente, à tous les outils utilisant la bibliothèque des ATerms.

Ces travaux pourraient être poursuivis en étudiant différents algorithmes de gestion mémoire parallèles, afin d'en proposer des versions tirant avantage des spécificités de la bibliothèque des ATerms. Le but serait de fournir une bibliothèque concurrente, qui pourrait être instanciée une seule fois et partagée par les différents composants inclus dans un environnement, alors qu'actuellement elle est instanciée une fois par composant l'utilisant. Ceci résulterait en des échanges entre composants en temps constant, et en un gain mémoire important.

6.7. Java : optimisation de la liaison dynamique

Participants : Karel Driesen [ACL McGill], Olivier Zendra.

En 2002, O. Zendra a continué ses recherches portant sur l'optimisation des programmes Java. En effet, un des plus gros problèmes qui se pose aux nombreux utilisateurs du langage Java est celui de la vitesse d'exécution des programmes. Il est donc capital de fournir des solutions à ce problème.

Après avoir travaillé sur la détection de régularités au sein des comportements des programmes Java, afin d'en tirer parti pour proposer des techniques d'optimisation portables, O. Zendra a cherché à bien caractériser les interactions matériel-logiciel, pour pouvoir les exploiter et les modéliser. En effet, même si des optimisations portables sont possibles, elles peuvent être encore sensiblement améliorées par la prise en compte de l'environnement d'exécution (JVM et/ou matériel).

Pour cela, O. Zendra a collaboré avec le laboratoire ACL de McGill University pour développer une expérimentation visant à compléter mes premiers travaux en simulant des prédicteurs de branchement de processeurs, avec différents patterns d'exécution. Les résultats de cette expérimentation [27][25][29] confirment que certaines structures de contrôle permettent d'optimiser la liaison dynamique d'une façon qui soit robuste aux changements de processeur. Cependant, l'impact des patterns de types s'avère considérable sur les prédicteurs, et en cas de prédiction de branchement manquée, le coût sur la liaison dynamique peut être important. Néanmoins, lorsque le nombre de types est faible, quel que soit leur pattern, les méthodes classiques de liaison dynamique en Java peuvent être avantageusement remplacées par des structures alternatives, dont la liaison dynamique à arbre de branchement binaire.

6.8. Récapitulatif et programme de travail

Compte tenu des effectifs actuels de l'action, bien conscients de nos limites, nous envisageons le programme de recherche suivant :

- **à court terme, *c.à.d.* sur environ 2 ans :**
  - description complète de la syntaxe, de la sémantique statique et dynamique du langage FunTalk ; démonstration de propriétés fondamentales sémantiques sur papier et début de leur formalisation dans un assistant de preuve ;
  - construction d'un compilateur et peut-être d'une machine virtuelle jouet pour FunTalk (en utilisant *e.g.* SmallEiffel ou Self) ; syntaxe du langage SmallTalk2K ;
  - étude, à travers l'analyse globale à la base du compilateur SmallEiffel, de la possibilité de résoudre le problème de sécurité du langage Eiffel sans modifier la définition du langage (en utilisant l'expérience sur les `match-type` en Match-O) ;
  - étude et implantation d'une bibliothèque d'interfaçage homme-machine pour Eiffel, en utilisant le langage Eiffel lui-même et la programmation par contrat et les agents ;
  - méta théorie du ρ-cube et des $\mathbf{P}^2_{\beta}\mathbf{T}\,\mathbf{S}$ et étude des différentes méthodes de formalisation et de preuves des langages et calculs à objets : choix d'une méthode ;
  - maturation du langage à prototypes Lisaac ; étude de la portabilité du système d'exploitation Lisaac sur d'autres processeurs ; portage des produits GNU sur l'OS Isaac ;
  - étude des optimisations du langage Java, notamment en utilisant les techniques développées pour SmallEiffel et de l'impact des architectures matérielles sur les performances des programmes à objets ;
- **à moyen terme, soit environ 4 ans :**
  - réalisation du compilateur et de la machine virtuelle pour FunTalk certifié (et peut-être extrait automatiquement) ; bootstrap d'un compilateur optimisé FunTalk écrit en FunTalk et début de sa certification semi-automatique ; validation de FunTalk avec exemples non triviaux ; certification partielle du compilateur optimisé et/ou de la machine virtuelle FunTalk ;
  - étude d'une logique possible pour le ρ-cube (peut-on envisager un isomorphisme à la Curry-Howard ?) et des $\mathbf{P}^2_{\beta}\mathbf{T}\,\mathbf{S}$ (peuvent-ils être à la base d'un futur noyau d'un assistant à la preuve dans l'esprit du système Coq ?) ;

  - continuation de l'étude de nouvelles théories typées facilitant la description des liaisons (*i.e. binders*) dans les langages de programmation ;
  - continuation de l'étude de différentes méthodes de formalisation et de preuve des langages et calculs concurrents et à objets ;
  - maturation du système Isaac avec en particulier la diffusion du système Isaac et de ses outils de compilation ;
  - étude de l'impact de l'utilisation de l'héritage dynamique pour l'implantation des systèmes d'exploitation.
  - modélisation et prédiction de l'impact des architectures matérielles sur l'exécution des programmes à objets ;

- **à long terme, soit plus de 4 ans :**

  - continuation de l'étude de nouvelles théories typées, comme par exemple celles présentées en [9][10] ;
  - certification complète du compilateur optimisé et/ou de la machine virtuelle FunTalk ; *plug-in* des transformations CPS dans ce compilateur, afin de le transformer en un compilateur SmallTalk2K !
  - design et développement d'un premier environment complet (compilateur et/ou machine virtuelle et espace de travail, comme browser, workspace, debugger) pour SmallTalk2K en SmallTalk2K ou en FunTalk.

7. Contrats industriels

7.1. Le contrat plan état-région (CPER)

Participants : Dominique Colnet, Luigi Liquori.

- L. Liquori et D. Colnet sont promoteurs du projet : *SmallTalk2K : Construction d'un Compilateur et d'un Environnement « Sûr » pour un Langage à Objet*, financé par la Région Lorraine pour l'année 2001 et 2002, avec 30KEuro[7] ;
- O. Zendra a été rédacteur d'un dossier de demande de subvention jeune chercheur *Qualité et Optimisation du Code dans les Programmes Java*, auprès de la Région Lorraine, qui lui a accordé la somme de 7KEuro, pour l'année 2003-2004.

7.2. Opérations développement du logiciel (ODL)

Participants : Dominique Colnet, Philippe Ribet.

Dans le cadre du développement du logiciel SmallEiffel, l'action Miró a obtenu pour l'année 2001 un financement de l'INRIA de 15KEuro et un ingénieur expert pour 12 mois. Ce contrat a été reconduit en 2002. Le but principal de P. Ribet est l'étude et l'implantation d'une bibliothèque d'interfaçage homme-machine pour Eiffel, en utilisant le langage Eiffel lui même et la programmation par contrat.

8. Actions régionales, nationales et internationales

8.1. Projets européens

- J. Despeyroux est responsable locale (*site leader*) du groupe de travail européen IST *Types*, qui fait suite au projet européen ESPRIT BRA *Types for proofs and programs*. Ce groupe de travail (projet no 29001) a commencé le 1er aout 2000, pour une durée de trois ans. Il comprend 34

sites (regroupés en sites et sous-sites) répartis en Europe (Finlande, France, Allemagne, Grande Bretagne, Italie, Pays-Bas et Suède), dont 5 sites industriels, notamment, en France, Dassault-Aviation, France Télécom et Trusted Logic. L'adresse Web de la page d'accueil de ce groupe est : http://www.dur.ac.uk/TYPES/. L. Liquori y participe également.

- J. Despeyroux et L. Liquori participent au projet IST *Appsem II : Applied Semantics*.
- C. Casetti et L. Liquori du Polytechnique de Turin ont participé à l'expression d'intérêt (EoI) de la Communauté Européenne pour la formation d'un réseau d'excellence (NoE) nommé *« Design and dimensioning of the 3rd Generation Internet"* (appel à proposition prévu début 2003).

8.2. Action QSL (CPER)

L'action Miró participe activement au CPER (Contrat Plan État Région) sous forme de rapporteur de propositions d'actions, participation à la conception de la plate-forme logicielle, organisation de journées séminaires, et diffusion auprès des industriels d'Alsace et Lorraine.

8.3. Groupes de recherches (GDR)

L. Liquori participe aux GDRs *Mélanges de systèmes algébriques et de systèmes logiques* et *Objets et Composants Logiciels : spécification, vérification, sémantique*. D. Colnet et O. Zendra participent au GDR *Objets et Composants Logiciels : implémentation de langages à objets*.

8.4. Action Recherche Coopérative (ARC)

A. Ciaffaglione, J. Despeyroux et L. Liquori participent à l'ARC INRIA 2003 *Concert : « Compilateurs Certifiés »* avec les équipes Cristal, Foc-Cnam, Lemme, Mimosa et Oasis [30].

8.5. Accueils de chercheurs

L´action Miró a accueilli en 2001-2002 les chercheurs suivants :

- avril 2001 (30 jours) : Richard Walker, Australian National University (ANU), de Canberra, Australie, pour effectuer un travail conjoint sur SmallEiffel,
- septembre 2001 (3 jours) : Gilles Barthe, projet INRIA Lemme (*Reasoning about Javacard*),
- novembre 2001 (5 jours) : Dan Dougherty, WPI (*Normal forms and reduction for theories of binary relations*),
- décembre 2001 (4 jours) : Claudio Casetti, École Polytechnique de Turin (*Enhancing the TCP Protocol* et *Advanced TCP/IP*),
- décembre 2001 (3 jours) : Rossano Gaeta, Università di Torino (*ATM : theory and applications*),
- mars 2002 : (2 jours) : Yves Bertot, projet INRIA Lemme (*Certification d'un compilateur pour un langage impératif*),
- mars 2002 : (1 jour) : Pierre Cointe et Mario Südholt, action INRIA Klee, Nantes (*AOP : Aspect Oriented Programming*),
- juillet 2002 : (3 jours) ; Gilles Barthe, projet INRIA Lemme (*Pure Pattern Type Systems*),
- septembre 2002 ; (1 jour) : Stephane Fechter, S.P.I.-Lip6, (*BBFoc*),
- octobre 2002 ; Jean Privat, LIRMM, Montpellier, (*Compilation séparée pour SmallEiffel*).

9. Diffusion des résultats

9.1. Diffusion du projet Miró

Luigi Liquori a commencé un petit « tour de France » de projets de recherches, notamment à l'INRIA. Nous envisageons la visite de quelque projet INRIA corrélées à notre travail de recherche pour présenter, discuter et réviser le projet Miró à l'aide de séminaires informels, notre but étant de profiter au maximum des compétences

de l'INRIA pour mieux adapter notre programme de recherche et/ou trouver des collaborations. Voici pour mémoire la liste des rencontres et présentations effectuées à ce jour :

- mars 2001 : INRIA Roquencourt (Cristal & Moscova),
- juin 2001 : INRIA Sophia (Certilab & Lemme & Mimosa),
- juillet 2001 : École des Mines de Nantes (OCM, de P. Cointe),
- octobre 2001 : ENS Lyon (Plume),
- janvier 2002 : Comité de projets INRIA Sophia Antipolis, France ;
- septembre 2002 : Thomson Multimedia, Rennes (P. Baudelaire).

9.2. Participation à des colloques, séminaires, invitations

- D. Colnet a présenté à l'université Pierre et Marie Curie de Jussieu le logiciel SmallEiffel, suite à une invitation ;
- J. Despeyroux a participé à la conférence FLOC-02 *Federated Logic Conference* ;
- L. Liquori à été *invited speaker* au workshop WRLA-02 *Workshop on Rewriting Systems and Applications*, où il a exposé *« Rewriting calculi with (out) Types »* [21] ; dans le GDR LAC il a également donné l'exposé *Pure Pattern Type Systems* ;
- B. Sonntag a participé au workshop OOOSWS-02 *Workshop on Object-Orientation and Operating Systems*, où il a exposé *« Dynamic inheritance : a powerful mechanism for operating system design »* [24] ;
- F.X. Kuntz a participé à la conférence TOOLS-02 *Technology of Object Oriented Languages and Systems*, où il a présenté le papier *« Lisaac : the power of simplicity at work for operating system »* [23] ;
- O. Zendra a participé à la conférence CC-02 *Compiler Construction*.

9.3. Administration de la recherche

- D. Colnet est membre du Département de Formation Doctorale (DFD) de Nancy. Il est membre actif du groupe ECMA (http://www.ecma.ch) de normalisation du langage Eiffel (TC39-TG4).
- J. Despeyroux est coéditeur, avec Robert Harper (CMU) de l'édition spéciale du *Journal of Functional Programming* sur les *Logical Frameworks and Meta-languages*. Cette édition devrait paraître en mars 2003 (vol. 13/2).
- J. Despeyroux a organisé l'école d'été internationale *Theory and Practice of Formal Proofs*, organisée sous l'égide du WG européen ESPRIT *Types*, à Giens, du 2 au 13 septembre 2002 (voir http://www-sop.inria.fr/certilab/types-sum-school02/). Les notes de cours sont accessibles en rapport INRIA, ainsi qu'en ligne.
- L. Liquori est membre du conseil des opérations CPER-QSL (depuis juin 02) et co-responsable de la plate-forme QSL (mai-septembre 02), (http://plateforme-qsl.loria.fr/plateforme_QSL) ;
- L. Liquori est depuis 2000 responsable des relations de l'École des Mines de Nancy avec les universités italiennes. En 2001-2002, 2 élèves en DEA d'informatique sont arrivés à Nancy ainsi que 7 élèves à l'École des Mines. En 2002-2003 on prévoit 8 élèves à l'École des Mines, dont 1 élève en doctorat.
- L. Liquori a organisée Semaine Départementale de l'École des Mines de Nancy à Turin, Italie (juin 02) :

  - visites auprès des Centre de Recherche de Fiat, Motorola, Radio Televisione Italiana, Altran Italia, Reteitaly (Leader Voip), de l'Institut Polytechnique de Turin, du Département d'Informatique Université de Turin, de l'Institut Nationale G. Ferraris ;
  - présentation du LORIA (http://www.loria.fr/~lliquori/PAPERS/En-LORIA-pp.pdf) et compte rendus de la semaine (http://www.loria.fr/~lliquori/mines@to.html) ;

- L. Liquori est co-éditeur de la lettre du Loria.

9.4. Participations à des jurys

- D. Colnet a participé aux comité de programme de TOOLS USA-02. Il est membre de la commission nationale de l'INRIA pour l'attribution des ODL (Opérations de Développement Logiciel) depuis 2000 ;
- J. Despeyroux a fait partie des membres de la Commission d'Évaluation de l'INRIA de janvier 1999 à août 2002 ;
- L. Liquori a participé en 2002 au jury de thèse de l'UHP-Nancy 1 de H. Dubois ; il a été membre du comité de programme de LMO-03.

9.5. Thèses et stages

Thèses et DEAs dans l'action (2001-2002) :

- A. Ciaffaglione, (doctorant INPL en cotutelle France-Italie) ; *Towards Certified Software for (manipulating) Reals and Objects* : le sujet de thèse porte, en outre, sur le développement et la formalisation en Coq d'un nouveau langage à objets, FunTalk, et de son compilateur (encadrants : L. Liquori 25% et C. Kirchner 25%) ;
- B. Sonntag, doctorant CCH-INRIA, 2001-2002 : *Intégration de concepts objets au cœur même des systèmes d'exploitation* (encadrant : D. Colnet) ;
- Cyril Proch, DEA, 2002 : *Lisaac : implantation et définition d'un langage à base de prototypes* (encadrants : D. Colnet 50% et O. Zendra 50%) ;
- K. Narayanaswamy, Stagiaire IIT, Inde, 5-6/2002 : *Formalisation d'un petit langage à objet en Coq* (encadrant : J. Despeyroux).

9.6. Enseignement

- D. Colnet a animé en 2002 à l'ESIAL (U.H.P. - Nancy 1) le cours *Programmation à objets* ; depuis septembre 2002 il est professeur à l'Université Nancy II ;
- J. Despeyroux a été responsable locale à Sophia Antipolis du DEA MDFI (Mathématiques Discrètes et Fondements de l'Informatique) de Marseille entre octobre 1998 et septembre 2002. Elle était responsable d'un cours d'option de ce DEA, sur la mécanisation des preuves.
- L. Liquori a été responsable en 2002 du cours *Réseaux et télécommunications* à l'école des Mines de Nancy : en outre il a participé en 2002 a un module de DEA IAEM Nancy ayant pour titre : *Sémantique des langages de programmation* ;
- O. Zendra anime en 2002 à l'U.H.P. - Nancy 1 des seances de TD's sur le langage de programmation Java.

10. Bibliographie

Bibliographie de référence

[1] M. ABADI, L. CARDELLI. *A Theory of Objects.* Springer Verlag, 1996.

[2] K. BRUCE. *A Paradigmatic Object-Oriented Programming Language : Design, Static Typing and Semantics.* in « Journal of Functional Programming », numéro 2, volume 4, 1994, pages 127-206.

[3] K. BRUCE. *Subtyping is not a Good Match for Object-oriented Programming Languages.* in « Proc. of ECOOP », série LNCS, volume 1241, Springer Verlag, 1997.

[4] S. COLLIN, D. COLNET, O. ZENDRA. *Type Inference for Late Binding. The SmallEiffel Compiler.*. in « Joint Modular Languages Conference », série LNCS, volume 1204, Springer Verlag, pages 67-81, 1997.

[5] D. COLNET. *Compilation, langages à objets et programmation par contrats. Expérience acquise dans le cadre du projet « SmallEiffel The GNU Eiffel Compiler »*. Thèse d'Habilitation à Diriger des Recherches, Université Henri Poincaré - Nancy 1, 2000.

[6] D. COLNET, L. LIQUORI. *Match-O, a Statically Safe (?) Dialect of Eiffel.* in « Proc. of TOOLS », IEEE Computer Society, 2000.

[7] D. COLNET, L. LIQUORI. *SmallTalk-2K : Construction d'un Compilateur et d'un Environnement « Sûr » pour un Langage à Objet.* 2000, Demande de Subvention à la Région Lorraine pour l'an 2001-2002 Soutien aux jeunes équipes de recherche et aux projets émergents.

[8] J. DESPEYROUX. *Proof of translation in Natural Semantics.* in « Proc. of LICS », IEEE Computer Society, 1986.

[9] J. DESPEYROUX, P. LELEU. *Metatheoretic Results for a Modal lambda-Calculus.* in « Journal of Functional and Logic Programming », numéro 1, volume 2000, 2000.

[10] J. DESPEYROUX, P. LELEU. *Recursion over Objects of Functional Type.* in « Mathematical Structures in Computer Sciences », numéro 4, volume 11, 2001.

[11] K. FISHER, F. HONSELL, J. C. MITCHELL. *A Lambda Calculus of Objects and Method Specialization.* in « Nordic Journal of Computing », numéro 1, volume 1, 1994, pages 3-37.

[12] P. D. GIANANTONIO, F. HONSELL, L. LIQUORI. *A Lambda Calculus of Objects with Self-inflicted Extension.* in « Proc. of OOPSLA », The ACM Press, pages 166-178, 1998.

[13] S. N. KAMINNNNN. *Inheritance in Smalltalk-80 : A Denotational Definition.* in « Proc. of POPL », éditeurs T. A. PRESS., pages 80-87, 1988.

[14] L. LIQUORI. *An Extended Theory of Primitive Objects : First Order System.* in « Proc. of ECOOP », série LNCS, volume 1241, Springer Verlag, pages 146-169, 1997.

[15] L. LIQUORI. *On Object Extension.* in « Proc. of ECOOP », série LNCS, volume 1445, Springer Verlag, pages 498-552, 1998.

[16] B. MEYER. *Eiffel, The Language.* Prentice Hall, 1994.

[17] C. SCHÜRMANN, J. DESPEYROUX, F. PFENNING. *Primitive Recursion for Higher-Order Abstract Syntax.* in « Theoretical Computer Science », numéro 1-2, volume 266, 2001, pages 1-57.

[18] D. UNGAR, R. B. SMITH. *Self : The power of simplicity.* in « Proc. of OOPSLA », The ACM Press, pages 227-241, 1987.

[19] O. ZENDRA, D. COLNET, S. COLLIN. *Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler.*. in « Proc. of OOPSLA », volume 32(10), The ACM Press, pages 125-141, 1997.

Communications à des congrès, colloques, etc.

[20] G. BARTHE, H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *Pure Patterns Type Systems.* in « Proc. of POPL », The ACM Press, 2002.

[21] H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *Rewriting Calculi with (out) Types.* in « Proc. of WRLA », volume 72, Electronic Notes in Theoretical Computer Science, 2002, To appear.

[22] H. CIRSTEA, C. KIRCHNER, L. LIQUORI, B. WACK. *The Rho Cube : Some Results, Some Problems.* in « In Proc. of FLOC-HOR », 2002, Also as RR LORIA A02-R-470.

[23] B. SONNTAG, D. COLNET. *Lisaac : the power of simplicity at work for operating system.* in « Proc. of TOOLS », The ACS Press, pages 45-52, 2002.

[24] B. SONNTAG, D. COLNET, O. ZENDRA. *Dynamic inheritance : a powerful mechanism for operating system design.* in « In Proc. of ECOOP-OOOSWS », série LNCS, Springer Verlag, pages 5, 2002.

[25] O. ZENDRA, K. DRIESEN. *Stress-testing Control Structures for Dynamic Dispatch in Java.* in « Proc. of JVM », Usenix, pages 105-118, 2002.

Rapports de recherche et publications internes

[26] D. DOUGHERTY, F. LANG, P. LESCANNE, L. LIQUORI, K. ROSE. *A Generic Object-calculus based on Addressed Term Rewriting Systems.* Rapport de Recherche, numéro RR-4549, INRIA, 2002, http://www.inria.fr/rrrt/rr-4549.html.

[27] D. GU, O. ZENDRA, K. DRIESEN. *The Impact of Branch Prediction on Control Structures for Dynamic Dispatch in Java.* Rapport de recherche, numéro LORIA A02-R-131 - INRIA RR-4547, INRIA, 2002, http://www.inria.fr/rrrt/rr-4547.html.

[28] P.-E. MOREAU, O. ZENDRA. *GC^2 : A Generational Conservative Garbage Collector for the ATerm Library.* Rapport de recherche, numéro LORIA A02-R-126 - INRIA 4548, INRIA, 2002, http://www.inria.fr/rrrt/rr-4548.html.

[29] O. ZENDRA, K. DRIESEN. *Evaluation of Control Structures for Dynamic Dispatch in Java.* Rapport de recherche, numéro LORIA A02-R-023 - INRIA 4370, INRIA, 2002, http://www.inria.fr/rrrt/rr-4370.html.

Divers

[30] CRISTAL, FOC-CNAM, LEMME, MIMOSA, OASIS, MIRÓ. *Concert : « Compilateurs Certifiés ».* 2002, ARC INRIA.

[31] C. PROCH. *Lisaac : implantation et définition d'un langage à base de prototypes.* 2002, Stage de DEA.

Bibliographie générale

[32] M. ABADI, L. CARDELLI. *On Subtyping and Matching.* in « Proc. of ECOOP », série LNCS, volume 952, Springer Verlag, pages 145-167, 1995.

[33] M. ABADI, A. D. GORDON. *A Calculus for Cryptographic Protocols : The Spi Calculus.* in « Proc. of ACCS », The ACM Press, pages 36-47, 1997.

[34] O. AGESEN. *The Cartesian Product Algorithm : Simple and Precise Type Inference Of Parametric Polymorphism.* in « Proc. of ECOOP », série LNCS, volume 952, Springer-Verlag, pages 2-26, 1995.

[35] A. APPEL. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[36] D. ASPINALL. *Proof General Home Page.* http://www.dcs.ed.ac.uk/, 2002.

[37] F. BARBANERA, M. FERNANDEZ, H. GEUVERS. *Modularity of Strong Normalization in the algebraic-λ-cube.* in « Journal of Functional Programming », numéro 6, volume 7, 1997, pages 613-660.

[38] H. BARENDREGT. *Lambda Calculi with Types.* éditeurs S. ABRAMSKY, D. GABBAI, T. MAIBAUM., in « Handbook of Logic in Computer Science », volume II, Oxford University Press, 1992, pages 118-310.

[39] B. BARRAS. *Coq in Coq.* Rapport Technique, numéro 3026, INRIA, Roquencourt, 1996, http://www.inria.fr/rrrt/rr-3026.html.

[40] Y. BERTOT. *A certified compiler for an imperative language.* rapport technique, numéro RR-3488, INRIA, 1998.

[41] Y. BERTOT, L. THÉRY. *The CtCoq System : Desing and Architecture.* rapport technique, numéro 3540, INRIA, 1998, http://www.inria.fr/rrrt/rr-3540.html.

[42] G. BLASHEK. *Type-safe OOP with prototypes : the concepts of Omega.* in « Structured Programming », numéro 12, volume 12, 1991, pages 1-9.

[43] V. BONO, M. BUGLIESI, L. LIQUORI. *A Lambda Calculus of Incomplete Objects.* in « Proc. of MFCS », série LNCS, volume 1113, Springer Verlag, pages 218-229, 1996.

[44] V. BONO, M. BUGLIESI, M. DEZANI-CIANCAGLINI, L. LIQUORI. *Subtyping for Extensible, Incomplete Objects.* in « Fundamenta Informaticae », numéro 4, volume 38, 1999, pages 325-364.

[45] V. BONO, K. FISHER. *An Imperative, First-Order Calculus with Object Extension.* in « In Proc. of ECOOP », série LNCS, numéro 1445, Springer Verlag, pages 462-497, 1998.

[46] V. BONO, L. LIQUORI. *A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects.* in « Proc. of CSL », série LNCS, volume 933, Springer Verlag, pages 16-30, 1995.

[47] P. BOROVANSKÝ, C. KIRCHNER, H. KIRCHNER, P.-E. MOREAU, C. RINGEISSEN. *An Overview of ELAN.* in « Proc. of WRLA », volume 15, Electronic Notes in Theoretical Computer Science, 1998.

[48] G. BOUDOL. *The π-Calculus in Direct Style.* in « Proc. of POPL », The ACM Press, pages 228-241, 1997.

[49] K. BRUCE. *Foundations of Object-Oriented Languages : Types and Semantics.* The MIT Press, 2002, http://www.cs.williams.edu/~kim/FOOLbook.html.

[50] K. BRUCE. *A Paradigmatic Object-Oriented Programming Language : Design, Static Typing and Semantics.* in « Journal of Functional Programming », numéro 2, volume 4, 1994, pages 127-206.

[51] K. BRUCE. *Increasing Java's expressiveness with `ThisType` and Match-bounded Polymorphism.* rapport technique, Williams College, 1997.

[52] K. BRUCE. *Subtyping is not a Good Match for Object-oriented Programming Languages.* in « Proc. of ECOOP », série LNCS, volume 1241, Springer Verlag, 1997.

[53] K. BRUCE. *Typing in Object-Orinted Language : Achieving Expressiveness and Safety.* rapport technique, Williams College, 1997.

[54] K. BRUCE, L. CARDELLI, G. CASTAGNA, T. H. O. GROUP, G. LEAVENS, B. PIERCE. *On Binary Methods.* in « Theory and Practice of Object Systems », numéro 3, volume 1, 1996.

[55] K. BRUCE, L. CARDELLI, C. B. PIERCE. *Comparing Object Encoding.* in « Proc. of TACS », série LNCS, volume 1281, Springer Verlag, pages 415-438, 1997.

[56] K. BRUCE, A. SHUETT, R. VAN GENT. *PolyTOIL : a Type-Safe Polymorphic Object Oriented Language.* in « Proc. of ECOOP », série LNCS, volume 952, Springer Verlag, pages 27-51, 1995.

[57] M. BUGLIESI, G. DELZANNO, L. LIQUORI, M. MARTELLI. *Object Calculi in Linear Logic.* in « Journal of Logic and Computation », numéro 1, volume 10, 2000, pages 75-104.

[58] J. BURSTEIN. *`Rupiah` : an Extension to Java Supporting Match-bounded Parametric Polymorphism, `ThisType`, and Exact Typing.* Bachelor thesis, Williams College, 1998.

[59] L. CARDELLI. *Obliq : A Language with Distributed Scope.* in « Computing Systems », numéro 1, volume 8, 1995, pages 27-59.

[60] L. CARDELLI, G. GHELLI, A. D. GORDON. *Mobility Types for Mobile Ambients.* in « Proc. of ICALP », série LNCS, numéro 1644, Springer Verlag, pages 230-239, 1999.

[61] L. CARDELLI, A. D. GORDON. *Mobile Ambients.* in « Theoretical Computer Science », numéro 1, volume 240, 2000.

[62] Y. CASEAU, F. LABURTHE. *CLAIRE : Combining Objects and Rules for Problem Solving.* in « Proc. of ICLP », éditeurs T. M. PRESS., pages 245-259, 1999.

[63] G. CHAMBERS. *The Cecil language specifications and rationale.* rapport technique, numéro 93-03-05, University of Washington, Dept. of Computer Science and Engineering, 1993.

[64] C. CHAMBERS, D. UNGAR. *Customization : Optimizing Compiler Technology for Self, a Dinamically-typed Object-Oriented Programming Language.* in « Proc. of PLDI », pages 146-160, 1989.

[65] H. CIRSTEA. *Calcul de Réécriture : Fondements et Applications.* Thèse de Doctorat, Université Henri Poincaré - Nancy I, 2000.

[66] H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *Matching Power.* in « Proc. of RTA », série LNCS, volume 2030, Springer Verlag, pages 168-183, 2001.

[67] H. CIRSTEA, C. KIRCHNER, L. LIQUORI. *The Rho Cube.* in « Proc. of ESOP/FOSSACS », série LNCS, volume 2051, Springer Verlag, pages 77-92, 2001.

[68] D. CLEMENT, J. DESPEYROUX, T. DESPEYROUX, G. KAHN. *A Simple Applicative Language : Mini-ML.* in « Proc. of LFP », ACM Press, 1986.

[69] D. COLNET, P. COUCAUD, O. ZENDRA. *Compiler Support to Customize the Mark and Sweep Algorithm.* in « Proc. of ISMM », pages 154-165, 1998.

[70] D. COLNET, O. ZENDRA. *Global System Analysis at Work.* in « Journal of Object-Oriented Programming », numéro 1, volume 14, 2001, pages 10-13.

[71] D. COLNET, O. ZENDRA. *Optimizations of Eiffel programs : SmallEiffel, The GNU Eiffel Compiler.* in « Proc. of TOOLS », IEEE Computer Society, pages 341-350, 1999.

[72] T. COQUAND, G. HUET. *The Calculus of Constructions.* in « Information and Computation », numéro 2/3, volume 76, 1988, pages 95-120.

[73] CRISTAL. *The Objective Caml.* 2000, http://pauillac.inria.fr/ocaml/.

[74] V. CROIZIER. *FunTalk : Formalisations et Validation.* 2001, Stage de DEA.

[75] F. DAMIANI, S. DROSSOPOULOU, M. DEZANI-CIANCAGLINI, P. GIANNINI. *Fickle : Dynamic Object Reclassification.* in « Proc. of ECOOP », série LNCS, Springer Verlag, 2001.

[76] J. DESPEYROUX. *A Higher-order specification of the pi-calculus.* in « Proc. of the IFIP-TCS », 2000.

[77] J. DESPEYROUX. *Sémantique Naturelle : Spécifications et Preuves.* rapport technique, numéro RR-3359, INRIA, 1998, http://www.inria.fr/rrrt/rr-3359.html.

[78] J. DESPEYROUX, A. FELTY, A. HIRSCHOWITZ. *Higher-order Abstract Syntax in Coq.* in « Proc. of TLCA », volume 902, Springer Verlag, pages 124-138, 1995.

[79] J. DESPEYROUX, A. HIRSCHOWITZ. *Higher-Order Syntax and Induction in Coq.* in « Proc. of LPAR », série LNCS, volume 822, Springer Verlag, pages 159-173, 1994.

[80] J. DESPEYROUX, P. LELEU. *A Modal λ-calcul with iteration and case constructs.* in « Proc. of TYPES », série LNCS, Springer Verlag, 1998.

[81] J. DESPEYROUX, P. LELEU. *Primitive recursion for higher-order abstract syntax with dependant types.* in « Proc. of IMLA », 1999, FLoC satellite workshop.

[82] J. DESPEYROUX, F. PFENNING, C. SCHÜRMANN. *Primitive Recursion for Higher-Order Abstract Syntax.* in « Proc. of TLCA », série LNCS, numéro 1210, Springer Verlag, pages 147-163, 1997.

[83] K. FISHER. *Type System for Object-Oriented Programming Languages.* thèse de doctorat, University of Stanford, 1996.

[84] K. FISHER, D. GROSSMANN, D. MACQUEEN, R. PUCELLA, J. REPPY, J. RIECKE, S. WEIRICH. *The Moby Programming Language.* 2000, http://www.cs.uchicago.eduindex.html.

[85] K. FISHER, J. H. REPPY. *Extending Moby with Inheritance-Based Subtyping.* in « Proc. of ECOOP », série LNCS, volume 1850, Springer Verlag, pages 83-107, 2000.

[86] K. FISHER, J. H. REPPY. *The Design of a Class Mechanism for Moby.* in « Proc. of PLDI », série SIGPLAN Notices, volume 34, The ACM Press, pages 37-49, 1999.

[87] C. FOURNET, G. GONTHIER, J.-J. LÉVY, L. MARANGET, D. RÉMY. *A Calculus of Mobile Agents.* in « Proc. of CONCUR », série LNCS, volume 1119, Springer Verlag, pages 406-421, 1996.

[88] M. J. GABBAY, A. M. PITTS. *A New Approach to Abstract Syntax with Variable Binding.* in « Formal Aspects of Computing », 2001, Special issue in honour of Rod Burstall. To appear..

[89] G. GILLARD. *A formalization of a concurrent object calculus up to alpha-conversion.* in « Proc. of CADE », série LNCS, Springer Verlag, 2000.

[90] G. GILLARD. *Formalisation des langages concurrents et à objets modulo l'alpha-conversion.* Ph.D. thesis, Nice University, 2001.

[91] A. D. GORDON. *A Mechanisation of Name-Carrying Syntax up to Alpha-Conversion.* in « Proc. of HUG », série LNCS, volume 780, Springer-Verlag, 1994.

[92] J. HANNAN, F. PFENNING. *Compiler Verification in LF.* in « Proc. of LICS », pages 407-418, 1992.

[93] W. A. HOWARD. *The Formulae-as-Types Notion of Construction.* éditeurs J. P. SELDIN, J. R. HYNDLEY., in « To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism », Academic Press, 1980, pages 479-490.

[94] G. HUET, G. KAHN, C. PAULIN-MOHRING. *The Coq Proof Assistant - A tutorial, Version 6.1.* rapport

technique, numéro 204, INRIA, 1997, http://www.inria.fr/rrrt/rt-0204.html, Version révisée distribuée avec Coq. http://coq.inria.fr/.

[95] A. IGARASHI, B. PIERCE, P. WADLER. *Featherweight Java : A minimal core calculus for Java and GJ.* in « Proc. of OOPSLA », The ACM Press, pages 132-146, 1999.

[96] F. LANG, P. LESCANNE, L. LIQUORI. *A Framework for Defining Object-Calculi (Extended Abstract).* in « Proc. of FM », série LNCS, volume 1709, Springer Verlag, pages 963-982, 1999.

[97] P. LEQUANG, Y. BERTOT, L. RIDEAU, L. POTTIER. *The Pcoq System.* rapport technique, INRIA, 1999, http://www-sop.inria.fr/lemme/pcoq/.

[98] L. LIQUORI, G. CASTAGNA. *A Typed Lambda Calculus of Objects.* in « Proc. of Asian », série LNCS, volume 1179, Springer Verlag, pages 129-141, 1996.

[99] L. LIQUORI. *An Extended Theory of Primitive Objects.* rapport technique, numéro CS-23-96, Computer Science Department, University of Turin, Italy, 1996.

[100] L. LIQUORI. *The Deep Blue Calculus.* 1998, http://www.loria.fr/~lliquori/PAPERS/deep-blue.ps.gz, Manuscript.

[101] Z. LUO. *ECC : An Extended Calculus of Constructions.* in « Proc. of LICS », IEEE Computer Society, pages 385-395, 1990.

[102] Z. LUO, R. POLLACK. *The LEGO Proof Development System : A User's Manual.* rapport technique, numéro ECS-LFCS-92-211, University of Edinburgh, 1992.

[103] O. MADSEN, K. MOLLER-PEDERSEN. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[104] G. MASINI, A. NAPOLI, D. COLNET, D. LÉONARD, K. TOMBRE. *Object-Oriented Languages.* Academic Press, Lyon, 1991.

[105] J. MCKINNA, R. POLLACK. *Pure Type Systems Formalized.* in « Proc. of TLCA », série LNCS, volume 664, Springer Verlag, pages 289-305, 93.

[106] J. C. MICHELL. *Toward a Typed Foundation for Method Specialization and Inheritance.* in « Proc. of POPL », The ACM Press, pages 109-124, 1990.

[107] D. MILLER. *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification.* in « Proc. of ELP », série LNCS, volume 475, Springer Verlag, pages 253-281, 1991.

[108] R. MILNER, J. PARROW, D. WALKER. *A Calculus of Mobile Processes I and II.* in « Information and Computation », numéro 1, volume 100, 1992, pages 1-77.

[109] T. NIPKOW, D. VON OHEIMB. *Java Light is Type-Safe - Definitely.* in « Proc. of POPL », The ACM Press, pages 161-170, 1998.

[110] éditeurs J. NOBLE, A. TAIVALSAARI, I. MOORE., *Prototype-Based Object-Oriented Programming : Concepts, Languages, and Applications.* Springer Verlag, 1999.

[111] J. NORDLANDER. *Reactive Objects and Functional Programming.* thèse de doctorat, Dept. of Computing Science, Chalmers University of Technology, Göteborg, Sweden, 1999.

[112] U. OF CAMBRIDGE COMPUTER LABORATORY. *The HOL System.* 1993.

[113] J. PALSBERG, T. JIM. *Type Inference for Simple Object Types is NP-Complete.* in « Nordic Journal of Computing », numéro 3, volume 4, 1997, pages 259-286.

[114] J. PALSBERG. *Efficient Inference of Object Types.* in « Proc. of LICS », pages 186-195, 1993.

[115] L. C. PAULSON. *Isabelle : The next 700 theorem provers.* éditeurs P. ODIFREDDI., in « Logic and Computer Science », Academic Press, 1990, pages 361-386.

[116] L. C. PAULSON. *Isabelle : A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[117] J. PETERSON, K. HAMMOND, L. AUGUSTSSON, B. BOUTEL, W. BURTON, J. FASEL, A. GORDON, J. HUGHES, P. HUDAK, T. JOHNSSON, M. JONES, E. MEIJER, S. PEYTON-JONES, A. REID, P. WADLER. *Haskell 1.4 : A Non-strict, Purely Functional Language.* 1997, http://www.haskell.org/onlinereport/.

[118] S. PEYTON-JONES. *The implementation of functional programming languages.* Prentice Hall, Inc., 1987.

[119] R. POLLACK. *The Theory of LEGO : A Proof Checker for the Extended Calculus of Constructions.* thèse de doctorat, University of Edinburgh, 1994.

[120] K. R. RAJENDRA, E. TEMPERO, H. M. LEVY, A. P. BLACK, N. C. HUTCHINSON, E. JUL. *Emerald : a general-purpose programming language.* in « Software Practice and Experience », numéro 1, volume 21, 1991, pages 91-118.

[121] J. REPPY. *Local CPS conversion in a direct style compiler.* rapport technique, Bell Labs, Lucent Technologies, 2000.

[122] J. RIECKE, C. STONE. *Privacy via Subsumption.* in « Proc. of FOOL », 1998, Also in *Theory and Practice of Object Systems*.

[123] D. RÉMY. *From Classes to Objects via Subtyping.* in « Proc. of ESOP », série LNCS, volume 1381, Springer Verlag, pages 200-220, 1998.

[124] S. SALVATI. *FunTalk : Formalisations et Démonstrations.* 2001, Stage de DEA.

[125] N. SHANKAR, S. OWRE, J. M. RUSHBY. *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, 1993.

[126] R. SIMON, R. STAPF, B. MEYER. *Full Eiffel on .NET.* 2002, http://www.inf.ethz.ch/personal/meyer/publications/msdn/eiffel-net/.

[127] B. SONNTAG. *Utilisation de la segmentation mémoire du processeur à partir d'un langage de haut niveau.* in « Proc. of *Conférence Française sur les Systèmes d'Exploitation*, CFSE », pages 107-116, 2001.

[128] A. TAIVALSAARY. *Kevo, a prototype-based object-oriented language based on concatenation and module operations.* rapport technique, numéro LACIR 92-02, University of Victoria, 1983.

[129] L. VAN BENTHEM JUTTING, J. MCKINNA, R. POLLACK. *Checking Algorithms for Pure Type Systems.* in « Proc. of Types », série LNCS, volume 806, Springer Verlag, pages 19-61, 1994.

[130] V. VAN OOSTROM. *Lambda Calculus with Patterns.* Technical Report, numéro IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[131] B. WERNER. *Une Théorie des Constructions Inductives.* Thèse de Doctorat, Université Paris 7, 1994.

[132] O. ZENDRA, D. COLNET, P. COUCAUD. *With SmallEiffel, The GNU Eiffel Compiler, Eiffel joins the Free Software community.*. in « GNU Bulletin », volume 25, 1998.

[133] O. ZENDRA, D. COLNET. *Coping with aliasing in the GNU Eiffel Compiler implementation.* in « Software - Practice and Experience », numéro 6, volume 31, 2001, pages 601-613.

[134] O. ZENDRA, K. DRIESEN, F. QIAN, L. HENDREN. *Evaluation of the runtime performance of control flow structures for dynamic dispatch in Java.* Poster and extended abstract. OOPSLA Conference Companion, The ACM Press, 2001.

[135] O. ZENDRA. *Traduction et optimisation dans les langages à objets.* Thèse de Doctorat, Université Henri Poincaré - Nancy 1, 2000.

[136] THE PYTHON ORGANIZATION. *The Python Language.* 2002, http://www.python.org.

[137] THE RUBY ORGANIZATION. *The Ruby Language.* 2002.

[138] THE SMLNJ TEAM. *The Standard ML of New Yersey.* 2002, http://cm.bell-labs.com/cm/cs/what/smlnj/index.html/.

[139] THE TWELF TEAM. *The Twelf Proof Assistant.* rapport technique, CMU, 2002.

[140] THE VERIFICARD TEAM. *The Verificard IST Project.* 2002, http://www.verificard.com/.

[141] WEB SITE DE L'AMBASSADE DE FRANCE AU ÉTAS-UNIS. *Catching bugs would save big bucks.* Site web de l'ambassade de France au Étas-Unis, juillet, 2002, http://www.info-france-usa.org/fr.

[142] WEB SITE FUNDACIÓ JOAN MIRÓ. 2002, http://www.info-france-usa.org/fr/.