**Département de formation doctorale en informatique** **École doctorale IAE + M**
**UFR STMIA**

# Compilation, langages à objets et programmation par contrats.

**Expérience acquise dans le cadre du projet**

# "SmallEiffel The GNU Eiffel Compiler"

## MÉMOIRE

présenté et soutenu publiquement le 22 décembre 2000

pour l'obtention de l'

**Habilitation de l'Université Henri Poincaré – Nancy I**

**(Spécialité Informatique)**

par

Dominique Colnet

**Composition du jury**

| | |
|---|---|
| *Président :* | Jean-Laurent MALLET |
| *Rapporteurs :* | Denis CAROMEL<br>Jean-Marc JÉZÉQUEL<br>Claude KIRCHNER |
| *Examinateur :* | Claude GODART |

**Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503**

Mis en page avec la classe thloria.

# Remerciements

Je tiens à remercier en tout premier lieu Olivier Zendra avec qui j'ai réalisé en juillet 1995 la toute première auto-compilation, *bootstrap*, de SmallEiffel. Même si je n'arrive pas à m'empêcher de lui reprocher ici même son petit coté *grognon*, SmallEiffel ne serait pas ce qu'il est aujourd'hui sans le travail d'Olivier !

Je tiens ensuite à remercier Philippe Coucaud, avec qui Olivier et moi avons passé d'intenses, et souvent nocturnes, séances de déverminage pour concevoir, tester, et mettre au point le ramasse-miettes.

Je tiens à remercier tout particulièrement Jean-Marc Jézéquel Chargé de Recherche à l'IRISA qui, en 1996 déjà, m'envoyait les premiers rapports d'anomalies depuis le lointain Japon. Si Jean-Marc n'a que deux étoiles dans la liste des contributeurs (page 82) c'est d'une part pour qu'on ne m'accuse pas de soudoyer mes rapporteurs, et d'autre part à cause des membres de son équipe qui ont déjà pris tout ce qui nous restait d'étoiles.

Je tiens à remercier Claude Kirchner, Directeur de Recherche à l'INRIA Lorraine, rapporteur en 1989 de ma Thèse d'Université, d'avoir accepté, onze années plus tard, de récidiver pour mon habilitation.

C'est un plaisir pour moi de remercier Jean-Laurent Mallet, Professeur à l'école de Géologie, directeur et responsable du célèbre projet Gocad (`http://pangea.standford.edu/gocad`), de me faire l'honneur de présider ce jury.

Je tiens à remercier Denis Caromel, Professeur à l'Université de Nice, fin connaisseur d'Eiffel, d'avoir accepté d'être rapporteur.

Je remercie enfin Claude Godart, Professeur à l'ESTIN, directeur de l'équipe ECOO à laquelle j'appartiens, de participer à ce Jury.

Merci également aux relecteurs des versions intermédiares de ce document : Jacques Jaray, Noelle Carbonell, Olivier Festor, Suzanne Collin et Olivier Zendra.

En dernier lieu, je tiens à remercier les différentes institutions, le CRIN, le LORIA, le département informatique, l'ESIAL ainsi que les contribuables français qui sont les véritables mécènes du projet SmallEiffel.

*A Marie, Lucien, Bénédicte, Suzanne et Suzanne.*

## Résumé

Ce mémoire d'habilitation à diriger des recherches présente de façon synthétique les activités de recherche que j'ai menées dans le cadre du projet "SmallEiffel The GNU Eiffel Compiler".

Le chapitre 1 présente le projet SmallEiffel, projet qui occupe une grande partie de mon temps depuis 1994, date à laquelle j'ai commencé l'écriture du premier compilateur. J'essaye également dans ce chapitre d'expliquer quelles furent mes motivations en démarrant ce projet ambitieux.

L'hypothèse du système fermé, chapitre 2, est un point essentiel dans les résultats de SmallEiffel. Grâce à cette approche, non seulement nous sommes en mesure de prédire statiquement le type du receveur pour un très grand nombre d'appels, mais aussi, de proposer une implantation très efficace de la liaison dynamique, point clef de l'implantation des langages à objets. C'est également grâce à l'hypothèse du système fermé que nous sommes en mesure de spécialiser la gestion mémoire ainsi que l'implantation de la généricité.

Le chapitre 3 intéressera peut-être ceux qui souhaitent commencer l'écriture de la partie frontale d'un compilateur sans utiliser ni *lex* ni *yacc*. En effet, comme j'essaye de le montrer dans ce chapitre, il est possible d'utiliser une approche plus flexible, simple et très performante, sans avoir recours à un constructeur d'analyseurs syntaxiques. Ceci étant, lex et yacc restent des outils adaptés à beaucoup d'autres tâches plus particulièrement dans un environnement C/UNIX.

Le chapitre 4 présente l'*aliasing*, les travaux du domaine et surtout, comment s'en servir pour optimiser son application. Grâce à l'utilisation de la programmation par contrats et aux règles d'exportation d'Eiffel, nous montrons comment il est possible de mettre en œuvre cette technique de façon plus systématique et plus sûre.

Je passe en revue dans le chapitre 5 l'ensemble de mes publications avant de conclure et de dresser un bilan de mon travail dans le chapitre 6.

L'annexe A décrit succinctement le travail réalisé pour le projet ProMic dans le cadre d'une convention de recherche avec obligations de résultats. Le développement du logiciel ProMic, nous a permis de valider une seconde fois, l'intérêt d'Eiffel et de la programmation par contrats dans un autre contexte : un environnement interactif de développement pour micro-contrôleurs synthétisables.

L'annexe B vient compléter le chapitre 3 en donnant le texte source d'un programme écrit en *scheme* qui permet de calculer les symboles directeurs d'une grammaire LL(1).

J'ai cru bon d'ajouter en annexe C de ce mémoire la liste des personnes qui ont contribué à l'effort important que représente le projet SmallEiffel. Enfin, pour donner un aperçu de l'ouvrage, l'historique des 25 versions diffusées à ce jour est ajouté en annexe D.

http://SmallEiffel.loria.fr

# Table des matières


**Table des figures** **xi**

**xiii**

**1 Introduction** **1**

1.1 Problématique et contexte . . . 1
1.2 De l'édition structurelle à la programmation par objets . . . 2
1.3 De la programmation par objets à Eiffel . . . 3
1.4 SmallEiffel The GNU Eiffel Compiler . . . 4
1.5 Guide de lecture . . . 5

**2 Analyse globale - L'hypothèse du système fermé** **7**

2.1 Introduction . . . 7
2.2 Calcul du code vivant - Calcul des types vivants . . . 8
2.3 Implantation de la liaison dynamique par sélection dichotomique . . . 9
2.4 Spécialisation des méthodes vivantes selon le type du receveur . . . 10
2.5 Généricité dans un système fermé . . . 12
2.6 Héritage multiple dans un système fermé . . . 12
2.7 Spécialisation du ramasse-miettes . . . 13
2.8 Discussion, conclusion et perspectives . . . 14

**3 Analyse syntaxique et analyse lexicale : un mode d'emploi** **17**

3.1 Introduction . . . 17
3.2 Les objectifs de notre implantation . . . 18
3.3 Analyse syntaxique et lexicale en une seule phase . . . 19
3.4 Suite d'entités syntaxiques identiques . . . 20
3.5 Analyse descendante récursive – Utilisation du positionnement exact . . . 21
3.6 Exemple de fonction de niveau lexical . . . 22
3.7 Comparaison des performances avec un analyseur ascendant . . . 23
3.8 Discussion et conclusion . . . 25

**4 Aliasing et programmation par contrats 27**
4.1 Introduction . . . . . . . . 27
4.2 L'aliasing : définition . . . . . . . . 28
4.3 Travaux du domaine . . . . . . . . 28
4.4 Intérêt de l'aliasing pour implanter un compilateur . . . . . . . . 30
4.5 Utilisation d'un fournisseur d'alias . . . . . . . . 31
4.5.1 Mise en oeuvre du modèle de conception Singleton en Eiffel . . . . . . . . 31
4.5.2 Implantation d'un fournisseur d'alias . . . . . . . . 32
4.6 Promouvoir et contrôler l'usage du fournisseur d'alias . . . . . . . . 33
4.7 Aliasing pour les objets modifiables . . . . . . . . 34
4.8 Résultats expérimentaux . . . . . . . . 36
4.9 Conclusions . . . . . . . . 38

**5 Résumés de mes publications 39**
5.1 Livres . . . . . . . . 39
5.2 Publications internationales avec comité de lecture éditées dans une revue . . . . 41
5.3 Publications internationales avec comité de lecture . . . . . . . . 42
5.4 Publications nationales avec comité de lecture . . . . . . . . 47
5.5 Thèse . . . . . . . . 47
5.6 Articles de vulgarisation scientifique . . . . . . . . 48
5.7 Edition d'actes . . . . . . . . 50
5.8 Rapports techniques et supports d'enseignements . . . . . . . . 50

**6 Bilan et perspectives 55**
6.1 Bilan et apports de nos travaux . . . . . . . . 55
6.1.1 Eiffel et analyse globale . . . . . . . . 56
6.1.2 Analyse statique globale et prédiction de type . . . . . . . . 56
6.1.3 Compilation sur nécessité . . . . . . . . 57
6.1.4 Spécialisation du code généré . . . . . . . . 57
6.1.5 Une technique novatrice d'implantation de la liaison dynamique . . . . . . 58
6.1.6 Un mode d'emploi pour l'analyse syntaxique . . . . . . . . 59
6.1.7 Impact de l'aliasing et programmation par contrats . . . . . . . . 59
6.1.8 Application: SmallEiffel, The GNU Eiffel Compiler . . . . . . . . 60
6.2 Perspectives à court, moyen et long terme . . . . . . . . 62
6.2.1 Extensions prévues d'Eiffel . . . . . . . . 62
6.2.2 Amélioration de l'analyse globale . . . . . . . . 63
6.2.3 Couplage avec des techniques d'analyse dynamique . . . . . . . . 63

6.2.4 Généralisation à des systèmes non figés . . . . . 63
6.2.5 Impact d'une modification . . . . . 64
6.2.6 Généralisation et extension à d'autres langages . . . . . 64
6.2.7 Sécurité du typage . . . . . 64
6.2.8 Micro-contrôleurs synthétisables : le projet ProMic . . . . . 66
6.2.9 Un nouveau langage à objets ? . . . . . 66

**Annexe** **69**

**A ProMic, Programmation des Micro-contrôleurs** **69**

**B Calcul des symboles directeurs** **75**
B.1 Représentation de la grammaire . . . . . 75
B.2 Calcul des symboles directeurs . . . . . 76
B.3 Utilisation - Exemple . . . . . 79

**C SmallEiffel Contributors** **81**

**D SmallEiffel's History of changes** **85**

**Bibliographie** **95**

**Index** **103**

# Table des figures

2.1 Implantation de la liaison dynamique par sélection dichotomique . . . . . . . . . 10
2.2 Spécialisation des méthodes vivantes selon le type du receveur . . . . . . . . . . . . 11

3.1 Analyse lexicale avec utilisation du contexte (a_then_part) . . . . . . . . . . . . 20
3.2 Utilisation d'itérations en analyse syntaxique descendante (a_compound) . . . . 21
3.3 Analyse syntaxique descendante à l'aide de fonctions (a_e0 a_r1) . . . . . . . . . 22
3.4 Analyse lexicale d'un identificateur (a_identifier) . . . . . . . . . . . . . . . . . . 23
3.5 Performances de l'analyse syntaxique ascendante vs. descendante . . . . . . . . . 24

4.1 Bootstrap de SmallEiffel avec et sans aliasing des string . . . . . . . . . . . . . . 37

6.1 Violation des règles de typage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

A.1 ProMic : capture d'écran numéro 1 . . . . . . . . . . . . . . . . . . . . . . . . . . 71
A.2 ProMic : capture d'écran numéro 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 72
A.3 ProMic : capture d'écran numéro 3 . . . . . . . . . . . . . . . . . . . . . . . . . . 73

B.1 Notation en scheme et notation BNF . . . . . . . . . . . . . . . . . . . . . . . . . 76

## Avant propos

J'ai découvert l'informatique en 1979, à l'Université de Metz, alors que je commençais un DEUG. C'est en partie à cause du traceur de courbes, que l'on pilotait en *basic* avec le calculateur Hewlett Packard, dont l'écran se réduisait à une seule ligne de *led* rouge, que j'ai décidé de faire un petit peu plus d'informatique. Je tiens à m'excuser auprès du traceur de courbes pour les *bugs* dans mes programmes qui provoquèrent de nombreuses collisions du stylet d'écriture contre la bordure métallique de la table de traçage.

Depuis cette époque, vingt années ce sont écoulées et mon intérêt pour l'informatique et les ordinateurs en général ne cesse de grandir. Cela m'étonne parfois moi-même ...

# Chapitre 1

# Introduction

"We called Smalltalk Smalltalk so that nobody would expect anything from it."

Alan Kay [We84]

## 1.1 Problématique et contexte

De nouveaux langages apparaissent régulièrement qui répondent à de nouveaux besoins, permettent aux développeurs de mieux exprimer leurs idées et introduisent de nouveaux concepts — en général de plus haut niveau. Ceci implique fort logiquement la réalisation de nouveaux compilateurs, toujours plus puissants.

De plus, l'accroissement de la complexité des tâches effectuées par les programmes informatiques nécessite des puissances de calcul toujours plus élevées, qui sont fournies en partie par l'augmentation de la puissance du matériel (microprocesseurs) et en partie par les progrès réalisés par les logiciels, dans le domaine de la compilation et plus précisément de l'optimisation des programmes.

C'est à ce niveau, *l'optimisation des programmes*, que se situe la problématique générale de notre travail, avec pour objectif de répondre, — au moins partiellement, car il semble qu'une réponse définitive soit utopique — à cette éternelle question: *Comment améliorer les performances des programmes compilés, c'est à dire comment faire en sorte que les programmes soient plus rapides et moins consommateurs de ressources?*

Une question à la fois aussi simple et aussi vaste peut être abordée de façons très différentes et avoir des réponses multiples, dépendant du domaine traité.

Nos recherches menées durant ces cinq dernières années se sont donc focalisées sur *la compilation et l'optimisation pour les langages de classes*. Ces derniers sont un sous-ensemble des *langages à objets* [MNC+89] et comprennent entre autres Smalltalk [GR83], Objective C [Cox83], Object Pascal [Sch86], C++ [Str86], Eiffel [Mey92] et Java [GJS96]. Comme l'indique le titre de mon mémoire, nous avons plus spécifiquement effectué nos recherches sur et avec le langage Eiffel, langage de classes particulièrement élégant, facile d'utilisation et puissant, offrant des constructions telles qu'héritage multiple, types paramétriques avec généricité contrainte ou non contrainte, assertions pour le support de la conception par contrats [Mey88]. Dans le cadre de ce travail, nous avons également eu pour objectif de proposer des optimisations de haut niveau et qui soient *portables*.

La première partie de notre travail, du point de vue chronologique comme du point de vue de l'importance des efforts fournis et des contributions apportées, concerne donc la conception et l'implantation d'un système de compilation efficace, c'est à dire capable de compiler très rapidement les programmes tout en générant un code spécialisé très fortement optimisé. Nous nous sommes focalisés dans cette première partie sur la compilation et la prise en compte des aspects *spécifiques aux langages de classes*, en mettant tout particulièrement l'accent sur *l'optimisation de la liaison dynamique*. En effet, cette dernière est un aspect des langages de classes qui est fondamental non seulement du point de vue conceptuel mais aussi en ce qui concerne les performances. Afin de *spécialiser* autant que possible l'ensemble du programme compilé, nous avons adopté une *approche globale* au système. Nous nous appuyons ainsi sur des algorithmes d'analyse globale, de prédiction de type et d'expansion en ligne (*inlining*) puissants. Nous proposons une nouvelle méthode d'implantation de la liaison dynamique [ZCC97], plus performante que les techniques actuelles, qui constitue une contribution majeure. Ces algorithmes permettent d'effectuer de façon automatique des optimisations ciblées sur les points importants pour les langages de classes.

Un second axe pour nos recherche a été le problème de la gestion mémoire automatique des applications compilées. Nous avons choisi de confier la génération et l'optimisation du ramasse-miettes au compilateur, afin d'avoir un processus aussi automatisé que possible, facilitant ainsi le travail du développeur d'applications. Nous nous sommes donc penchés sur la conception et l'intégration au processus de compilation d'un système de *génération* de ramasse-miettes *automatiquement adaptés* aux applications compilées. Nous proposons un système qui optimise les ramasse-miettes en étendant et généralisant au problème de la gestion automatique de la mémoire les algorithmes et techniques de nos premiers travaux, notamment l'analyse globale, la prédiction de type et la duplication-spécialisation de code. Nous montrons que ce système produit des ramasse-miettes optimisés ayant de très bonnes performances, du niveau des meilleurs ramasse-miettes actuels.

Enfin, nous avons exploré une troisième direction de recherche, complémentaire aux précédentes, ayant bien entendu toujours pour but d'optimiser les programmes. Il s'agit de l'étude de l'*aliasing* dans un langage de classes et de son impact sur le processus de compilation. L'aliasing est en effet souvent considéré comme néfaste et devant être évité. Il peut cependant présenter un certain nombre d'avantages, notamment en termes de confort de développement et de performances à l'exécution, qui font qu'il peut également être très précieux. Nos recherches ont eu pour but de tenter de caractériser et de mesurer l'impact de l'aliasing, ce qui est assez rarement fait, ainsi que de tirer parti des avantages de l'aliasing tout en maîtrisant ses inconvénients. Nos recherches dans ce domaine ont été moins profondes que celles mentionnées précédemment, mais elles présentent des résultats initiaux très intéressants et prometteurs.

## 1.2 De l'édition structurelle à la programmation par objets

Pour ceux qui s'en souviennent, le titre de ma thèse d'Université [Col89] est "*Prototypage, programmation objet : application à l'édition structurelle*". Comme ne l'indique pas forcément le titre, j'ai d'abord commencé mon travail de thèse en m'intéressant à l'édition structurelle. Il est amusant de voir maintenant, avec le recul, que l'édition structurelle est, à ma connaissance, un thème de recherche complètement abandonné. Pour être franc, je dois également avouer que l'édition syntaxique ne fut pour moi qu'un thème de travail particulier, une sorte de challenge technique, un défi à relever.

En effet, désireux d'implanter un éditeur syntaxique performant, je me souviens avoir effectué une première tentative de réalisation pratique, en Pascal, sur les premiers terminaux alphanumérique, 24 ou 25 lignes je ne sais plus, mais avec 80 ou 132 colonnes au choix.

Avec la diffusion du système UNIX et de son langage C, les sources de la première tentative en Pascal furent vite oubliés pour repartir sur de nouvelles bases. L'expérience acquise en Pascal, nous permit alors d'atteindre rapidement, puis de dépasser les fonctionnalités de la première version.

Les contrats et les relations de l'équipe Génie Logiciel, mon équipe d'appartenance de l'époque, m'amenèrent ensuite à effectuer une tentative avec le langage LTR3, langage développé pour l'armée française. Cette tentative infructueuse fut rapidement abandonnée pour une tentative utilisant le langage Ada, plus connu. Ce nouveau changement de langage se révéla être rapidement un échec, non pas à cause de déficiences dans la définition du langage, car Ada était un langage véritablement moderne pour l'époque[1], mais à cause de la médiocrité du compilateur dont nous disposions.

Après un bref retour au langage C sous UNIX, je découvre Lisp grâce à l'excellente implantation de Jerôme Chailloux, Le_Lisp de l'INRIA [CDH84]. Le fait de pouvoir travailler dans un environnement véritablement interactif comme celui de Le_Lisp décuple nos possibilités. L'éditeur syntaxique commence à prendre de l'allure même si le terminal alpha-numérique, caractères vert sur fond d'écran noir, *scrolling* programmable, possède toujours 80 ou 132 colonnes au choix. Notons au passage que Le_Lisp permettait de programmer son application de façon portable, indépendamment des caractéristiques du terminal. Un beau progrès pour l'époque.

Ensuite, les choses s'accélèrent quand je commence à utiliser la couche objets Ceyx [Hul83] [Col86] qui fonctionne sur l'interprète Le_Lisp. Ce sont les premières définitions de classes et les premiers envois de messages avec liaison dynamique. Pour couronner le tout, un Macintosh avec une souris et un *bitmap* noir et blanc, plus exactement un Lisa, arrive dans notre équipe. En outre Le_Lisp et Ceyx sont véritablement portables et portés sur Lisa. Grâce aux 128 Kilo Octets du Lisa, sa puissance de calcul, processeur Motorola 68010, le travail avance très vite. Gépi [CCCD88], notre éditeur syntaxique voit enfin le jour.

## 1.3 De la programmation par objets à Eiffel

Le projet Gépi m'a donné l'occasion d'essayer plusieurs langages et surtout m'a permis de découvrir l'expressivité offerte par la liaison dynamique des langages à objets. Sur ce dernier point, la moitié d'un chapitre de ma thèse ([Col89] chapitre 7) est consacré à l'utilisation de la liaison dynamique pour décompiler un arbre abstrait. Cette même technique est utilisée dans SmallEiffel pour l'indenteur de programme, `pretty`, et pour l'extracteur d'interface, `short`.

En parallèle avec mon travail sur l'édition structurelle, nous avons commencé par écrire un premier polycopié sur les langages à objets [CMN+86]. Le travail a ensuite continué pour aboutir à un ouvrage entier, comparant de façon exhaustive, tous les langages à objets de l'époque ([MNC+89] voir page 39). C'est à l'occasion de la rédaction de cet ouvrage que nous avons découvert Eiffel.

Toujours dans le cadre de cet ouvrage, nous avons eu l'occasion d'étudier l'implantation des différents langages et, dès cette époque, je me souviens avoir eu l'envie de réaliser mon propre compilateur. Pour être plus précis, je souhaitais compiler des programmes Smalltalk dans le but d'en optimiser l'exécution.

Suite à la décision d'utiliser Eiffel pour l'enseignement en Licence d'informatique en 1990-1991, j'ai été amené à utiliser ce langage de plus en plus souvent. C'est en partie à cause de la

1. La première version du langage Ada ne peut pas être considérée comme faisant partie de la famille des langages à objets; aucun polymorphisme de variable ni bien sûr de liaison dynamique dans cette première version. Un chapitre complet de notre ouvrage [MNC+89] est consacré à une comparaison de Smalltalk avec Ada.

mauvaise qualité des premières implantations d'Eiffel que j'ai repris mon idée d'implanter mon propre compilateur.

J'ai donc abandonné l'idée de travailler sur Smalltalk afin de faire d'une pierre deux coups : un travail de recherche intéressant et un produit gratuit, utilisable aussi pour l'enseignement.

J'ai commencé à travailler sur SmallEiffel en mai-juin 1994 pour obtenir une première version auto-compilable du compilateur en juin 1995. Cette première version a ensuite été utilisée de façon interne par les étudiants de licence d'Informatique et par les élèves de l'ESIAL avant d'être disponible gratuitement à l'extérieur (`http://SmallEiffel.loria.fr`).

## 1.4 SmallEiffel The GNU Eiffel Compiler

Contrairement à ce que pourrait laisser penser son nom, SmallEiffel n'est pas un compilateur pour un sous-ensemble du langage Eiffel, mais bien un véritable compilateur pour le langage Eiffel [Mey92]. Le *Small* de SmallEiffel est emprunté à Smalltalk [Gol84], langage et environnement que j'ai pratiqués à la fois lors de la rédaction du livre sur les langages à objets [MNC+89] mais aussi de façon intensive dans la réalisation du projet ReDraw [CC94].

En fait, il faut bien reconnaître que l'appellation *SmallEiffel The GNU Eiffel Compiler* est véritablement trompeuse quand on sait que la distribution actuelle de SmallEiffel comprend :

- un compilateur Eiffel vers C ANSI, composé des commandes `compile` et `compile_to_c`,
- un compilateur Eiffel vers la machine virtuelle Java, la commande `compile_to_jvm` qui produit du *bytecode* Java,
- un outil de documentation capable d'extraire l'interface des classes Eiffel, la commande `short` qui peut produire du texte brut mais aussi du source TEX, LATEX ou *html* par exemple,
- un indenteur de programme, `pretty`, qui sert à réordonner le contenu d'un fichier Eiffel en tenant compte du niveau d'imbrication des constructions,
- un désassembleur de *bytecode* Java, la commande `print_jvm_class` originalement écrite pour vérifier le *bytecode* produit par `compile_to_jvm`,
- une bibliothèque de classes d'intérêt général comprenant les principales structures de données habituelles :
  - diverses sortes de tableaux: ARRAY, FIXED_ARRAY, ARRAY2, ARRAY3 ainsi que NATIVE_ARRAY pour l'interfaçage de bas niveau avec C ou Java par exemple,
  - des listes chaînées à chaînage simple ou double, munies d'un cache mémoire automatique : LINKED_LIST et TWO_WAY_LINKED_LIST,
  - les tables de hachage dont la représentation est inspirée de la *Standard Template Library* (STL) C++ : la classe DICTIONARY,
  - des ensembles : les classes COLLECTION et SET,
  - des itérateurs externes [ZC99a] : ITERATOR, ITERATOR_ON_COLLECTION, ITERATOR_ON_STRING, ITERATOR_ON_DICTIONARY_ITEMS, ITERATOR_ON_SET, ITERATOR_ON_DICTIONARY_KEYS, ITERATOR_ON_LINKED_LIST, etc.,
  - une bibliothèque de calcul numérique sans perte de précision dont la hiérarchie est inspirée de celle de Smalltalk mais dont l'implantation est optimisée : les classes NUMBER, SMALL_INTEGER, LARGE_POSITIVE_INTEGER, LARGE_NEGATIVE_INTEGER, INTEGER_FRACTION, NUMBER_FRACTION,

  - des classes de manipulation de fichiers et de répertoires indépendamment du système d'exploitation utilisé [2] : FILE_TOOLS, BASIC_DIRECTORY, DIRECTORY,
  - des classes de manipulation d'heures et de dates dans différents langages : BASIC_TIME, TIME_IN_FRENCH, TIME_IN_ENGLISH, TIME_IN_ITALIAN, ...,
  - une bibliothèque comportant les algorithmes de tri les plus courants : classes COLLECTION_SORTER et REVERSE_COLLECTION_SORTER,
  - les classes d'entrées-sorties sur disque ou sur écran/clavier : INPUT_STREAM, STD_INPUT, STD_FILE_READ, OUTPUT_STREAM, STD_OUTPUT, STD_FILE_WRITE, STD_FILE_READ_WRITE, STD_INPUT_OUTPUT,
  - et bien d'autres classes encore sans oublier la définition Eiffel des objets les plus élémentaires STRING, INTEGER, REAL, DOUBLE, CHARACTER, BOOLEAN, POINTER, et BIT_N ,
- une interface bidirectionnelle vers les langages C, C++ et Java; cette interface permet à la fois aux programmes Eiffel d'appeler une fonction dans un de ces langages, mais également aux fonction externes d'invoquer des méthodes Eiffel,
- un ensemble de programmes de démonstration ainsi que l'indispensable Foire Aux Questions (FAQ).

S'agissant d'un projet GNU, l'ensemble des sources de SmallEiffel fait nécessairement partie de la distribution. Ces textes sources sont très majoritairement écrits en Eiffel, seule par exemple l'implantation du ramasse-miettes nous a obligé à utiliser quelques centaines de lignes de C ainsi qu'une ligne d'assembleur [3].

## 1.5 Guide de lecture

Les personnes ne connaissant pas mon parcours, peuvent commencer par feuilleter le chapitre 5 qui dresse la liste de mes publications. Notons également que l'index en fin de document permet d'accéder directement au résumés de mes publications.

Le chapitre 2 effectue la synthèse de plusieurs publications [CCZ97; ZCC97; CCZ98a; CZ99] concernant la technique de compilation originale utilisée pour SmallEiffel. Dans une certaine mesure, ce chapitre résume bien selon moi, une partie du travail que j'ai réalisée en encadrant Olivier Zendra pour sa Thèse d'Université [Zen00].

Si l'on souhaite profiter de notre expérience dans le but d'écrire un analyseur syntaxique, il est possible de lire le chapitre 3 indépendamment du reste du document. Ce chapitre qui n'a pas encore fait l'objet de publication est un mode d'emploi permettant d'écrire un analyseur syntaxique sans avoir recours à un outil de construction d'analyseur syntaxique. Il me semble utile de prendre connaissance de ce chapitre avant de se plonger dans la documentation d'outils comme *lex* ou *yacc* par exemple.

Le chapitre 4 peut lui aussi être lu indépendamment du reste du document. Il concerne les travaux que nous avons effectués autour de l'*aliasing* et présente les résultats publiés dans [ZC99b;

2. Ce qui nous a permis d'écrire en Eiffel un seul programme d'installation de SmallEiffel pour toutes plates-formes ! Même si GNU/Linux reste mon système favoris, voici la liste des systèmes actuellement supportés dans l'ordre alphabétique : AIX, Amiga, BeOS, BSD, DOS, FreeBSD, GNU/Hurd, HP-UX, IRIX, GNU/Linux, MacOS, NetBSD, NeXT, OS/2, OSF1, QNX, SCO, Solaris, VMS, Windows 95, Windows NT, XENIX. Notons enfin qu'il est actuellement possible d'utiliser SmallEiffel comme *cross compiler* pour produire du code pour Palm-Pilot.

3. Le fichier de configuration du ramasse-miettes pour l'architecture SUN SPARC utilise très exactement une instruction assembleur en raison de son architecture à fenêtre de registres, *window register*. Sur cette architecture, la fonction ANSI C `setjmp` ne sauvegarde pas le contenu de tous les registres.

ZC00b; ZC00a]. L'*aliasing* est le fait de référencer une même donnée via deux variables différentes, ou plus généralement par deux chemins différents. Cette astuce commune de programmation est souvent utilisée dans le but d'optimiser les programmes et est même selon nous inhérente à la programmation par objets. Nous montrons dans ce chapitre comment il est possible d'éviter les aspects néfastes de l'*aliasing* et de profiter ainsi de nombreuses possibilités d'optimisation de manière plus sûre.

Comme il se doit, le document se termine par un bilan accompagné de perspectives, ceci au chapitre 6. L'index situé dans les toutes dernières pages permettra je l'espère de se déplacer rapidement dans le document.

# Chapitre 2

# Analyse globale - L'hypothèse du système fermé

"La question fondamentale à laquelle se heurte tout débutant en informatique est de savoir comment l'électricité peut être domestiquée à un point de réalisation si spectaculaire."

Francis Scheid [Sch74]

## 2.1 Introduction

La programmation à objets est devenue une tendance majeure en informatique. En effet, l'usage intensif de l'héritage et de la liaison dynamique tend à rendre le code plus extensible et réutilisable. Cependant, certaines préoccupations demeurent en ce qui concerne les performances des programmes à objets, notamment à cause de la liaison dynamique.

Afin d'atteindre des performances similaires à celles des langages traditionnels comme C, les systèmes à objets doivent implanter cette liaison dynamique efficacement. Les travaux récents sur la prédiction de type représentent un premier pas dans cette direction en permettant le remplacement de nombreux appels polymorphiques par des appels monomorphiques directs. La méthode utilisée pour implanter les appels polymorphiques restants a également un impact important en termes de vitesse d'exécution.

Bien évidemment, les techniques d'optimisation des programmes à objets doivent non seulement produire des exécutables rapides, mais doivent également être elles-mêmes suffisamment peu coûteuses pour ne pas se voir cantonnées à la dernière compilation des applications juste avant livraison. Des compilateurs rapides sont en effet indispensables pour permettre de développer avec des langages à objets de façon incrémentale.

Dans ce chapitre, nous présentons les résultats obtenus dans ces domaines durant nos recherches [CCZ96; CCZ97; ZCC97; CCZ98a; CZ99; CZ00] dans le cadre du projet SmallEiffel. Ce projet, appliqué et fortement utilisé pour la recherche aussi bien qu'hors de la recherche, nous a amenés à la conclusion quelque peu surprenante que la compilation séparée n'est pas indispensable pour faire du développement incrémental. Nous affirmons en effet que des compilateurs intégrant une prédiction de type *globale* au système, combinée avec une implantation efficace de la liaison dynamique, peuvent être suffisamment rapides pour être utilisés dans le cadre d'environnements de développement incrémental.

L'hypothèse du *système fermé* consiste à supposer qu'aucune nouvelle classe ne sera créée lors de l'exécution et aussi qu'aucune nouvelle primitive (attribut ou routine) ne sera ajoutée dynamiquement. Bien entendu, cela suppose que l'on est capable de recenser, avant l'exécution, toutes les classes pouvant être instanciées. Pour rester dans le cadre de cette hypothèse, il faut également pouvoir inspecter la totalité du graphe d'appel avant l'exécution. Cette hypothèse est très contraignante pour des langages fortement dynamiques comme Smalltalk [GR83] ou Self [US87] par exemple.

Dans le système SmallEiffel, nous avons implanté et validé un algorithme de prédiction de type globale permettant un haut degré de spécialisation du code généré. Nous avons également proposé et expérimenté une méthode nouvelle d'implantation de la liaison dynamique, offrant d'excellentes performances. Cette méthode consiste à supprimer totalement toutes les tables de pointeurs de fonctions — *Virtual Function Tables*, ou VFT — ou les structures similaires, en les remplaçant par de simples, mais très efficaces, tests de type. Comme tous les appels indirects sont remplacés par des appels directs, cette implantation de la liaison dynamique permet d'importantes optimisations supplémentaires telles que l'expansion de code en ligne (*inlining*), et ceci même au sein des appels polymorphiques. Ces optimisations, qui portent sur l'implantation de concepts fondamentaux dans les langages de classe, sont très importantes car elles permettent aux développeurs de se concentrer sur une bonne programmation à objets sans crainte de surcoûts excessifs. Notre méthode, détaillée plus loin dans ce chapitre, n'est pas limitée à Eiffel [Mey92], mais peut au contraire s'appliquer à tout langage de classes [MNC$^+$89] sans création ou modification dynamique des classes. Elle peut donc par exemple être appliquée à C++ [Str86] mais pas à Smalltalk [GR83] ni Self [US87].

La suite de ce chapitre est organisée de la façon suivante. La section 2.2 explique comment le système à compiler est exploré par analyse statique globale de façon à rassembler les informations de types et à collecter le code vivant. La section 2.3 présente notre technique pour implanter la liaison dynamique de façon particulièrement efficace. La section 2.4 présente à l'aide d'un exemple notre méthode de génération de code optimisé par duplication-spécialisation. Puis la section 2.5 traite de la généricité et la section 2.6 concerne l'héritage multiple. La spécialisation du ramasse-miettes est résumée en section 2.7. Enfin, la section 2.8 conclut et présente les perspectives.

## 2.2 Calcul du code vivant - Calcul des types vivants

Un aspect essentiel dans la stratégie de compilation de SmallEiffel réside dans le fait que le système à traduire est toujours considéré dans sa globalité [CCZ97]. Cette stratégie rejette totalement la possibilité de compilation séparée et une classe n'est jamais compilée en dehors de son contexte d'utilisation. Ainsi, déclencher une compilation implique d'être en mesure de donner le point de départ de l'exécution, la *racine du système*, à savoir une classe de départ et une procédure de départ dans cette classe. Dans un langage ordinaire, comme Java ou C++ par exemple, la racine du système est en fait la fonction `main`, le programme principal.

La première étape dans le processus de compilation consiste à collecter le *code vivant*, c'est à dire le code susceptible d'être exécuté. Dans le contexte des langages à objets, le calcul du code vivant va de pair avec le calcul des *classes vivantes*, les classes qui sont susceptibles d'être instanciées lors de l'exécution.

La racine du système est utilisée pour initialiser l'algorithme de calcul du code vivant avec une seule catégorie d'objets vivants, la classe de départ, et une seule méthode vivante, la procédure de départ. Ensuite, de façon récursive, l'algorithme de calcul du code vivant inspecte toutes les instructions des méthodes vivantes jusqu'à obtention de l'ensemble du code vivant. Ainsi, lorsqu'une instruction d'instanciation est rencontrée dans le code vivant, le type instancié est alors ajouté à l'ensemble des *types vivants*[4]. Notons au sujet des instructions ou expressions d'instanciation qu'il est indispensable de pouvoir déterminer statiquement le type d'objet créé. Dans un autre langage qu'Eiffel, une trop grande dynamicité comme par exemple une expression d'instanciation prenant en argument une variable donnant le type d'objet à créer, pourrait rendre impossible la construction de l'ensemble des types vivants. Néanmoins, même pour des langages extrêmement dynamiques comme Smalltalk par exemple, la très grande majorité des expressions d'instanciation indique clairement le type instancié.

Pour compléter l'ensemble du code vivant, en plus des instructions et des expressions d'instanciation, il faut considérer tous les appels de méthodes. Par exemple, pour un appel `foo.bar` dans le code vivant, l'algorithme actuel ajoute dans le code vivant le code source de la méthode `bar` qui est associée au type statique de `foo`. Ainsi, si FOO correspond au type statique de `foo`, la définition de `bar` dans la classe FOO est ajoutée au code vivant[5]. Afin de prendre en compte la liaison dynamique, sont également ajoutées dans le code vivant toutes les redéfinitions de `bar` associées aux types vivants qui sont des sous-types de FOO[6].

Le processus de calcul du code vivant ainsi que de l'ensemble des types vivants est à la fois récursif et itératif. Sans entrer dans les détails, car cet algorithme est réparti dans de nombreuses classes, 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.

## 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 projet SmallEiffel consiste 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 2.1.

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. Le lecteur intéressé trouvera dans [ZCC97] et [Zen00] toutes les informations concernant les avantages de cette implantation de la liaison dynamique sans VFT.

4. Comme Eiffel offre la possibilité de définir des classes génériques, parler de classe vivante dans ce cas ne suffit plus. Il est nécessaire d'introduire la notion de type vivant pour distinguer entre elles les différentes dérivations d'une même classe générique : ARRAY`[INTEGER]` et ARRAY`[DOUBLE]` par exemple.

5. Lorsque le type statique 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. De toute évidence, ce cas de figure ne doit pas être très fréquent et il n'est donc pas urgent d'effectuer cette amélioration. Ceci étant, cette remarque fait apparaître un besoin criant de pouvoir *garantir* que le calcul du code vivant est correct. Travailler sur une meilleure validation des processus mis en œuvre durant la compilation est une tâche qui semble très complexe à formaliser.

6. Cette dernière phase dans le calcul du code vivant porte le nom de *falling down* dans le code source de SmallEiffel.

```
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
```

FIG. 2.1 – *Le code généré pour implanter la liaison dynamique effectue une sélection dichotomique. On suppose pour simplifier dans cet exemple que* `typeId`*, le type de l'objet receveur, appartient à l'ensemble {*`1`*,*`2`*,*`3`*,*`4`*}.*

## 2.4 Spécialisation des méthodes vivantes selon le type du receveur

La spécialisation automatique du code en fonction du type du receveur est un point essentiel dans notre stratégie de compilation [CZ99]. Cette spécialisation correspond 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 2.2 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 respectivement la valeur vrai, dans la classe PEACH et la valeur faux dans la classe APPLE, l'expression conditionnelle est respectivement, 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.

Pour chaque type vivant donné, toutes les méthodes vivantes de ce type sont spécialisées en appliquant la technique vue précédemment. Ainsi, 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. Comme en raison du principe le plus élémentaire de la programmation par objets, cette variable référence toujours un objet (ie. `Current` n'est jamais `Void`), tous les appels concernant `Current` sont 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. C'est probablement pour cette raison que les scores d'inférence de type (on dit aussi *prédiction de type*) atteignent des niveaux aussi élevés. Par exemple, dans la version -0.76 de SmallEiffel, le générateur de code C, la commande `compile_to_c`, s'auto-compile avec un score supérieur à 90% de prédiction du type : pour un envoi de message donné, 90 fois sur 100, le type dynamique du receveur est connu statiquement !

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 même, plus petits que les exécutables obtenus à partir de classes C++ (cf. par exemple [CCZ97]). Cette bonne surprise est pour une part liée au fait que la spécialisation des méthodes a souvent tendance à élaguer du code comme dans l'exemple de la figure 2.2. D'autre

```
deferred class FRUIT                                      1
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
```

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

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

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

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

FIG. 2.2 – *Spécialisation du code généré pour la méthode* `display` *définie dans la classe* FRUIT. *Les cadres* **1**, **2** *et* **3** *correspondent respectivement aux textes des classes* FRUIT, PEACH *et* APPLE. *Les cadres* **4** *et* **5** *montrent le code C généré pour les classes* PEACH *et* APPLE.

part, l'implantation de la liaison dynamique à l'aide de code de sélection dichotomique permet le remplacement direct, *inlining*, de nombreuses méthodes. En particulier, toutes les fonction d'écriture et de lecture d'attributs ne sont jamais ni appelées, ni même définies [ZCC97].

La technique de duplication/spécialisation est particulièrement bien adaptée dans 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, nous appelons *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`).

Pour donner un exemple courant, dans un système fermé qui utilise des chaînes de caractères, le type STRING est un type feuille s'il n'existe aucune sous-classe de STRING dans le système fermé. Dans ce cas, tous les envois de message portant sur une expression de type STRING sont des appels directs sans liaison dynamique. En fait, tout se passe comme si le compilateur ajoutait automatiquement l'équivalent du mot clef *final* de Java pour toutes les classes feuilles.

## 2.5 Généricité dans un système fermé

Dans l'hypothèse d'un système fermé, la prise en compte de la généricité est extrêmement simple [CZ99]. Chaque dérivation d'un type générique correspond à un type différent considéré dans la suite comme un type ordinaire à part entière. Ainsi, deux dérivations génériques[7] de la classe ARRAY comme ARRAY[INTEGER] et ARRAY[CHARACTER] par exemple, sont considérées de façon indépendante, exactement comme s'il s'agissait de deux classes ordinaires, non génériques. Selon ce principe, une opération vivante pour une des deux dérivations ne l'est pas forcément pour l'autre. Un système donné peut par exemple n'utiliser que la méthode `put` avec les instances du type ARRAY[INTEGER] et uniquement la méthode `item` pour les instances du type ARRAY[CHARACTER].

L'expérience acquise dans les différents projets compilés avec SmallEiffel, ainsi que les retours d'information nous provenant des utilisateurs de SmallEiffel, montrent que la taille des exécutables est très satisfaisante. On aurait pu en effet craindre que cette approche naïve pour implanter la généricité, basée sur une duplication/spécialisation, ne conduise à une taille de code trop importante. En fait, les exécutables produits par SmallEiffel sont souvent plus petits que ceux produits par les compilateurs commerciaux. Comme dans l'exemple de la figure 2.2, le code spécialisé a souvent tendance à être plus compact que le code qui ne l'est pas. Ceci est d'autant plus vrai pour le code des classes génériques qui servent la plupart du temps à implanter des conteneurs: tableaux d'objets, listes d'objets, ensembles d'objets, etc. En fait, il est assez difficile de produire du code pour implanter un tableau alors que l'on ne connaît pas encore la taille des éléments de ce tableau.

Le fait de procéder par duplication de code pour chaque dérivation générique permet de rester uniforme avec les autres spécialisations déjà effectuées. Par exemple, l'opération d'accès dans un tableau de type ARRAY[PEACH] est une opération qui ne peut que retourner un objet de type PEACH ou `Void`[8]. Ainsi, si le type PEACH est lui-même un type feuille, une expression comme celle qui suit ne nécessite pas de code de liaison dynamique pour l'appel de la méthode `display` :

```
array_of_peach.item(i).display
```

## 2.6 Héritage multiple dans un système fermé

Dans un système fermé la prise en compte de l'héritage multiple est très simple. En effet, le graphe d'héritage est complètement connu et figé au moment de la traduction ce qui permet de détecter, pendant la compilation, les éventuels conflits.

Notons qu'il est difficile d'imaginer la prise en compte de l'héritage multiple dans un contexte de compilation *complètement* séparée. Dans la pratique, la plupart des compilateurs considèrent aussi les super-classes de la classe à compiler (ie. la compilation d'une classe n'est donc pas complètement séparée).

Dans un contexte d'héritage multiple, l'implantation de la liaison dynamique avec des tableaux de pointeurs de fonctions (VFT) est complexe car elle nécessite d'associer au sélecteur un seul index applicable à toutes les sous-classes. Le problème connexe lié au compactage des tables d'indirection a d'ailleurs fait l'objet de nombreux travaux [DMSV89] [ES90] [AR92] [DHV95] [DH95] [DH96].

7. Certains auteurs parlent d'instanciation d'un type générique et n'utilisent pas le terme dérivation. Nous préférons dérivation dans ce cas pour éviter la confusion avec l'instanciation d'une classe qui correspond à la création d'un nouvel objet.

8. Bien entendu, nous considérons ici que le système sur lequel on travaille est globalement valide (*system validity rules* [Mey92]). Nous montrons plus loin, page 65, comment il est possible de mettre autre chose qu'une PEACH dans un tableau de type ARRAY[PEACH].

Notre implantation de la liaison dynamique à l'aide de code de sélection dichotomique contourne complètement ce problème. Chaque sous-classe est prise en compte par une branche différente du code de sélection, exactement comme dans le cas de l'héritage simple. Avec notre implantation, l'héritage multiple n'implique donc aucun surcoût à l'exécution par rapport à l'héritage simple.

## 2.7 Spécialisation du ramasse-miettes

Toujours dans le but d'améliorer les performances du système à l'exécution, le ramasse-miettes est lui aussi spécialisé. Le travail présenté dans [CCZ98a] présente en détail comment nous avons spécialisé l'implantation de l'algorithme marquage-balayage, *mark-sweep*, compte tenu de la connaissance globale du système.

Du point de vue placement en mémoire les objets sont regroupés par type : un bloc de mémoire donné ne contient qu'un seul type d'objets. Un avantage de cette représentation est de pouvoir ranger les objets consécutivement, les uns à coté des autres, sans perte de mémoire entre les objets pour un lien de chaînage par exemple. Dans un tel bloc d'objets, la phase de première allocation est optimale puisqu'il suffit de prendre les objets dans l'ordre ou ils se présentent (*Linear Allocation Chunk* [JL96] page 32). De même, en phase de balayage, le bloc est parcouru par une procédure elle-même spécialisée dans le traitement des objets du bloc.

Notons que cette spécialisation des fonctions de balayage permet également d'optimiser le déclenchement des opérations de *finalisation*, les opérations qu'il faut déclencher sur un objet juste avant que le ramasse-miettes ne recycle la place mémoire de l'objet sortant. En effet, on sait statiquement si un type d'objets donné nécessite ou non une opération de finalisation [9].

A chaque type d'objet est associé une procédure de marquage spécialisée qui tient compte du type des différents attributs de l'objet. Considérons par exemple qu'un système fermé utilise à la fois des objets de type TRIANGLE et des objets de type POINT. On suppose qu'un point est simplement composé de deux attributs de type DOUBLE pour mémoriser ses coordonnées. Un TRIANGLE possède trois attributs de type POINT, les trois sommets du triangle. Comme un objet de type POINT ne référence pas d'objets alloués dans le tas, le marquage d'un objet de type POINT se limite à modifier l'indicateur de marquage. La fonction C correspondante, automatiquement générée, est réduite à sa plus simple expression :

```
void markPOINT(Point *point) {
    point->mark_flag=MARKED_FLAG;
}
```

Une fonction aussi simple que la fonction précédente est généralement remplacée en ligne (*inlining*) par le compilateur C.

Le marquage d'un objet de type TRIANGLE consiste, si l'objet n'est pas déjà marqué, à déclencher le marquage sur les trois sommets qui correspondent dans la fonction suivante aux attributs `point1`, `point2` et `point3` :

```
void markTRIANGLE(Triangle *triangle) {
    if (triangle->mark_flag!=MARKED_FLAG){
        triangle->mark_flag=MARKED_FLAG;
        if (triangle->point1 != NULL) markPOINT(triangle->point1);
        if (triangle->point2 != NULL) markPOINT(triangle->point2);
        if (triangle->point3 != NULL) markPOINT(triangle->point3);
    }
}
```

9. En Eiffel la procédure *dispose* définie dans la classe MEMORY est l'équivalent de la fonction *finalize* de Java

Comme on peut le voir dans les exemples qui précèdent, l'algorithme de marquage-balayage n'est pas modifié, seule sa mise en œuvre est plus efficace. Les fonctions de marquage présentées précédemment sont *codées en dur*, sans faire intervenir une quelconque structure de données auxiliaire décrivant la structure des objets. On supprime ainsi un niveau d'interprétation lors de l'exécution, le niveau qui consiste à regarder si un objet possède des attributs et si ces attributs sont ou non des références vers d'autres objets dans le tas.

Pour avoir tous les détails sur la spécialisation de l'algorithme de marquage-balayage le lecteur intéressé peut consulter [CCZ98a]. Plus généralement, la technique de spécialisation pourrait tout aussi bien s'appliquer à n'importe quel autre système de gestion mémoire : comptage de références, *reference counting*, algorithme par recopie, *copying garbage collection*, etc. [JL96]. Pour ne donner qu'un exemple, on peut imaginer, dans le cas d'un algorithme par comptage de références, de définir pour chaque type vivant une version spécialisée de la phase de mise à jour du compteur de références. Comme dans le cas du marquage spécialisé vu précédemment, cette implantation spécialisée permet d'éviter un niveau d'interprétation. En outre, il serait également possible de déterminer statiquement quels sont les types d'objets qui sont potentiellement à l'origine des cycles; et ainsi compenser un des plus gros inconvénient de l'algorithme par comptage de références.

## 2.8 Discussion, conclusion et perspectives

Comme nous l'avons montré précédemment, le fait de travailler dans un système fermé offre d'excellentes opportunités d'optimisation. En outre, certains aspects comme l'implantation du mécanisme d'exceptions, l'algorithme de validation globale du système, *system level validity rules*, pour ne citer que ceux-là, pourront faire l'objet de travaux futurs.

Il est intéressant de noter ici que la définition d'Eiffel est très bien adaptée à la méthode de compilation globale. Les extensions prévues d'Eiffel avec les *agents*[10], les *Tuples*[11], ainsi que les nouvelles possibilités d'introspection ne remettent pas en cause, *a priori*, notre principe de compilation globale.

Les résultats que nous avons obtenus permettent d'utiliser Eiffel, langage de haut niveau, pour des domaines d'applications parfois surprenants, comme la réalisation de systèmes embarqués ou la réalisation de jeux vidéo[12]. Au moins en ce qui concerne la livraison définitive du logiciel, la méthode de compilation de SmallEiffel semble nettement plus avantageuse que les techniques plus classiques, utilisées par les compilateurs commerciaux.

Concernant l'hypothèse du système fermé, la question qui se pose est de savoir si cette technique est compatible avec des langages plus populaires, comme Java ou C++ par exemple, et aussi de dresser la liste des inconvénients inhérents à cette technique.

Le principal inconvénient de notre méthode est qu'elle requiert la connaissance de l'ensemble du système, ce qui empêche la compilation séparée ainsi que la production de bibliothèques pré-compilées. Ceci peut sembler un problème pour du développement incrémental. Néanmoins, dans ce contexte, le point le plus important pour le développeur est la vitesse de (re)compilation.

10. Un agent dans la future définition d'Eiffel est comparable à une fermeture, *closure* dans un langage fonctionnel, avec un aspect d'évaluation partielle. L'objectif de cet ajout consiste essentiellement à pouvoir augmenter l'expressivité des assertions présentes dans les contrats.

11. Possibilité de manipuler des N-uplets typées de valeurs pour la prise en compte plus aisée de fonctions à nombre variable d'arguments par exemple.

12. La société EUGEN-systèmes réalise avec SmallEiffel des logiciels de jeux vidéos nécessitant des temps de réponses précis sur des machines grand public.

L'impossibilité de fournir des bibliothèques pré-compilées semble être un problème lorsque leur code source est confidentiel, pour des raisons commerciales par exemple. Cependant, une solution triviale consiste à utiliser des techniques de cryptage. Bien entendu, cette solution ne résout pas le problème des bibliothèques partagées à l'exécution (dites *bibliothèques dynamiques*). En effet, les programmes générés avec SmallEiffel peuvent interagir avec les bibliothèques existantes (faites par exemple avec des compilateurs C/C++), et SmallEiffel peut être utilisé pour produire des *bibliothèques statiques*, mais il est en revanche actuellement impossible de produire des bibliothèques dynamiques avec SmallEiffel.

Nous pensons pourtant que notre technique peut aisément être adaptée pour produire des bibliothèques pré-compilées. Néanmoins, le code pré-compilé serait très probablement moins spécialisé et donc moins optimisé que le code d'un exécutable autonome compilé à partir d'un système connu entièrement. Ce problème de "frontière" entre système connu et système inconnu, son impact sur la production de bibliothèques, et plus généralement sur la compilation, nous semble être un sujet de recherche tout à fait intéressant. Il se rapproche d'ailleurs des nombreuses recherches menées actuellement sur la *compilation dynamique*.

Notre méthode utilise seulement des informations statiques pour améliorer l'efficacité du code, notamment la liaison dynamique. L'adjonction à notre compilateur de la prise en compte du flot d'exécution augmenterait les performances du code généré, en permettant le remplacement d'un plus grand nombre de sites de liaison dynamique par des appels statiques monomorphiques. Ceci peut néanmoins provoquer une augmentation conséquente des temps de compilation.

# Chapitre 3

# Analyse syntaxique et analyse lexicale : un mode d'emploi

> "à qui sourit la vie
> éviter les péages
> jamais souffrir
> juste faire hennir
> les chevaux du plaisir"
>
> Alain Bashung [Osez Joséphine]

Ce chapitre décrit l'implantation de l'analyseur syntaxique et lexical de SmallEiffel. Contrairement à certaines idées reçues, nous pensons que l'utilisation d'un constructeur d'analyseur, comme lex/yacc par exemple, n'est pas toujours souhaitable. Nous montrons dans ce chapitre comment nous avons procédé ainsi que les avantages de cette approche.

Le présent chapitre n'a pas fait l'objet de publications. Je l'ai rédigé très récemment car je pense que notre expérience pratique en matière d'analyse syntaxique devait aussi faire partie de ce mémoire. J'ai également appris très récemment que l'analyseur syntaxique de GNU Ada est conçu de façon très similaire ([CGS95] section 3.2 pages 6-7).

## 3.1 Introduction

L'analyse syntaxique et l'analyse lexicale sont des points de passage obligés lors du développement de la partie frontale d'un compilateur. La plupart du temps, les outils qui manipulent des programmes, pour les mettre en forme ou pour extraire de la documentation par exemple, nécessitent eux aussi l'utilisation de techniques d'analyse syntaxique et lexicale. Par exemple, parmi les outils de la distribution de SmallEiffel, quatre intègrent un analyseur syntaxique d'Eiffel : `compile_to_c` le traducteur vers C, `compile_to_jvm` le traducteur vers la machine virtuelle Java, `pretty` l'indenteur de programme et `short` qui extrait les interfaces.

Pour un langage donné, pour lequel on dispose d'une grammaire LL(k) ou LR(k) avec $k < 3$, ce qui est généralement vrai pour les langages de programmation, l'analyse syntaxique ne présente auncun problème théorique particulier. Néanmoins, le passage de la théorie à la pratique, lorsqu'un même analyseur doit être partagé par plusieurs outils (`compile_to_c`, `compile_to_jvm`, `pretty`, `short`), pose de sérieux problèmes de mise en œuvre d'un point de vue génie logiciel.

La solution adoptée dans SmallEiffel et reprise ensuite pour le langage C dans le logiciel Pro-

Mic [13] remet en question certaines idées préconçues que l'on peut avoir sur la manière d'aborder ce problème. Cette solution assez originale qui fusionne en une seule phase analyse syntaxique et analyse lexicale est présentée en détail dans la suite.

La suite de ce chapitre est organisée de la façon suivante. La section 3.2 présente les objectifs de notre implantation. Nous montrons ensuite dans la section 3.3 comment les phase d'analyse syntaxique et d'analyse lexicale sont regroupées en une seule et même phase. La section 3.4 concerne la prise en compte des suites d'entités syntaxiques identiques. La section 3.5 montre l'utilisation de la récursivité ainsi que la mémorisation positionnement exact dans le texte source. Un exemple de fonction de niveau lexical est donné en section 3.6. Avant de conclure en section 3.8, quelques éléments de mesure de performances sont donnés à la section 3.7.

## 3.2 Les objectifs de notre implantation

Parmi nos objectifs, le plus important consiste à être en mesure de mettre en commun une seule et unique définition de l'analyseur syntaxique pour tous les outils qui en ont besoin : c'est le même texte source qui décrit à la fois l'analyseur syntaxique de `compile_to_c`, de `compile_to_jvm`, de `pretty` et de `short`. L'avantage de cette unicité permet évidemment de garantir que chacune des quatres applications citées précédemment accepte exactement le même langage source. Ainsi, qu'il s'agisse d'une évolution du langage source ou bien d'une correction dans la grammaire, ou encore de la modification d'un libellé de message d'erreur, chaque outil réagit de façon strictement identique puisque c'est le **même** analyseur qui est toujours utilisé. Cet aspect est particulièrement important pour les messages d'erreurs syntaxiques qui doivent être identiques pour tous les outils.

Durant les six années du projet, il nous est assez souvent arrivé d'avoir à modifier l'analyseur syntaxique, soit pour corriger des erreurs mais aussi pour implanter de nouvelles fonctionalités. En outre, Eiffel est un langage en continuelle évolution sémantique et syntaxique (la nouvelle version d'Eiffel, Eiffel version 5 est en cours de préparation).

Un autre objectif de l'implantation consiste à pouvoir repérer très précisément la position des unités lexicales : fichier source, numéro de ligne ainsi que numéro de colonne dans la ligne. Un tel repérage est indispensable pour réussir la composition de messages d'erreurs précis commme ceux de SmallEiffel par exemple qui peuvent comporter une ou plusieurs positions dans différents textes source. Cette information exacte de positionnement peut aussi être utilisée dans les contrôles sémantiques eux-mêmes, pour savoir par exemple, si deux utilisations d'une variable se rapportent ou non au même endroit de déclaration (il suffit pour cela de comparer la position de chacune des deux déclarations).

Pour donner encore un exemple nécessitant une information de positionnement complète, citons enfin l'exemple des bulles d'aides du débogueur de ProMic. Dans ce débogueur, si on laisse le pointeur de la souris arrêté sur une position ligne/colonne d'une fenêtre source, le débogueur affiche les informations qu'il est en mesure de trouver grâce à cette position [14]. S'il s'agit par exemple d'une variable active, liée ou vivante, le débogueur affiche automatiquement dans la bulle d'aide qui apparaît la valeur de cette variable ainsi que les informations qui lui sont associées (type et endroit de stockage en mémoire par exemple). Il va de soi que la réalisation d'un

13. ProMic est un environnement de programmation que nous avons développé dans le cadre d'une convention de recherche. L'environnement ProMic comprend entre autres choses un analyseur syntaxique pour C ANSI écrit sur le même modèle que l'analyseur syntaxique de SmallEiffel. ProMic est destiné à la programmation de micro-contrôleurs synthétisables (voir page 69).

14. Sur le même modèle que les bulles d'aides de `ddd` (*Data Display Debugger*) qui est une excellente interface graphique pour gdb (*gnu debugger*).

tel mécanisme de bulles d'aide peut difficilement se passer de la connaissance du positionnement exact des unités lexicales.

Notons au sujet du repérage précis des unités lexicales que le très célèbre tandem d'outils lex/yacc [AJ74] [Ben96] permet facilement d'obtenir la ligne sur laquelle a été trouvée un lexème mais permet plus difficilement d'avoir accès à la colonne. En outre, comme nous le montrerons plus tard, la solution que nous avons mise en œuvre aboutit à de bien meilleurs résultats en terme de vitesse d'exécution.

Un autre aspect de comparaison des méthodes d'analyse syntaxique concerne la facilité de la mise en œuvre du rattrapage des erreurs syntaxiques. Sur ce point, il est beaucoup plus facile de le faire sur un analyseur descendant que sur un analyseur ascendant. Pour cette raison et parce qu'il peut être implanté récursivement, de façon simple, nous avons choisi l'analyse descendante.

## 3.3 Analyse syntaxique et lexicale en une seule phase

Traditionnellement [AU77], un traducteur est composé entre autres choses d'un analyseur lexical qui fonctionne en amont et d'un analyseur syntaxique qui poursuit le travail en aval. Plus encore, tous les ouvrages de compilation que nous connaissons [Gri71] [AU77] [CGV80] [TS85] [ASU86] [ASU89] [WM94] [Muc97] [App98] invitent à séparer complètement les deux phases que sont l'analyse syntaxique et l'analyse lexicale. Suivant ce principe à la lettre, le couple d'outils lex/yacc [AJ74] [Ben96], un des plus célèbres outils de génération d'analyseurs, impose de fait la séparation de ces deux phases. Il faut en effet deux fichiers pour utiliser lex/yacc, un pour décrire les unités lexicales et l'autre pour décrire les règles syntaxiques. En outre, l'analyseur lexical fonctionne sans aucune information contextuelle de la part de l'analyseur syntaxique.

L'expérience acquise dans le cadre du projet SmallEiffel, sur le langage Eiffel, et dans le projet ProMic, sur C ANSI, nous a permis d'expérimenter en grandeur nature une autre approche du problème. Cette approche consiste à fusionner en une seule phase le processus d'analyse lexicale et syntaxique de telle sorte que l'analyseur lexical ne travaille qu'au dernier moment, sur nécessité, grâce à des informations contextuelles données par l'analyseur syntaxique.

De cette façon, au cœur d'une construction syntaxique donnée, l'analyseur syntaxique s'informe auprès de l'analyseur lexical de la présence d'un mot-clef particulier [15]. Par exemple, dans une instruction conditionnelle Eiffel *if-expression-then*, après l'analyse de la partie *expression*, l'analyseur syntaxique demande à l'analyseur lexical si le mot-clef *then* est bien présent à la position courante du flux d'entrée.

La fonction `a_then_part` de la classe EIFFEL_PARSER (figure 3.1) correspond à cette partie de l'analyse syntaxique. Appelée immediatement après le mot-clef *if*, la fonction `a_then_part` commence par déclencher l'analyse syntaxique de l'expression booléenne en appelant `a_expression`. Après avoir mémorisé l'expression analysée dans la variable locale `expression`, l'appel de la fonction `a_keyword("then")` correspond à demander à l'analyseur lexical si le mot-clef *then* est bien présent dans le flux d'entrée. Comme on peut le constater, toujours dans la fonction `a_then_part` de la figure 3.1, le rattrapage d'erreur est facilement programmable : l'absence du mot-clef *then* provoque l'émission d'un message d'avertissement, `warning`, avant de poursuivre l'analyse syntaxique comme si le mot-clef *then* avait été rencontré.

Pour terminer l'explication du code présenté à la figure 3.1, notons que l'appel à la fonction `get_comment` permet d'avoir accès au dernier commentaire rencontré. La fonction `add_comment` permet pour sa part d'associer ce commentaire à `last_expression`, la dernière expression ré-

15. Traditionnellement, lorsque l'on utilise lex, on demande d'avancer d'un lexème dans le flux d'entrée sans donner d'information particulière à l'analyseur lexical.

```
a_then_part(ifthenelse: IFTHENELSE): BOOLEAN is
      -- Rule: then_part -> expression "then"
   local
      expression: EXPRESSION;
   do
      if a_expression then
         Result := true;
         expression := last_expression.add_comment(get_comment);
         if not a_keyword("then") then
            warning("Added %"then%".");
         end;
         ifthenelse.add_if_then(expression,a_compound);
      end;
   end;
```

FIG. 3.1 – *La fonction* `a_then_part` *de la classe* EIFFEL_PARSER *chargée de l'analyse syntaxique d'une partie de l'instruction if-expression-then.*

sultant de l'appel à `a_expression`. Notons, que si les commentaires ne servent pas aux outils de traduction (`compile_to_c` et `compile_to_jvm`), ils sont indispensables pour l'outil d'extraction d'interfaces (`short`) et pour la mise en page (`pretty`). Encore une fois, notons qu'un outil comme lex n'est pas adapté à nos besoins du fait qu'il enlève les commentaires du flux d'entrée [16].

## 3.4 Suite d'entités syntaxiques identiques

Du point de vue analyse syntaxique, l'ensemble de l'analyseur utilise une stratégie descendante [Gre64] [Gre65] [Fos68] et est écrit récursivement sauf lorsqu'il s'agit de suites d'entités syntaxiques identiques qui sont alors traitées itérativement.

Dans la pratique, les suites d'entités syntaxiques sont très fréquentes : suite de définitions de fonctions, suite de déclarations de variables, suite d'instructions, suite de clauses de renommage, suite de clients. Dans la définition d'Eiffel par exemple, on dénombre environ 25 sortes de suites différentes. Habituellement, la modélisation d'une suite d'entités par des règles de grammaire nécessite un minimum de deux règles et parfois plus, selon qu'il y a ou non un marqueur indiquant la fin de la suite, selon qu'il y a ou non un séparateur et selon le fait que le séparateur en question est optionnel ou non. Bien entendu, puisque nous avons décidé de coder l'analyseur directement en Eiffel, il serait compliqué et inutile de modéliser les suites d'entités syntaxiques par des règles de grammaire puisqu'une simple itération convient.

Ainsi, la fonction `a_compound` de la figure 3.2 montre comment l'analyse d'un bloc d'instructions Eiffel est réalisée itérativement. La fonction `a_instruction`, comme vu précédemment, déclenche l'analyse d'une instruction et, le cas échéant, rend accessible son résultat dans l'attribut `last_instruction` (figure 3.2). L'appel de la fonction `skip` permet de sauter le point virgule qui est le séparateur optionnel dans une suite d'instructions. Comme on peut le constater ici encore, l'appel `skip(';')` permet d'effectuer l'avancée dans le flux des lexèmes avec une information contextuelle précise. Notons finalement que le rattrapage d'une erreur syntaxique dans une suite peut facilement être programmé, par exemple en avançant jusqu'au prochain séparateur par exemple.

16. Bien entendu, il serait possible d'utiliser lex sans lui demander de traiter les commentaires comme des commentaires mais comme des unités lexicales ordinaires.

```
a_compound: COMPOUND is
      -- Rule: compound -> { instruction ";" ... }
   local
      header_comment: COMMENT;
      instruction: INSTRUCTION;
      instruction_list: ARRAY[INSTRUCTION];
   do
      from
         header_comment := get_comment;
      until
         not a_instruction
      loop
         instruction := last_instruction;
         if skip(';') then end; -- Optional Separator
         instruction := instruction.add_comment(get_comment);
         if instruction_list = Void then
            !!instruction_list.with_capacity(4);
         end;
         instruction_list.add_last(instruction);
      end;
      if header_comment /= Void or else instruction_list /= Void then
         !!Result.make(header_comment,first_one,instruction_list);
      end;
   end;
```

FIG. 3.2 – *Exemple d'analyse d'une suite d'unités syntaxiques de façon itérative. Version simplifiée de la fonction* `a_compound` *de la classe* EIFFEL_PARSER *chargée d'analyser une suite d'instructions.*

## 3.5 Analyse descendante récursive – Utilisation du positionnement exact

Lorsqu'il ne s'agit pas d'une répétition d'unités syntaxiques identiques, chaque règle de grammaire est traduite par une fonction ou par une procédure. Cette méthode bien connue est décrite dans [ASU89] pages 61–66. Elle permet de prendre en compte les problèmes de priorité et d'associativité des opérateurs. Les différentes fonctions peuvent s'appeler récursivement selon la structure de la grammaire. La figure 3.3 donne un exemple de traduction de deux règles de grammaire en utilisant cette méthode. Dans la fonction, `a_r1`, le mot clef *implies* est explicitement le symbole directeur qui permet de choisir la bonne alternative. Dans le cas de productions ne commençant pas par un terminal du langage, il faut avoir préalablement calculé les symboles directeurs. Un petit programme *scheme* donné en annexe B permet de les calculer automatiquement et de vérifier que la grammaire est bien LL(1).

Quantitativement, et simplement pour fixer les idées, sur les 125 fonctions d'analyse syntaxique et lexicale de la classe EIFFEL_PARSER, on dénombre environ 60 fonctions dont l'écriture est directement guidée par une règle de grammaire.

Toujours dans la figure 3.3, on peut voir dans la fonction `a_r1` un exemple d'utilisation de l'information de positionnement exact. Après l'appel `a_keyword("implies")`, l'appel de la fonction `position` construit un objet de la classe POSITION qui indique très exactement la position du premier caractère du mot clef *implies*. Cette position est utilisée pour construire un nœud CALL_INFIX_IMPLIES de l'arbre abstrait. En cas d'erreur, la même information de positionne-

```
a_e0: BOOLEAN is
      -- Rule: e0 -> e1 r1
   do
      Result := a_e1;
      if Result then
         a_r1(last_expression);
      end;
   end;

a_r1(left_part: like last_expression) is
      -- Rule: r1 -> "implies" e1 r1 | ε
   local
      infix_implies: CALL_INFIX_IMPLIES;
      start_position: POSITION;
   do
      if a_keyword("implies") then -- "implies" e1 r1
         start_position := position(start_line,start_column);
         if a_e1 then
            !!infix_implies.make(left_part,start_position,last_expression);
            a_r1(infix_implies);
         else
            error(start_position,"Expression expected after 'implies'.");
         end;
      else -- ε
         last_expression := left_part;
      end;
   end;
```

FIG. 3.3 – *Exemple de traduction de deux règles de grammaire en utilisant une fonction par règle. Les fonctions* `a_e0` *et* `a_r1`*, de la classe* EIFFEL_PARSER*, correspondent respectivement aux règles* e0 *et* r1 *( $\epsilon$ est la notation pour le mot vide).*

ment exact est utilisée pour composer le rapport d'anomalie (appel de la fonction `error`).

## 3.6 Exemple de fonction de niveau lexical

L'analyseur lexical est composé d'un ensemble de fonctions, chacune de ces fonctions étant spécialisé dans la reconnaissance d'une seule sorte de lexème : une fonction pour les identificateurs, une fonction pour les nombres entiers, une fonction pour les nombres réels, etc. Une fonction de niveau lexical se contente de détecter ou non la présence du lexème correspondant à la position courante dans le texte source. Si le lexème est présent, la fonction avance dans le texte source en modifiant la position courante, sinon, la position courante reste inchangée.

Comme les fonctions de niveau syntaxique vues précédemment, les fonctions de niveau lexical font elles aussi partie de la classe EIFFEL_PARSER. La figure 3.4 montre un exemple d'une fonction de niveau lexical, la fonction `a_identifier` chargée de détecter la présence d'un identificateur. Dans cette fonction, `current_char` donne l'accès au caractère courant, et `next_char` avance sur le prochain. La fonction `skip_comments` permet de consommer les commentaires qui peuvent suivre l'identificateur. Si l'identificateur correspond en fait à un mot clef du langage, fonction `isa_keyword`, on recule à la colonne de départ (`original_column`). Notons ici qu'il est indispensable de travailler sur un texte source complètement disponible en mémoire centrale et

```
a_identifier: BOOLEAN is
   local
      original_column: INTEGER; stop: BOOLEAN;
   do
      if current_char.is_letter then
         from
            original_column := column;
            tmp_name.reset(position(line,original_column));
            tmp_name.extend(current_char);
         until
            stop
         loop
            next_char;
            inspect
               current_char
            when 'a'..'z','0'..'9','A'..'Z','_' then
               tmp_name.extend(current_char);
            else
               stop := true;
            end;
         end;
         if tmp_name.isa_keyword then -- Going back:
            current_char := tmp_name.buffer.first;
            column := original_column;
         else  -- It is an identifier:
            Result := true;
            skip_comments;
         end;
      end;
   end;
```

Fig. 3.4 – *La fonction* `a_identifier` *de la classe* EIFFEL_PARSER*, fonction de niveau lexical chargée de reconnaître un identificateur.*

non pas directement sur fichier en raison des fréquents retours en arrière dans la plupart des fonctions de niveau lexical.

## 3.7 Comparaison des performances avec un analyseur ascendant

Dans le but de mesurer les performances de notre implantation avec celle d'un analyseur syntaxique ascendant, nous avons comparé l'analyseur syntaxique de SmallEiffel avec celui qui est disponible dans le paquetage *Gobo* (`http://www.gobosoft.com/`). L'analyseur syntaxique du langage Eiffel qui fait partie de la distribution *Gobo* est un analyseur ascendant généré à partir des outils *Gelex/Geyacc* qui font également partie de cette même distribution.

En fait, le couple d'outils *Gelex/Geyacc* est le pendant Eiffel des outils C *lex/yacc*. *Gelex* est le générateur d'analyseur lexical et *Geyacc* est le générateur d'analyseur syntaxique. Comme dans le cas de *lex/yacc* les analyseurs produits sont guidés par des tables. Selon l'auteur même de la distribution, Eric Bezault, les algorithmes de calcul des tables ainsi que les algorithmes de compactage des tables sont strictement identiques à ceux de *lex/yacc*. La différence entre ces deux générateur d'analyseurs réside dans le fait que *lex/yacc* produit du code C alors que

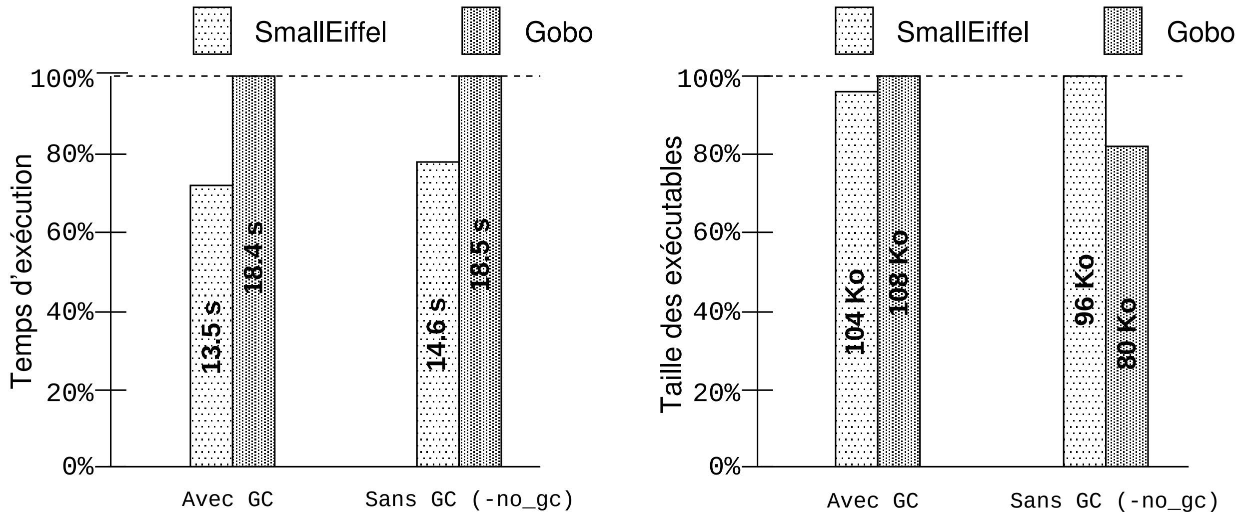


FIG. 3.5 – *Comparaison des performances de l'analyseur syntaxique de SmallEiffel (version -0.76) et de l'analyseur syntaxique de Gobo (version 1.6). Mesures effectuées sur Pentium III 550 Méga-hertz avec un jeu d'essai d'environ 1 400 000 lignes de code source Eiffel en entré.*

*Gelex/Geyacc* produit du code Eiffel.

Afin de comparer les deux analyseurs syntaxiques dans les mêmes conditions, nous avons préalablement procédé au déshabillage complet de l'analyseur syntaxique de SmallEiffel. Concrètement, cela consiste à retirer de l'analyseur syntaxique toutes les actions sémantiques consistant à construire au fur et à mesure de l'analyse syntaxique la représentation abstraite du programme.

Ainsi, les deux analyseurs comparés sont des analyseurs pour le même langage source : le langage Eiffel. Ces deux analyseurs sont dépourvus d'action sémantiques afin de ne mesurer que la partie purement syntaxique. Notons que l'analyseur syntaxique de *Gobo* produit des messages beaucoup plus rudimentaires que celui de SmallEiffel et qu'il s'arrête sur la première erreur le cas échéant (i.e. aucun rattrapage d'erreur dans l'analyseur de *Gobo*).

La figure 3.5 présente les résultats de la comparaison effectuée sur un jeu d'essais d'environ 1 400 000 lignes de code source, code source Eiffel provenant de la distribution de SmallEiffel. Le temps d'exécution de SmallEiffel est meilleur d'environ 25% à 30% selon que l'analyseur est ou non compilé avec un ramasse-miettes. Dans tous les cas, les exécutables sont de petite tailles, variant de 80 Ko à 108 Ko. Ceci montre que le fait d'utiliser des tables ne permet pas d'obtenir un gain substantiel du point de vue de la taille des exécutables.

De manière générale, les résultats de cette mesure montrent que l'analyse syntaxique de code source est extrêmement rapide, y compris sur des machines désormais courantes sur le marché. *Gobo* analyse environ 75 000 lignes de code source par seconde et SmallEiffel passe le cap des 100 000 lignes de code source analysé par seconde.

SmallEiffel (version -0.76) est composé d'environ 80 000 lignes de code source Eiffel et se recompile environ en 6.5 secondes sur la même machine que celle utilisée à la figure 3.5. Ainsi, le temps passé pour l'analyse syntaxique par rapport au temps total de compilation ne représente que 12% du temps total de compilation. Dans la pratique de l'écriture des compilateurs, cette constatation indique, à mon avis, qu'il est inutile de mémoriser sur disque la représentation abstraite des programmes dans l'espoir d'économiser du temps de calcul lors de la prochaine re-compilation. Ceci justifie pleinement le choix que nous avons fait dans SmallEiffel de ne pas mémoriser sur disque la représentation abstraite des classes d'une compilation sur l'autre.

Notons pour finir que l'efficacité de l'analyseur syntaxique n'était pas la toute première priorité de notre implantation (voir section 3.2). En outre, il est certainement possible d'améliorer encore les performances en modifiant la mémorisation de la position courante dans le texte source. En effet, l'implantation actuelle est simpliste puisqu'elle consiste à détecter le changement de ligne à l'aide d'un test dans la fonction `next_char`. Il serait probablement plus efficace de ne mémoriser que l'indice du caractère courant en considérant l'ensemble du fichier source comme une seule et très longue chaîne de caractères. Dans ce cas, l'information ligne/colonne peut bien entendu être recalculée sur nécessité lors de l'affichage d'un message d'erreur.

## 3.8 Discussion et conclusion

Nous avons présenté dans ce chapitre l'implantation de l'analyseur syntaxique et lexical de SmallEiffel. Cette méthode a également été utilisée avec succès pour le langage C ANSI dans le cadre du projet ProMic (annexe A). Comme nous l'avons mentionné au début de ce chapitre, une technique similaire a été utilisée pour le compilateur GNU Ada.

Cette approche peut au premier abord paraître surprenante sachant que la grande majorité des publications sur le sujet favorisent très largement l'utilisation de générateur d'analyseurs syntaxique et d'analyseurs lexical comme *lex/yacc*. Nous montrons dans ce chapitre qu'il est très simple de construire, sans outil particulier, un analyseur syntaxique commun à plusieurs applications, précis dans le repérage des unités lexicales, et qui plus est performant. Enfin, la simplicité de mise en œuvre du rattrapage d'erreur en analyse syntaxique descendante est également un avantage par rapport aux méthodes ascendantes à base de tables.

L'ensemble analyseur syntaxique et analyseur lexical est codé complètement en Eiffel dans la classe EIFFEL_PARSER qui correspond en fait à un objet unique (*Singleton* Pattern [GHJV95] [JTM99]). Cette classe construit une représentation abstraite commune aux différents outils de la distribution de SmallEiffel : `compile_to_c`, `compile_to_jvm`, `pretty` et `short`.

Le positionnement des unités lexicales est réalisé de façon précise, ligne et colonne, ce qui permet entre autres choses de préparer des messages d'erreurs détaillés mais aussi de renseigner précisément le débogueur ainsi que les outils de mise en page ou d'extraction d'interface.

L'utilisation de l'analyse syntaxique descendante offre la possibilité de programmer facilement le rattrapage des erreurs syntaxiques. Enfin, le fait que l'analyse lexicale soit toujours effectuée à l'aide d'une information contextuelle est probablement la raison principale des excellentes performances des analyseurs produits avec cette technique.

Pour conclure sur l'analyse syntaxique et sur l'analyse lexicale, il convient d'ajouter que ce travail n'est qu'une très petite partie dans la réalisation du projet SmallEiffel et d'un compilateur en général. En terme de nombre de lignes de code, l'analyseur syntaxique, c'est à dire la classe EIFFEL_PARSER qui fait environ 4500 lignes, ne correspond qu'à environ 6% du code source de SmallEiffel. L'analyse syntaxique n'est qu'un point de passage obligé avant de pouvoir se consacrer aux autres aspects de la compilation : vérifications sémantiques, optimisations, génération de code, analyse de flots, allocations de registres, etc.

# Chapitre 4

# Aliasing et programmation par contrats

“Quand il s’agit de tromperie,
De duperie,
De volerie,
Il est toujours bon, sur ma foi,
D’avoir les femmes avec soi,

Georges Bizet [Carmen, $2^{ième}$ Acte]

## 4.1 Introduction

Ce chapitre présente les travaux que nous avons effectués concernant l’*aliasing* dans le but premier d’optimiser SmallEiffel lui-même. La technique présentée ici permet, *a priori*, d’optimiser n’importe quel autre genre de logiciel. L’originalité de nos travaux par rapport à ceux du domaine de l’aliasing tient à l’approche “positive” envers l’aliasing que nous avons adoptée, et qui, à notre connaissance, a rarement été suivie.

En effet, l’aliasing est connu depuis déjà longtemps (voir par exemple [Mey88]), et est en général considéré comme quelque chose de dangereux qui devrait être évité autant que possible dans les programmes. Beaucoup de travaux se sont donc concentrés sur la détection, la prévention et les moyens d’éviter l’aliasing [Hog91] [HLW$^{+}$92] [Min96].

Il semble cependant très difficile, si ce n’est impossible, d’éviter l’aliasing dans les langages à objets et des techniques ont donc également été recherchées pour contrôler l’aliasing.

Pour notre part, nous pensons que l’aliasing est inhérent à la programmation à objets et ne peut être supprimé sans que l’on perde une bonne partie des facilités de développement. Nous sommes également convaincus que l’aliasing offre des avantages certains et significatifs en termes de performances et qu’il peut donc être utilisé avec profit.

Pour ces raisons, l’implantation de SmallEiffel, fait un usage intensif, mais soigneusement pensé de l’aliasing, essayant de réconcilier performance et sûreté à l’aide de techniques pour la plupart déjà connues. Dans ce chapitre, qui rapporte notre expérience avec SmallEiffel, nous montrons donc comment certains choix de conception simples peuvent permettre un usage plus sûr de l’aliasing, même lorsque le langage, comme Eiffel, n’intègre pas de support particulier pour l’aliasing.

Ce chapitre est organisé comme suit. La section 4.2 rappelle tout d’abord brièvement ce qu’est l’aliasing. Les principales références du domaine sont données en section 4.3. Puis dans la section 4.4, nous expliquons pour quelles raisons nous considérons que l’aliasing est très utile dans un compilateur. La section 4.5 introduit ensuite le concept de *fournisseur d’alias* et détaille comment celui-ci peut être mis en œuvre en Eiffel en utilisant le modèle de conception *Singleton* [GHJV95] [JTM99]. Puis nous décrivons dans la section 4.6 comment la programmation par contrats, *design by contract*, et les assertions Eiffel peuvent être utilisées pour rendre l'emploi de l'aliasing plus sûr. La section 4.7 considère le problème des objets modifiables. Les bénéfices dus à l'utilisation de l'aliasing sont discutés en section 4.8. Finalement, la section 4.9 conclut.

## 4.2 L'aliasing : définition

L'*aliasing* est le fait de *référencer une même donnée via deux variables différentes*, ou plus généralement, deux chemins différents, les *alias*. La donnée en question peut être aussi simple qu'un booléen, ou aussi compliquée qu'une grande structure de données ou un objet complexe. L'aliasing est permis dans la plupart des langages informatiques, langages applicatifs dont les langages à objets ainsi que dans les langages fonctionnels.

Bien entendu, l'aliasing est extrêmement courant dans de nombreux types de programmes. Hormis le cas des langages purement fonctionnels [17], on comprend aisément le danger potentiel de l'aliasing. En effet, lorsque le développeur manipule un objet aliasé, il n'est pas certain que lui seul contrôle les modifications qui peuvent survenir sur cet objet, puisqu'un autre alias peut être utilisé pour changer son état. L'aliasing brise donc, en quelque sorte, la séquentialité et l'atomicité apparentes de certains morceaux de code. Ce problème est bien entendu exacerbé lorsqu'on est dans un contexte d'exécution parallèle, par exemple dans un programme *multi-threads*.

En revanche, lorsque le développeur est conscient des aliasings qui sont présents dans le programme sur lequel il travaille, il peut les gérer, et en tirer parti.

Il est à noter que, comme le montre notre revue des travaux du domaine ci-dessous, l'aliasing est la plupart du temps considéré comme une pratique néfaste. Nous avons adopté dans nos recherches une approche qui consiste plutôt à le considérer comme une *technique* utile, comme nous le détaillerons à partir de la section 4.4

## 4.3 Travaux du domaine

Dans cette section, nous présentons brièvement les principaux travaux sur l'aliasing et l'analyse de pointeurs. Comme l'aliasing peut dans certains cas être à l'origine d'importants problèmes, voir de cauchemars de déverminage, ces travaux se sont souvent focalisés sur la prévention de l'aliasing et les moyens de l'éviter. La détection des alias a aussi été un sujet de recherche actif, notamment en relation avec les techniques d'optimisation des programmes.

John Hogg décrit dans [Hog91] une méthode basée sur des modes d'accès permettant de limiter l'aliasing et de créer des *ilôts* (*islands*) d'objets. Un *ilôt* est un ensemble d'objets qui peuvent se référencer les uns les autres librement, mais pour lesquels il n'existe qu'une porte d'accès vers et depuis le monde extérieur, chargée de contrôler et de restreindre l'aliasing. Ainsi, l'aliasing *intra ilôt* est permis, ce qui permet d'utiliser des modèles de conception utilisant l'aliasing, tandis que l'aliasing *inter-ilôts* est restreint, ce qui permet aux *ilôts* d'éviter les problèmes potentiellement créés par l'aliasing.

Les *types ballons* (*balloons types*) [Alm97] présentent certaines similarités avec les *ilôts*. Cependant, les *types ballons* sont plus puissants, car il s'agit d'extensions aux langages de programmation intégrées au système de type. Ils permettent un contrôle plus complet vis-à-vis du partage de l'état interne d'un type de donnée.

17. Dans le cas d'un langage fonctionnel "pur", l'aliasing n'est a priori pas dangereux car les entités partagées ne peuvent pas être modifiées.

Un certain nombre d'analyses d'alias ou de pointeurs ont été réalisées et divers algorithmes furent développés par les chercheurs depuis une dizaine d'années. Dans [WL95], Wilson et Lam détaillent une analyse de pointeurs sensible au contexte, basée sur des fonctions de transfert partielles (*Partial Transfer Functions*) chargées de "résumer" les effets d'une routine, et montrent des résultats encourageants sur des programmes C. Néanmoins, [Ruf95] indique que la prise en compte du contexte n'améliore pas toujours l'analyse et que dans les cas qu'ils ont étudiés, l'analyse d'alias sans prise en compte du contexte peut être à la fois "étonnamment précise" (*"surprisingly precise"*) et efficace.

Une technique pour éviter les alias est décrite dans [Min96], basée sur des *objets non partageables* (*unshareable objects*) référencés par des *pointeurs uniques* qui peuvent être déplacés mais pas copiés. Quand une affectation implique des pointeurs uniques, la source de l'affectation est nullifiée (remise au pointeur `null`), garantissant ainsi que la destination de l'affectation sera la seule et unique référence restant sur l'objet. Ceci permet l'implantation, avec un certain coût à l'exécution, d'objets qui ne sont jamais sujets à l'aliasing, en plus des objets normaux qui eux peuvent être aliasés.

L'analyse d'alias présentée dans [DMW98] est effectuée à bas niveau, directement sur le code exécutable. L'avantage principal de cette approche est que les aspects bas niveau liés aux pointeurs (essentiellement arithmétique de pointeurs) peuvent être pris en compte, alors qu'ils ne le sont pas dans les analyses de haut niveau (langage).

Ghiya et Hendren montrent dans [GH98] comment deux analyses de pointeurs existant — la *points-to analysis* et la *connection analysis* — peuvent être comparées et comment elles peuvent être utilisées pour étendre et améliorer certaines optimisations classiques dans un compilateur, comme l'extraction d'invariants de boucles, l'élimination des sous-expressions communes et les tests de dépendance sur les tableaux (*array dependence testing*).

Une protection d'alias flexible (*Flexible alias protection*) est proposée dans [NVP98]. Grâce à l'addition d'une information explicite à chaque déclaration afin de spécifier le *mode d'aliasing* des objets, elle permet de contrôler le degré d'aliasing et les modifications autorisées depuis le code client. Cette technique impose une extension des langages à objets actuels, mais n'implique aucun surcoût à l'exécution et permet d'éviter les problèmes potentiellement causés par l'aliasing tout en gardant un style de programmation à objets relativement naturel.

Un système de type formel pour une partie du travail précédent a été conçu dans [CPN98]. Il se base sur des *ownership types* pour contrôler la visibilité des références et les chemins d'accès aux objets, permettant de garder la représentation interne des objets confinée (*representation containment*).

Des extensions à Java sont proposées dans [BV99] pour implanter des *types confinés* (*confined types*). Ces derniers sont des types dont les instances sont confinées dans les modules (i.e. ne peuvent être référencées depuis l'extérieur), ou *packages*, auxquels ils appartiennent. Les *types confinés* n'impliquent aucun surcoût à l'exécution et permettent de déceler statiquement, à la compilation, les erreurs d'aliasing et de dissémination.

Trois types d'analyses d'alias sont comparés dans [DMM98]. La première est basée sur les seules informations de compatibilité de types. La seconde étend la précédente en ajoutant des informations de haut niveau à propos des attributs. La troisième étend la seconde avec une analyse ne prenant pas en compte le flot de données (*flow insensitive*). Des évaluation statiques et dynamiques montrent que la prise en compte de la compatibilité des types est d'un faible coût, mais fournit des informations d'aliasing imprécises, qui peuvent sensiblement être améliorées par les deux autres analyses. L'analyse basée sur les types apparaît très efficace et permet des optimisations supplémentaires du programme comme l'élimination des affectations redondantes par exemple.

Une nouvelle analyse de pointeurs inter-procédurale, prenant en compte le contexte et le flot d'exécution, est présentée dans [RR99] pour des programmes utilisant plusieurs *threads* où les pointeurs partagés peuvent être mis à jour de façon concurrente. Cet algorithme supporte un large ensemble de constructions et est de complexité polynomiale. Des résultats expérimentaux montrent que cet algorithme est assez précis et a de bonnes propriétés de convergence sur les bancs d'essais testés.

Les problème liés aux conversions forcées, *type casting*, sont abordés dans [YHR99]. Un schéma d'analyse de pointeurs réglable permettant de gérer les structures en présence de conversions de types y est présenté. Des résultats sont donnés pour quatre implantations de ce schéma. Ils montrent qu'il est important de distinguer les différents champs des structures et confirment qu'il est possible d'émettre à faible coût des hypothèses conservatrices afin de préserver la portabilité.

Comme nous l'avons précédemment mentionné, ces travaux et les nôtres ont des approches assez différentes. La plupart d'entre eux se focalisent sur la prévention et la limitation de l'aliasing, alors que nous avons choisi, lors de la conception de SmallEiffel, d'essayer de *nous baser sur* l'aliasing, d'une façon disciplinée, afin d'obtenir les meilleures performances possibles sans sacrifier la sécurité.

## 4.4 Intérêt de l'aliasing pour implanter un compilateur

Pour obtenir une implantation efficace d'un compilateur, l'utilisation de l'aliasing est extrêmement fréquente, naturelle, voire même indispensable. Considérons par exemple le cas des identificateurs. A chaque fois que le compilateur rencontre un identificateur, il lui faut vérifier par exemple s'il s'agit d'une variable locale, s'il s'agit d'un argument, d'une variable globale ou bien d'une variable non déclarée. Pour ce faire, le compilateur doit chercher le nom de l'identificateur qu'il vient de rencontrer, dans la liste des variables locales courantes, puis, s'il ne l'a pas trouvé, dans la liste des arguments et enfin dans celle des variables globales.

On comprend aisément pourquoi l'opération qui correspond à comparer deux noms de variables est extrêmement fréquente. Pour effectuer ce travail efficacement, il est capital d'éviter de coûteuses comparaisons sur les *contenus* de chaînes de caractères.

L'idée de base de l'aliasing dans ce cas consiste à utiliser exactement la même chaîne de caractères quand plusieurs symboles (variable, nom de fonction, nom de type, nom de classe, etc.) ont le même nom. Ainsi, il est possible de comparer deux noms de symboles avec une simple comparaison de références (opérateur `=`) au lieu de devoir examiner l'ensemble des caractères qui composent le nom (méthode `is_equal`). Ceci évite les duplicata d'une même chaîne, économisant ainsi une importante quantité de mémoire. Il semble également y avoir un gain possible en terme de vitesse. Cependant, le coût de la gestion d'un tel système d'aliasing des noms de symboles peut gommer une partie de ces avantages et doit donc être évalué plus précisément.

Bien entendu, les chaînes de caractères ne sont pas les seuls objets "aliasables" dans un compilateur, mais comme ce sont des objets communs et bien compris de tous, nous les utiliserons comme exemple de base tout au long de ce chapitre. En fait, de nombreux autres types d'objets plus complexes sont eux aussi aliasés dans l'implantation de SmallEiffel comme par exemple certains nœuds de l'arbre abstrait ainsi que les objets qui représentent les classes pour ne citer que ceux-ci.

De manière générale, l'utilisation de l'aliasing n'est pas limitée aux compilateurs, mais peut en fait être appliquée à la plupart des types de logiciels, tels que les moteurs de recherche, les interfaces graphiques, les interprètes, systèmes de réécriture, etc.

## 4.5 Utilisation d'un fournisseur d'alias

Les noms de symboles partagés sont apparentés aux *interned atoms* offerts par Lisp ou Smalltalk, ou aux *interned strings* de Java. Contrairement à ces langages, Eiffel ne propose pas un tel mécanisme qui doit donc être émulé explicitement par le développeur Eiffel.

Dans SmallEiffel, nous implantons ce type d'aliasing à l'aide d'un *fournisseur d'alias* ou "*aliaseur*". En effet, bien que la gestion locale des objets aliasés soit très facile et intuitive dans une routine ou un petit algorithme, ce n'est plus le cas lorsque l'aliasing doit être géré à travers l'ensemble du système. Par exemple, bien que les noms de symboles soient seulement créés dans l'analyseur syntaxique, ils sont manipulés par toutes les autres classes durant toutes les autres phases de la compilation.

Pour cette raison, les noms d'identificateurs sont fournis par un autre objet que l'analyseur syntaxique lui-même. Cet objet spécial est appelé dans SmallEiffel et dans ce chapitre le STRING_ALIASER. Il présente certaines similarités avec un pont (*bridge*) donnant accès à des îlots (*islands*) d'objets [Hog91]. Cependant, contrairement à un objet "pont", notre fournisseur d'alias n'a pas pour but de fournir un accès seulement temporaire aux objets qu'il contient (aliasing dynamique [HLW+92]) mais également un accès contrôlé et à long terme (aliasing statique).

Dans cette optique, l'objet STRING_ALIASER doit être l'unique objet de sa classe dans l'ensemble du système. Le modèle de conception *Singleton* [GHJV95] [JTM99] est ici parfaitement adéquat.

### 4.5.1 Mise en oeuvre du modèle de conception Singleton en Eiffel

La raison d'être du modèle de conception *Singleton* est, comme son nom l'indique, de garantir qu'une classe n'a qu'une seule instance dans un système et de fournir un accès global à cette instance.

L'implantation C++ proposée dans [GHJV95] garantit que la classe Singleton est instanciée une seule fois grâce à un test explicite lors de l'exécution.

En Java il est possible de garantir sans aucun coût à l'exécution qu'une seule instance d'une classe est créée. Ceci se fait en rendant le constructeur privé, en créant une seule instance statique dans le bloc d'initialisation statique et en redéfinissant la méthode `clone` afin qu'elle retourne une référence vers cette instance statique.

En Eiffel, un Singleton peut être implanté en se basant sur les mécanismes d'assertions et de routines *once* de la manière suivante:

```
class A_SINGLETON
   ...
   ...
   ...
feature {NONE}  -- Singleton Part:

   singleton_memory: A_SINGLETON is
      once
         Result := Current;
      end;

invariant

   is_actually_singleton: Current = singleton_memory

end -- A_SINGLETON
```

La fonction `singleton_memory` est une fonction `once` [Mey92]. Une telle fonction est exécutée une seule fois, la première fois qu'elle est appelée. Les appels consécutifs retournent exactement le même résultat qui est automatiquement mémorisé pour toutes les instances de la classe. Comme `singleton_memory` est exportée à NONE, elle est privée et peut être appelée uniquement depuis l'objet `Current`, de type A_SINGLETON ou un de ses sous-types. Ainsi, la fonction de classe `singleton_memory` mémorise la référence vers le premier objet qui l'appelle, le premier objet crée de cette catégorie.

Les invariants en Eiffel sont des assertions qui sont déclenchées à chaque fois qu'un client appelle une routine sur une instance de la classe englobant l'invariant. Les invariants sont hérités par les descendants des classes où ils apparaissent. L'invariant `is_actually_singleton` garantit qu'un seul objet de type A_SINGLETON (ou de l'un de ses sous-types) est jamais créé dans le système. En effet, il vérifie que la référence sur l'objet `Current` est exactement celle stockée par `singleton_memory`. Si plus d'un objet de type A_SINGLETON était créé, cet invariant serait faux et déclencherait une exception sur le second objet, car `Current` ferait référence à ce second objet alors que `singleton_memory` référencerait le premier.

Il est important de noter que, comme la solution Java mentionnée précédemment, et contrairement à l'implantation référence en C++, l'implantation Eiffel d'un singleton n'implique absolument aucun surcoût en mode optimisé. En effet, les invariants, comme les autres assertions, sont conçus pour être utilisés durant la phase de développement du code, mais ne sont plus activés dans le code final livré aux utilisateurs.

Bien entendu, à cause de la sémantique des routines `once` d'Eiffel, chaque classe qui doit être un singleton doit implanter la fonction `singleton_memory` et son invariant `is_actually-_singleton`. Hériter simplement de A_SINGLETON n'est pas une solution viable car il serait ainsi impossible d'avoir plus d'un Singleton dans le système, puisque comme nous l'avons indiqué précédemment, le Singleton est commun au type dans lequel il apparaît, ainsi qu'à tous ses sous-types.

### 4.5.2 Implantation d'un fournisseur d'alias

Grâce à l'implantation proposée ci-dessus pour les Singletons, la spécification de notre fournisseur d'alias devient très simple. En plus du code pour le modèle de conception Singleton, notre STRING_ALIASER offre uniquement une fonction, `item`, chargée de fournir une référence vers la chaîne partagée correspondant au modèle spécifié. En voici l'interface complète :

```
class interface STRING_ALIASER

   item (model: STRING): STRING
      require
         model /= Void
      ensure
         Result.is_equal(model)

invariant

   is_actually_singleton: Current = singleton_memory

end -- STRING_ALIASER
```

La première fois qu'`item` est appelée avec un certain nom comme argument, elle le retourne et mémorise la chaîne `model` dans un DICTIONARY – une sorte de table de hachage – privé. Les appels suivants avec, soit la même chaîne modèle, soit une chaîne modèle différente ayant le même contenu, retourneront tous (la référence vers) le tout premier objet chaîne mémorisé.

Comme pour l'invariant, la pré-condition, *require*, et la post-condition, *ensure*, ne sont pas générées en mode optimisé, le mode habituellement utilisé pour la livraison.

## 4.6 Promouvoir et contrôler l'usage du fournisseur d'alias

Nous avons décrit dans les sections précédentes comment l'unicité du fournisseur d'alias pouvait être garantie. Il est bien sûr également très important que dans tout le système tous les objets utilisent ce fournisseur d'alias et qu'aucun ne garde ses propres duplicata des chaînes aliasées. De cette façon, une seule et unique version de la chaîne est créée par le fournisseur d'alias.

Toutes les classes du système doivent donc respecter la "règle du jeu" et utiliser le fournisseur d'alias. Elles doivent également prendre part à la sécurité de l'ensemble du système en vérifiant que leurs propres fournisseurs eux aussi utilisent bien le fournisseur d'alias. Ceci peut facilement être implanté en Eiffel à l'aide des contrats, les pré-conditions et les post-conditions des routines.

L'exemple suivant, LOCAL_NAME, montre une classe utilisée par SmallEiffel pour représenter un identificateur correspondant au nom d'une variable locale. La procédure de création `make` prend deux arguments : un pour enregistrer la position dans le code source, l'autre pour la chaîne qui contient de nom de l'identificateur.

```
class interface LOCAL_NAME
-- Some local variable name.
creation

   make (sp: POSITION; n: STRING)
      require
         sp /= Void;
         n = string_aliaser.item(n)  -- (#1)
      ensure
         start_position = sp;
         to_string = n               -- (#2)

   ...
   ...
   ...
end -- LOCAL_NAME
```

Dans l'extrait de code ci-dessus, la pré-condition `(#1)` sert à vérifier que l'appelant passe bien un argument qui est une chaîne *aliasée* par le STRING_ALIASER. En effet si le contenu de `n` est le même que celui d'une chaîne déjà rencontrée et mémorisée par l'aliaseur de chaînes, `string_aliaser.item(n)` retourne une référence vers l'objet mémorisé. Donc si la chaîne `n` a été fournie par le STRING_ALIASER, l'objet mémorisé correspondant au contenu de `n` est `n` elle-même.

Similairement, la post-condition `(#2)` garantit aussi le comportement correct de la classe englobante, c'est à dire que la chaîne aliasée a bien été mémorisée par l'objet LOCAL_NAME courant, qui est donc un nouvel *alias* vers l'objet partagé.

Le même genre de contrat s'applique par exemple dans CLASS_NAME, la classe qui représente les noms de classes, ainsi que dans ARGUMENT_NAME la classe qui représente les noms d'arguments.

## 4.7 Aliasing pour les objets modifiables

Bien entendu, promouvoir l'utilisation du fournisseur d'alias n'est pas suffisant pour travailler avec l'aliasing en toute sécurité. Il est également nécessaire de contrôler strictement qui modifie les objets aliasés et où.

Les chaînes Eiffel sont des objets modifiables : leur contenu peut être changé, étendu, raccourci, etc. Dans le compilateur SmallEiffel, néanmoins, une chaîne aliasée ne devrait jamais être modifiée. S'il en était autrement, il en résulterait le chaos le plus total dans le processus de compilation.

Puisqu'une classe comme STRING, issue de la bibliothèque standard, contient de nombreuses routines de modification publiquement accessibles, la meilleure façon de contrôler leur utilisation serait d'encapsuler l'objet STRING dans, par exemple, une classe IMMUTABLE_STRING qui comporterait les accesseurs vers l'objet STRING mais aucun modificateur, ou au moins des modificateurs sélectivement exportés. Les sous-classes de STRING — qui n'existent pas actuellement dans le contexte du compilateur SmallEiffel — pourraient aussi être encapsulées de la même façon, par IMMUTABLE_STRING ou l'une de ses sous-classes.

Cette solution, bien que très satisfaisante du point de vue conceptuel, n'a pas été mise en oeuvre dans SmallEiffel car elle aurait impliqué un coût important tant en termes de mémoire (objets IMMUTABLE_STRING supplémentaires) qu'en vitesse d'exécution (niveau d'indirection supplémentaire).

Il est néanmoins possible de prendre des mesures pour s'assurer que les chaînes aliasées ne sont pas modifiées. Pour ce faire, des assertions peuvent être ajoutées au sein même de la classe STRING_ALIASER. Ceci requiert dans STRING_ALIASER un second DICTIONARY, dans lequel sont conservées les copies (pas des alias) des chaînes aliasées. Puis, à chaque fois qu'`item` est appelée sur STRING_ALIASER, une assertion doit être déclenchée, vérifiant que les contenus des deux dictionnaires sont égaux. Comme les copies conservées dans le second dictionnaire ne sont pas accessibles de l'extérieur, une violation d'assertion à ce point indiquerait que l'un des objets aliasés a été modifié, signalant ainsi un comportement incorrect de l'un des alias. Ceci n'est cependant pas parfait, car ces vérifications sont réalisées de façon asynchrone, c'est à dire seulement lorsqu'`item` est appelée.

En revanche, aliaser des objets modifiables (et effectivement modifiés) pose moins de problème lorsqu'on a affaire à des objets qui ont étés *conçus pour être aliasés*.

De tels objets sont par exemple les instances du type BASE_CLASS, qui représentent des classes après leur analyse syntaxique. Ces objets sont bien plus complexes que des chaînes de caractères car ils contiennent toutes les informations concernant la classe qu'ils représentent : nom, ancêtres, attributs, routines, etc. Si une classe C est présente dans le système, définie dans un fichier `c.e`, une seule instance de BASE_CLASS y correspondant est jamais créée, quel que soit le nombre d'occurrences de C dans le code client. Évidemment, aliaser ce type d'objets est indispensable. Faire autrement, si c'est possible, ne pourrait résulter qu'en la création d'un nombre considérable d'instances de BASE_CLASS et serait — au mieux — totalement inefficace.

Le mécanisme utilisé ici est très semblable à celui précédemment détaillé pour les chaînes de caractères. Le fournisseur d'alias pour les objets de type BASE_CLASS est le Singleton EIFFEL_PARSER. Son rôle est de garantir que le fichier source de chaque classe n'est analysé syntaxiquement qu'une seule fois, sur demande, pour d'évidentes raisons de performance, et de fournir des alias sur les objets de type BASE_CLASS. En effet, le Singleton EIFFEL_PARSER est le seul objet autorisé à créer des instances de BASE_CLASS. Grâce au mécanisme d'export sélectif d'Eiffel, cette propriété peut être *contrôlée statiquement*. L'extrait de code suivant montre la clause d'exportation de la seule méthode de création de BASE_CLASS :

```
class BASE_CLASS
   ...
creation {EIFFEL_PARSER} make
   ...
end -- BASE_CLASS
```

Ainsi, à chaque fois qu'un objet de type BASE_CLASS apparaît dans le code client, on sait statiquement qu'il s'agit d'un objet aliasé qui vient — directement ou non — de la classe EIFFEL_PARSER. Les vérifications appropriées sont donc effectuées à la compilation et aucun surcoût n'apparaît à l'exécution, que ce soit en mode optimisé ou lors du développement. Ceci contraste avec les objets STRING que nous avons précédemment détaillés, qui peuvent aussi être créés hors du STRING_ALIASER et donc nécessitent du code supplémentaire dans le client pour contrôler leur origine lors de l'exécution.

Ce mécanisme d'export sélectif est aussi extrêmement utile pour obtenir un contrôle fin sur les modifications des objets de type BASE_CLASS. Contrairement à ce qui se produit pour les objets STRING, aucune "enveloppe" supplémentaire n'est nécessaire et aucun surcoût n'apparaît lors de l'exécution, ni au niveau mémoire, ni au niveau vitesse. Les routines modificatrices (i.e. les mutateurs) peuvent n'être exportées que vers le type des objets qui ont besoin d'y accéder, de la façon suivante :

```
class BASE_CLASS
   ...
feature {TYPE}
   smallest_ancestor(type, other: TYPE): TYPE is
   ...
feature {EIFFEL_PARSER}
   add_index_clause(index_clause: INDEX_CLAUSE) is
   ...
feature {RUN_CLASS, PARENT_LIST}
   collect_invariant(rc: RUN_CLASS) is
   ...
end -- BASE_CLASS
```

Il est intéressant de remarquer que ce mécanisme du langage Eiffel — l'export sélectif — concilie avec succès le passage élégant du niveau conceptuel à l'implantation et la recherche de performances maximales. Il n'y a à notre connaissance aucun équivalent direct dans d'autres langages à objets communs tels que C++ ou Java.

Avec l'aliasing de BASE_CLASS, il est également capital de faciliter la réalisation sur nécessité de certaines opérations et d'éviter de refaire les calculs plusieurs fois. Par exemple, la remontée de l'arbre d'héritage pour trouver l'origine d'une routine héritée est une opération très fréquente. Comme elle implique des calculs coûteux qui se propagent aux ancêtres de la BASE_CLASS, il est beaucoup plus efficace de ne la déclencher qu'une seule fois et de modifier la liste des parents en conservant ses résultats intermédiaires pour les réutiliser par la suite. Grâce à l'aliasing, toutes les occurrences d'un certain nom de classe dans le code partagent le même objet BASE_CLASS et donc la même liste d'ancêtres. Ainsi, une fois qu'un contrôle sémantique est effectué sur l'une de ces occurrences, l'information correspondante devient immédiatement — et pratiquement sans aucun coût supplémentaire — disponible pour toutes les autres occurrences.

On peut remarquer que de cette façon, les alias ne reçoivent pas de notifications des modifications de l'objet aliasé. En effet, il n'y avait aucun besoin de notification dans notre application.

Il est cependant extrêmement facile de rajouter un tel système de notification en cas de besoin, grâce au Singleton EIFFEL_PARSER qui gère toutes les demandes d'alias.

## 4.8 Résultats expérimentaux

Comme nous l'avons montré ci-dessus, l'aliasing est très présent dans l'ensemble du compilateur SmallEiffel et concerne de nombreux types d'objets. Afin de quantifier précisément l'impact de l'aliasing, les performances de deux versions du compilateur devraient être comparée : une avec aliasing (partout où c'est possible) et une autre sans aucun aliasing. Cette dernière aurait impliqué une réécriture majeure et très longue du compilateur SmallEiffel. De plus, comme indiqué précédemment, il semble même impossible d'écrire un compilateur sans avoir au moins certaines formes d'aliasing, par exemple pour BASE_CLASS. En conséquence, nous avons dû restreindre nos mesures de performances au seul aliasing des chaînes de caractères.

Pour quantifier l'impact de l'aliasing des chaînes durant le processus de compilation, nous avons donc développé une seconde version de SmallEiffel sans l'aliasing des chaînes. Il est intéressant de noter que ceci fut grandement facilité par le fait que l'ensemble du processus d'aliasing était concentré dans le STRING_ALIASER. Tous les tests d'égalité référentielle entre deux STRINGs (opérateur `=`) ont été remplacés par la comparaison structurelle (méthode `is_equal`).

Nous avons mesuré les performances des deux versions du compilateur lors de l'auto-compilation de chacune de ces deux version par elle-même [18]. Une auto-compilation constitue un banc d'essai significatif, car il s'agit d'une véritable application, complètement écrite en Eiffel et représentant environ 80000 lignes de codes pour environ 300 classes vivantes.

La taille mémoire semble pouvoir bénéficier de l'aliasing des chaînes. Le nombre total de chaînes (mots) rencontrées durant l'auto-compilation est précisément de 79838. Tous ces mots contiennent 637014 octets, ce qui correspond à une longueur de mot moyenne de 8 octets.

Sachant qu'un objet de type STRING a une occupation mémoire de 3 mots plus le contenu de la chaîne elle-même (les caractères), on peut facilement calculer la taille mémoire totale théorique de tous les objets de type STRING non aliasés sur une machine 32 bits :

$$\begin{array}{rclcll} \text{nb\_chaînes} & \times & (\text{taille\_objet\_string} & + & \text{longueur\_moyenne}) & = \\ 79838 & \times & (12 & + & 8) & = \\ & & 1596760 \text{ octets} & = & 1559 \text{ KO} & \end{array}$$

Ces 79838 occurrences de chaînes correspondent à seulement 4265 chaînes différentes, dont la taille mémoire peut être estimée à :

$$4265 \times (12 + 8) = 85300 \text{ octets} = 83 \text{ KO}$$

Donc lorsque l'aliasing est utilisé, la taille mémoire théorique des objets de type STRING dans le compilateur est réduite de 94,6 % par rapport à ce qu'elle est sans l'aliasing des chaînes.

Ces valeurs théoriques ne prennent pas en compte l'espace supplémentaire nécessaire pour la gestion mémoire, comme les entêtes des `malloc` C ou les données pour la gestion du ramasse-miettes [CCZ98a]. Nos résultats expérimentaux (figure 4.1) montrent qu'en fait la quantité de mémoire économisée grâce à l'aliasing est de 2456 KO.

18. Les résultats ont été obtenus sur un PC Pentium II 333 MHz avec 128 MO de RAM, sous Linux, noyau 2.0.36, et avec SmallEiffel -0.78 et egcs-1.1.1.

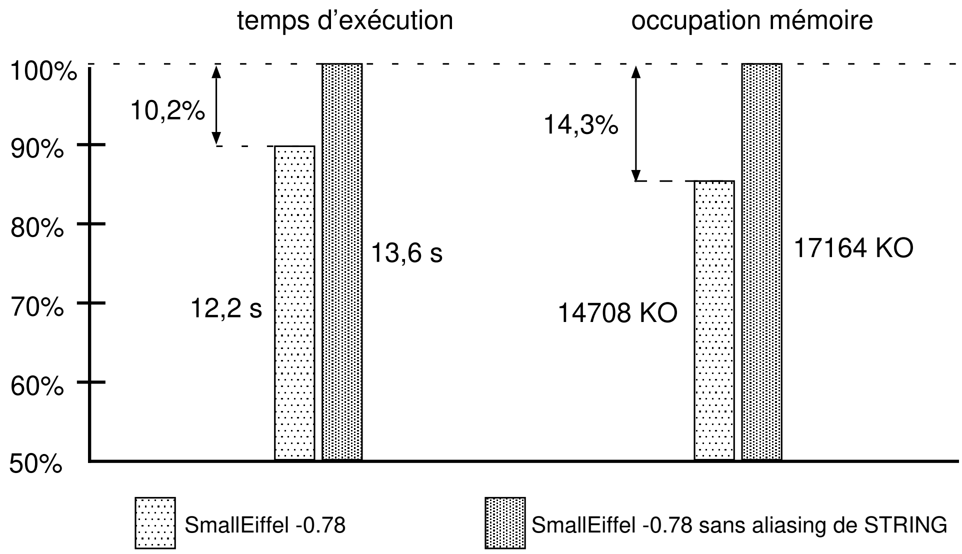


FIG. 4.1 – *Comparaison de performances : SmallEiffel avec et sans l'aliasing des chaînes de caractères*

Comme la taille mémoire maximale de l'ensemble du compilateur (lorsqu'il compile SmallEiffel) est d'environ 14708 KO, le bénéfice *venant du seul aliasing des chaînes de caractères* représente une diminution de plus de 14 % et est donc très significatif. Ceci est d'autant plus vrai si l'on se rappelle que SmallEiffel n'est pas seulement un projet de recherche, mais également un produit gratuit utilisé dans le monde entier par — entre autres — de nombreuses personnes sur de petits ordinateurs individuels pour lesquels une mémoire vive de 32 MO n'est pas rare du tout (ou même 16 MO, à l'époque où le projet SmallEiffel a débuté).

Cette réduction d'occupation mémoire est aussi un point positif en terme de vitesses d'exécution, car diminuer la taille mémoire du programme a également pour conséquence d'alléger le travail du ramasse-miettes et de diminuer les échanges avec le disque (*swapping*).

En pratique, les effets de l'aliasing des chaînes sur le temps d'exécution sont bien plus que des conséquences indirectes. En effet, comme nous l'avons écrit dans la section 4.4, une des raisons principales à l'utilisation de l'aliasing des chaînes dans le compilateur SmallEiffel est de pouvoir faire des comparaisons rapides de chaînes dans le code client, en comparant simplement deux références avec l'opérateur `=` au lieu de comparer (les contenus de) deux objets avec la fonction `is_equal`. Cependant, l'inconvénient de l'aliasing des chaînes est que la gestion du STRING_ALIASER induit un certain coût supplémentaire. En effet, à chaque fois qu'une nouvelle chaîne est rencontrée, il est nécessaire de vérifier si elle est déjà dans l'aliaseur et, dans le cas contraire, le nouvel objet doit être créé puis inséré dans l'aliaseur. Il était donc raisonnable de se demander si le coût nécessaire pour aliaser les chaînes n'allait pas annuler le gain en temps d'exécution procuré par les comparaisons de références.

Les expérimentations que nous avons menées montrent (figure 4.1) que *le seul aliasing des chaînes* conduit en fait à une amélioration de plus de 10 % du temps de compilation total, ce qui est tout à fait considérable.

Encore une fois, il est important de souligner que cet accroissement sensible des performances vient uniquement de l'aliasing des chaînes. Il n'est pas déraisonnable de penser que les autres types d'aliasing (sur les BASE_CLASSes, LOCAL_NAMEs, etc.) amènent eux aussi des gains substantiels en vitesse. L'aliasing sous diverses formes semble donc jouer un rôle important dans les très bonnes performances (voir [CCZ97]) du compilateur SmallEiffel.

## 4.9 Conclusions

Bien que l'aliasing présente des inconvénients importants et soit souvent considéré (au moins) avec méfiance, il offre également un certain nombre d'avantages qui peuvent être utiles aux développeurs.

Nous avons souligné ces avantages dans le domaine de la compilation et décrit comment nous avons mise en oeuvre l'aliasing dans SmallEiffel, The GNU Eiffel Compiler. Nous avons montré comment des choix de conception raisonnables nous permettaient de faire cela d'une façon plus sûre et comment comment certains idiomes du langage Eiffel rendaient un aliasing plus sûr et facile à implanter. Dans ce cadre, le *design by contract*, implanté à l'aide des assertions d'Eiffel, ainsi que les règles d'exportation sélectives, se sont révélés des outils puissants et utiles. Le modèle de conception Singleton, pour lequel nous avons proposé une implantation Eiffel novatrice, a également été crucial.

Les résultats en termes d'utilisation mémoire et de temps d'exécution ont été précisément caractérisés et se sont avérés très positifs. En utilisant l'aliasing sur les seuls objets de type STRING, nous avons pu réduire la taille mémoire de ces objets d'un facteur 20. Sur l'ensemble de la taille mémoire du programme, ceci revient à une diminution de plus de 14 %. Similairement, la vitesse d'exécution a été améliorée de plus de 10 %.

Les très bonnes performances atteintes par SmallEiffel en termes de vitesse et de mémoire semblent ainsi venir en bonne partie de l'aliasing. Nous sommes donc convaincus que l'aliasing est une *technique* tout à fait *utile* et fondamentale pour la programmation à objets. Considérer l'aliasing comme un phénomène nuisible serait à notre avis une erreur. Nous pensons également que les informations que nous fournissons dans ce chapitre peuvent être très utiles non seulement aux développeurs de compilateurs mais aussi, de façon plus générale, à tout développement avec les langages à objets. Des travaux complémentaires sont néanmoins nécessaires pour pouvoir plus précisément évaluer l'impact de l'aliasing sur différents types de programmes, liés à la compilation ou non. Enfin, pour avoir dû développer des techniques d'usage de l'aliasing dans un langage ne le supportant pas, nous estimons absolument nécessaire que les langages à objets intègrent eux-mêmes dès leur conception la notion d'aliasing.

# Chapitre 5

# Résumés de mes publications

"Quand le marquis le vit venir, il point encontre lui, il le frappe au premier coup en l'œil et il l'abat mort de ce coup; il frappe à dextre et à senestre, et lui et sa gent, et ils en occisent beaucoup. Quand ceux-là virent que leur seigneur était mort, ils commencent à se déconfire et ils tournent en fuite."

Robert De Cléry [La conquête de Constantinople]

Ce chapitre contient la liste exhaustive de mes publications, classées par catégorie, chacune accompagnée de son résumé et parfois même d'un commentaire approprié.

## 5.1 Livres

### [MNC+89] Les langages à objets

G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre
Livre paru chez InterEdition dans la collection IIA, Paris, 1989.
Cet ouvrage à vocation encyclopédique, passe en revue les différentes facettes de la programmation objet et effectue un vaste tour d'horizon sur les langages à objets (550 pages, version Française diffusée à plus de 6000 exemplaires).
LORIA 89-R-126.

**Résumé :** Cet ouvrage est le fruit d'un travail d'enseignement et de recherche commencé en 1985 au CRIN (Centre de Recherches en Informatique de Nancy). Il s'adresse à tous les membres de la communauté informatique, des néophytes aux spécialistes. Il passe en revue les différentes facettes de la programmation objet et effectue un vaste tour d'horizon des langages à objets :

- présentation didactique des concepts de base de la programmation objet,
- historique et classification des langages à objets : langages de classes, langages de frames, langages d'acteurs et langages hybrides,
- description comparative de nombreux langages à objets : Smalltalk, Objective-C, Flavors, CLOS, Simula, C++, Eiffel, KRL, etc.,
- étude comparative de la programmation avec un langage de classes (Smalltalk) et avec un langage plus classique (Ada).

Il fournit en outre le source commenté de l'implantation de trois langages à objets.

Cet ouvrage de référence, grâce aux nombreux exemples qu'il contient, constitue également un support de cours idéal pour les enseignants désireux d'initier leurs étudiants aux langages à objets.

**Remerciements :** J'en profite pour remercier ici tous les autres auteurs de ce livre, Gérald Masini, Amédeo Napoli, Daniel Léonard, Karl Tombre ainsi que Yvan Noiret membre de l'équipe de rédaction durant les premières années[19]. Jamais je n'aurais pu imaginer la masse de travail que représente la préparation d'un tel ouvrage.

### [MNC+91b] Object-Oriented Languages

G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre
Academic Press, 1991, New York.
Ce livre est la traduction en Anglais du livre [MNC+89]. La traduction a été demandée par l'éditeur Anglais suite aux bons résultats de la version française (ventes de plus de 5000 exemplaires lors des trois premières années). Nous avons entièrement supervisé la traduction.

**Note 1 :** Il faut bien avouer que mon niveau d'anglais de l'époque ne m'a pas permis de participer autant que je l'aurais souhaité à cette traduction. Je pense que les autres auteurs de cet ouvrage seront d'accord avec moi pour féliciter tout particulièrement Gérald Masini pour le travail fourni.

**Note 2 :** Alors que l'éditeur Français nous a toujours envoyé chaque année[20] le nombre d'exemplaires vendus au cours de l'année écoulée, curieusement, nous n'avons jamais rien reçu de la part de l'éditeur New-Yorkais. Inutile donc de dire que nous n'avons jamais été rétribué pour cette édition en langue Anglaise.

### [MNC+91a] Linguaggi per la programmazione a ogetti

G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre
Gruppo Editoriale Jackson,1991, Milano.
Traduction italienne du livre [MNC+89].

**Note 1 :** Tout comme pour la version Anglaise, nous n'avons jamais eu aucun retour en ce qui concerne le nombre d'exemplaires produits en Italien[21].

**Note 2 :** Selon mon nouveau collègue de bureau, Luigi Liquori, Italien d'origine, cette traduction semble être de bien piètre qualité, ce qui confirme également l'avis d'Amédéo Napoli, qui comme son nom l'indique est également originaire de ces terres ensoleillées.

19. Si ma mémoire est bonne, Yvan a participé aux réunions hebdomadaires de travail pendant les 3 premières années. Entre la toute première réunion de travail et le moment où le manuscrit final a été envoyé, environ 5 années se sont écoulées ! La publication [CMN+86] marque le véritable début du travail de rédaction de l'ouvrage.

20. Nous avons reçu il y a quelque mois une dernière lettre de l'éditeur Français nous indiquant que les derniers exemplaires en stock seraient passés au pilon !

21. Bien évidemment, aucun retour sonnant et trébuchant non plus pour cette version Italienne. Pour conclure l'aspect financier concernant cet ouvrage, seule la version française a fait l'objet d'une rétribution. Lors de la signature du contrat d'édition, 10000 Francs ont été donné au CRIN en dédommagement de l'utilisation des moyens informatiques pour la production des originaux à tirer. Environ 6500 exemplaires tirés, à raison d'environ 25 Francs de droits d'auteurs par exemplaire, sachant qu'il y avait cinq auteurs et que chaque auteur a touché une part égale, sachant qu'il faut déclarer cette somme dans la colonne des droits d'auteurs de votre déclaration d'impôts, je vous laisse faire le calcul !

## 5.2 Publications internationales avec comité de lecture éditées dans une revue

### [ZC00a] Coping with aliasing in the GNU Eiffel Compiler implementation

O. Zendra, D. Colnet
Software Practice and Experience (*SP&E*), Volume??, pages??-??
A paraître
LORIA ??-R-??

**Abstract :** This paper report our experience about *aliasing* usage in the implementation of SmallEiffel, the GNU Eiffel Compiler. The SmallEiffel compiler source code makes intensive use of aliasing in order to achieve very good performance both in terms of memory and execution time. Thanks to the design by contract capabilities of the Eiffel language, aliasing can be handled very safely. The singleton pattern appears to be crucial in implementing alias provider objects. We propose an efficient implementation of this pattern made easy by some Eiffel idioms. This technique, which requires no language modification, is very appropriate for compilation, but can also be applied to a wide range of applications.

**Note :** Jean-Marc Jézéquel m'avait demandé de jeter un coup d'œil sur son dernier livre en préparation, un livre sur les *design pattern* et la programmation par contrats, pour relecture avant de l'envoyer au tirage. C'est en lisant le manuscrit non définitif que j'ai eu l'idée d'améliorer la traduction du *Singleton pattern* en utilisant l'invariant de la classe du *Singleton* lui-même. C'est cette idée qui est à l'origine de la présente publication [ZC00a]. C'est cette idée que Jean-Marc a pris en compte dans sa version définitive [JTM99]. Enfin, c'est cette idée qui a mis en évidence un bug dans SmallEiffel car l'invariant du *Singleton* entraînait le compilateur dans une récursivité infinie !

### [ZCC97] Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler.

O. Zendra, D. Colnet, S. Collin
12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)
SIGPLAN Notices, Volume 32, Number 10, pages 125-141
October 1997, Atlanta
LORIA 97-R-140

**Abstract :** SmallEiffel is an Eiffel compiler which uses a fast simple type inference mechanism to remove most late binding calls, replacing them by static bindings. Starting from the system's entry point, it compiles only statically living code, which saves compiling and then removing dead code. As the whole system is analyzed at compile time, multiple inheritance and genericity do not cause any overhead.

SmallEiffel features a coding scheme which eliminates the need for virtual function tables. Dynamic dispatch is implemented without any array access but uses a simple static binary branch code. We show that this implementation makes it possible to use modern hardware very efficiently. It also allows us to inline more calls even when dynamic dispatch is required. Some more dispatch sites are removed after the type inference algorithm has been performed, if the different branches of a dispatch site lead to the same code.

The advantage of this approach is that it greatly speeds up execution time and considerably decreases the amount of generated code.

**Note :** OOPSLA est vraiment la première conférence dans le domaine des objets, avec en moyenne mille à deux mille visiteurs, son niveau de sélection extrêmement difficile et dont les actes sont publiés dans une revue ACM. Inutile de dire que cela fait vraiment très peur de présenter son travail à OOPSLA ..., de plus comme je présentais mon travail en fin de séjour, je n'ai pas vraiment réussi à profiter sereinement du joli paysage des alentours d'Atlanta. Vraiment inoubliable !

### [CC94] Syntactic Analysis of Technical Drawing Dimensions

S. Collin, D. Colnet
International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)
Octobre 1994, Volume 8 Number 5, pages 1131-1148
LORIA 94-R-036.
Ce papier décrit le système d'extraction et d'interprétation de la cotation des dessins techniques ReDraw, que nous avons développé en Smalltalk.

**Abstract :** The transfer of technical drawings from paper to CAD software data-bases and the construction of a 3D model of an object from several 2D views drawn on a draft, are topical problems. There are several different types of data in an industrial drawing, such as graphics, nomenclature and dimensioning. Dimensioning gives the measurement of the parts with more precision than direct scale measurements on the drawing. This article deals with the problem of dimension extraction and interpretation.

We present a syntactic approach to technical drawing dimensions analysis. A specific grammar is used to describe dimensions of drawings. This grammar can be graphically designed by combining different graphic primitives. The algorithm used for analysis can start at different points of the grammar. The analysis proceeds bottom-up and top-down according to previously obtained results.

**Note :** J'ai participé à l'implantation de ReDraw, logiciel complètement développé en Smalltalk et capable de repérer et de classifier les cotations sur un dessin technique. L'expressivité de Smalltalk est véritablement appréciable pour ce genre de projet qui fut pour moi l'occasion de mettre en pratique le célèbre principe MVC, *Model View Controler*, déjà étudié dans le cadre de [MNC+89]. Par ailleurs, en plus d'éviter les problèmes liés à la surcharge, la notation à mot clefs de Smalltalk se révèle particulièrement lisible. Depuis peu, j'ai redécouvert Smalltalk avec plaisir grâce à *squeak*, un système Smalltalk complètement auto-compilé (*bootstrapped*) et maintenu par *Walt Disney Imagineering*. Les dessins animés Disney soutiennent ce projet (`http://squeak.cs.uiuc.edu`) dans le but d'améliorer l'automatisation de la réalisation des dessins animés. La démonstration de l'orchestre interactif de squeak est très impressionnante.

## 5.3 Publications internationales avec comité de lecture

### [CL00a] Match-O, a Dialect of Eiffel with Match-Types

D. Colnet and L. Liquori
35th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'2000)
Sydney, Australia, Novembre 2000,

IEEE Computer Society PR00462, pages 1-12
LORIA A00-R-270.

**Abstract :** It is well known that the Eiffel language allows covariant redefinition. Regardless of system-level validity rules, Eiffel is not type safe. In this paper, we present a dialect of Eiffel called Match-O, which prohibits covariant redefinition. We introduce a new kind of types, the *match-types*, inspired by the papers of Kim Bruce. The scope of this project is many-fold: – allowing binary methods; – keeping sound "mytype method specialization", i.e. anchored type using `Current`; – allowing subtyping in all other sound cases. We claim that match-types can be added in the Eiffel type system to eliminate type unsoundness without blocking many interesting Eiffel programs (e.g. the ones with "binary methods"). We have implemented a compiler for Match-O and we have experimented our dialect on a large system using the original source code of SmallEiffel itself.

## [ZC00b] Vers un usage plus sûr de l'aliasing avec Eiffel

O. Zendra, D. Colnet
Actes du Colloque Langages et Modèles à Objets, LMO'00
Mont Saint-Hilaire, Québec, janvier 2000
HERMES, pages 183-194
LORIA A00-R-022.

**Abstract :** Le code source du compilateur SmallEiffel fait un usage intensif de l'aliasing afin d'atteindre les meilleures performances, tant en termes de mémoire que de vitesse d'exécution. Cette technique semble très appropriée à la compilation mais peut aussi s'appliquer à une large gamme d'applications. Grâce aux capacités de programmation par contrat du langage Eiffel, l'aliasing peut être géré d'une façon assez sûre. Le modèle de conception singleton se révèle également crucial pour l'implantation d'objets fournisseurs d'alias. Nous présentons ici une implantation efficace de ce modèle rendue possible par certains idiomes d'Eiffel.

**Note :** Version française de [ZC00a]. L'*aliasing* consiste à utiliser à plusieurs endroits ou dans plusieurs contextes, un pointeur vers un seul et même objet. L'objectif de cette technique est soit d'assurer une cohérence de l'information dans un réseau d'objets, soit d'optimiser la place mémoire ou le temps d'exécution. A lire si l'on souhaite comprendre pourquoi SmallEiffel va aussi vite mais aussi pour avoir quelques idées pour optimiser sa propre application.

## [ZC99a] Adding external iterators to an existing Eiffel class library

O. Zendra, D. Colnet
32th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99)
Melbourne, Australia, Novembre 1999,
IEEE Computer Society PR00462, pages 188-199
LORIA 99-R-318

**Abstract :** This paper discusses common iteration schemes and highlights the interest of using explicit iterators. The advantages of external iterators are compared to those of internalized iterators.
The integration of an iterator class hierarchy to an existing library without modifying the latter is detailed. This integration brings an extra level of abstraction to the library, which

thus becomes more flexible, more adapted to certain design patterns and hence can be used in a higher-level way. Such an integration is not only possible, but can even be done in an optimized way, taking into account the specific structure of the collection traversed.
A slight extension of existing class libraries can also be implemented that does not cause any compatibility problem and does not break existing code, but allows even further abstraction and makes it easier for the developer to use high-level, optimized, external iterators.

### [CZ99] Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler

D. Colnet, O. Zendra
29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)
Nancy, France, Juin 1999
IEEE Computer Society PR00275, pages 341-359
LORIA 99-R-061.

**Abstract :** The design of the Eiffel language makes it possible to perform global optimizations on Eiffel programs. In this paper, we describe some of the techniques we used in SmallEiffel, The GNU Eiffel Compiler, to generate highly efficient executables for Eiffel programs. Most of these techniques — related to global analysis or not — may also be applied to other object-oriented languages.

### [ZC99b] Towards safer aliasing with the Eiffel language

O. Zendra, D. Colnet
Intercontinental Workshop on Aliasing in Object-Oriented Systems (IWAOOS'99)
June 1999, Lisbonne, Portugal
ECOOP'99 Workshop Reader pages 153–154
Springer Verlag
LORIA 99-R-317

**Abstract :** The SmallEiffel compiler makes intensive use of aliasing in order to achieve very good performance both in terms of memory and execution time. This technique is very appropriate for compilation, but can also be applied to a wide range of applications. Thanks to the design by contract capabilities of the Eiffel language, aliasing can be handled very safely. The singleton pattern appears to be crucial in implementing alias provider objects. We propose an efficient implementation of this pattern made easy by some Eiffel idioms.

### [KLC99] Design of a customizable processor IP

K. Limam, P. Vernel, J.Besse, D. Colnet
International Workshop on IP Based Synthesis and System Design
pages 181-185
December 1999, Grenoble
LORIA 99-R-370

**Abstract :** Since the beginning the users wish electronics more and more powerful, easy to use and increasingly reduced in volume. Thanks to the availability of the FPGA these wishes are closed to be reached. In this paper we presents how a customizable processor IP, The ProMic Virtual Machine, was designed by a very small team and what tools were used.

**Note :** Voir aussi annexe A, page 69 pour plus d'informations sur ProMic.

### [CCZ98a] Compiler Support to Customize the Mark and Sweep Algorithm

D. Colnet, P. Coucaud, O. Zendra
Proceedings of 1st ACM SIGPLAN International Symposium on Memory Management (ISM-M'98)
pages 154-165
October 1998, Vancouver, BC, Canada
LORIA 98-R-233

**Abstract :** Mark and sweep garbage collectors (GC) are classical but still very efficient automatic memory management systems. Although challenged by other kinds of systems, such as copying collectors, mark and sweep collectors remain among the best in terms of performance.

This paper describes our implementation of an efficient mark and sweep garbage collector tailored to each program. Compiler support provides the type information required to statically and automatically generate this customized garbage collector. The segregation of objects by type allows the production of a more efficient GC code. This technique, implemented in SmallEiffel, our compiler for the object-oriented language Eiffel, is applicable to other languages and other garbage collection algorithms, be they distributed or not.

We present the results obtained on programs featuring a variety of programming styles and compare our results to a well-know and high quality garbage collector.

### [CCZ97] Type Inference for Late Binding. The SmallEiffel Compiler

S. Collin, D. Colnet, O. Zendra
Proceedings of Joint Modular Languages Conference
1997, Lintz, Austria
Lecture Notes in Computer Sciences
Springer-Verlag, Volume 1204, pages 67-81
LORIA 97-R-056.

**Abstract :** The SmallEiffel compiler uses a simple type inference mechanism to translate Eiffel source code to C code. The most important aspect in our technique is that many occurrences of late binding are replaced by static binding. Moreover, when dynamic dispatch cannot be removed, inlining is still possible. The advantage of this approach is that it speeds up execution time and decreases considerably the amount of generated code. SmallEiffel compiler source code itself is a large scale benchmark used to show the quality of our results. Obviously, this efficient technique can also be used for class-based languages without dynamic class creation : for example, it is possible for C++ or Java and not possible for Smalltalk.

### [CC92b] Syntactical Analysis of Technical Drawing Dimensions

S. Collin, D. Colnet
Proceedings of Workshop on Syntactical and Structural Pattern Recognition
1992, Bern (Suisse)

LORIA 92-R-???

**Résumé :** Analyse de dessins techniques à l'aide de grammaires graphiques (*plex-grammaire*). Cet article explique comment décrire une grammaire graphique et comment l'analyseur utilise une grammaire graphique pour effectuer la reconnaissance. L'analyseur est capable, lorsqu'il ne trouve pas un motif complet, de localiser un (ou des) fragments de motifs.

## [CC90] Analysis of Dimension in Mechanical Engineering Drawings

S. Collin, D. Colnet
Proceedings of IAPR Workshop on Machine Vision Applications
Tokyo, 1990, pages 105–107
LORIA 90-R-167
Article sur l'analyse et la compréhension automatique de dessins techniques. Application de techniques issues de la programmation par objets.

**Abstract :** Many papers engineering drawings are still in use and there is a demand for reliable means of transforming them into digital information and incorporating them into a CAD (Computer Aided Design) system. To be effiently used, paper engineering drawings dimensions must be analysed allowing for example 3D model interpretation. We describe the two majors parts of our system : vectorization and dimension identification. The first part, vectorization, uses a specific algorithmm to recognize as soon as possible, text and arrows. This is a typical case where a priori knowledge can be introduced as well in low-level processing at higher-level. The second part is achieved in the same way, finding as soon as possible some acceptable dimension. Two differents actors are introduce two implements the dimension identification : the "Grouper", which groups elements, trying to build dimension., and the "Checker" which controls the validity of the set of grouped elements. These two actors are working alternately and share a common view of the drawing. The firt low-level part, is already implements in C. The second, the high-level one is implemented in Objective-C, which seems powerful for handling the complexity of the system. We conclude with remainings problems and further works.

## [CCCD88] GÉPI : un Générateur d'Environnements de Programmation Intégrés

G. Canals, D. Colnet, S. Cruzlara, J.C. Derniame
Software Engineering & Its Application
Toulouse, Décembre 1988, pages 406–421
LORIA 88-R-153

**Résumé :** Nous présentons l'état actuel et les développements envisagés du système GÉPI, un générateur d'environnements de programmation. Un environnement de programmation est construit à partir de la description lexico-syntaxique du langage cible, et de la description des différents outils qui composeront l'environnement.

Les environnements générés possèdent les propriétés suivantes :

- intégration : les différents outils collaborent et coopèrent entre eux durant la construction d'un programme ;

- modularité : la configuration d'un environnement peut évoluer dynamiquement, en cours d'utilisation. On peut enlever les outils inutiles ou rajouter des outils dont on a besoin ;
- interactivité : les environnements construits utilisent une interface utilisateur évoluée : multifenêtrage, menus, souris...

## 5.4 Publications nationales avec comité de lecture

### [CC92a] Grammaire pour la reconnaissance de la cotation dans les dessins techniques

S. Collin, D. Colnet
BIGRE 80 - Actes de CNED'92 ; Traitement de l'écriture et des documents
Nancy, juillet 92, pages 57–64
LORIA 92-R-180

**Résumé :** Ce papier décrit le système d'extraction et d'interprétation de la cotation des dessins techniques que nous avons développé en Smalltalk. Nous exposons en particulier comment il est possible de modéliser l'ensemble de la cotation par une grammaire, et plus particulièrement une *plex-grammaire*. L'étude se décompose en deux grandes parties : d'une part la vectorisation du document, l'extraction des entités de base du dessin (segments), et extraction des éléments de construction de la cotation (flèches et blocs de texte), et d'autre part l'interprétation de la cotation. Le système réalisé montre la puissance du modèle syntaxique dans le cas de formes dont on connaît parfaitement la description. Ce modèle peut intégrer facilement d'autres définitions de cotation ; l'analyseur syntaxique construit ne contenant aucune heuristique sur le contenu de la grammaire.

## 5.5 Thèse

### [Col89] Prototypage, programmation objet : application à l'édition structurelle

D. Colnet
Thèse de l'université de Nancy I, 1989
LORIA 89-T-018

**Résumé :** Cette thèse peut être lue sous deux angles différents selon que l'on s'intéresse à l'édition syntaxique ou à la programmation objet.

Le chapitre 1 effectue un survol des différentes caractéristiques de GEPI, un Générateur d'Environnements de Programmation Intégrés. Le langage de description de langages défini pour ce générateur est présenté au chapitre 2.

Les trois chapitres qui suivent (3,4,5) sont consacrés à la programmation objet et peuvent être lus indépendamment : vocabulaire, notions de base, intérêt de la programmation objet, comparaison avec une approche plus classique et enfin, la description de Leloup, le langage à objets défini pour le projet.

Les chapitres 6 et 7 sont consacrés à l'implantation du manipulateur de documents de GÉPI [GEPI88]. L'implantation est décrite en Leloup et peut être vue comme un exercice en grandeur nature d'utilisation d'un langage à objets.

Le dernier chapitre fait le bilan du projet dans son ensemble, aussi bien en ce qui concerne le générateur lui-même qu'en ce qui concerne l'approche utilisée pour son implantation : la programmation objet en Leloup, un langage très proche de Smalltalk.

## 5.6 Articles de vulgarisation scientifique

### [CZC99b] La programmation à objets. Application au langage Eiffel (1).

P. Coucaud, O. Zendra, D. Colnet
Linux Magazine, juillet 1999, No 8, pages 63-66
LORIA 99-R-073.

**Résumé :** Cet article introductif présente les notions principales de la programmation à objets: objets, classes, instanciation, héritage. Nous illustrons notre propos d'exemples réalisés en Eiffel. Parallèlement à l'étude des concepts de ce langage, nous faisons quelques comparaisons avec les autres grands langages à objets: Java, C++ et Smalltalk.

### [CZC99c] La programmation à objets. Application au langage Eiffel (2).

P. Coucaud, O. Zendra, D. Colnet
Linux Magazine, septembre 1999, No 9, pages 52-55
LORIA 99-R-089.

**Résumé :** Après une introduction aux concepts objets dans un premier article [CZC99b], celui-ci en étudie de plus complexes, mais aussi plus puissants, comme l'héritage multiple, la généricité, ainsi que les assertions et leur utilisation dans le cadre de la programmation par contrat.

### [ZCC99a] La programmation à objets. Application au langage Eiffel (3).

O. Zendra, D. Colnet, P. Coucaud
Linux Magazine, octobre 1999, No 10, pages 60-63
LORIA 99-R-126.

**Résumé :** Cette troisième partie de notre introduction à la programmation à objets détaille deux de ses plus fondamentaux concepts: le polymorphisme et la liaison dynamique. Nous finissons également de présenter la syntaxe d'Eiffel, avec les structures de contrôle évoluées.

### [ZCC99b] La programmation à objets. Application au langage Eiffel (4).

O. Zendra, D. Colnet, P. Coucaud
Linux Magazine, novembre 1999, No 11, pages 56-59
LORIA 99-R-228.

**Résumé :** Ce dernier article de notre série sur la programmation à objets permet de mettre en application les concepts que nous avons présentés dans nos précédents articles [CZC99b] [CZC99c] [ZCC99a]. Nous utilisons les éléments du langage Eiffel que nous avons précédemment introduits pour programmer divers exemples de mise en oeuvre des concepts objets

dans les bibliothèques de SmallEiffel, le compilateur GNU Eiffel (http://SmallEiffel.loria.-fr).

### [CZC99a] Gestion mémoire: manuelle ou automatique?

P. Coucaud, O. Zendra, D. Colnet
Magazine *Programmez !*, février 1999, No 7, pages 54–57
LORIA 99-R-021.

**Résumé :** La mémoire disponible sur les machines actuelles tend à augmenter. Les applications en utilisent elles aussi de plus en plus. Ainsi, la mémoire disponible devient rapidement un facteur limitant et il est bon alors d'éviter tout gaspillage afin de permettre aux programmes de fonctionner.
Cet article fait le point sur les systèmes actuels de gestion automatique de la mémoire, et présente celui utilisé dans le compilateur SmallEiffel.

### [ZCC98a] SmallEiffel: l'Eiffel à Très Grande Vitesse

O. Zendra, D. Colnet, P. Coucaud
Magazine *Programmez !*, octobre 1998, No 3, pages 64–67
LORIA 98-R-241.

**Résumé :** Faisant suite à celui publié dans le numéro 2 de Programmez! [CCZ98b], cet article détaille certains aspects du langage Eiffel de son compilateur GNU Eiffel, nommé SmallEiffel. Le puissant mécanisme d'assertions et leur utilisation dans le cadre de la programmation par contrat sont présentés, ainsi que les capacités que possède SmallEiffel dans ce domaine.

### [CCZ98b] Eiffel: la puissance de la recherche au service de vos programmes

D. Colnet, P. Coucaud, O. Zendra
Magazine *Programmez !*, juillet 1998, No 2, pages 82–83
LORIA 98-R-201.

**Résumé :** Eiffel est un langage objet de haut niveau issu de la recherche. Il permet l'implantation effective de la programmation par contrat, notamment grâce à ses puissants mécanismes d'assertions. SmallEiffel est le compilateur GNU Eiffel reconnu par la FSF (Free Software Foundation). Il a été développé au sein de l'équipe ECOO du laboratoire Loria se situant à Nancy.

**Note :** Magazine vendu en kiosque avec un CD ROM de SmallEiffel inclus.

### [ZCC98b] With SmallEiffel, The GNU Eiffel Compiler, Eiffel joins the Free Software community.

O. Zendra, D. Colnet, P. Coucaud
*GNU Bulletin*, 1998, No 25

### [CMN+89] La montée des langages à objets

D. Colnet, G. Masini, A. Napoli, K. Tombre, D. Léonard
Le Monde Informatique
Numéro 376, 1989
LORIA 89-R-128
Article de vulgarisation sur les langages à objets

**Résumé :** Cet article de vulgarisation est consacré à la montée des langages à objets dans le domaine industriel. Plus particulièrement destiné aux « décideurs », il présente, sans être trop technique, le vocabulaire courant en programmation objet, une ébauche de classification-comparaison des différents langages à objets ainsi que divers aspects liés à leurs utilisations.

**Note :** Je me rappelle qu'il avait fallu batailler avec les relecteurs de notre article qui voulaient, en bons journalistes du *Monde Informatique*, nous faire utiliser tout le vocabulaire médiatico-informatique de l'époque

## 5.7 Edition d'actes

### [DCT89] Actes Les Mardis Objets du CRIN

éditeurs : D. Colnet, G. Masini, A. Napoli, K. Tombre
LORIA 89-R-072.
Ce fascicule regroupe les textes de la plupart des conférences sur les langages à objets que nous avons organisées, chaque mardi au CRIN, du 20 Octobre 1987 au 31 Mai 1988.

**Résumé :** Les textes sont divisés en cinq parties, indépendamment de leur ordre chronologique. La première est consacrée aux exposés portant sur des aspects théoriques liés à la programmation objet, avec des textes de Roland Ducournau et Michel Habib (héritage multiple), Yves Caseau (LORE), Alain Quéré (le polymorphisme en ML) et Jacques Ferber (réflexivité). La deuxième partie regroupe les interventions de Bernard Paul Serpette (implantation de l'interprète Le-Lisp), Yves Potard et Jacques Duthen (le langage Le-Loup) et Nicolas Graubé et Pierre Cointe (CLOS), toutes liées à LISP. La troisième partie est consacrée aux langages de représentation des connaissances, avec les présentations de Kris van Marcke (KRS), François Rechenmann (Shirka), Anne Adam-Nicolle (Airelle), Philippe Dugerdil (OBJLOG) et Zeina Zein (Glances). La quatrième partie est dédiée aux acteurs et comprend deux textes de Jean-Pierre Briot (Actalk) et Oscar Nierstrasz (Hybrid). Enfin, la dernière partie regroupe les présentations de Thierry Gueguen (programmation avec un langage de classes), Marie Meysembourg (le système Guide) et Philippe Gautron (le système SOS), traitant de la méthodologie objet et des applications de la programmation.

## 5.8 Rapports techniques et supports d'enseignements

### [CL00b] Match-O, a Statically Safe(?) Dialect of Eiffel.

D. Colnet, L. Liquori
Technical Report, 2000
LORIA A00-R-103

**Abstract :** It is well known that the Eiffel language is *type unsafe*. The main reason of this unsoundness is the possibility to covariantly specialize the type of the arguments in method definition. Starting from the GNU implementation of the Eiffel compiler, SmallEiffel, developed at the LORIA lab in Nancy, we modify the compiler in order to avoid such unsoundness without eliminating statically so many programs (eg. binary methods). This was made possible by an extension of the type system with a new kind of types, the *match types*, inspired by the theoretical works of Kim Bruce [Bru93] [Bru94] [BSvG95] [Bru97] and Ph.D. Thesis of Jonathan Burstein [Bur98].

The new type system, together with *whole system analysis*, can statically achieve both expressivity and safety. What we obtain is, in fact another language that we call, in honor of the type theoretical relation on match types (ie. matching), Match-O (read *"macho"*). The scope of this project is many-fold: – setting the bases of rigorous semantics definition of Match-O; – allowing binary methods blocking unsound subtyping; – allowing mytype method specialization, ie. the typing of `Current`; – allowing subtyping in all sound cases; – bootstrapping the Match-O compiler (the compiler for Match-O is written in Match-O) thus validating our choices for quite a large program of about **80K** lines of code; – validating some type theoretical choices in a Eiffel-like setting; – guaranteeing (with the help of the flag "*-unsafe*" in the Match-O compiler) backward compatibility with all Eiffel programs.

### [CZ00] Targeting the Java Virtual Machine with Genericity, Multiple Inheritance, Assertions and Expanded Types.

D. Colnet O. Zendra
Technical Report, 2000
LORIA A00-R-137

**Abstract :** The Java Virtual Machine is now widely available on many architectures and systems, which makes this virtual machine a very appealing target as an execution environment, not only for Java programs but also for Java extensions as well as other high-level languages.

In this paper, we focus on using the current Java Virtual Machine without modification to support assertions, genericity, multiple inheritance and expanded types. An experiment was realized by implementing a new back-end for our compiler, SmallEiffel, The GNU Eiffel compiler.

### [CSZ99] NICE ELKS 2000 proposal.

D. Colnet, E. Stapf, O. Zendra
Technical Report, 1999
LORIA 99-R-359

**Abstract :** This document presents the proposal we designed for the Non-profit International Consortium for Eiffel (NICE) for the new Eiffel Kernel Libraries Standard 2000 (ELKS 2000). We explain our motivation in designing this new standard and detail the new ELKS 2000 version of the standard, which represents a significant evolution of the ELKS'95 standard. We also keep track of the first revisions of ELKS 2000 thanks to early feedback from the Eiffel community.

### [Col99] Eiffel, SmallEiffel, conception, programmation, objets, contrats, et cetera.

D. Colnet
Polycopié de cours, et sujets d'exercices, 1994-2000

**Résumé :** Ce document d'environ 150 pages correspond au polycopié du cours Eiffel que je donne à l'ESIAL (Ecole Supérieure d'Informatique et Applications) ainsi qu'en Licence d'Informatique. Ce polycopié, mis à jour régulièrement, contient également les transparents du cours ainsi qu'une cinquantaine de sujets d'exercices.

### [CCZ96] Compiling Late Binding in Object-Oriented Languages with a Copy-and-Adapt Technique. Application to the Eiffel Language.

S. Collin, D. Colnet, O. Zendra
Technical Report, 1996
LORIA 96-R-068

**Abstract :** We present in this paper a method for late-binding compilation in object-oriented languages. The most important aspect in our technique is that many occurrences of late binding are replaced by static binding. This is possible after considering the calling context of the routine and the type of the target Current. The advantage of this approach is that it decreases considerably the amount of generated code. Another important feature is that the compilation of Eiffel routines is done only for those which exist at run time. We validated our ideas and finalized our technique by realizing a compiler for the Eiffel language. This compiler, which is completely written in Eiffel, allows us to use itself to evaluate the quality of our results.

### [CZC98] Using Type Inference to Customize the Garbage Collector in an Object-Oriented Language. The SmallEiffel Compiler.

D. Colnet, O. Zendra, P. Coucaud
Technical Report 1998
LORIA 98-R-196

**Abstract :** Although many garbage collection algorithms can be found in the literature, we think that the efficiency of a collector is closely linked to the quality of its implementation. This paper is not intended to present a new algorithm for garbage collection, but a way to optimize its implementation. In this paper we show how type inference information can help generating an efficient collector customized for each application. At compile time, SmallEiffel produces a mark-sweep collector tailored to the application. For example, the run through the object graph during the mark phase is hard coded thanks to the knowledge of object structure. In order to demonstrate our very good results, we also present a benchmark suite including typical execution patterns. The whole source code of the compiler itself is used as an additional real-size benchmark. We compare our results with two excellent production-level products. The technique we present is applicable to many kinds of collection algorithms, distributed or not.

### [Col86] De CEYX

D. Colnet
Rapport technique, 1986
LORIA 86-T-035.
Rapport technique d'enseignement et de recherche sur l'utilisation de CEYX, une couche objets au-dessus de LISP.

**Résumé :** CEYX est un ensemble d'outils destinés à faciliter la tâche des programmeurs Le_Lisp :

- description et manipulation de structures : l'équivalent du struct C ou du record PASCAL est disponible sous CEYX ;
- mise en œuvre des mécanismes d'instanciation d'héritage et de transmission de message : on peut adopter un style de programmation à la Smalltalk ;
- documentation automatique des structures définies.

**Note :** Avec Ceyx, je découvre la programmation par objets ! En relisant le résumé qui précède, le lecteur averti constatera que l'on sort déjà du cadre fonctionnel pur.

### [CMN+86] Les Langages Orientés Objets

Rapport technique, 1986
D. Colnet, G. Masini, A. Napoli, Y. Noiret, K. Tombre
LORIA 86-R-077

**Résumé :** Après avoir présenté les principes généraux de la programmation orientée objet, nous étudions en détail SMALLTALK, langage orienté objets par excellence. Nous voyons ensuite sur deux exemples, Objective-C et Ceyx, comment il est possible d'ajouter une couche orientée objets à un langage de programmation "classique", qu'il soit compilé (C) ou interprété (Lisp). L'étude est complétée par les représentations centrées objets utilisées en intelligence artificielle, qui sont tout à la fois ancêtres et héritières des langages orientés objets. Nous terminons par un rapide bilan sur les particularités des langages orientés objets et les apports qu'on peut en attendre dans les années à venir.

# Chapitre 6

# Bilan et perspectives

> “The most exciting phrase to hear in science, the one that heralds new discoveries, is not "*Eureka!*" (I found it!) but "*That's funny ...*"”
>
> Isaac Asimov

La première partie de ce chapitre dresse le bilan de nos travaux (6.1) et la deuxième essaye de dresser quelques perspectives (6.2).

## 6.1 Bilan et apports de nos travaux

Comme on l'aura compris, l'hypothèse du système fermé dont l'impact est résumé au chapitre 2, est essentielle dans les bons résultats du projet SmallEiffel [CCZ97]. Cette hypothèse rend possible une implantation simple, originale et efficace de la liaison dynamique [ZCC97]. L'implantation est basée sur la génération de code de sélection dichotomique qui permet d'éviter la définition systématique de nombreuses fonctions au niveau du code machine.

L'analyse globale du système permet de recenser l'ensemble des types vivants ainsi que de dresser le graphe d'héritage complet du système. L'implantation de la liaison dynamique en présence d'héritage multiple est immédiate. Ceci contraste fortement avec les implantations à base de VFT [ES90].

L'hypothèse du système fermé simplifie la prise en compte de la généricité. Toutes les dérivations sont recensées statiquement et tout se passe comme s'il y avait duplication du code source autant de fois qu'il y a de dérivations différentes [CZ99]. Tout se passe ensuite comme pour les classes ordinaires (non génériques) : on ne considère que le code vivant et on spécialise ce code vivant selon le type du receveur.

Enfin, toujours grâce à cette hypothèse du système fermé, nous sommes également en mesure de produire un ramasse-miettes spécialisé pour chaque application [CCZ98a]. A chaque type vivant est associée une fonction de marquage spécialisée et un allocateur spécialisé. Les objets sont rangés en mémoire selon leur type, un bloc de mémoire donné ne contient que des objets de même type dynamique. A chaque type est associée une fonction de balayage spécialisée.

Nous reprenons dans la suite plus en détails les points que nous considérons comme les plus importants dans le bilan de nos travaux.

### 6.1.1 Eiffel et analyse globale

La définition du langage Eiffel se prête extrêmement bien à l'optimisation globale. Réciproquement, l'optimisation globale donne d'excellents résultats sur Eiffel. La raison principale de cette adéquation entre le langage et la technique d'optimisation réside dans le fait que le langage Eiffel est, somme toute, un langage assez statique. Les possibilités d'introspection sont limitées et permettent uniquement de consulter l'état du système. Eiffel ne permet ni d'ajouter une nouvelle classe lors de l'exécution, ni de modifier dynamiquement les méthodes et/ou les attributs d'une classe.

Dire que le langage Eiffel est un langage *statique* n'en fait pas pour autant un langage faiblement expressif. Il suffit pour s'en convaincre de rappeler quelques un des aspects de ce langage à objets :

- définition générique de classe,
- programmation par contrats (pré-conditions, post-conditions, invariants de classes, invariants d'itérations, etc.),
- définition de types ancrés,
- mécanisme d'exportation permettant un dosage fin des relations entre les objets,
- gestion mémoire automatique,
- héritage multiple,
- mécanisme d'exceptions,
- ...

Conçu pour le génie logiciel, Eiffel fait partie, à mon avis, des rares langages véritablement adaptés à un travail en équipe. Eiffel est a priori un langage généraliste, prévu pour une large gamme d'applications, comme les applications de gestion ou de simulation par exemple. Les optimisations réalisées par SmallEiffel en font aussi un excellent candidat pour les domaines ou le temps de réaction est critique comme les systèmes embarqués, ou les jeux vidéo.

Au sujet des jeux vidéo, nous sommes en contact avec une société Parisienne qui réalise des jeux vidéo[22]. Cette société Parisienne développe des jeux vidéo grand publique, *War Games*, en Eiffel. Selon Cédric Le Dressay, Manager d'Eugen System, le code généré par SmallEiffel est plus compact et plus performant que l'ancien code produit par Borland Delphi (sorte de Pascal Objet sans héritage multiple). Le même constat est fait lorsque qu'il compare le code produit par SmallEiffel avec le code produit (beaucoup plus lentement) par ISE Eiffel, le compilateur Eiffel comercialisé par la société de Betrand Meyer (`http://www.eiffel.com`). Toujours selon Cedric Le Dressay, une des raisons de cette efficacité est liée à notre l'implantation de la liaison dynamique et au fait que cette implantation n'induit aucun surcoût lors d'un usage intensif de l'héritage multiple.

### 6.1.2 Analyse statique globale et prédiction de type

Afin de pouvoir améliorer le processus de compilation et le code généré de façon portable et à haut niveau, nous nous sommes tout d'abord intéressés à l'analyse statique de l'ensemble du code source à compiler. Le but global de cette analyse est l'extraction du maximum d'informations du système compilé.

22. Parmi les utilisations commerciales de SmallEiffel, je tiens à remercier ici la société Eugen System pour sa récente donation de 5000 Francs à notre projet. Je remercie également Cédric Le Dressay pour les jeux vidéo, compilés avec SmallEiffel, qu'il m'a promis pour Noël 2000.

Une des premières étapes conceptuelles de cette analyse statique est l'exécution d'un algorithme de prédiction de type. Alors que beaucoup de travaux de recherche ont porté sur l'inférence de type avec d'intéressants résultats, les compilateurs couramment utilisés en production ne semblent guère s'appuyer sur les avantages de puissants algorithmes de prédiction de type. En effet, les implantations de tels algorithmes ne sont souvent pas assez rapides pour être utilisées dans le cadre du développement incrémental. La prédiction de type a néanmoins été incorporée dans nos travaux (section 2.2), afin de rassembler des informations permettant d'optimiser tant le processus de compilation que le code généré.

De nombreux algorithmes d'inférence de type existent dans la littérature, avec des résultats variables selon la précision des informations qu'ils donnent et le coût qu'ils représentent pour le processus de compilation. L'algorithme qui a fait l'objet de nos travaux se limite à une analyse du programme source purement statique, *context-sensitive*, mais *flow insensitive*. Il n'est donc pas optimal en terme de complétude de l'information récupérée, même s'il en est proche. En revanche, il est très peu coûteux et ne ralentit quasiment pas la compilation. Ceci représente un bon compromis qui en permet l'incorporation dans les compilateurs utilisés en production logicielle.

Il faut également noter que, en analysant l'ensemble du système compilé, cet algorithme peut permettre de déceler un certain nombre d'erreurs plus sûrement et plus tôt, facilitant ainsi le travail du développeur d'applications. Ceci constitue la première implantation existante (même partielle) du *system-level validity checking* (vérification de cohérence globale) du langage Eiffel [Mey92].

### 6.1.3 Compilation sur nécessité

Les informations de typage et d'instanciations d'objets rassemblées par l'analyse statique globale permettent entre autres de connaître le code qui est vivant, c'est à dire potentiellement exécuté.

Il est selon nous important de prendre en compte ce type d'information *dès les premières étapes de la compilation*. En effet, le code mort, qui ne pourra jamais être exécuté, n'a pas besoin d'être analysé complètement, et peut ne pas donner lieu à génération de code. Il est ainsi possible d'alléger de manière non négligeable la tâche du compilateur, en diminuant la quantité de code à compiler réellement. Ceci permet donc de dégager des ressources, notamment en temps d'exécution, qui peuvent être affectées à des phases ultérieures de la compilation (optimisation du code généré) ou qui servent simplement à améliorer les temps de réponse du processus de compilation lui-même. Ceci contraste avec les compilateurs commerciaux, où il arrive de voir une étape supplémentaire (et donc coûteuse en temps) à la fin de la compilation, dite de “suppression de code mort”.

Notre compilation sur nécessité permet donc en général un gain important en vitesse d'exécution de SmallEiffel, d'où un gain en temps de développement total pour l'utilisateur du compilateur.

### 6.1.4 Spécialisation du code généré

Nous considérons comme absolument primordial le fait que des informations collectées lors de l'analyse statique, puissent servir à générer un code compilé qui soit le plus optimisé possible.

Une méthode qui permet à ces informations et cette optimisation de s'exprimer au mieux est la duplication suivie d'une spécialisation du code généré (section 2.4). Très brièvement, cette technique consiste, pour toute primitive présente dans une classe, à recopier un exemplaire de la primitive dans chacune des sous-classes de celle où elle est définie. Les versions recopiées dans les sous-classes ne sont pas identiques à la version originale, mais elles sont au contraire spécialisées, c'est à dire que leur code est adapté et optimisé par rapport au contexte propre à chacune des sous-classes. Ainsi, au lieu d'avoir une version commune complexe, traitant tous les cas possibles, avec de nombreux branchements conditionnels, la duplication-spécialisation génère de nombreux clones simples, adaptés chacun à un sous-ensemble de cas, ayant un code très linéaire. De même, mais en général dans une moindre mesure, les structures de données peuvent être spécialisées en fonction du contexte calculé par l'analyse statique. Ces versions dupliquées et spécialisées semblent être très prédictibles et exécutées très efficacement par les microprocesseurs actuels.

Ces optimisations de haut niveau, et donc portables, permettent ainsi d'améliorer sensiblement les performances du système compilé, notamment en temps d'exécution. La duplication-spécialisation a également un impact très positif, sur le code généré car, en simplifiant le code des primitives, elle facilite grandement un certain nombre d'optimisations classiques de plus bas niveau, telles que les évaluations partielles par calcul statique des expressions et les *inlinings*, comme nous l'avons montré dans nos travaux (voir en particulier les sections 2.2 et 2.3).

Ces optimisations portent non pas sur les points faciles à optimiser mais sur ceux qui sont liés aux concepts objets et ont donc un fort impact. Elles éliminent ainsi le surcoût potentiel dû à l'utilisation de ces concepts, ce qui permet réellement aux développeurs de se concentrer sur une programmation à objets de qualité.

Il est important de souligner encore une fois qu'étant extraites de la sémantique du système compilé, les optimisations de haut niveau mentionnées ci-dessus ne dépendent pas de l'architecture cible.

Notons enfin que nous avons appliqué cette technique d'optimisation par spécialisation non seulement au code du programme compilé, mais également au code du ramasse-miettes automatiquement généré par SmallEiffel (section 2.7). Là encore, les résultats obtenus ont été particulièrement positifs [CCZ98a].

### 6.1.5 Une technique novatrice d'implantation de la liaison dynamique

Nos travaux nous ont amenés à une réflexion approfondie sur la liaison dynamique [ZCC97] [Zen00], résultant en la conception d'une technique d'implantation nouvelle et performante résumée à la section 2.3.

Cette technique consiste, pour chaque site d'appel de méthode polymorphique, à générer une séquence de code qui remplace la table d'indirection classique par un arbre de branchement binaire spécialisé, dont les branches représentent les différents types dynamiques possibles pour le receveur à ce site d'appel. Grâce aux informations contextuelles fournies par l'analyse globale et l'algorithme de prédiction de type, les types dynamiques réellement possibles à chaque site sont connus et limités, ce qui permet d'avoir des arbres de branchement en général très réduits. Le cheminement dans ces arbres de branchement est déterminé en examinant l'identifiant de type dynamique codé dans l'objet receveur.

Ainsi, au lieu d'avoir un même code de liaison dynamique à chaque site d'appel polymorphique, comme dans les méthodes classiquement utilisées de nos jours, on a avec la méthode que nous avons développée des sites d'appels spécialisés, dont le code n'est capable de traiter que la liaison dynamique propre au site considéré, et cela de façon très efficace.

Il faut noter que tout le processus de liaison dynamique se trouve ainsi dans le code de l'application et non pas dans ses données (mis à part le type dynamique associé au receveur), ce qui garantit de bonnes propriétés de localité, permettant d'obtenir des performances accrues.

Les mesures de performances effectuées et présentées dans ([Zen00] section 2.7 et [CCZ97]) montrent que cette nouvelle méthode est très nettement supérieure aux méthodes classiques à base de VFT lorsque le nombre de types dynamiques possibles est faible, ce qui représente la majorité des cas. Dans le cas où de nombreux types dynamiques sont possibles (sites *mégamorphiques*, rares), notre méthode semble globalement équivalente aux méthodes à base de VFT. Cette nouvelle technique apparaît donc comme relativement simple à mettre en oeuvre, très efficace et supportant bien la "montée en charge".

Ici encore, notre implantation et notre optimisation de la liaison dynamique permet aux développeurs de s'appuyer sans réserve sur ce mécanisme, sans crainte d'aucun surcoût gênant, ce qui facilite une bonne programmation à objets.

### 6.1.6 Un mode d'emploi pour l'analyse syntaxique

Le chapitre 3 décrit l'implantation de l'analyseur syntaxique et de l'analyseur lexical. Nous montrons comment il est possible d'intégrer les deux analyseurs dans la même phase. L'analyseur lexical travaille toujours à l'aide d'une information contextuelle, fournie par l'analyseur syntaxique. L'information contextuelle permet de simplifier l'écriture des fonctions de niveau lexical tout en assurant de très bonnes performances (section 3.7).

La plupart des constructions syntaxiques et lexicales sont implantées par de simples itérations (section 3.4). Les autres constructions syntaxiques sont implantées de manière récursive en analyse descendante (section 3.5). Le fait de ne pas utiliser de tables d'analyse permet de programmer simplement le rattrapage d'erreurs.

Le même analyseur est partagé par tous les outils de la distribution SmallEiffel. L'implantation n'utilise aucun outil extérieur comme lex ou yacc, ce qui, en plus du traitement aisé des rattrapages d'erreurs, permet d'effectuer un repérage précis des unités lexicales. En effet, un repérage précis, en ligne et colonne, est indispensable pour des outils de qualité (précision des messages d'erreurs et implantation de débogueur).

Nous avons également expérimenté avec succès cette technique dans le cadre du projet ProMic, pour l'écriture d'un analyseur syntaxique et lexical du langage C ANSI. En outre, nous avons très récemment appris que l'analyseur syntaxique du compilateur GNU Ada est implanté de façon très similaire (cet aspect est évoqué brièvement dans [CGS95] section 3.2 pages 6-7). Sauf à se plonger dans les arcanes du code source de GNU Ada ou de SmallEiffel, nous n'avons pas trouvé de publication concernant cette approche de l'analyse syntaxique. Le chapitre 3, "Analyse syntaxique et lexicale : un mode d'emploi" représente une première tentative pour combler ce manque de publication sur le sujet. En espérant que notre expérience puisse servir à d'autres.

### 6.1.7 Impact de l'aliasing et programmation par contrats

La version actuelle de SmallEiffel, version `-0.76`, environ 80 000 lignes de codes pour 300 classes, s'auto-compile en 6.5 secondes sur un Pentium cadencé à 550 Mhz. Ces performances proviennent en grande partie de l'usage intensif de l'aliasing dans l'implantation du compilateur. Nous montrons dans le chapitre 4 comment il est possible de mettre en œuvre cette technique, de façon systématique et plus sûre.

L'exemple présenté dans ce chapitre concerne le partage des noms des symboles. Nous montrons comment il est possible d'économiser la mémoire et d'optimiser la comparaison des noms, en réduisant la comparaison de deux noms à une simple comparaison de deux pointeurs. Bien entendu, l'aliasing est applicable à de nombreux autres domaines, interface graphique ou pattern matching par exemple. L'exemple donné dans le cadre de la compilation est facilement transposable à un autre domaine.

Les assertions et les règles d'exportation jouent un rôle essentiel dans la mise en œuvre sécurisée de l'aliasing. L'implantation Eiffel du pattern Singleton utilise la notion d'invariant de classe pour garantir l'unicité de l'objet Singleton.

L'aliasing existe et est connu dans les systèmes à objets (ou même plus classiques) depuis longtemps. Il est généralement considéré comme un phénomène néfaste, puisqu'il favorise diverses erreurs et rend le déverminage plus difficile. De nombreuses recherches ont donc eu pour objectif la suppression ou au mieux la prévention de l'aliasing.

Il existe pourtant peu de travaux quantifiant *l'impact* de l'aliasing sur un système en termes de performances. Comme l'aliasing permet de toute évidence un certain nombre d'optimisations dans le codage d'une application (comparaisons de références à la place de comparaisons de contenu, notamment), nous avons eu l'idée de considérer l'aliasing non pas comme un phénomène, néfaste qui plus est, mais bien comme une technique, potentiellement très utile. Notre démarche, détaillée dans le chapitre 4 de ce mémoire, a donc consisté à prendre le contre-pied des travaux habituels sur l'aliasing et à caractériser l'impact de l'aliasing dans un compilateur écrit en Eiffel, pour comprendre comment mieux l'utiliser.

L'aliasing s'est révélé délicat à singulariser, dans la mesure où il est entrelacé avec l'ensemble du code du compilateur. Il a cependant été possible d'extraire l'aliasing des chaînes de caractères du reste du compilateur. Cet aliasing apparaît comme ayant un impact positif et non négligeable en termes d'accélération de l'exécution du compilateur, ainsi qu'en ce qui concerne la mémoire consommée.

Il a également été possible de dégager certaines règles de codage (*patterns*) qui permettent une utilisation plus sûre de cette technique utile mais potentiellement dangereuse.

### 6.1.8 Application: SmallEiffel, The GNU Eiffel Compiler

Le résultat principal de nos recherches, en sus des publications qu'elles ont permises, a été le développement et la diffusion d'un compilateur pour le langage Eiffel nommé SmallEiffel.

En effet, les buts éminemment pratiques de nos travaux 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. Afin de pouvoir implanter nos idées efficacement, en nous concentrant autant que possible sur les algorithmes eux-mêmes sans perdre de temps à les implanter, nous avons choisi d'utiliser un langage à objets de haut niveau, offrant des constructions puissantes et ayant un fort pouvoir expressif: le langage Eiffel. Comme aucun compilateur Eiffel de qualité suffisante n'était disponible, et encore moins le code source correspondant, nous avons très rapidement décidé de créer *ex nihilo* notre propre compilateur, SmallEiffel, 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`). Notre souci de vulgarisation s'est également manifesté par la publication de nombreux articles techniques [CCZ98b; ZCC98b; ZCC98a; CZC99a; CZC99b; CZC99c; ZCC99a; ZCC99b]. Cette diffusion de nos travaux auprès du public hors secteur académique nous a permis de disposer, en plus de la validation de la communauté scientifique, d'une validation la plus large possible et également la plus "pratique" possible.

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 `-0.76` environ 80 000 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 210 000 lignes d'Eiffel. A part quelques briques de base très localisées, l'ensemble des constituants du compilateur est entièrement écrit en Eiffel : les analyses lexicale, syntaxique et sémantique, la prédiction de type, la génération et l'optimisation de code, etc... La bibliothèque standard distribuée avec SmallEiffel, qui a également été l'objet d'efforts importants de notre part (voir [CSZ99; ZC99a] et `http://SmallEiffel.loria.fr/libraries/classes.html`), est évidemment celle utilisée pour le développement du compilateur lui-même. On peut y vérifier que tout est écrit en Eiffel, même des classes très basiques, comme STRING ou ARRAY.

L'intérêt majeur que le codage "tout Eiffel" présente mérite d'être souligné. En effet, le fait d'avoir très peu de briques de base écrites en C, et donc l'immense majorité du code en Eiffel portable, permet bien évidemment de simplifier considérablement, voire d'éliminer, les problèmes de portabilité. La conception s'en trouve aussi facilitée, puisqu'il est ainsi possible de rester la plupart du temps au niveau d'un langage de haut niveau, Eiffel, dans lequel les algorithmes et concepts s'expriment nettement mieux qu'en C, langage bas niveau que nous considérons plutôt comme un bon macro-assembleur relativement portable. Cela aide également fortement le déverminage, grâce notamment aux assertions pour la conception par contrats intégrées au langage Eiffel, qui diminuent sensiblement le besoin d'un débogueur extérieur. Cela représente aussi une masse de code Eiffel qui va elle-même servir de test et de validation expérimentale pour l'ensemble du compilateur et des algorithmes qui y sont implantés. Enfin, le simple fait de ne pas avoir à développer deux fois, une fois en Eiffel, une fois en C, les mêmes algorithmes et/ou structures de données représente un gain d'efforts de développement qu'il est important de ne pas sous-estimer.

Intégrant les techniques d'optimisation décrites dans ce mémoire, 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++, pour des programmes aux fonctionnalités équivalentes.

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++.

Dans un souci d'uniformité et d'efficacité, les techniques développées précédemment et appliquées au code généré l'ont été également au niveau du ramasse-miettes. Ainsi, de façon relativement originale, SmallEiffel n'intègre pas un même ramasse-miettes à tous les programmes qu'il compile, mais produit automatiquement un système de gestion mémoire adapté à chaque application générée, basé sur un ramasse-miettes à marquage-balayage. Ce dernier est donc différent pour chaque application, étant spécialisé en tenant compte des informations fournies par l'analyse globale. Seules les routines et les structures de données réellement nécessaires à la gestion mémoire des objets du programme compilé sont produites.

Ce ramasse-miettes, portable, a lui aussi fait preuve d'excellentes performances, notamment quand on le compare à un ramasse-miettes beaucoup plus mûr, ayant fait l'objet de beaucoup plus d'optimisations et de réglages que lui et qui a pu être intégré à SmallEiffel à des fins de comparaison.

Enfin, il faut noter que, étant écrit dans un langage portable (Eiffel), générant du code portable (C ANSI ou *bytecode* Java) et n'appliquant que des techniques d'optimisation de haut niveau, SmallEiffel est très facile à porter sur de nouvelles architectures, et ne nécessite dans la plupart des cas qu'une simple recompilation.

Ces qualités montrent donc de façon probante l'efficacité et l'applicabilité 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.

## 6.2 Perspectives à court, moyen et long terme

Les bons résultats obtenus ces dernières années ne doivent bien sûr pas être considérés comme ultimes et ne pouvant être améliorés. Au contraire, ils nous incitent à poursuivre les recherches présentées dans ce mémoire.

A court ou moyen terme, nous souhaiterions donc étudier les extensions possibles aux techniques présentées ci-dessus, notamment en améliorant l'analyse statique et en la couplant avec des techniques d'analyse dynamique. Nous pensons également qu'il peut être intéressant de généraliser nos travaux à des systèmes non figés, en étudiant tout particulièrement l'impact d'une modification ponctuelle sur le système. Des extensions à d'autres langages modernes et couramment utilisés comme Java seraient bien entendu bienvenues. Enfin, afin de maîtriser plus effectivement les techniques d'optimisations, il nous semble judicieux d'étudier et de bien comprendre l'impact des architectures matérielles.

Tout ceci doit se faire en gardant à l'esprit la vision à long terme qui sous-tend le travail présenté dans ce mémoire, à savoir la production de compilateurs plus "intelligents" et le passage, en ce qui concerne les optimisations, du stade actuel un peu "artisanal" à un stade plus scientifique et industriel, où les optimisations sont mieux comprises, maîtrisées et appliquées, avec pour but final l'obtention de programmes toujours plus rapides.

### 6.2.1 Extensions prévues d'Eiffel

La définition du langage Eiffel est, une nouvelle fois, en cours d'évolution. Parmi mes perspectives, j'envisage d'implanter ces nouveautés en tirant profit du principe d'analyse globale.

La nouvelle définition prévoit l'ajout d'un nouveau type de base, *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, 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 en CAML par exemple.

De nouvelles possibilités d'introspection sont également en projet. Ces dernières n'étant que consultatives (ie. combien de méthodes dans une classes, type des arguments d'une méthode, etc.), cela ne devrait pas remettre en cause l'hypothèse du système fermé.

### 6.2.2 Amélioration de l'analyse globale

Bien qu'il donne d'excellents résultats actuellement, 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 — CPA*, [Age95]) proposé par Ole 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.

### 6.2.3 Couplage avec des techniques d'analyse dynamique

Les travaux présentés dans ce mémoire 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'est à dire 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.

### 6.2.4 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 parait donc indispensable d'étudier des solutions permettant d'éviter cette dégradation. Ceci pourrait impliquer de fortes modifications des algorithmes actuellement utilisés dans notre système 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 changeant. L'intégration dans ce cadre de compilation dynamique des techniques présentées dans ce mémoire représente donc pour nous une direction de travail importante.

### 6.2.5 Impact d'une modification

Fort naturellement, la compilation dynamique conduit à s'intéresser au problème de l'impact d'une modification du source disponible (par chargement dynamique de classe par exemple) sur le reste du système, afin de mettre à jour le code compilé.

Plus généralement, ce problème présente également un intérêt considérable même dans le cas de la compilation statique de systèmes figés. En effet, l'incrémentalité des recompilations est l'une des voies possibles pour offrir au développeur un environnement de compilation lui permettant des cycles d'édition-compilation-corrections les plus rapides possibles. Une meilleure compréhension de l'impact d'une modification ponctuelle du source par le développeur est ainsi utile non seulement au compilateur mais également au développeur car elle peut permettre à l'interface utilisateur de l'environnement de mieux mettre en valeur l'ensemble du code affecté, ce qui nous semble constituer une aide au développement intéressante.

### 6.2.6 Généralisation et extension à d'autres langages

Comme nous l'avons indiqué à plusieurs reprises, il nous parait indispensable de ne pas limiter nos recherches à un seul langage. En plus d'Eiffel, qui nous a servi pour implanter nos idées tout au long de ce travail, il serait donc intéressant de pouvoir disposer d'autres implantations. Nous estimons qu'il devrait être assez facile d'appliquer ce travail à d'autres langages de classes statiques sans chargement dynamique de classes, comme C++. Il est par contre probable que le passage nécessaire vers des langages couramment utilisés et plus dynamiques, comme Java ou Self, demande plus de travail. Cependant, cela représenterait également une opportunité pour développer de nouvelles techniques adaptées à ces nouveaux contextes.

Il faut remarquer que les résultats que nous avons obtenus ont porté sur un langage, Eiffel, où la liaison dynamique s'effectue sur le receveur. Il nous semble également intéressant et relativement aisé de s'attacher à les généraliser aux cas de langages offrant des multiméthodes et du polymorphisme paramétrique.

### 6.2.7 Sécurité du typage

Grâce au dynamisme et avec les compétences de Luigi Liquori [LC96] [Liq97] [GHL98] [Liq98], qui a rejoint notre équipe en janvier 2000, nous avons commencé un travail sur les problèmes de typage dans les langages à objets [CL00b] [CL00a].

Si l'on fait abstraction des règles de cohérence globale, *system validity rules* [Mey92] [Mey94], Eiffel n'est pas un langage statiquement sûr. Aucun compilateur Eiffel actuel n'implante ces

```
break_it is
   local
      apple: APPLE;
      peach: PEACH;
      array_of_fruit: ARRAY[FRUIT];
      array_of_peach: ARRAY[PEACH];
   do
      !!apple;
      !!array_of_peach.make(1,1);
      array_of_fruit := array_of_peach;  -- (1)
      array_of_fruit.put(apple,1);       -- (2)
      peach := array_of_peach.item(1);   -- (3)
   end;
```

FIG. 6.1 – *Violation des règles de cohérence globale. Comment affecter une variable de type* APPLE *avec une instance de la classe* PEACH.

règles. Ainsi, un programme accepté par le compilateur peut aboutir, lors de l'exécution, à une incohérence de type.

Par exemple, si l'on considère à nouveau l'arbre d'héritage de la page 11, qui présente la classe abstraite FRUIT et ses deux sous-classes APPLE et PEACH, le programme de la figure 6.1 est un programme accepté par les compilateurs Eiffel. Ce programme est en réalité incorrect puisqu'il affecte un objet de la classe APPLE dans une variable de type PEACH (APPLE n'est pas un sous-type de PEACH). En effet, prise une à une, toutes les instructions de la figure 6.1 respectent les règles de typage du manuel de référence Eiffel. L'instruction `(1)` est valide puisque ARRAY[PEACH] est considéré comme un sous-type de ARRAY[FRUIT]. Statiquement, l'instruction `(2)` est valide puisque qu'il est correct de mettre une APPLE dans un ARRAY[FRUIT]. Dynamiquement, on ne peut pas dire la même chose puisqu'il s'agit de mettre une APPLE dans un ARRAY[PEACH].

Enfin, l'instruction `(3)` est elle aussi statiquement valide puisqu'elle consiste à ranger dans une variable de type PEACH un objet provenant d'un ARRAY[PEACH]. Dynamiquement, on risque[23] d'affecter une variable de type PEACH avec une APPLE incorrectement stockée dans un ARRAY[PEACH].

Plus généralement, en utilisant le type ANY à la place du type FRUIT, il est possible d'écrire des programmes, acceptés par les compilateurs Eiffel, pouvant potentiellement affecter n'importe quelle instance dans une variable de n'importe quel type.

A court terme, notre objectif consiste à utiliser, encore une fois, la connaissance globale du système pour rejeter statiquement un programme comme celui de la figure 6.1. Néanmoins, les choses ne sont pas si simples car l'application brutale des règles de cohérence globale risquent de rejeter un trop grand nombre de programmes, pourtant corrects, même lors de l'exécution. Un autre problème consiste à être en mesure d'expliquer à l'utilisateur, le plus clairement possible pourquoi son programme est rejeté. Cette tâche n'est pas évidente car différents endroits dans différentes classes peuvent être à l'origine du problème.

L'exemple de la figure 6.1 met en évidence, de façon simple, un grave problème de sécurité susceptible d'effrayer, à juste titre, un bon nombre d'utilisateur d'Eiffel. Dans la pratique le problème semble se produire très rarement. A titre personnel, ce problème ne m'est arrivé qu'une

23. SmallEiffel génère du code supplémentaire qui permet de déclencher une exception dans le cas où un objet du mauvais type devient le receveur d'une opération spécialisée pour un autre type. Comme l'objet de type APPLE n'est jamais le receveur d'un envoi de message dans le programme de la figure 6.1, ce programme incorrect s'exécute sans déclenchement d'exception !

seule fois en neuf ans, sur un total approximatif de 400 000 lignes. Même si SmallEiffel génère du code de vérification qui déclenche dynamiquement une exception, une seule fois même en neuf ans, c'est déjà une fois de trop. En outre, les expérimentations faites dans le cadre de Match-O [CL00b] [CL00a] nous permettent d'espérer un nouveau gain dans le score de prédiction de types.

Pour finir sur cet aspect de sécurité du typage, notons que ce problème n'est pas spécifique à Eiffel puisqu'il est possible d'écrire en Java un programme identique à celui de la figure 6.1. Comme dans le cas d'Eiffel, le compilateur Java accepte statiquement le programme globalement incorrect. Le seul avantage de Java par rapport à Eiffel, est que la définition du langage nous permet de savoir que l'exception exhibant le problème de type se produit lors de l'exécution de l'instruction `(2)`. Lors du déroulement de cette instruction, une exception signale une tentative d'écriture incorrecte dans un tableau (tentative de rangement d'un objet de la classe APPLE dans un emplacement d'un objet de type ARRAY[PEACH]).

### 6.2.8 Micro-contrôleurs synthétisables : le projet ProMic

Le projet ProMic (annexe A) nous a permis de valider dans un autre contexte, celui de la génération de code pour micro-contrôleurs synthétisables, les idées développées dans le cadre du projet SmallEiffel. Comme évoqué précédemment, la même technique d'analyse syntaxique a été utilisée avec succès. Le principe d'analyse globale, ne prenant en compte que le code vivant à lui aussi été utilisé, supprimant ainsi la nécessité d'écrire un éditeur de liens.

En effet, comme dans le cas de programmes Eiffel, nous procédons à l'analyse globale des programmes sources, écrits en C ANSI, dans le but d'optimiser le micro-code généré. En collaboration avec le CRITT Microlor (Centre Régional d'Innovation et de Transfert de Technologie), nous avons défini une machine virtuelle générique [KLC99] permettant de prendre en compte la flexibilité de ces architectures.

Suite aux bon résultats de la convention de recherche ProMic (voir annexe A), une jeune société, Microlor Systems, vient d'être créée en juin dernier. Cette société pourrait envisager, après le rachat du code source du logiciel ProMic, la signature d'une nouvelle convention de recherche.

Un thème de recherche envisageable consisterait par exemple à automatiser le choix des paramètres de la machine virtuelle pour optimiser le code. Un autre thème pourrait concerner l'introduction du parallélisme en utilisant les possibilités de reconfiguration dynamique des nouveaux micro-contrôleurs synthétisables (FPGA).

### 6.2.9 Un nouveau langage à objets ?

Luigi Liquori et moi allons travailler sur la définition d'un nouveau langage à objets. Pour mettre en œuvre ce projet, nous avons effectué une demande de crédit qui vient d'être acceptée par la région Lorraine (subvention de 100 Kf pour 2000).

Nous envisageons de travailler dans un premier temps sur la définition formelle du système de types ainsi que sur la spécification formelle de la sémantique opérationnelle du langage. La définition de notre nouveau langage sera basée sur les objectifs suivants :

- Nous pensons que la sécurité est un aspect primordial. Le système de types doit garantir *statiquement* que le programme est sûr. Par exemple, l'équivalent du message *doesNotUnderstand* de Smalltalk, ou du *core dump* Unix ne doit plus exister.
- Les constructions et les entités de base du langage doivent rester aussi simples que possible et en nombre raisonnable. Comme en Smalltalk par exemple, il devrait être possible de

définir, grâce aux éléments de base, les constructions algorithmiques habituelles (conditionnelles ou itérations par exemple).

– Le langage doit offrir de bonnes propriétés permettant d'envisager une utilisation répartie des moyens de calcul ainsi que la prise en compte d'une forme de parallélisme ou de coopération.
– La persistance des objets ainsi que l'accès réparti à ces objets est également un point qui nous semble important. De ce point de vue, il nous semble intéressant de considérer les programmes eux mêmes comme des objets, dans le but de partager ces programmes sur le réseau.

Pour commencer à travailler sur cet ambitieux projet, j'envisage dans un premier temps de compléter mes connaissances sur le *RhO Calcul* [CKL00], le langage B[24] ainsi que sur les langages à base de prototypes[25].

Dominique COLNET
**Version du 22 Décembre 2000**

24. `http://www.atelierb.societe.com`
25. `http://www.sun.com/research/self`

# Annexe A

# ProMic, Programmation des Micro-contrôleurs

## Cadre administratif

Le projet ProMic a été réalisé dans le cadre d'une convention de recherche en collaboration avec le CRITT Microlor (Centre Régional d'Innovation et de Transfert de Technologie), dans le cadre du projet Jessica :

`http://www.ensem.u-nancy.fr/MICROLOR/`

Ce projet de recherche avec transfert de technologie (convention de recherche No 1 99 F 350 00 51514 12 2) a été réalisé durant la période janvier 1999 à juillet 2000 pour un montant de 300 Kf.

## Objectifs

L'objectif du projet consistait à concevoir et à réaliser un environnement de développement pour micro-contrôleurs synthétisables. Pour des raisons d'étude de marché évidentes, le langage source choisi est C ANSI.

Un micro-contrôleur synthétisable, aussi appelé FPGA de l'anglais *Fexible Programmable Gate Array*, est un composant programmable, comportant un grand nombre de portes logiques. Grossièrement, les portes peuvent être combinées les unes avec les autres afin de réaliser un composant spécialisé pour une tâche donnée. Le composant peut-être reconfiguré plusieurs fois, parfois même dynamiquement, selon le fabricant de composants.

Dans le but de programmer cette nouvelle génération de composant avec un langage de haut niveau (le langage C est un langage de haut niveau quand on parle de la programmation de FPGA), nous avons participé à la définition d'une machine virtuelle. La machine virtuelle PROMIC est comparable à une machine RISC dotée d'un grand nombre de paramètres, fournis au moment même de la configuration du FPGA :

- nombre de registres variable,
- taille d'une adresse RAM variable,
- taille d'une adresse ROM variable,
- hauteur de pile ajustable,
- taille d'un mot machine variable (8 bits, 10 bits, 12bits, ... 16bits),

– ports d'entrée-sortie en nombre variable,
– ...

Conjointement au travail de l'équipe chargée d'implanter la machine virtuelle sur le matériel (le FPGA), nous avons participé à la définition de cette machine virtuelle [KLC99]. En effet, la machine virtuelle doit à la fois être une machine cible adaptée au langage source (le langage C), mais doit aussi être implantée dans le micro-contrôleur de façon efficace.

L'aspect recherche du projet réside dans la nature du code cible. Le jeux d'instructions de la machine virtuelle est composé d'un noyau dur d'instructions du genre de celles que l'on trouve habituellement dans les machines RISC. En plus de ce noyau dur d'instructions, il est possible de spécifier d'autres instructions à prendre en compte dans la génération du code cible.

Compte tenu du paramétrage de la machine virtuelle, le générateur de code doit décider s'il est possible ou non de traduire le programme utilisateur. En cas d'impossibilité, le traducteur doit aider aux choix des différents paramètres. Le choix des paramètres a une incidence directe sur la nature (et donc sur le coût) du composant à utiliser. La taille du code cible, les paramètres choisis et l'ensemble des instructions utilisées détermine le coût du composant FPGA à utiliser (est-il par exemple nécessaire d'utiliser un composant à 40 Francs pièce ou bien un composant à 20 Francs pièce suffit-il ?). Le problème à traiter dans ce projet concerne donc à la fois des aspects d'optimisation et de compactage de code ainsi que des problèmes de résolution de contraintes (choisir les paramètres afin d'utiliser le composant le moins coûteux).

## ProMic

L'environnement de développement ProMic que nous avons développé complètement en Eiffel est un logiciel qui intègre :

– un compilateur d'un sous-ensemble du langage C ANSI pour la machine virtuelle PROMIC,
– un assembleur pour la machine virtuelle PROMIC,
– un simulateur de la machine vituelle PROMIC,
– un générateur de code pour le configurateur de FPGA,
– une débogeur pour les programmes C et pour les programmes écrits en assembleur PROMIC,
– une interface entre C et le langage assembleur PROMIC,
– une interface utilisateur graphique basée sur l'API Windows,
– une interface en ligne de commande disponible sur presque tous les systèmes (Unix, Windows, Macintosh, etc.).

Bien entendu l'expérience acquise en matière de compilation dans le cadre du projet SmallEiffel fut capitale pour ce projet. La même technique d'implantation et les mêmes outils Eiffel sont utilisés. ProMic est entièrement développé en Eiffel et réutilise une part importante du code de SmallEiffel.

## Bilan

Les résultats du projets ProMic sont positifs : la machine virtuelle se révèle très performante (*Best IP prize 1999 au workshop* [IWIPBSSD99]) et l'environnement de développement réalisé est très réactif et correspond aux attentes des utilisateurs (les trois entreprises régionales qui ont participé au financement de ce projet). Une société de commercialisation de ProMic vient de voir le jour en en Avril 2000. Un contrat de cession du code source du logiciel ProMic à cette société est en cours d'élaboration.

## Captures d'écran de l'interface utilisateur

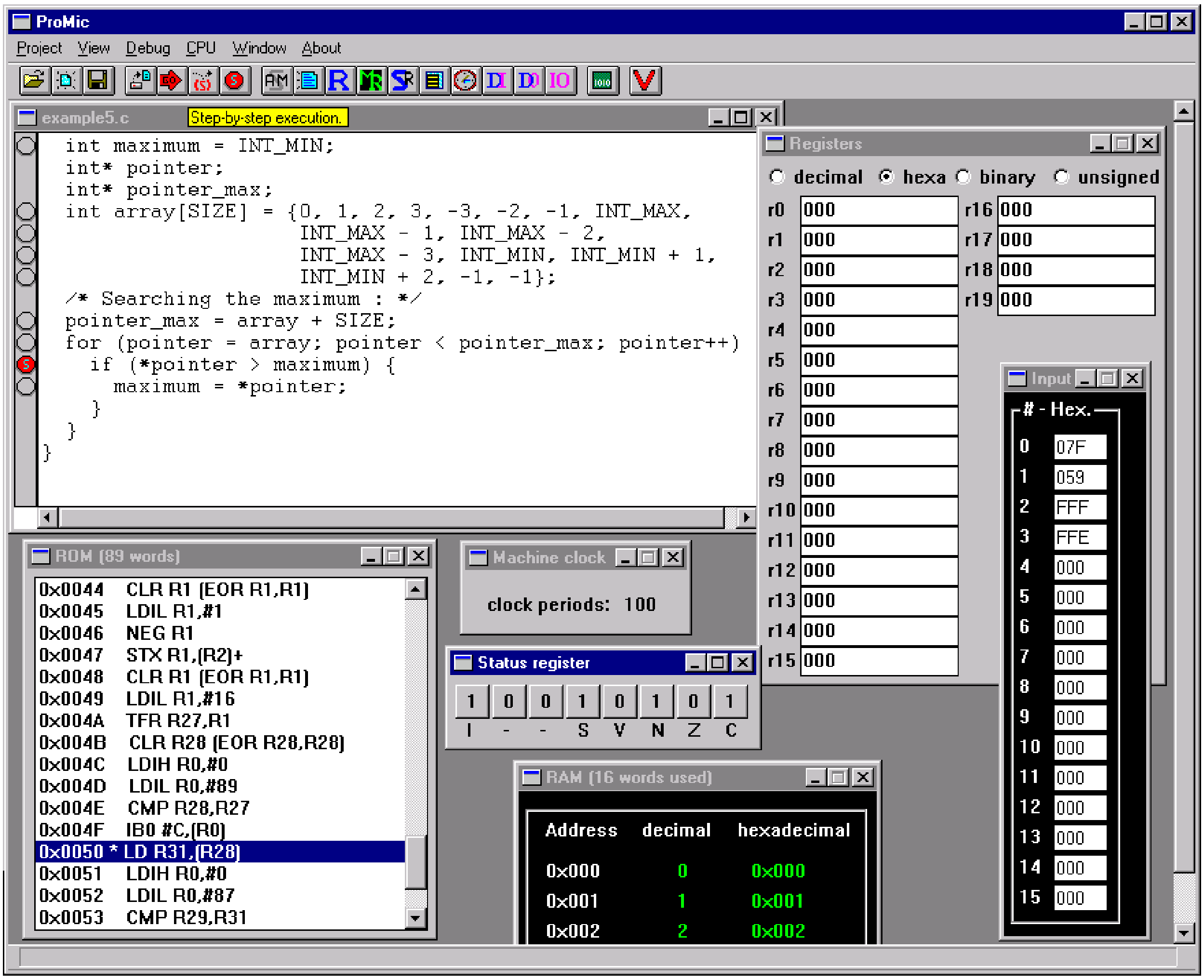


Fig. A.1 – *L'interface utilisateur graphique de ProMic sous Windows. En haut à gauche le texte source C ANSI. En bas à gauche le code pour la machine virtuelle. En haut à droite, la fenêtre des registres recouverte partiellement par la fenêtre Input.*

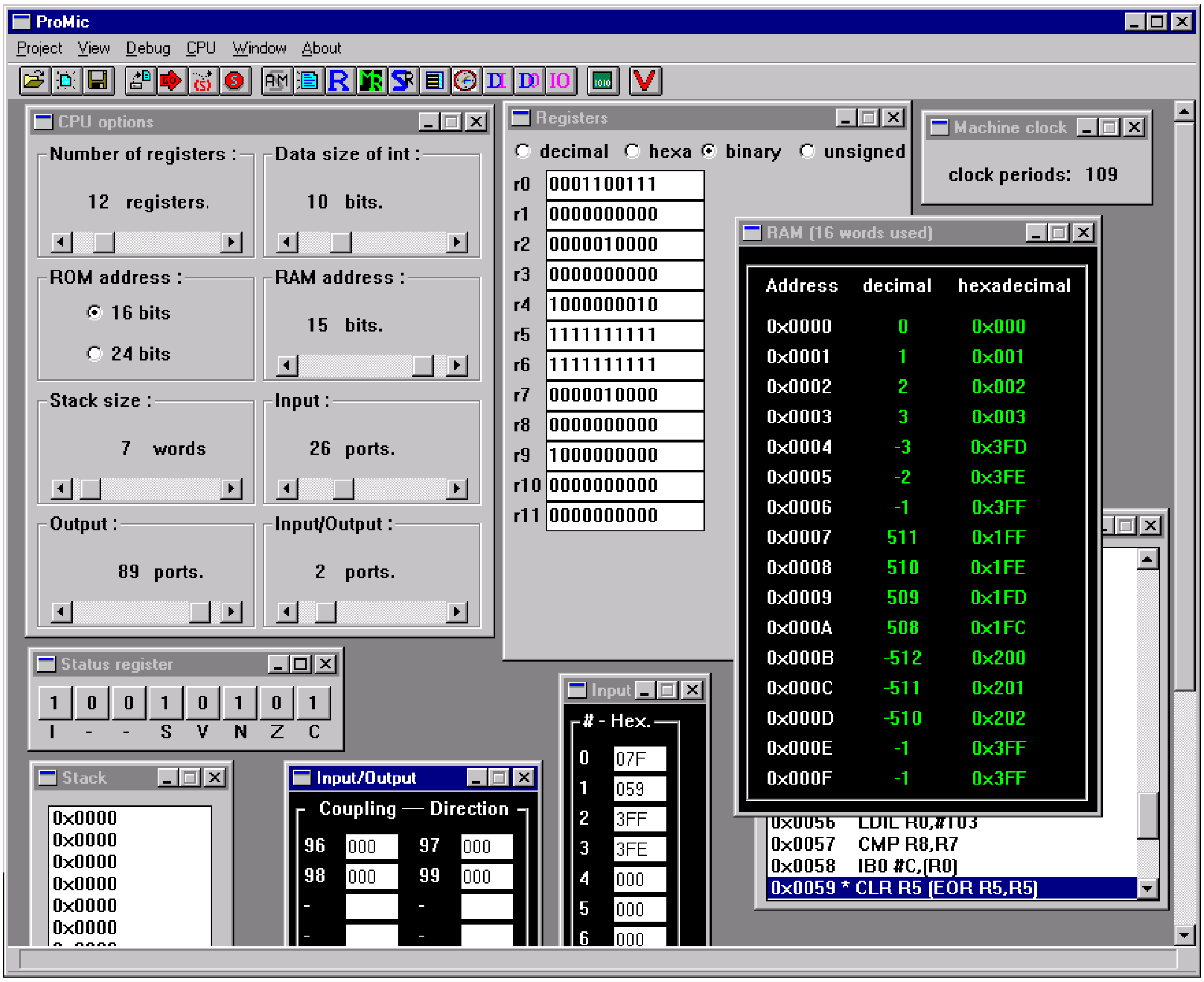

Fig. A.2 – *L'interface utilisateur graphique de ProMic sous Windows. En haut à gauche la fenêtre de paramétrages. En dessous, le registre d'état de la machine. En haut à droite, la vue sur la mémoire RAM.*

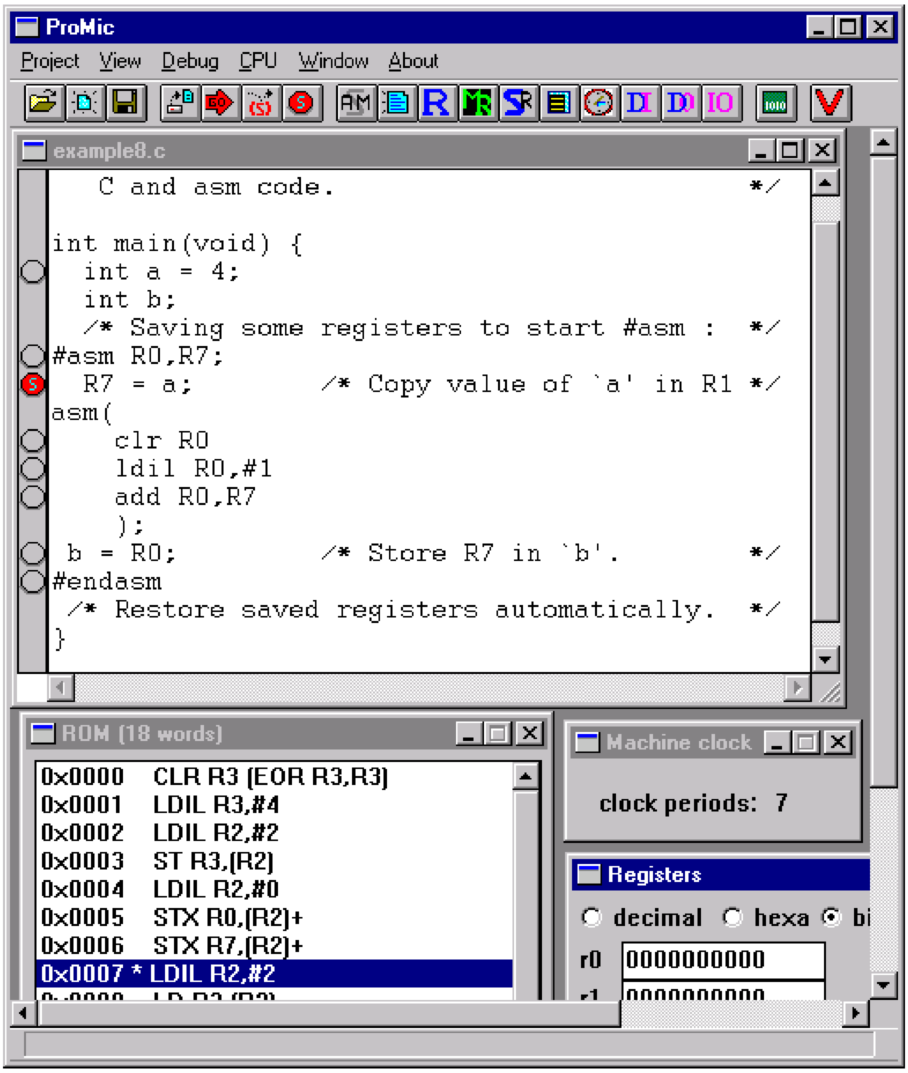


Fig. A.3 – *L'interface utilisateur graphique de ProMic sous Windows. Gros plan sur la fenêtre source qui montre la macro* `#asm #endasm`*, qui permet d'introduire une section de code assembleur au cœur d'un programme C ANSI.*

# Annexe B

# Calcul des symboles directeurs

Étant donné la volatilité des informations sur internet, il m'a semblé utile de mémoriser dans ce document un petit programme écrit en *scheme* qui permet de calculer les symboles directeurs d'une grammaire LL(1) pour l'écriture d'un analyseur syntaxique descendant comme présenté au chapitre 3.

Le programme présenté ci-après est un extrait de `ftp://iecc.com/pub/file/cgram-ll1.gz`, document comprenant également une grammaire ANSI C LL(2) elle-même utilisée pour le projet ProMic. Il convient de remercier ici l'auteur initial de ce programme, *William D. Clinger*, ainsi que *Mohd Hanafiah Abdullah* pour la mise à disposition du document.

## B.1 Représentation de la grammaire

La grammaire est représentée sous la forme d'un programme *scheme*. Plus précisément, une grammaire est représentée par une liste d'entrées, chaque entrée étant elle-même une liste donnant les différentes productions d'un non-terminal donné.

Le première entrée de la grammaire correspond à l'axiome de la grammaire. Le premier élément d'une entrée (`car`) est un non-terminal. Le reste d'une entrée (`cdr`) correspond à la liste des productions pour ce non-terminal. Chaque production est une liste de symboles sachant que le mot vide est représenté par la liste vide `()`, un non-terminal est représenté par un symbol *scheme* et un terminal est représenté par une chaîne de caractères (`"..."`).

La figure B.1 donne un exemple de grammaire notée en *scheme* ainsi que la notation BNF (Backus-Naur Form) habituelle (notation utilisée dans [ASU86]).

```
(define g '(
    (S   ("id" ":=" E "\;")
         ("while" E S)
         ("do" S A "od"))
    (A   ()
         (S A))
    (E   (T E'))
    (E'  () ("+" T E') ("-" T E'))
    (T   (F T'))
    (T'  () ("*" F T') ("/" F T'))
    (F   ("id") ("(" E ")"))))
```

$$
\begin{array}{lll}
\texttt{S} & \rightarrow & \texttt{id := E ;} \\
 & & \mid \texttt{while E S} \\
 & & \mid \texttt{do S A od} \\
\texttt{A} & \rightarrow & \epsilon \\
 & & \mid \texttt{S A} \\
\texttt{E} & \rightarrow & \texttt{T E'} \\
\texttt{E'} & \rightarrow & \epsilon \mid \texttt{+ T E'} \mid \texttt{- T E'} \\
\texttt{T} & \rightarrow & \texttt{F T'} \\
\texttt{T'} & \rightarrow & \epsilon \mid \texttt{* F T'} \mid \texttt{/ F T'} \\
\texttt{F} & \rightarrow & \texttt{id} \mid \texttt{( E )}
\end{array}
$$

FIG. B.1 – *A gauche la notation en scheme d'une grammaire avec son équivalent en notation BNF à droite.*

## B.2 Calcul des symboles directeurs

La fonction `LL1?` retourne vrai ou faux selon que la grammaire `g` que l'on passe en argument respecte ou non la propriété LL(1). Durant son exécution, cette fonction imprime, le cas échéant, quelles sont la (ou les) production(s) n'ayant pas d'ensembles de symboles directeurs disjoints :

```
(define (LL1? g)
  (define (loop dsets)
    (cond ((null? dsets) #t)
          ((disjoint? (cdr (car dsets))) (loop (cdr dsets)))
          (else (display "Failure of LL(1) condition ")
                (write (car dsets))
                (newline)
                (loop (cdr dsets)))))
  (define (disjoint? sets)
    (cond ((null? sets) #t)
          ((null? (car sets)) (disjoint? (cdr sets)))
          ((member-remaining-sets? (caar sets) (cdr sets))
           #f)
          (else (disjoint? (cons (cdr (car sets)) (cdr sets))))))
  (define (member-remaining-sets? x sets)
    (cond ((null? sets) #f)
          ((member x (car sets)) #t)
          (else (member-remaining-sets? x (cdr sets)))))
  (loop (director-sets g)))
```

Pour une grammaire donnée `g`, la fonction `director-sets` calcule l'ensemble des symboles directeurs pour chaque production. Dans un ensemble de symboles directeurs, le symbole de fin de chaîne (ie. fin de fichier du source) est représenté par le symbole scheme $. Une fois la condition LL(1) respectée, c'est le résultat de cette fonction qui permet d'écrire l'analyseur syntaxique selon la méthode présentée au chapitre 3 (utilisation des symboles directeurs en 3.5). Les ensembles de symboles directeurs donnés dans le résultat respectent l'ordre des productions de la grammaire `g` donnée :

```
(define (director-sets g)
  (let ((follows (follow-sets g)))
    (map (lambda (p)
           (let ((lhs (car p))
                 (alternatives (cdr p)))
             (cons lhs
```

```
                    (map (lambda (rhs)
                           (let ((f (first rhs g '())))
                             (if (member "" f)
                                 (union (lookup lhs follows)
                                        (remove "" f))
                                 f)))
                         alternatives))))
          g)))
```

La fonction `first` retourne l'ensemble des *premiers*. Les arguments étant respectivement, `s`, une chaîne de symboles de la grammaire `g` et `recursion` une liste de non-terminaux apparaissant en partie gauche durant le calcul récursif des *premiers*. Dans le résultat, le mot vide ($\epsilon$) est représenté par la chaîne *scheme* vide (`""`). Un message d'erreur est imprimé dans le cas ou l'on détecte une récursivité gauche :

```
(define (first s g recursion)
  (cond ((null? s) '(""))
        ((memq (car s) recursion)
         (display "Left recursion for ")
         (write (car s))
         (newline)
         '())
        ((and (null? (cdr s)) (string? (car s))) s)
        ((and (null? (cdr s)) (symbol? (car s)))
         (let ((p (assoc (car s) g))
               (newrecursion (cons (car s) recursion)))
           (cond ((not p)
                  (error "No production for " (car s)))
                 (else (apply union
                              (map (lambda (s) (first s g newrecursion))
                                   (cdr p)))))))
        (else (let ((x (first (list (car s)) g recursion)))
                (if (member "" x)
                    (append (remove "" x)
                            (first (cdr s) g recursion))
                    x)))))
```

La fonction `follow-sets` calcule l'ensemble des *suivants* pour la grammaire `g` passée en argument. Dans le résultat, le symbole *scheme* $ représente la fin du flux d'entrée. Comme pour la fonction précédente, des messages d'erreur sont imprimés lorsque l'on détecte une récursivité gauche :

```
(define (follow-sets g)
  (define (loop g table)
    (let* ((new (map (lambda (x) (cons x (fol x g table)))
                     (map car g)))
           (new (cons (cons (caar new) (union '($) (cdar new)))
                      (cdr new))))
      (if (equal-table? table new)
          table
          (loop g new))))

  (define (fol x g t)
    (define (fol-production p)
      (let ((lhs (car p))
            (alternatives (cdr p)))
```

```
        (do ((l alternatives (cdr l))
             (f '() (union (fol-alternative x (car l)) f)))
            ((null? l)
             (if (member "" f)
                 (union (lookup lhs t)
                        (remove "" f))
                 f)))))
    (define (fol-alternative x rhs)
      (cond ((null? rhs) '())
            ((eq? x (car rhs))
             (union (first (cdr rhs) g '())
                    (fol-alternative x (cdr rhs))))
            (else (fol-alternative x (cdr rhs)))))
    (apply union (map fol-production g)))

  (loop g
        (cons (list (caar g) '$)
              (map (lambda (p) (cons (car p) '()))
                   (cdr g)))))
```

Les dictionnaires utilisés pendant le calcul sont représentés avec de simples listes d'association dans lesquelles on utilise le prédicat `eq?` pour les comparaisons :

```
(define (lookup x t)
  (cdr (assq x t)))

(define (equal-table? x y)
  (cond ((and (null? x) (null? y)) #t)
        ((or (null? x) (null? y)) #f)
        (else (let ((entry (assoc (caar x) y)))
                (if entry
                    (and (equal-as-sets? (cdr (car x)) (cdr entry))
                         (equal-table? (cdr x) (remove entry y)))
                    #f)))))
```

Comme à l'habitude en Lisp, les ensembles sont représentés à l'aide de simple listes :

```
(define (equal-as-sets? x y)
  (and (every? (lambda (a) (member a y)) x)
       (every? (lambda (a) (member a x)) y)))

(define (union . args)
  (define (union2 x y)
    (cond ((null? x) y)
          ((member (car x) y)
           (union (cdr x) y))
          (else (cons (car x)
                      (union (cdr x) y)))))
  (cond ((null? args) '())
        ((null? (cdr args)) (car args))
        ((null? (cddr args)) (union2 (car args) (cadr args)))
        (else (union2 (union2 (car args) (cadr args))
                      (apply union (cddr args))))))

(define (every? p? l)
  (cond ((null? l) #t)
        ((p? (car l)) (every? p? (cdr l)))
```

```
        (else #f)))

(define remove
  (lambda (item ls)
    (cond
     ((null? ls) '())
     ((equal? (car ls) item) (remove item (cdr ls)))
     (else (cons (car ls) (remove item (cdr ls)))))))
```

## B.3 Utilisation - Exemple

Du point de vue de l'utilisateur souhaitant écrire un analyseur récursif descendant selon la méthode présentée au chapitre 3, seules les fonctions `LL1?` et `director-sets` sont importantes. La première étape consiste à préparer sa grammaire pour appeler la fonction `LL1?`. Si la grammaire ne respecte pas la condition LL(1), il faut autant que possible modifier la grammaire à l'aide des messages d'erreur fournis. Une fois cette étape passée avec succès, la fonction `director-sets` permet d'obtenir les ensembles directeurs pour chaque production.

Par exemple, en utilisant la grammaire de la figure B.1 qui est bien une grammaire LL(1), l'appel de la fonction `director-sets` retourne sans message d'erreur le résultat suivant :

```
((S  ("id") ("while") ("do"))
 (A  ("od") ("id" "while" "do"))
 (E  ("id" "("))
 (E' ("id" "while" "do" ";" ")") ("+") ("-"))
 (T  ("id" "("))
 (T' ("id" "while" "do" ";" ")" "+" "-") ("*") ("/"))
 (F  ("id") ("(")))
```

Dans le résultat, les non-terminaux apparaissent dans le même ordre que celui utilisé dans la grammaire initiale. Par exemple, les trois sous listes en face du non-terminal `S` correspondent respectivement aux symboles directeurs de chacune des trois productions du non-terminal `S`. Ainsi, la mise en correspondance du résultat précèdant avec la grammaire initiale donne :

| | | |
|---|---|---|
| S → | id := E ; | { id } |
| | \| while E S | { while } |
| | \| do S A od | { do } |
| A → | $\epsilon$ | { od } |
| | \| S A | { id while do } |
| E → | T E' | { id ( } |
| E' → | $\epsilon$ | { id while do ; ) } |
| | \| + T E' | { + } |
| | \| - T E' | { - } |
| T → | F T' | { id ( } |
| T' → | $\epsilon$ | { id while do ; ) + - } |
| | \| * F T' | { * } |
| | \| / F T' | { / } |
| F → | id | { id } |
| | \| ( E ) | { ( } |

Dans le cas où il n'est pas possible d'obtenir une grammaire LL(1), le programme calcule quand même les symboles directeurs. Comme la grammaire n'est pas LL(1), pour une règle donnée, les ensembles des différentes productions ne sont pas toujours disjoints. Il est alors souvent possible de lever l'indéterminisme en regardant un deuxième symbole à l'avance durant l'analyse syntaxique des règles concernées.

# Annexe C

# SmallEiffel Contributors

- ⋆⋆⋆ Frank ARNAUD: bug report of `test_tag1.e`. Reported a problem with Windows NT /-Microsoft C / empty struct. Reported a bug about renaming/redefinition of Eiffel externals. New maintainer of the Eiffel FAQ.
- ⋆⋆⋆ Eric BEZAULT: lots of really good (short!) bug reports: rename/select, statically computed expressions, weird uses of inheritance clause, insisted to give SmallEiffel a -case_insensitive flag...
- ⋆⋆⋆ Xavier CREGUT: has been using SmallEiffel for a long time. Many bug reports. Reported in a very small program a tricky bug dealing with inheritance/genericity and VNCG rule.
- ⋆⋆⋆ Geoff ELDRIDGE: since many years, Geoff is one of the best SmallEiffel's supporter thanks to the excellent elj-win32 distribution. Keep up the good work Geoff ! As a sidenote, Geoff is also the initial designer of the BASIC_TIME class.
- ⋆⋆⋆ Fabrice FRANCESCHI: initial sketch of the SmallEiffel Web site.
- ⋆⋆⋆ Enrico GOBBETTI: many (excellent) bug reports and many good bug-fixes .
- ⋆⋆⋆ James GRAVES: initial design of exception handling, on which we based our final implementation. Maintainer of SmallEiffel's RPMs for RedHat Linux.
- ⋆⋆⋆ Alain LE GUENNEC: many well-isolated bug reports plus some very good bug-fixes.
- ⋆⋆⋆ Philippe RIBET: fixed a bug in the ensure clause of INTEGER.`append_in_format`. Optimized INTEGER.`gcd`. Prepared the first precompiled DOS version. Debugging of class BIT_N. (Worked with Antoine GUARRIGUES).
- ⋆⋆⋆ S.u.S.E. GNU/Linux: since mid-1998, the S.u.S.E. GNU/Linux distribution is our main development OS. Indeed, we have already received for free all versions from 4.3 to 6.2. Also note that SmallEiffel The GNU Eiffel Compiler is part of the S.u.S.E. GNU/Linux distribution. We are grateful to S.u.S.E for being very supportive to us. We hope the S.u.S.E. GNU/Linux team will keep up the good work, being successful and providing us with free copies of their distribution ;-)
- ⋆⋆ Cyril ADRIAN: fixed some bugs in STD_FILE_READ (`read_character`, `read_integer`). Found bug of `test_req2.e`. Found problem with xor. Author of `spread_illness`. Found a bug in character constants, etc. (Worked with Antony LE LABOUSSE).
- ⋆⋆ Reimer BEHRENDS: isolated a bug about once precomputed functions and many other bugs.
- ⋆⋆ Nigel BREE: fixed a tricky bug in the runtime stack. Improved portability for Windows95/NT. Wrote the install script for Windows95/NT.
- ⋆⋆ Thierry DE CARVALHO: found a bug in inspect/when. Fixed a bug in STD_FILE_READ.-`connect_to`.

⋆⋆ Emmanuel CECCHET: updates of the Linux executable. Found some bugs in creation calls. Provided a lot of useful comments.

⋆⋆ Valery CROIZIER: reported bugs (in very numerous mails :-). Provided hints to optimize the C code.

⋆⋆ Andre DOHERTY: used to maintain the SmallEiffel ready for OS2 version.

⋆⋆ Antoine GUARRIGUES: worked with Philippe RIBET (see above).

⋆⋆ Wai-Ming HO: many well-isolated bug reports, plus one tricky Garbage Collector bug with fix.

⋆⋆ Jean-Marc JÉZÉQUEL: reported some bugs on ARRAY inheritance. Isolated a bug on VNCG. Beta testing.

⋆⋆ Alexander KJELDAAS: fixed a bug in INSTALL.CSH. Reported a problem with mixed rename/redefine and a problem with run time stack.

⋆⋆ Philippe LAHIRE: reported some bugs with repeated inheritance. Isolated some problems with select.

⋆⋆ Antony LE LABOUSSE: worked with Cyril ADRIAN (see above).

⋆⋆ Jean-Lin PACHERIE: many bug reports.

⋆⋆ Marc SCALZOLARO: reported some bugs and added some tests for expanded class types.

⋆⋆ Fridtjof SIEBERT: for some well isolated bug reports.

⋆⋆ Glen STONE: improved the type inference score as well as code generation (more dispatch functions are now removed). Also sent some good bug reports.

⋆⋆ Brian STRELIOFF: reported some bugs. Gave me some good hints about compatibility with C++ and warning messages of gcc.

⋆⋆ Jerome THOEN: author of some (optimized) features to avoid memory leaks.

⋆⋆ Dan WILDER: uploads new versions of SmallEiffel to sunsite (for Linux).

⋆⋆ Alan ZIMMERMAN: clear bug report with an excellent fix for the exception trace. For one aspect, the problem was specific to the CYGWIN compiler and its implementation of longjmp. The second bug was related to the unwinding of exception trace immediately before a polymorphic call.

⋆ Thomas AGLASSINGER: implementation of the low level C code to make BASIC_DIRECTORY available on Amiga.

⋆ Christophe ALEXANDRE: found a bug in REAL.`put_real_format`.

⋆ Wilfried BERGER: done some tuning of the GC for Windows platforms.

⋆ Oscar N. BRIA: reported a bug for mixed rename/redefine.

⋆ Emmanuel BOUYER: was the first one to use SmallEiffel for his PhD Thesis. He is also a good bug reporter.

⋆ Robert W. BREWER: a good bug report about inlining of $ operator.

⋆ Lars BRUECKNER: wrote some code for STD_FILE_READ.

⋆ Romaric CHARTON: find two bugs in ifthenelse expression, and argument checking of external feature.

⋆ Grant CHESTON: reported a bug with attribute renaming.

⋆ Frank CHIRON: fixed a bug in class BIT_N.

⋆ Vincent CROIZIER: good work to fix the priority of operator infix "∧" and for Precursor implementation.

⋆ Stephane D'ALU: added some features in STRING (split, ...). Reported some bugs for class BIT.

* Oliver ELPHICK: fixed a bug in misc/INSTALL.SH and help us to make our decision about Eiffel source distribution.
* Joel FAEDI: (not yet) solved some problems using gcc.
* Brigitte JARAY: one of the first beta testing girls.
* Zahi Al JAMOUS: found a syntaxical bug, and some others. First Lebanese user of SmallEiffel.
* Stephane JANTZEN: beta testing of externals.
* Patrice KHAWAM: reported some bugs.
* Mostefa KASSAB: reported a bug about deferred features.
* Frédéric LEJAL: provides us precompiled versions for several platforms.
* Nicole LEVY: found some syntactical bugs producing a core dump.
* Masato MOGAKI: reported a bug with $ operator.
* Jean-Louis NÉBUT: reported bug about renaming.
* Steve NORMAN: fixed a bug in floating-point constants.
* Francois PENNANEACH: a good bug report.
* Todd PLESSEL: a good bug report.
* Francois RAGUIN: isolated a bug of REAL.infix "∧"
* Philippe REITZ: bug report dealing with inheritance.
* Christophe REMY: isolated a bug about argument passing.
* Claude SANCHEZ: allowed me to use some more memory (256Mb...) to be able to do the very first bootstrap.
* Gregoire SCHNAKENBOURG: fixed a bug in REAL/DOUBLE keybord input.
* Adrian SIEBER: for some good bug reports.
* Brian STEFANISH: patch for assertion tags.
* John TYRREL: comments and some bug reports.
* Johannes WEHRLI: done some beta testing on VMS.
* Bruce WIELINGA: first author of SmallEiffel/lib_rand.
* Dietmar WOLZ: beta testing and information of SmallEiffel with Boehm-Demers-Weiser garbage collector. Also gave me some hints to optimize ARRAY add_last. Provided a very interesting benchmark comparing SmallEiffel to other C/C++/Eiffel compilers.

# Annexe D

# SmallEiffel's History of changes

Version numbering uses negative numbers. The first distributed version was numbered `-0.99`, the second one `-0.98`, the third `-0.97`, and so on. Version number `-0.0` should correspond to the very best Eiffel compiler on this planet `:-)`
Latest version first.

**Release - 0.76 - Saturday June 3th, 2000**
• Added a brand new class SET[E->HASHABLE].
• Full compatibility with the new ELKS'2000 ARRAY class. Most modifications are simple renamings which are automatically pointed out by obsolete warnings. For obvious uniformity reasons, some other classes of the library have also been modified. As an example, `empty` is always replaced with `is_empty` in all classes. The name `all_cleared` is always replaced with `all_default`. The most difficult problem is due to the fact that `is_equal` in ARRAY no longer uses the elements `is_equal` to compare them, but the basic '=' infix operator. For uniformity reasons, this has also been done for all COLLECTIONs as well as class DICTIONARY.
• Added feature `is_equal_map` for all COLLECTIONs (ARRAY, FIXED_ARRAY, LINKED_LIST, TWO_WAY_LINKED_LIST) and for class DICTIONARY. This is the replacement for the old implementation of `is_equal` (ie. `is_equal_map` use `is_equal` to compare elements).
• Added feature `reindex` in class ARRAY.
• For feature `dispose` of class MEMORY (as well as redefinitions), the class invariant is no longer triggered.
• Added features `atan2` and `pow` both in class REAL and DOUBLE.
• Internal implementation of BASIC_TIME revisited (using now a double for memorization).
• Implementation of class DICTIONARY completely revisited. Also added features `add`, `item_map_in` and `key_map_in` in the new class DICTIONARY.
• Various bug fixes and improvements.

**Release - 0.77 - Saturday February 12th, 2000**
• Added in directory SmallEiffel the new **install** command (class `install.e` with a precompiled **install.exe** for Windows). This `install.e` class is the code for the installer of the SmallEiffel distribution it comes with, and is designed to be as portable as possible. This automatic installation program has already been tested for Windows/lcc-win32, Windows/bcc32, Linux/gcc, Solaris/gcc, DEC-Alpha/gcc and FreeBSD/gcc, ... To install SmallEiffel under some UNIX like system, just type `make` in the SmallEiffel directory. To install SmallEiffel under Windows, just run `install.exe`. On other systems, C compile and run the `install.c` file.

• Added C++ externals support (creation of C++ objects, deletion of C++ objects, member function calls, static function calls etc.). See `lib_show/external/C++` for examples.
• Added NUMBER library, for infinite precision and infinitely large numbers. See `lib_show/number` for examples.
• Added ITERATOR library, for external iterators on data structures. See `lib_show/iterator` for examples.
• Added class BASIC_DIRECTORY and class DIRECTORY to handle in a portable way directories as well as system path notations. See `lib_show/basic_directory` for examples. This class is also implemented for Java bytecode (command `compile_to_jvm`).
• Added class BASIC_TIME, TIME_IN_FRENCH, TIME_IN_ENGLISH and TIME_IN_ITALIAN to handle date and time. See `lib_show/basic_time` for examples. Low level routines for Java byte-code are not yet implemented If you have some time to do this, you just have to add the missing Java code in `sys/runtime/SmallEiffelRuntime.java` (please contribute your work on the SmallEiffel mailing list).
• Added class BIT_STRING for very large bit sequences, with reference semantics.
• Features `deep_clone` and `deep_equal` of class GENERAL are now implemented for the C compilation mode (does not work yet with `compile_to_jvm`).
• Added feature `skip_remainder_of_line` and feature `reach_and_skip` in class INPUT_STREAM. Warning: in order to be consistent with other features, the behavior of features `skip_separators` and `skip_separators_using` has changed.
• Added feature `to_hexadecimal` and feature `to_hexadecimal_in` in class INTEGER.
• Improved incremental recompilation of the generated C code (also fixed a very old bug in incrementality).
• Exceptions handling: the `default_rescue` feature is now supported.
• Various bug fixes and improvements (type inference score increased, ...).

**Release - 0.78 - Saturday June 05th, 1999**
• New `-html2` flag for command short generates a colorized HTML short form for classes.
• The garbage collector now takes into account feature dispose of class MEMORY for reference objects.
• Obsolete classes (obsolete keyword) now supported.
• Assertion tags are now displayed when an assertion fails.
• Added environment variable in loadpath files. Syntax: `$some_var`
• Added the `-no_style_warning` flag to suppress warnings when the recommended styles guidelines for Eiffel are not strictly followed.
• Added the `-version` flag to show SmallEiffel's version.
• Enhanced ease of use with and adaptability to various C compilers (SYSTEM_TOOLS). Files `compiler.system`, `linker.system` and `o_suffix.system` are now obsolete and replaced by a unique file compiler.se common to all systems. See the System configuration page for more information.
• Class LINKED_LIST now replaces obsolete class LINK_LIST (simple renaming). Class TWO_WAY_LINKED_LIST now replaces obsolete class LINK2_LIST (simple renaming).
• Fixed "implicit renaming" bug.
• Cleaned all source code of tabulations at beginning of line (made code look ugly when using an editor whose tabs were not 8).
• Some new ELKS'95 features implemented.
• Validity rule VEEN fixed.

• Added class COLLECTION_SORTER and REVERSE_COLLECTION_SORTER to the library.
• Fixed file renaming portability bug (".d files bug").
• Validity rule VCFG.1 is now enforced.
• Validity rule VAPE is now enforced.
• Classes mentioned in a cecil.se file are now automatically made live. Makes it easier to link with external libraries.
• Various other bug fixes.

**Release - 0.79 - Tuesday December 22nd, 1998**
• The new Eiffel construct `Precursor` as described OOSC2 is now implemented.
• The Eiffel expression `strip` is now implemented.
• Nested loadpath files now allowed.
• Extended anchored definition to accept `infix` and `prefix` feature names. For example, this kind of declaration is now accepted:

```
foo: like infix "+" ...
```

• Classes COLLECTION2, ARRAY2 and FIXED_ARRAY2 of `lib_std` completely revisited.
• Fixed a bug related to calls of the form `f.g.h`; on expanded objects.
• Fixed a bug related to inheritance of generic classes.
• Fixed bugs related to assertions checking in case of exceptions (rescue clause, retry). Improved cycle detection in assertions.
• Fixed bug in GC related to recycling of "monsters" (very large resizable objets).
• Fixed an incredible bug in the implementation of the like Current type mark. `;-)`.
• Fixed many others bugs ($ operator, GC for alpha DEC, ...).
• System customization file for the BeOS system added in the "`sys`" sub-directory (more on system customization).
• Contents of the environment variable `SmallEiffel` must be now set with the absolute path of the file "system.se" which is in the sub-directory "sys" of the installation directory. Under a UNIX-like system, the value of the `SmallEiffel` environment variable may be for example: `/usr/lib/SmallEiffel/sys/system.se` Commands are also more robust when this environment variable contains non-alphanumeric characters.
• No more ensure assertion in GENERAL.`get_environment_variable`.
• Commands `compile_to_jvm` and `print_jvm_class` completely revisited. Java byte-code can be now used with the `-verify` Java option).
• Validity rule VDRD.6 is now enforced.
• Balancing rule (automatic promotion) for INTEGER, DOUBLE and REAL is now implemented.
• Recursive once routines now work correctly.
• Unmodified generated C files are not touched anymore.
• Assertions correctly checked for all external C routines.
• Exception handling now works when an exception occurs in external C code.
• Cleaned the compiler and decreased its memory footprint.

**Release - 0.80 - Thursday July 9th, 1998**
• Exception handling implemented. Class EXCEPTIONS added in `lib_std`.
• Execution trace stack implementation completely revisited: more comprehensive information is now available, and the overhead incurred by this stack is greatly reduced (executables are about 3 times faster when running in `-all_check` mode).
• Added option `-no_main` in command `compile_to_c` to avoid generation of the C main function.

This is useful when one wants to start execution from outside before calling some Eiffel routines via the cecil interface.
• Behavior of compilation flag -trace of command compile_to_c changed to allow step-by-step execution (embryo of Eiffel source code debugger).
• Feature ARRAY.resize completely revisited (added two features in class NATIVE_ARRAY: clear and move).
• Garbage Collector optimized (the GC should be more agressive and some benchmarks are included in directory SmallEiffel/misc/benchmarks/gc/*/bench.e).
• Associativity of infix operator "^" is now correctly handled.
• Fixed bugs in PLATFORM for Minimum_double, Minimum_real and Minimum_character_code.
• Fixed some others bugs about export rules, about expanded objects with expanded attributes.
• Fixed bugs in pretty.

**Release - 0.81 - Thursday April 9th, 1998**
• First finalized version of the garbage collector. Currently supported architectures are SPARC Solaris, HP-UX, Linux, MacOS, Windows 95 and NT (see file SmallEiffel/sys/gc for details). Added flag -no_gc to suppress the GC (see man/compile_to_c).
• Changed the default for the generation of C code. Now, the C code and object files are kept by default. (This previously required using option -c_code, which becomes obsolete.) Added flag -clean which removes all the C and object files of the system. (This corresponds to the old default behavior.)
• Fixed a bug related to buffered input under Windows/MS VC.
• Fixed a bug in misc/INSTALL.SH script.
• Added require is_connected for all features put_* of class OUTPUT_STREAM.
• Fixed a bug in command short (require/ensure assertion of deferred routines are now printed).
• Added one useful hook for mode -html1 in command short (see hook Mcn).
• Added directory SmallEiffel/sys/runtime which contains the C runtime.
• Fixed a bug about instanciation of class ANY itself (not so common!).
• Fixed a bug about inlining of operator $.
• Fixed a bug about cyclic anchored definitions.

**Release - 0.82 - Friday January 16th, 1998**
• Became the official GNU Eiffel compiler.
• Added first HTML mode to command short. Flag -html1. Thanks to Matthias Klause.
• Fixed bugs in command pretty.
• It is now possible to rename/redefine external "SmallEiffel" features.
• Files *.hlp of directory SmallEiffel/man no longer exists. All the documentation is now in *.html files plus corresponding automatically generated *.txt files.
• Manifest arrays creation has been optimized.
• Validity rule VHRC.2 is now enforced.
• Fixed bugs about export clause.
• Feature GENERAL.hash_code no longer exists. Added ELKS class HASHABLE in order to be compatible with others compilers/libraries.
• Fixed a bug in pre-computed once functions.
• Fixed a bug in REAL/DOUBLE keybord input.
• Reintroduced left hand side cast in C code for better performances (allowed by the C ANSI standard).

• Obsolete features (obsolete keyword) now generate a Warning.

**Release - 0.83 - Friday September 19th, 1997**
• Added command **short** to the distribution.
• Added directory `contrib` in the distribution. This directory contains some scripts to use gdb as source level debugger for SmallEiffel.
• Added flag `-no_warning` to commands: `compile_to_c`, `pretty`, and `compile_to_jvm`.
• Added flag `-case_insensitive` to command `compile_to_c`.
• Fixed a bug for inherit/select.
• Fixed a bug for some pre-computed once function.

**Release - 0.84 - Monday August 18th, 1997**
• Class BIT is now implemented in Java bytecode (command `compile_to_jvm`).
• Added external specification to call Java code (when using `compile_to_jvm`).
• To fit on a single 3.5 inches disk, unsplitted C code for large commands (`compile_to_c` and `compile_to_jvm`) is no longer in the distribution as well as the old `lib_test` directory.
• Changed the algorithm to load classes in order to allow upper case letters in files names (priority is always given to lower case file names).
• Added STRING.`substring_index` (from ELKS written by Fridtjof SIEBERT).
• Added one more file in `SmallEiffel/sys/ directory` in order to customize object files suffix (thus, using Borland C compiler on Windows is now possible).
• Changed the behavior of `end_of_input` of INPUT_STREAM (this flag is true _after_ last character has been read).
• Fixed a bug in manifest arrays (mixed objects including Void).
• Fixed a bug dealing with some statically computed expressions.
• Fixed a bug in repeated inheritance.

**Release - 0.85 - Thursday July 3rd, 1997**
• First beta-release of commands `compile_to_jvm` and `print_jvm_class`!
• The new name for class C_ARRAY is now NATIVE_ARRAY (because it works both with Java and C). The old C_ARRAY name is temporarily accepted with a warning from the compiler.
• For readability and to avoid confusion with Java names, external tags have changed (see for new names in `man/external` file). Old names are temporarily accepted with a warning giving the new name to use.
• Added feature `to_hexadecimal` and feature `to_hexadecimal_in` in class CHARACTER.
• Feature io, `std_input`, `std_output` and `std_error` of class GENERAL are no longer frozen.
• Some changes in STD_FILE_READ for features `read_integer`, `read_double` and `read_real` (added comments, precondition and solved the problem with the trailing separator).
• Feature `die_with_code` of GENERAL now accepts any INTEGER code (not just predefined `exit_success_code` and `exit_failure_code`).
• Fixed a bug in STRING. The following expression is now true:

```
("a%/0/b").count = 3
```

Just try this on your favorite Eiffel compiler ;-)
• Many changes in basic input/output in order to be compatible with Java: No more class STD_FILE (the name is now free for an ELKS implementation). Two new classes: INPUT_STREAM and OUTPUT_STREAM.
• Feature `unread_character` of class INPUT_STREAM is now implemented in Eiffel.

• Added OUTPUT_STREAM.put_pointer to view a POINTER.

**Release - 0.86 - Sunday April 13th, 1997**
• All reported bugs at this time have been fixed.
• Added flag -verbose to commands: compile, compile_to_c and clean. When this new flag is not present, commands now work silently unless some error (or warning) occurs.
• Features BOOLEAN.infix "or" and BOOLEAN.infix "and" are now written in pure Eiffel. As a consequence, it is very important for the SmallEiffel programmer to make the distinction between BOOLEAN.infix "and then" and BOOLEAN.infix "and" (respectively for BOOLEAN.infix "or else" and BOOLEAN.infix "or"). When left-hand-side argument produces no side effect, the semi-strict operator (BOOLEAN.infix "and then"/"or else") may run faster.
• Fixed REAL.sin (the old one was calling sqrt :-).
• Fixed a bug in INTEGER.append_in (you can now print Minimum_integer).
• Added feature in_range in class COMPARABLE.
• Result type of REAL.infix "^" is now DOUBLE for ELKS compatibility.
• Feature remove is now implemented in all subclasses of COLLECTION (i.e. ARRAY, FIXED_ARRAY, LINK_LIST and LINK2_LIST).
• Added feature add for all subclasses of COLLECTION.
• Conversion DOUBLE/STRING: ANSI C sscanf and sprintf is now used to avoid loss of precision.
• Added some more class invariant code generation when compiling in -invariant_check mode. Class invariant is now also checked before exit of a routine.
• Added feature file_tools in class GENERAL to ease access to class FILE_TOOLS.
• Changed printing format for basic *_REF classes. For example, instruction print(1); now prints 1.
• Feature truncated_to_integer of DOUBLE is now ELKS compatible (added feature rounded in class DOUBLE to replace the old truncated_to_integer). Same changes in class REAL.

**Release - 0.87 - Tuesday January 7th, 1997**
• Added a new flag -trace to ease debug (see man/compile_to_c file).
• More inlining at Eiffel level (-boost mode only).
• Class BIT_N completely revisited. It may be as fast as C.
• Added class C_ARRAY[E] to deal directly with C arrays at Eiffel level. Thus there are no more external "CSE" or c_inline_c in classes STRING/ARRAY/FIXED_ARRAY (only full Eiffel). Eiffel code is nice and STRING/ARRAY/FIXED_ARRAY may run faster.
• Fixed a bug in STD_FILE_READ.read_double.
• Ordering of C output to increase gcc inlining.
• According to man/compile_to_c) , flag -debug_check now works (debug instructions are no longer generated in mode -all_check).
• Unused local variables removed at Eiffel level (warning added for -debug_check mode only).
• Default class ANY now inherits PLATFORM (as in ETL).
• Fixed a bug in floating-point constants.
• Added directory sys to customize default C compiler, default C linker and default loading path.
• Fixed some bugs with outside expanded types.
• Fixed a bug with rename/select.
• Redefinition of once routine is now allowed.

• Feature GENERAL.`conforms_to` is now implemented.
• Rule VFFD.7 is now enforced.
• Feature `force` implemented for all COLLECTION.
• Added conversions features CHARACTER.`to_bit`, INTEGER.`to_bit`, BIT_N.`to_character` and `BIT_N.to_integer`.

**Release - 0.88 - Wednesday October 30th, 1996**
• Fixed bugs dealing with inherit/rename/select.
• Added warning for missing colon in actual arguments list.
• Warning : INSTALL procedure has changed and you have to set manually the default loading path (see `misc/INSTALL` for details).
• Added some VMS customization.
• Fixed bugs when printing run-time stack.
• Warning added for missing colon in actual parameter list.
• Added flush in class STD_FILE_WRITE.
• No more left hand side cast in C code (because some C compilers don't like them).

**Release - 0.89 - Sunday September 15th, 1996**
• Calling Eiffel from C is now implemented (see `man/cecil`) file).
• Object creation uses C `calloc` instead `malloc+memset`.
• Object creation is inlined.
• ARRAY/FIXED_ARRAY put and item are now inlined (`-boost` only).
• Added feature `capacity` and `resize` in FIXED_ARRAY.
• Added some classes in `std_lib`: LINK2_LIST (two way linked list), COLLECTION2 (deferred), ARRAYED_COLLECTION (deferred), LINKED_COLLECTION (deferred) and FIXED_ARRAY2 in std_lib.
• More user's routines are inlined (`-boost` only).

**Release - 0.90 - Friday August 23rd, 1996**
• Added random number generator library (`SmallEiffel/lib_rand`).
• Added checking of assertions for external "CSE" features.
• Anchoring on expanded types are now allowed.
• Multiple level of anchoring definition allowed.
• Fixed a bug in STRING.`from_external`.
• Fixed a bug with mixed rename/redefine.
• Inheritance loop detection.
• Anchoring loop detection.
• Fixed a bug with renaming infix/prefix.
• Command clean also use the make suffix.
• Added ELKS95 sign in INTEGER/REAL/DOUBLE.
• Feature make is the default root feature name for `compile`.
• Optimized ARRAY.`add_last` and STRING.`extend`.
• Changed STRING.`out` (no more enclosing %" printed).
• Remove warning gcc messages for 64 bits machine (alpha/DECK).
• Added option `-no_split` for a better finalization (see `man/compile_to_c` file).

**Release - 0.91 - Wednesday July 24th, 1996**
• Incremental C compiling mode implemented (see option `-c_code` in help file `man/compile`) .
• Added command clean in help file `man/clean`.
• Added class FILE_TOOLS.
• No more empty C struct in generated C code (to avoid problems with Microsoft Windows C compiler).
• Fixed one more bug with conformance rule VNCG.

**Release - 0.92 - Saturday July 20th, 1996**
• Fixed some more bugs with conformance rule VNCG.
• Added math functions in DOUBLE/REAL (`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, ... ANSI C names).

**Release - 0.93 - Thursday July 18th, 1996**
• Fixed some more bugs with conformance rule VNCG.
• Type BIT is now implemented.
• Added ELKS 95 STRING.`left_adjust` and STRING.`right_adjust`.

**Release - 0.94 - Friday July 5th, 1996**
• Fixed some bugs with conformance rule VNCG.
• Static expressions are used to detect pre-computable once routines.
• Added features to_external and from_external both in classes ARRAY and FIXED_array.
• Fixed a bug for inheritance of ARRAY or FIXED_ARRAY.
• Contents of end of class comment checked.
• Added documentation file for external calls (see help file `man/external`).

**Release - 0.95 - Thursday May 30th, 1996**
• It is now possible to inherit ARRAY.
• It is now possible to inherit STRING.
• Flag `-cc` of `compile_to_c` has changed.
• The PLATFORM class is now conform to ELKS95.
• Added feature `twin` (like the one of TowerEiffel) in GENERAL.
• Added features `to_external` and `from_external` in class STRING.
• Fixed some bugs in once pre-computed routines.

**Release - 0.96 - Friday May 10th, 1996**
• Internal renaming to suppress some warning messages and for a smooth integration with C++ software.
• DOS and Macintosh better portability.

**Release - 0.97 - Thursday May 2nd, 1996**
• Inside of compiler cleaned.
• Pre-Computing of some once functions.
• Added class COLLECTION, LINK_LIST and FIXED_ARRAY in `lib_std`.

**Release - 0.98 - Friday March 15th, 1996**
• Command `pretty` added.
• Implements Eiffel expanded clause.

**Release - 0.99 - Saturday February 17th, 1996**

- The first version available on the net. Before being made available, the very first SmallEiffel had been tested since September 1995 by students of the University Henri Poincaré

**Release - 1.00 - July 1995**

- The very first bootstrap. SmallEiffel is born.

# Bibliographie


[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, 1996.

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 2–26. Springer-Verlag, 1995.

[AJ74] A. V. Aho and S. C. Johnson. LR Parsing (YACC - Yet Another Compiler Compiler). *ACM Computing Surveys*, 1974.

[Alm97] P. S. Almeida. Ballon Types: Controlling Sharing of State in Data Types. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Sciences*, pages 32–59. Springer-Verlag, June 1997.

[App98] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. ISBN 0-521-58388-8.

[AR92] P. André and J.-C. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *7th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*, volume 27, number 10 of *SIGPLAN Notices*, pages 110–126. ACM Press, October 1992. Vancouver, BC, Canada.

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

[ASU89] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilateurs. Principes, techniques et outils*. InterEditions, Paris, 1989. ISBN 2-7296-0295-X.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[BBDL97] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraint for Incomplete Objects. In *Proceedings of TAPSOFT/CAAP*, volume 1214 of *Lecture Notes in Computer Sciences*, pages 465–477. Springer-Verlag, 1997.

[BBDL99] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping for Extensible, Incomplete Objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

[Ben96] J.P. Bennett. *Introduction to Compiling Techniques. A first course using ANSI C, lex and yacc.* McGraw-Hill, London, 1996. ISBN 007709221X.

[Bru93] K. Bruce. Safe Type Checking in a Statically-typed Object-oriented Programming Language. In *Proceedings of 26th ACM Conference on Principles of Programming Languages (POPL'99)*, pages 285–298, 1993.

[Bru94] K. Bruce. A Paradigmatic Object–Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[Bru97] K. Bruce. Subtyping is not a Good Match for Object-oriented Programming Languages. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Sciences*, June 1997.

[BSvG95] K. Bruce, A. Shuett, and R. van Gent. PolyToil: a Type-safe Polymorphic Object Oriented Language. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 27–51. Springer-Verlag, 1995.

[Bur98] J. Burstein. *`Rupiah`: an Extension to Java Supporting Match-bounded Parametric Polymorphism, `ThisType`, and Exact Typing.* PhD thesis, Williams College, 1998.

[BV99] B. Bokowski and J. Vitek. Confined Types. In *14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 33, number 10 of *SIGPLAN Notices*, pages 82–96. ACM Press, October 1999.

[CC90] S. Collin and D. Colnet. Analysis of Dimension in Mechanical Engineering Drawings. In *Proceedings of IAPR Workshop on Machine Vision Applications*, pages 105–107, Tokyo, Japan, 1990. LORIA 90-R-167.

[CC92a] S. Collin and D. Colnet. Grammaire pour la reconnaissance de la cotation dans les dessins techniques. In *BIGRE 80 - Actes de CNED'92 ; Traitement de l'écriture et des documents*, pages 57–64, Nancy, Juillet 1992. LORIA 92-R-180.

[CC92b] S. Collin and D. Colnet. Syntactical Analysis of Technical Drawing Dimensions. In *Proceedings of Workshop on Syntactical and Structural Pattern Recognition*, pages??–??, Berne, Suisse, 1992. LORIA 92-R-???

[CC94] S. Collin and D. Colnet. Syntactic Analysis of Technical Drawing Dimensions. In *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, volume 8, number 5, pages 1131–1148, October 1994. LORIA 94-R-036.

[CCCD88] G. Canals, D. Colnet, S. Cruz-Lara, and J. C. Derniame. GEPI: un Générateur d'Environnements de Programmation Intégrés. In *Software Engineering & Its Application*, pages 406–421, Toulouse, December 1988. LORIA 88-R-153.

[CCZ96] S. Collin, D. Colnet, and O. Zendra. Compiling Late Binding in Object-Oriented Languages with a Copy-and-Adapt Technique. Application to the Eiffel Language. Technical report, LORIA - UMR 7503, 1996. LORIA 96-R-068.

[CCZ97] S. Collin, D. Colnet, and O. Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. In *Joint Modular Languages Conference, JMLC'97*, volume 1204 of *Lecture Notes in Computer Sciences*, pages 67–81, Lintz, Austria, 1997. Springer Verlag. LORIA 98-R-056.

[CCZ98a] D. Colnet, P. Coucaud, and O. Zendra. Compiler Support to Customize the Mark and Sweep Algorithm. In *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pages 154–165, Vancouver, BC, Canada, October 1998. LORIA 98-R-233.

[CCZ98b] D. Colnet, P. Coucaud, and O. Zendra. Eiffel: la puissance de la recherche au service de vos programmes. *Programmez !* Numéro 2 avec SmallEiffel sur CD-ROM, pages 82–83, Juillet 1998. LORIA 98-R-201.

[CDH84] J. Chailloux, M. Devin, and J.M. Hullot. *Le_Lisp, a portable and Efficient System.* Rapport INRIA no. 319, Rocquencourt, 1984.

[CGS95] C. Comar, F. Gasperoni, and E. Schonberg. The GNAT Project: A GNU-Ada-9X Compiler. Technical report, New York University, USA, 1995. `http://www.cs.caltech.edu/~john-t/programming/gnat/`.

[CGV80] P.Y. Cunin, M. Griffiths, and J. Voiron. *Comprendre la Compilation.* Springer-Verlag, Berlin, 1980.

[CKL00] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *?*, Submitted paper. Extended Abstract, July 2000.

[CL00a] D. Colnet and L. Liquori. Match-O, a Dialect of Eiffel with Match-Types. In *35th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'2000), Sydney, Australia*, pages 1–12. IEEE Computer Society PR00???, November 2000. LORIA A00-R-270.

[CL00b] D. Colnet and L. Liquori. Match-O, a Statically Safe(?) Dialect of Eiffel. Technical report, LORIA - UMR 7503, 2000. LORIA A00-R-103.

[CMN+86] D. Colnet, G. Masini, A. Napoli, Y. Noiret, and K. Tombre. Les langages orientés objets. Technical report, Centre de Recherche en Informatique de Nancy, 1986. LORIA 86-R-077.

[CMN+89] D. Colnet, G. Masini, A. Napoli, K. Tombre, and D. Léonard. La montée des langages à Objets. *Le Monde Informatique* Numéro 376, pages 22–24, 1989. LORIA 89-R-128.

[Col86] D. Colnet. De ceyx. Technical report, Centre de Recherche en Informatique de Nancy, 1986. LORIA 86-E-035.

[Col89] D. Colnet. *Prototypage, programmation objet : application à l'édition structurelle.* Thèse de l'Université de Nancy I, 1989. LORIA 89-T-018.

[Col99] D. Colnet. *Eiffel, SmallEiffel, conception, programmation, objets, contrats, et cetera.* Polycopié du cours et sujets d'exercices, Licence d'Informatique et Ecole Supérieure d'Informatique et Applications de Lorraine, 1992-1999.

[Cox83] B. J. Cox. Object-Oriented Programming in C. *UNIX Review*, pages 67–70, Octobre/Novembre 1983.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33, number 10 of *SIGPLAN Notices*, pages 48–64. ACM Press, October 1998.

[CSZ99] D. Colnet, E. Stapf, and O. Zendra. NICE ELKS 2000 proposal. Technical report, LORIA - UMR 7503, 1999. LORIA 99-R-359.

[CZ99] D. Colnet and O. Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99), Nancy, France*, pages 341–350. IEEE Computer Society PR00275, June 1999. LORIA 99-R-061.

[CZ00] Dominique Colnet and Olivier Zendra. Targeting the Java Virtual Machine with Genericity, Multiple Inheritance, Assertions and Expanded Types. Technical Report A00-R-137, LORIA - UMR 7503, September 2000.

[CZC98] D. Colnet, O. Zendra, and P. Coucaud. Using Type Inference to Customize the Garbage Collector in an Object-Oriented Language. The SmallEiffel Compiler. Technical report, LORIA - UMR 7503, 1998. LORIA 98-R-196.

[CZC99a] P. Coucaud, O. Zendra, and D. Colnet. Gestion mémoire: manuelle ou automatique? *Programmez !* Numéro 7, pages 54–57, Février 1999. LORIA 99-R-021.

[CZC99b] P. Coucaud, O. Zendra, and D. Colnet. La programmation à objets. Application au langage Eiffel ($1^{ère}$ partie). *Linux Magazine* Numéro 8, pages 63–66, Juillet 1999. LORIA 99-R-073.

[CZC99c] P. Coucaud, O. Zendra, and D. Colnet. La programmation à objets. Application au langage Eiffel ($2^{ième}$ partie). *Linux Magazine* Numéro 9, pages 52–55, Septembre 1999. LORIA 99-R-089.

[DCT89] A. Napoli D. Colnet, G. Masini and K. Tombre. *Actes Les Mardis Objets du CRIN.* Centre de Recherche en Informatique de Nancy, Nancy, 1989. LORIA 89-R-072.

[DH95] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, volume 30, number 10 of *SIGPLAN Notices*, pages 141–155. ACM Press, October 1995. Austin, TX, USA.

[DH96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 306–323. ACM Press, 1996.

[DHV95] K. Driesen, U. Hölzle, and J. Vitek. Message Dispatch on Pipelined Processors. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 253–282. Springer-Verlag, 1995.

[DMM98] A. Diwan, K. S. McMkinley, and J. Eliot B. Moss. Type-Based Alias Analysis. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, volume 33, number 5, pages 106–117. ACM Press, June 1998.

[DMSV89] R. Dixon, T. McKee, P. Schweitzer, and M .Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *4th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24, number 10 of *SIGPLAN Notices*, pages 211–214. ACM Press, October 1989. New Orleans, LA, USA.

[DMW98] S. Debray, R. Muth, and M. Weippert. Alias analysis of Executable Code. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 12–24. ACM Press, 1998.

[ES90] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[FM95] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proceedings of FCT*, volume 965 of *Lecture Notes in Computer Sciences*, pages 42–61. Springer-Verlag, 1995.

[Fos68] J. M. Foster. A Syntax Improving Device. In *Computer Journal*, 1968.

[GH98] R. Ghiya and L. J. Hendren. Putting Pointer Analysis to Work. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 121–133. ACM Press, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addisson-Wesley, Reading, Massachusetts, 1995. ISBN 0201633612.

[GHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33, number 10 of *SIGPLAN Notices*, pages 166–178. ACM Press, October 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Gol84] A. Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addisson-Wesley, Reading, Massachusetts, 1984.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Gre64] S. A. Greibach. Formal Parsing System. In *Communications of the ACM*, volume 7, number 8, 1964.

[Gre65] S. A. Greibach. A New Normal Form Theorem for Context-Free Phrase Structure Grammars. In *Communications of the ACM*, volume 12, 1965.

[Gri71] D. Gries. *Compiler Construction for Digital Computer.* J. Wiley & Sons, New York, 1971.

[HLW+92] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2), April 1992.

[Hog91] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, volume 26, number 11 of *SIGPLAN Notices*, pages 271–285. ACM Press, November 1991.

[Hul83] J. M. Hullot. Ceyx: a Multiformalism Programming Environment. In *Proceedings of IFIP'83*, Paris, 1983.

[JL96] R. Jones and R. Lins. *Garbage Collection.* Wiley, 1996. ISBN 0-471-94148-4.

[JTM99] J. M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts.* Addisson-Wesley, 1999. ISBN 0-201-30959-9.

[KLC99] J.Besse K. Limam, P. Vernel and D. Colnet. Design of a customizable processor IP. In *International Workshop on IP Based Synthesis and System Design*, pages 181–185, Grenoble, December 1999. LORIA 99-R-380.

[LC96] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proceedings of Asian*, volume 1179 of *Lecture Notes in Computer Sciences*, pages 129–141. Springer-Verlag, 1996.

[Liq97] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Sciences*, pages 146–169. Springer-Verlag, 1997.

[Liq98] L. Liquori. On Object Extension. In *12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Sciences*, pages 498–552, 1998.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mey92] B. Meyer. *Eiffel, The Language.* Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-247925-7.

[Mey94] Bertrand Meyer. *Eiffel, The Language.* Prentice Hall, 1994.

[Min96] N. Minsky. Towards Alias-Free Pointers. In *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Sciences*, 1996.

[MNC+89] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets.* InterEditions, Paris, 1989. LORIA 89-R-126.

[MNC+91a] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Linguaggi per la programmazione a oggetti.* Gruppo Editoriale Jackson, Milano, 1991.

[MNC+91b] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Object-Oriented Languages.* Academic Press, New York, 1991. ISBN 0-12-477390-7.

[Muc97] S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, San Francisco, 1997. ISBN 1-55860-320-4.

[NVP98] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Sciences*, pages 158–185, 1998.

[Rém98] D. Rémy. From Classes to Objects via Subtyping. In *Proceedings of ESOP*, volume 1381 of *Lecture Notes in Computer Sciences*, pages 200–220. Springer-Verlag, 1998.

[RR99] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, volume 34, number 5, pages 77–90. ACM Press, May 1999.

[RS98] J.G. Riecke and C. Stone. Privacy via Subsumption. In *Electronic proceedings of FOOL-98*, 1998. To appear in *Theory and Practice of Object Systems.*

[Ruf95] E. Ruf. Context-Insensitive Alias Analysis Reconsidered. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, volume 30, number 6, pages 13–22. ACM Press, June 1995.

[Sch74] F. Scheid. *Introduction à l'informatique.* Série Schaum, Paris, 1974.

[Sch86] K. J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, Hasbrouck Heights, New Jersey, 1986.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Series in Computer Science, 1986.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, London, 1985. ISBN 0-07-065161-2.

[US87] David M. Ungar and Randall B. Smith. Self: The Power of Simplicity. In *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 227–241. ACM Press, 1987.

[We84] P.H. Winston and K.A. Prendergast editors. *The AI Business.* MIT Press, Cambridge, Massasuchetts, 1984.

[WL95] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, volume 30, number 6, pages 1–12. ACM Press, June 1995.

[WM94] R. Wilhelm and D. Maurer. *Les compilateurs théorie.construction.génération.* Masson, Paris, 1994.

[YHR99] S. H. Yong, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, volume 34, number 5, pages 91–103. ACM Press, May 1999.

[ZC99a] O. Zendra and D. Colnet. Adding external iterators to an existing Eiffel class library. In *32th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99), Melbourne, Australia*, pages 188–199. IEEE Computer Society PR00462, November 1999. LORIA 99-R-318.

[ZC99b] O. Zendra and D. Colnet. Towards safer aliasing with the Eiffel language. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems (IWAOOS'99) - ECOOP'99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Sciences*, pages 153–154. Springer Verlag, June 1999. LORIA 99-R-317.

[ZC00a] O. Zendra and D. Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software Pratice and Experience (SP&E)*, ?:A paraître, ??? 2000. *J. Wiley & Sons.*

[ZC00b] O. Zendra and D. Colnet. Vers un usage plus sûr de l'aliasing avec Eiffel. In *5ème Colloque Langages et Modèles à Objets (LMO'2000), Mont St-Hilaire, Québec*, pages 183–194. Hermes, Janvier 2000. LORIA A00-R-022.

[ZCC97] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, number 10 of *SIGPLAN Notices*, pages 125–141. ACM Press, October 1997. LORIA 97-R-140.

[ZCC98a] O. Zendra, D. Colnet, and P. Coucaud. SmallEiffel: l'Eiffel à Très Grande Vitesse. *Programmez !* Numéro 3, pages 64–67, Octobre 1998. LORIA 98-R-241.

[ZCC98b] O. Zendra, D. Colnet, and P. Coucaud. With SmallEiffel, The GNU Eiffel Compiler, Eiffel joins the Free Software community. *GNU Bulletin*, 25, 1998.

[ZCC99a] O. Zendra, D. Colnet, and P. Coucaud. La programmation à objets. Application au langage Eiffel ($3^{ième}$ partie). *Linux Magazine* Numéro 10, pages 60–63, Octobre 1999. LORIA 99-R-126.

[ZCC99b] O. Zendra, D. Colnet, and P. Coucaud. La programmation à objets. Application au langage Eiffel ($4^{ième}$ partie). *Linux Magazine* Numéro 11, pages 56–59, Novembre 1999. LORIA 99-R-228.

[Zen00] O. Zendra. *Traduction et optimisation globale dans les langages de classes.* Thèse de l'Université Henri Poincaré – Nancy 1, October 2000. LORIA A00-T-304.

# Index


Résumé des publications:
[CC90], 46
[CC92a], 47
[CC92b], 45
[CC94], 42
[CCCD88], 46
[CCZ96], 52
[CCZ97], 45
[CCZ98a], 45
[CCZ98b], 49
[CL00a], 42
[CL00b], 50
[CMN+86], 53
[CMN+89], 50
[CSZ99], 51
[CZ00], 51
[CZ99], 44
[CZC98], 52
[CZC99a], 49
[CZC99b], 48
[CZC99c], 48
[Col86], 53
[Col89], 47
[Col99], 52
[DCT89], 50
[KLC99], 44
[MNC+89], 39
[MNC+91a], 40
[MNC+91b], 40
[ZC00a], 41
[ZC00b], 43
[ZC99a], 43
[ZC99b], 44
[ZCC97], 41
[ZCC98a], 49
[ZCC98b], 49
[ZCC99a], 48
[ZCC99b], 48
=, 30, 36

accesseur, 34
Ada, 3
agent, 14, 62
Agesen, Ole, 63
Alain Bashung, 17
Alan Kay, 1
alias, 28
*aliasing*, v, 2, 27, 28, 41, 43
allocation, 13
analyse de flot, 63
analyse dynamique, 63
analyse globale, 7, 58, 61, 63
analyse lexicale, 17–25, 61
analyse sémantique, 61
analyse syntaxique, 17–25, 61
ANY, 65
appel mégamorphique, 59
appel polymorphique, 58
APPLE, 65
arbre de branchement binaire, 58
architecture matérielle, 58
ARGUMENT_NAME, 33
ARRAY, 12, 61
Asimov Isaac, 55
assertions, 1, 14, 28, 31, 32, 38, 60–62
auto-compilation, i, 10, 36, 61, 93

*balloon type*, 28
BASE_CLASS, 34
Bashung, Alain, 17
BASIC_DIRECTORY, 5, 86
BASIC_TIME, 5, 85, 86
bibliothèque standard, 4, 61
BIT_N, 5, 90
BIT_STRING, 86
Bizet Georges, 27
BNF, 75
*bootstrap*, *voir* auto-compilation
Borland Delphi, 56
*bytecode*, 4, 62

C, 4, 5, 18, 61, 62
C++, 64, *voir* langage ordinaire
C_ARRAY, 89
cache, 63
calcul des types vivants, 8
calcul du code vivant, 8
calcul statique des expressions, 58
*Cartesian Product Algorithm*, 63
Ceyx, 3, 53
Chailloux Jérôme, 3
CLASS_NAME, 33
classe feuille, 11
classe générique, 9
classe vivante, 8, 61
*closure*, 14, 62
code mort, 57, 63
code vivant, 8, 10, 57, 63
cohérence globale, 57
COLLECTION, 4, 85, 90, 92
commentaire, 19, 20, 22
COMPARABLE, 90
compilation à la demande, 61
compilation dynamique, 63, 64
compilation séparée, 64
`compile`, 4
`compile_to_c`, 4, 10
`compile_to_jvm`, 4
comptage de références, 14
conception par contrats, *voir* assertions
*confined types*, 29
contexte de compilation, 58
*copying garbage collection*, 14
Coucaud, *voir* Philippe Coucaud
`Current`, 10

développement incrémental, 14, 63
Delphi, 56
*Design By Contract (DBC)*, 28, 33
développement incrémental, 7
DICTIONARY, 4, 32, 34, 85
Disney, 42
*dispose*, 13
droits d'auteurs, 40
duplication de code, 57

édition structurelle, 2
Eiffel, v, 1, 4, 5, 13, 14, 60, 61
EIFFEL_PARSER, 19, 21, 22, 25, 34
*ensure*, 33
Eric Bezault, 23
Eugen System, 56
évaluation partielle, 10
exception, 65
export sélectif, 34, 38

FAQ, 5
fermeture, 14, 62
*final*, 11
finalisation, 13
*finalize*, 13
*flow sensitive analysis*, 63
Foire Aux Questions, 5
fonction *once*, 31
fournisseur, 33
fournisseur d'alias, 31, 34
FPGA, 66, 69
Francis Scheid, 7
FRUIT, 65

généricité, 12
Gépi, 3
*garbage collector*, 13
Gelex, 23, 24
GENERAL, 86
Georges Bizet, 27
gestion mémoire, 13, *voir* ramasse-miettes
Geyacc, 23, 24
Gigi, *voir* Luigi Liquori
GNU Ada, 17, 25, 59
GNU Eiffel, *voir* SmallEiffel
Gobo, 23, *voir* Eric Bezault
Gocad, i

héritage multiple, 12
*html*, 4

identifiant de type, 9, 58
ilôt, 28
inférence de type, 10, 57
*inlining*, 2, 9, 13
*inlining*, 58
introspection, 14, 62
invariant, 29, 32
Isaac Asimov, 55
ISE, 56
`is_equal`, 30, 36
*island*, 28

itérateur, 4, 43
ITERATOR, 4, 43, 86

Java, 4, 5, 62, 64, *voir* langage ordinaire
Jean-Marc Jézéquel, i, 41, 82
jeux vidéo, 56

Kay, Alan, 1

langage de classes, 64
langage ordinaire, 8
LaTeX, 4
Le_Lisp de l'INRIA, 3
lex, v, 5, 19–20, 23, 25
liaison dynamique, 2, 9, 58, 63
Liquori, *voir* Luigi Liquori
Lisa, 3
Lisp, 3
LL(1), 21, 75
LL(k), 17
LOCAL_NAME, 33
localité, 58
logiciel libre, 60
LR(k), 17
LTR3, 3
Luigi Liquori, 64

*mark-sweep*, 13
marquage-balayage, 13
Match-O, 50, 66
MEMORY, 13, 85, 86
Meyer, Bertrand, 1, 8, 57
micro-contrôleur synthétisable, *voir* FPGA
microprocesseur, 58
modificateur, 34
multiméthodes, 64
MVC, 42

NATIVE_ARRAY, 4, 89
NONE, 32
NUMBER, 4, 86

Olivier Zendra, i
*once* (routine), 31
optimisation, *voir* aliasing

PEACH, 65
Philippe Coucaud, i
POINTER, 5, 90
pointeur unique, 29
polymorphisme paramètrique, 64
portabilité, 1, 5, 61, 62, 64
position, *voir* POSITION
POSITION, 18, 22, 33
post-condition, 33
pré-condition, 33
prédiction de type, 2, 10, 57, 58, 61, 63
`pretty`, 3, 4
`print_jvm_class`, 4
profil d'exécution, 63
programmation par contrats, 1, 28, 33
ProMic, v, 18, 66, 69, 75

racine du système, 8
ramasse-miettes, i, 2, 13, 58, 61
receveur, 58, 64
recompilation incrémentale, 64
ReDraw, 4, 42
*reference counting*, 14
*require*, 33

sécurité du typage, 64
Scheid, Francis, 7
*scheme*, 21, 75
Self, 8, 64
SET, 85
`short`, 3, 4
*Singleton*, 25, 28, 31
SmallEiffel, 4–5, 57, 60, 64
SmallEiffel (naissance), 93
Small, 4
Smalltalk, 4, 9, 42
spécialisation de code, 2, 10, 57, 61, 63
*squeak* Smalltalk, 42
STL, 4
STRING, 11, 34
STRING, 61
STRING_ALIASER, 31, 32
symbole directeur, 21, 75
*system level validity*, 12, 14
*system level validity*, 64
système fermé, 7, 14

table de fonctions, 58
taille de code, 63
taille des exécutables, 61
terminal alphanumérique, 2
TeX, 4
*Tuple*, 14, 62

typage (sécurité), 64
type ballon, 28
type du receveur, 58
type dynamique, 9, 10, 58
type feuille, 11
type vivant, 9
types confinés, 29

validation (tests), 61
version (numéro de), 85
VFT, 9, 12, 55, 59
Virtual Functions Table, *voir* VFT
vitesse de compilation, 61, 63
Void, 10, 11

*Walt Disney*, 42

yacc, v, 5, 19, 23, 25

Zendra, *voir* Olivier Zendra