Université d'Évry Val d'Essonne

Mémoire d'Habilitation à diriger des recherches

Spécialité informatique

Sandrine Blazy

Sémantiques formelles

Soutenu le jeudi 23 octobre 2008 devant le jury :

Rapporteurs :

M. **Andrea Asperti**, professeur, Università di Bologna
M. **Roberto Di Cosmo**, professeur des universités, Université Paris 7
Mme. **Christine Paulin**, professeur des universités, Université Paris-Sud

Examinateurs :

M. **Andrew W. Appel**, professeur, Princeton University
Mme. **Catherine Dubois**, professeur des universités, ENSIIE
M. **Xavier Leroy**, directeur de recherches, INRIA Paris-Rocquencourt

À Léana et Valentin

À Jean Paul

Remerciements

Je remercie Christine Paulin ainsi qu'Andrea Asperti et Roberto Di Cosmo, qui ont bien voulu juger mes travaux et y consacrer du temps malgré leurs nombreuses obligations.

Je remercie de tout cœur Catherine Dubois, qui a su contribuer à l'aboutissement de ce travail. C'est grace à Catherine que je me suis replongée dans l'étude de la sémantique formelle des langages de programmation et que j'ai découvert l'assistant à la preuve Coq, après une année transitoire de conversion à la bioinformatique suite au décès de Philippe Facon.

Je suis infiniment redevable à Xavier Leroy. À son contact, au cours de nos travaux communs et de nos discussions quotidiennes, j'ai véritablement appréhendé la conception formelle orientée par la preuve. Sa disponibilité quasi-permanente m'a de plus permis de mettre à profit mon expérience précédente en sémantique formelle.

J'adresse un merci tout particulier à Andrew Appel, pour sa confiance et son soutien. Je lui témoigne toute ma gratitude pour avoir lu ce manuscrit dans une langue qui n'est pas la sienne.

Les travaux présentés dans ce mémoire sont pour un bon nombre d'entre eux des travaux menés en collaboration avec d'autres chercheurs. Je remercie Yves Bertot, Marc Frappier, Frédéric Gervais, Thérèse Hardin, Régine Laleau, Pierre Letouzey, Laurence Rideau, Marc Shapiro, Éric Soutif, Véronique Viguié Donzeau-Gouge pour les discussions et échanges enrichissants que nous avons eus ensemble. Je remercie également les étudiants que j'ai eu le plaisir d'encadrer.

Je remercie chaleureusement les membres du projet Gallium, en particulier Zaynah Dargaye, Damien Doligez, François Pottier, Didier Remy et Boris Yakobowski pour leur aide, leur soutien et leur amitié.

Un grand merci aussi à tous les membres de l'équipe CPR du laboratoire CEDRIC, ainsi qu'à Marie-Christine Costa, directrice du laboratoire CEDRIC.

Je tiens à remercier mes collègues de l'ENSIIE pour leur bienveillance et

leur gentillesse, en particulier Gérard Berthelot, Florent Chavand, Brigitte Grau et Mireille Jouve.

Enfin, je remercie Alain, Bernadette, Léana, Valentin et Jean Paul pour leurs encouragements continus ainsi que pour leur patience et leur aide durant ces derniers mois.

Table des matières

Table des matières v

Table des figures 1

1 Introduction 3

2 Introduction à la sémantique des langages de programmation 9

Ce chapitre décrit les prérequis.

2.1 Sémantiques formelles 10
2.1.1 Jugement d'évaluation 11
2.1.2 Sémantique opérationnelle 12
2.1.3 Sémantique axiomatique 14
2.1.4 Sémantique dénotationnelle 17
2.2 Correction des transformations de programmes 17
2.2.1 Équivalence observationnelle 18
2.2.2 Vérification formelle 20
2.3 Outils pour formaliser des sémantiques 20
2.3.1 Centaur 20
2.3.2 Coq 21

3 Évaluation partielle de programmes Fortran 90 23

Ce chapitre commente une partie du matériau publié dans [1, 2, 3, 4, 5, 6]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Philippe Facon. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Frédéric Paumier, Hubert Parisot, Nathalie Dubois, Pousith Sayarath et Romain Vassallo [stage 13, stage 14, stage15]. Ces références sont celles de la liste détaillée des étudiants que j'ai encadrés (page 117).

3.1 Positionnement par rapport à mes travaux de thèse de doctorat 24
3.2 Le langage Fortran 90 27
3.2.1 Particularités 27
3.2.2 Sémantique formelle 29
3.3 Une stratégie d'évaluation partielle « on-line » 31
3.4 Une première expérience de vérification formelle en Coq . . . 32
3.5 Bilan 34

4 Le compilateur certifié CompCert 37

5 Un front-end pour le compilateur CompCert 43

Ce chapitre commente une partie du matériau publié dans [7, 8, 9]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Xavier Leroy. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Zaynah Dargaye et Thomas Moniot [stage 4, stage 6].

5.1 CIL 44
5.2 Syntaxe abstraite de Clight 45
5.3 Sémantique formelle de Clight 47
5.4 Bilan sur Clight 52
5.5 Cminor 53
5.5.1 Formalisation en Coq de la sémantique de Cminor . . . 53
5.5.2 Un front-end pour Compcert 55

6 Un modèle mémoire pour le compilateur CompCert 61

Ce chapitre commente une partie du matériau publié dans [10, 11]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Xavier Leroy. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Thibaud Tourneur et François Armand [stage 7, stage 9].

6.1 Généricité 64
6.2 Propriétés requises pour décrire la sémantique de C 64
6.2.1 Séparation 64
6.2.2 Adjacence 65
6.2.3 Confinement 66
6.2.4 Propriétés impliquant les opérations d'accès à la mémoire 67
6.3 Propriétés relatives aux transformations de la mémoire 69
6.4 Modèle concret 70
6.5 Bilan 71

7 Quelles sémantiques pour la preuve de programmes ? 73

Ce chapitre commente une partie du matériau publié dans [12, 13]. Sauf mention contraire, les travaux présentés dans ce chapitre ont été effectués en collaboration avec Andrew W.Appel. Sur les thèmes présentés dans ce chapitre, j'ai encadré le stage de M2 de Sonia Ksouri ([stage 5], sur le thème de la logique du « rely guarantee ») et piloté le post-doctorat de Keiko Nakata (sur le thème de la logique de séparation).

7.1 Vérification formelle de programmes en logique de séparation 73
7.2 Une sémantique à petits pas pour Cminor 75
7.2.1 Concurrent Cminor 76
7.2.2 De CompCert à Concurrent Cminor 77
7.2.3 Quelles continuations pour Cminor ? 79
7.2.4 Une sémantique à continuations pour Cminor 80
7.3 Une logique de séparation pour Cminor 82
7.3.1 Langage d'assertions 82
7.3.2 Sémantique axiomatique 83
7.3.3 De la sémantique à continuations à la sémantique axiomatique 83
7.3.4 Des tactiques pour la logique de séparation 85
7.4 Une autre expérience en preuve de programmes 86
7.5 Bilan 88

8 Autres expériences 91
8.1 Allocation de registres 91
8.2 Un langage pour la réutilisation de composants de spécifications 94
8.3 Un langage pour la description de glossaires 95

9 Conclusion 99

Bibliographie 103

10 Curriculum vitae 121

Table des figures

2.1 Exemple de règles de sémantique opérationnelle 13
2.2 Équivalence entre les styles à grands pas et à petits pas 14
2.3 Exemple de règles de sémantique axiomatique 16
2.4 Exemple de règles de sémantique dénotationnelle 17
2.5 Diagramme commutatif exprimant la propriété de préservation sémantique 19
2.6 Diagramme commutatif (avec état) 19

3.1 Évaluation partielle 23
3.2 Un exemple d'évaluation partielle de programme. 26
3.3 Sémantique dynamique de Fortran 90 30
3.4 Évaluation partielle d'une instruction i 32

4.1 Compilation 37
4.2 Architecture du compilateur CompCert 39
4.3 Le langage intermédiaire Cminor 40
4.4 Le front-end et le back-end du compilateur CompCert 42

5.1 Syntaxe abstraite de Clight (types et expressions). a^* dénote 0, 1 ou plusieurs occurrences de la catégorie syntaxique a. 46
5.2 Syntaxe abstraite de Clight (instructions, fonctions et programmes). $a^?$ dénote une occurrence optionnelle de la catégorie a. 48
5.3 Valeurs, sorties d'instructions et environnement d'évaluation . . 49
5.4 Jugements de la sémantique de Clight. 50
5.5 Exemples de règles de sémantique de Clight 51
5.6 Jugements de la sémantique de Cminor. 56
5.7 Traduction de Clight à Csharpminor 57
5.8 Préservation sémantique de la traduction de Clight vers Csharpminor 58
5.9 Traduction de Csharpminor à Cminor 58

6.1 La mémoire d'un compilateur vue depuis un processus 62
6.2 Un exemple de bloc de mémoire 66
6.3 Spécification abstraite la mémoire (extrait). Le type `option` et la notation $\lfloor x \rfloor$ sont définis dans le chapitre 2, page 21. 68
6.4 Transformations de la mémoire durant la compilation 69
6.5 Extrait de l'implémentation de la mémoire. 70

7.1 Preuve de programme et compilation certifiée 74
7.2 Règle d'encadrement de la logique de séparation 75
7.3 Extrait de la sémantique à continuations de Cminor 81
7.4 Extrait de la sémantique axiomatique de Cminor 84
7.5 Chronologie des différentes sémantiques de Cminor 89
7.6 De CompCert à Concurrent Cminor 90

1 Introduction

Une méthode formelle fournit une notation mathématique pour décrire précisément le comportement attendu d'un programme (i.e. sa spécification formelle). La *vérification formelle* d'un programme est l'utilisation de règles précises pour démontrer mathématiquement à l'aide d'un assistant à la preuve que ce programme satisfait une spécification formelle. Les programmes dont il est question dans ce mémoire sont des transformations de programmes. Une transformation de programmes opère à langage constant (par exemple *l'évaluation partielle* de programmes, qui peut être vue comme une optimisation de programmes), ou bien, plus généralement, traduit un langage vers un autre (par exemple, la *compilation* d'un programme).

Les progrès des assistants à la preuve, ainsi que de l'ingéniérie de la preuve rendent désormais possible la vérification formelle de systèmes de plus en plus complexes. L'engouement pour la vérification formelle de transformations de programmes effectuées par des compilateurs témoigne de ces progrès [14, 15, 16, 17, 18, 19]. La complexité d'une transformation de programmes formellement vérifiée provient à la fois du langage sur lequel la transformation opère, et de la transformation en elle-même. La vérification formelle d'une transformation de programme établit la correction de cette transformation. Cette preuve de correction repose sur l'existence de sémantiques formelles décrivant les langages source et cible de la transformation. Dans ce contexte, la correction signifie la préservation de la sémantique des programmes transformés.

La *sémantique formelle* d'un langage de programmation permet de définir un modèle mathématique du sens de tout programme écrit dans ce langage. Les sémantiques formelles étudiées dans ce mémoire modélisent l'exécution des programmes. Ce sont des *sémantiques opérationnelles.* Dans une sémantique formelle, un programme est représenté par un *arbre de syntaxe abstraite* (i.e. un terme algébrique). La phase préliminaire d'analyse syntaxique, transformant un texte de programme en un arbre de syntaxe abstraite (conformément aux règles d'une grammaire définissant une syntaxe concrète) est donc sup-

posée déjà effectuée, et elle ne fait donc pas partie de la sémantique formelle. Il en est de même pour la phase ultime reconstruisant à l'inverse un texte de programme à partir d'un arbre de syntaxe abstraite.

Une première difficulté dans la définition d'une sémantique formelle est de choisir la *frontière entre syntaxe et sémantique.* D'une façon générale, les syntaxes abstraites des différents langages dont il est question dans ce mémoire sont assez fidèles aux syntaxes concrètes dont elles sont issues.

Une sémantique formelle est définie par un ensemble de règles d'inférence décrivant plus ou moins précisément des événements qui se produisent durant l'exécution d'un programme. Dans une sémantique formelle, ces événements sont représentés par des calculs exacts (i.e. par opposition à des calculs approchés effectués par exemple par une analyse statique afin de découvrir lors de la compilation par exemple, certaines erreurs dans un programme). Il existe également des sémantiques abstraites utilisées en interprétation abstraite, et qui abstraient des calculs effectués par les programmes (par exemple toute valeur numérique est abstraite en son signe) afin de rendre possible des analyses statiques de programmes. Dans ce mémoire, seules des sémantiques formelles dites concrètes (c'est-à-dire opérant des calculs exacts) sont considérées.

Définir une sémantique formelle nécessite également de choisir une des deux techniques de *plongement* du langage étudié dans le métalangage employé pour définir la sémantique formelle. Un plongement superficiel est plus simple car il réutilise la syntaxe et les concepts du métalangage. Par contre, il n'est pas adapté à l'étude de propriétés générales du langage. Un plongement profond permet de raisonner par induction sur les programmes, mais nécessite de définir la syntaxe du langage comme un type de données du métalangage. La plupart des formalisations présentées dans ce mémoire consistent en un plongement profond.

Une deuxième difficulté dans la définition d'une sémantique réside dans le choix de la *précision des événements observés* durant l'exécution d'un programme. Au minimum, les valeurs finales des variables d'un programme doivent être observées. La tentation est grande d'observer davantage d'événements. Par exemple, le graphe des appels d'un programme, la trace des accès à la mémoire ou encore la trace des entrées et sorties du programme peuvent être observés. Le choix de la précision résulte d'un compromis entre d'une part la confiance gagnée en observant davantage d'événements dans les sémantiques formelles, et d'autre part les modifications autorisées par les transformations étudiées.

Disposer d'un cadre formel pour spécifier des transformations de programmes permet d'établir des propriétés de *préservation sémantique.* Par exemple, si

un programme dont la sémantique est bien définie (c'est-à-dire conforme à la sémantique formelle du langage dans lequel est écrit ce programme) est transformé en un programme dont la sémantique est également bien définie, alors les deux programmes sont équivalents modulo équivalence observationnelle. Plus précisément, deux programmes sont considérés comme équivalents lorsqu'il n'existe pas de différence observable entre leurs exécutions respectives.

En outre, une troisième difficulté dans la définition d'une sémantique réside dans le *choix d'un style de sémantique*. Il existe principalement trois familles de sémantiques formelles : sémantique opérationnelle (à petits pas ou à grands pas), sémantique dénotationnelle et sémantique axiomatique. Les critères de choix d'un style sont variés et parfois antagonistes : simplicité de la sémantique, facilité à raisonner sur la sémantique, modélisation des programmes dont l'exécution ne termine pas, ou plus généralement précision des événements observés. Ce mémoire présente différentes expériences d'utilisation de ces styles.

D'autre part, les langages dont il est question dans ce mémoire sont des langages impératifs (à l'exception d'un langage déclaratif étudié lors d'une expérience isolée). La sémantique d'un langage impératif repose sur la modélisation d'un *état* qui est modifié au cours de l'exécution séquentielle des instructions du programme. Il est donc nécessaire de modéliser ces états ainsi que plus généralement les environnements d'exécution des programmes. Quel que soit le style de sémantique choisi, cette modélisation (i.e. le choix des structures de données définissant les environnements d'exécution) est rarement prise en charge par le formalisme associé au style de sémantique choisi.

Aussi, une quatrième difficulté dans la définition d'une sémantique d'un langage impératif réside donc dans la *modélisation des environnements* d'exécution des programmes. Ce mémoire propose deux modélisations dédiées à deux langages de programmation utilisés dans l'industrie. La première définit les environnements permettant l'évaluation partielle de programmes Fortran 90. La seconde définit un modèle assez complet de la mémoire utilisée par l'ensemble des langages d'un compilateur du langage C.

Enfin, définir la sémantique formelle d'un langage de programmation nécessite de disposer d'outils assistant la conception de cette sémantique, c'est-à-dire d'une part permettant de tester et exécuter les sémantiques formalisées et d'autre part de vérifier formellement des propriétés de préservation sémantique. Les sémantiques dont il est question dans ce mémoire ont ainsi été définies à l'aide de l'un des deux outils Centaur et Coq, présentés dans le chapitre suivant.

Une sémantique formelle décrit l'exécution d'un programme et définit donc

implicitement un interprète du langage considéré. Lors de la conception d'un langage, disposer d'un interprète permet de tester la sémantique de ce langage sur des exemples de programmes. Le premier outil présenté, l'environnement Centaur a été développé dans les années quatre-vingt et quatre-vingt-dix. Il proposait un langage permettant de définir des sémantiques formelles (dans un style à grands pas) à l'aide de règles d'inférence qui étaient traduites en prédicats Prolog, ainsi qu'un environnement de débogage des règles de sémantique.

Le second outil dont il est question dans ce mémoire est l'assistant à la preuve Coq. Définir une sémantique formelle rend possible la preuve de propriétés sur cette sémantique (par exemple des propriétés de déterminisme dans l'exécution d'une instruction, des propriétés d'équivalence entre différentes constructions syntaxiques, ou encore l'absence de définitions multiples d'une même entité d'un programme écrit dans un langage déclaratif). La facilité de preuve de ces propriétés dépend de la précision de la sémantique considérée.

Ce mémoire décrit les travaux que j'ai réalisés dans le cadre du projet CompCert portant sur la vérification formelle en Coq d'un compilateur du langage C. De nombreuses sémantiques formelles ont été étudiées dans ce projet. La vérification formelle en Coq de propriétés sémantiques a eu un impact sur la définition de ces sémantiques formelles. Ainsi, il a parfois été nécessaire de contraindre davantage une sémantique afin de faciliter ensuite la preuve de certaines de ses propriétés. Le principal critère de choix d'une sémantique a été la facilité à raisonner sur celle-ci. En effet, selon le choix, le principe d'induction associé à une sémantique est plus ou moins difficile à formuler, et les étapes de preuve associées sont plus ou moins difficiles à découvrir.

La dernière difficulté rencontrée pour définir des sémantiques formelles concerne la *validation* de ces sémantiques. Comme beaucoup de langages de programmation, les langages étudiés ne disposent pas de spécifications formelles précises. Leurs manuels de référence et standards sont rédigés en anglais et comportent des imprécisions, parfois volontaires (car dépendant par exemple de l'architecture de la machine sur laquelle sera compilé le programme). Par exemple, le standard C ISO précise que certains comportements de programmes sont indéfinis. C'est la diversité des propriétés sémantiques ayant été formellement vérifiées qui atteste de la validité de cette sémantique. Les propriétés dont il est question dans ce mémoire concernent principalement l'équivalence entre différents styles de sémantique, ainsi que la préservation sémantique de diverses transformations de programme.

Ce mémoire présente plusieurs définitions de sémantiques formelles et de

transformations de programmes, et expose les choix de conception répondant aux difficultés précédemment énoncées. Mon propos est de présenter ces formalisations de façon assez générale, en complément des articles déjà publiés sur ces travaux et joints en annexe de ce mémoire. J'adopte donc une présentation historique de ces expériences, au détriment parfois d'une présentation rigoureuse de résultats très détaillés.

Le chapitre 2 présente le cadre général d'étude des sémantiques formelles. Il passe en revue différents styles de sémantique employés dans les chapitres suivants et détaille le cadre théorique de la preuve de propriétés de préservation sémantique de transformations de programmes. Il introduit également l'environnement Centaur et l'assistant à la preuve Coq, qui ont été utilisés pour formaliser les sémantiques décrites dans ce mémoire.

Le chapitre 3 décrit une transformation de programmes inspirée de l'évaluation partielle et dédiée à la compréhension de programmes scientifiques écrits en Fortran. Il définit d'abord une sémantique formelle à grands pas d'un vaste sous-ensemble du langage Fortran 90. Ensuite, il précise quelle évaluation partielle a été considérée. Ces travaux sont le prolongement de mes travaux de doctorat. Un prototype d'environnement d'aide à la compréhension de programmes a été développé grâce à l'environnement Centaur. Enfin, ce chapitre décrit une première vérification formelle en Coq concernant l'évaluation partielle de programmes.

Cette première expérience a été mise à profit dans le projet CompCert, un projet différent et plus ambitieux, concernant la vérification formelle en Coq d'un compilateur réaliste du langage C, utilisable dans le domaine embarqué. Ce projet est présenté dans le chapitre 4. Les chapitres 5 et 6 détaillent les deux parties de ce projet auxquelles j'ai participé. Le chapitre 5 précise la sémantique formelle d'un vaste sous-ensemble du langage C ainsi que la sémantique formelle du principal langage intermédiaire du compilateur. Il introduit brièvement la vérification formelle de la traduction vers ce langage intermédiaire. Le chapitre 6 détaille le modèle mémoire qui est commun à tous les langages du compilateur. Disposer d'un modèle mémoire commun à plusieurs langages a nécessité de définir un modèle générique et un niveau d'abstraction adapté.

Le chapitre 7 décrit une formalisation en Coq d'une logique de séparation, un formalisme récent adapté à la preuve de propriétés relatives aux structures de données de type pointeur. Partant d'une sémantique définie dans le cadre du précédent projet, il a été nécessaire d'adopter un autre style de sémantique et de prouver des propriétés de cette sémantique. De plus, un langage d'assertions généraliste a été défini, ainsi que des tactiques dédiées à la preuve en logique de séparation.

Le chapitre 8 décrit trois expériences particulières. La première s'inscrit dans le cadre du projet CompCert. Elle étudie une formalisation de l'allocation de registres fondée sur des techniques d'optimisation combinatoire. La deuxième concerne la spécification en langage B de réutilisation de composants de spécifications formelles. La dernière est une expérience de sémantique formelle d'un langage déclaratif permettant la définition de glossaires de termes bancaires.

Enfin, le chapitre 9 présente des perspectives d'évolution des travaux décrits.

2 Introduction à la sémantique des langages de programmation

Ce chapitre décrit les prérequis.

Un langage impératif est un langage de programmation constitué de différentes catégories syntaxiques. Ce sont principalement les déclarations (de variables, de types, de fonctions), les expressions, les instructions et les fonctions (ou procédures). Lorsqu'un programme impératif est exécuté, des opérations (*i.e.* des instructions et éventuellement des expressions, selon le langage considéré) modifient l'état du programme, c'est-à-dire les valeurs de certaines variables (ou plus généralement de valeurs gauches) du programme, stockées dans une mémoire ou dans un environnement. Le caractère impératif du langage provient de l'exécution en séquence des instructions du programme.

Définir une sémantique formelle d'un langage permet de comprendre le comportement des programmes écrits dans ce langage. Pour un langage impératif, cette définition repose sur la modélisation d'états. La sémantique formelle d'un langage définit l'exécution des programmes et exprime comment évolue l'état du programme pendant l'exécution de chaque élément syntaxique. L'effet de l'exécution diffère selon les langages considérés. La forme des exécutions diffère également selon le style de sémantique adopté.

Les transformations de programmes étudiées dans ce mémoire opèrent sur des sémantiques formelles. D'une façon générale, les sémantiques formelles étudiées ne sont pas dédiées aux transformations de programmes considérées. Ces sémantiques sont adaptées aux transformations de programmes, mais elles sont aussi relativement indépendantes de ces transformations. Cette indépendance est nécessaire pour des langages de grande taille tels que Fortran 90 et C, car elle facilite la conception à la fois des sémantiques formelles, mais aussi des transformations de programmes. Aussi, un programme dont la sémantique est définie ne sera pas nécessairement transformé. Ceci explique la forme des propriétés de préservation sémantique étudiées.

Définir une sémantique formelle permet également de raisonner sur le comportement d'un programme, dans le but d'établir des propriétés d'équivalence entre sémantiques formelles, ou plus généralement de préservation sémantique

de transformations de programmes.

Ce mémoire décrit plusieurs expériences de définitions de sémantiques formelles pour des langages impératifs de grande taille (Fortran 90, C et Cminor, le principal langage intermédiaire d'un compilateur C). Pour chaque expérience, ces sémantiques formelles ont été définies « sur machine », d'abord dans le but de mieux comprendre les langages étudiés, mais aussi pour pouvoir définir ensuite diverses transformations de programmes opérant sur ces sémantiques. La préservation sémantique de ces transformations a été prouvée (d'abord à la main pour les transformations de programmes les plus anciennes, puis avec l'assistant à la preuve Coq). Ces expériences ont également été l'occasion de tester différents styles de sémantique (principalement opérationnel à grands pas, opérationnel à petits pas et axiomatique, et aussi dans une moindre mesure dénotationnel) et de vérifier formellement des propriétés d'équivalence entre ces styles, augmentant ainsi la confiance en ces sémantiques.

Le but de ce chapitre est de fournir le cadre général permettant de comprendre les expériences détaillées dans ce mémoire. Il ne s'agit pas de présenter la théorie justifiant les choix effectués, mais plutôt de poser brièvement les briques théoriques (et notions générales) qui sont nécessaires à la compréhension des chapitres suivants.

La suite de ce chapitre est organisée comme suit. Je présente d'abord comment définir des sémantiques formelles pour des langages impératifs et comment vérifier formellement des équivalences sémantiques entre différents styles. Ensuite, j'explique comment vérifier formellement des propriétés de préservation sémantique de transformations de programmes. Enfin, j'introduis les outils Centaur et Coq que j'ai utilisés pour spécifier des sémantiques et des transformations de programmes. Les notions détaillées dans ce chapitre sont illustrées d'exemples empruntés aux expériences dont il est question dans ce mémoire.

2.1 Sémantiques formelles

Il existe principalement trois styles de sémantiques formelles [20]. Les sémantiques opérationnelles sont adaptées à la vérification formelle de propriétés sémantiques. Les sémantiques axiomatiques sont adaptées à la preuve de programmes. Les sémantiques dénotationnelles permettent de définir des sémantiques plus abstraites à l'aide de formalismes mathématiques. Chaque style a ses avantages et ses inconvénients, et des dépendances existent entre ces styles. Les sémantiques opérationnelles et axiomatiques sont davantage détaillées dans cette section car elles ont été davantage étudiées.

2.1.1 Jugement d'évaluation

La sémantique formelle opérationnelle d'un langage définit sous la forme de règles les effets de l'exécution de chaque élément syntaxique (c'est pourquoi cette sémantique est souvent qualifiée de « dirigée par la syntaxe »). Des environnements d'exécution apparaissent également dans ces règles. Les différentes formes que peuvent prendre ces règles sont appelées des *jugements d'évaluation*. La forme la plus générale des jugements d'évaluation d'un langage impératif est $\sigma \vdash synt \Rightarrow sem, \sigma'$. Ce jugement d'évaluation relie un état initial σ et un élément syntaxique *synt* au résultat de l'évaluation de cet élément syntaxique. Dans le cas le plus général, ce résultat consiste en un élément sémantique (i.e. une observation) *sem* et un nouvel état σ'. L'état σ' est omis lorsque l'évaluation ne modifie jamais σ. De même, *sem* peut être omis, par exemple dans le cas de l'exécution d'une instruction qui a pour seul effet observable de modifier l'état du programme.

L'état σ représente l'état du programme (en particulier les valeurs qui sont calculées durant l'exécution du programme, et auxquelles il est possible d'accéder pendant l'exécution de l'élément syntaxique considéré) ainsi qu'éventuellement des environnements prenant en compte des caractéristiques particulières du langage étudié. L'état du programme comprend au moins un *état mémoire* associant à chaque adresse en mémoire une valeur nécessitant d'être conservée en vue d'une utilisation future.

Pour des langages impératifs avec pointeurs, la correspondance entre identificateurs de variables et adresses en mémoire est modélisée par un environnement. Afin de modéliser les règles de portée des variables, cet environnement peut être partagé en deux, afin par exemple de gérer différemment les variables locales des variables globales. Ainsi, un jugement possible d'évaluation d'une expression C sans effet de bord est $G, E \vdash a \Rightarrow v$. Il signifie qu'étant donné un environnement global G et un environnement local E, l'évaluation d'une expression a fournit une valeur v et ne modifie pas les environnements G et E.

Une sémantique formelle comprend différents jugements d'évaluation. Par exemple, les sémantiques formelles étudiées dans ce mémoire comprennent un jugement d'évaluation des expressions, ainsi qu'un jugement d'évaluation (ou exécution) des instructions. Dans le cas du langage C, un jugement d'évaluation des expressions en position de valeur gauche a été également défini. Une expression en position de valeur gauche désigne tout élément syntaxique pouvant recevoir une valeur (c'est-à-dire apparaissant en partie gauche d'une affectation) : variable, champ d'un enregistrement, déréférencement de pointeur,

case de tableau. L'évaluation d'une expression en position de valeur gauche produit une adresse en mémoire qu'il est par exemple nécessaire de calculer lors de l'exécution d'une affectation.

La différence entre ces jugements d'évaluation réside dans les éléments écrits à droite du symbole $\Rightarrow$: nature de l'élément sémantique observé et partie de l'environnement ayant été modifiée par l'évaluation. Une façon de marquer la différence consiste à utiliser autant de symboles qu'il existe de formes de jugements d'évaluation.

Dans la sémantique formelle de C, il a souvent été choisi de ne pas trop contraindre la syntaxe abstraite (par exemple en évitant de définir une catégorie syntaxique représentant les valeurs gauches). La contrainte a ainsi été considérée dans la sémantique seulement (par exemple, en définissant un jugement d'évaluation d'une expression en position de valeur gauche). Cette démarche a été préférée à la démarche inverse, qui avait été adoptée lors de l'expérience antérieure de formalisation d'une sémantique formelle de Fortran 90. Le passage à la preuve formelle a également milité en faveur de ce choix.

Enfin, la nature de l'élément sémantique considéré dépend de la précision des événements observés durant l'évaluation. Ainsi, l'exécution d'une instruction C que nous présentons calcule non seulement les effets de cette exécution, mais également une trace d'événements (d'entrées et sorties) ayant eu lieu durant cette exécution.

2.1.2 Sémantique opérationnelle

Une sémantique opérationnelle définit inductivement une relation d'évaluation d'un programme. Cette relation décrit un système de transitions entre les différents états du programme. Selon la nature des transitions, la sémantique opérationnelle est dite à grands pas (ou encore naturelle [21]) ou à petits pas (ou encore opérationnelle structurée).

La figure 2.1 montre les règles d'évaluation de trois éléments syntaxiques (une expression avec opérateur binaire, une séquence d'instructions et une boucle infinie) dans les deux styles de sémantique. Dans une sémantique à grands pas, une transition (notée $\Rightarrow$ dans la figure 2.1) représente l'exécution d'un élément syntaxique. Dans une sémantique à petits pas, une transition (notée $\mapsto_1$ dans la figure 2.1) représente une étape élémentaire de calcul.

Une sémantique à petits pas comprend des règles de transition (par exemple les règles (4) et (5)) et des règles de réduction (par exemple la règle (6)). L'évaluation complète d'un élément syntaxique est une relation notée $\mapsto^*$, re-

présentée par une séquence de transitions, obtenue en calculant la fermeture réflexive et transitive de la relation $\mapsto_1$. La relation $\mapsto^*$ peut également compter le nombre n de pas d'exécution ; elle est alors notée $\mapsto^n$. La relation $\mapsto^*$ modélise aussi des exécutions vers des *états bloquants* (i.e. à partir desquels aucune transition n'est possible), et donc en particulier des programmes dont l'exécution ne termine pas, mais aussi des états qui ne sont jamais atteints à partir d'états initiaux (et qui n'ont pas leur équivalent dans une sémantique à grands pas). Un état qui n'est pas bloquant est dit *sûr*. L'exécution d'un programme est sûre lorsque partant d'un état sûr, l'exécution du programme termine dans un état sûr. L'exécution sûre d'un programme entier réduit ce dernier en l'instruction vide `Skip`.

Sémantique à grands pas
(jugements $\Sigma \vdash \mathbf{Exp} \Rightarrow \mathbf{Val}$ et $\Sigma \vdash \mathbf{Inst} \Rightarrow \Sigma$) :

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2 \quad \texttt{eval_op_bin}(op, v_1, v_2) = v}{\sigma \vdash e_1\; op\, e_2 \Rightarrow v} \; (1)$$

$$\frac{\sigma \vdash i_1 \Rightarrow \sigma_1 \quad \sigma_1 \vdash i_2 \Rightarrow \sigma'}{\sigma \vdash i_1; i_2 \Rightarrow \sigma'} \; (2) \qquad \frac{\sigma \vdash i \Rightarrow \sigma_1 \quad \sigma_1 \vdash \texttt{loop}(i) \Rightarrow \sigma'}{\sigma \vdash \texttt{loop}(i) \Rightarrow \sigma'} \; (3)$$

Sémantique à petits pas
(jugement $\langle \mathbf{Exp}, \Sigma\rangle \mapsto_1 \langle \mathbf{Exp} \mid \mathbf{Val}, \Sigma\rangle$) :

$$\frac{\langle e_1, \sigma\rangle \mapsto_1 \langle e_1', \sigma\rangle}{\langle e_1\; op\, e_2, \sigma\rangle \mapsto_1 \langle e_1'\; op\, e_2, \sigma\rangle} \; (4) \qquad \frac{\langle e_2, \sigma\rangle \mapsto_1 \langle e_2', \sigma\rangle}{\langle v_1\; op\, e_2, \sigma\rangle \mapsto_1 \langle v_1\; op\, e_2', \sigma\rangle} \; (5)$$

$$\frac{\texttt{eval_op_bin}(op, v_1, v_2) = v}{\langle v_1\; op\, v_2, \sigma\rangle \mapsto_1 \langle v, \sigma\rangle} \; (6) \qquad \frac{\langle i_1, \sigma\rangle \mapsto_1 \langle i_1', \sigma_1\rangle}{\langle i_1; i_2, \sigma\rangle \mapsto_1 \langle i_1'; i_2, \sigma_1\rangle} \; (7)$$

$$\langle \texttt{Skip}; i_2, \sigma\rangle \mapsto_1 \langle i_2, \sigma\rangle \; (8) \qquad \langle \texttt{loop}(i), \sigma\rangle \mapsto_1 \langle i; \texttt{loop}(i), \sigma\rangle \; (9)$$

Fig. 2.1 – Exemple de règles de sémantique opérationnelle

Une sémantique à grands pas relie un programme à son résultat final sans préciser les étapes de calcul qui ont amené à ce résultat. Elle ne permet pas d'observer des programmes dont l'exécution ne termine pas. Les sémantiques à grands pas étant plus simples, elles sont donc souvent choisies afin de raisonner sur des programmes écrits dans des langages complexes tels que C.

Par contre, une sémantique à petits pas détaille tous les états intermédiaires

de l'exécution d'un programme et permet des observations plus précises. Les sémantiques à petits pas permettent donc d'établir des résultats d'équivalence plus forts. Par contre, elles exposent trop d'étapes de calculs, ce qui peut compliquer énormément les preuves de propriétés sémantiques. En effet, la preuve d'équivalence sémantique entre un programme p et un programme transformé p_t peut être rendue difficile par le fait qu'un état intermédiaire de l'exécution de p n'a pas nécessairement d'équivalent dans l'exécution de p_t.

Toute évaluation à grands pas d'un élément syntaxique s peut être interprétée comme étant une suite d'évaluations à petits pas de s. Réciproquement, tout suite d'évaluations à petits pas d'un élément syntaxique s menant à un état sûr peut être interprétée comme étant une évaluation à grands pas de s. Ce résultat d'équivalence sémantique entre les styles à grands pas et à petits pas est détaillé dans la figure 2.2 pour l'exécution des instructions.

Théorème. $\forall \sigma, i, \sigma', \;\; \sigma \vdash i \Rightarrow \sigma' \quad \Leftrightarrow \quad \langle i, \sigma \rangle \mapsto^* \langle \texttt{Skip}, \sigma' \rangle$

Schéma de la preuve :

1. $\forall \sigma, i, \sigma', \;\; \sigma \vdash i \Rightarrow \sigma' \quad \Rightarrow \quad \langle i, \sigma \rangle \mapsto^* \langle \texttt{Skip}, \sigma' \rangle$
2. $\forall \sigma, i, \sigma', \;\; \langle i, \sigma \rangle \mapsto_1 \langle i_1, \sigma_1 \rangle \;\wedge\; \sigma_1 \vdash i_1 \Rightarrow \sigma' \quad \Rightarrow \quad \sigma \vdash i \Rightarrow \sigma'$
3. $\forall \sigma, i, \sigma', \;\; \langle i, \sigma \rangle \mapsto^* \langle \texttt{Skip}, \sigma' \rangle \quad \Rightarrow \quad \sigma \vdash i \Rightarrow \sigma'$

FIG. 2.2 – Équivalence entre les styles à grands pas et à petits pas

2.1.3 Sémantique axiomatique

La sémantique axiomatique fournit un style adapté à la preuve de programmes. Prouver un programme consiste à le spécifier au moyen d'assertions écrites à l'aide de formules logiques et à établir ensuite que le programme satisfait sa spécification, c'est-à-dire que la spécification et le programme respectent les règles d'une sémantique axiomatique définissant les exécutions valides de chaque instruction du langage source. La technique employée pour prouver un programme annoté (par des assertions) réduit ce dernier en un ensemble de formules logiques (appelé conditions de vérification), ne faisant plus référence aux instructions du programme. Prouver un programme se ramène ainsi à vérifier la validité de formules logiques [22].

Des procédures de décision peuvent être utilisées pour décider de façon automatique de la validité des conditions de vérification d'un programme. Plus généralement, il existe des outils automatisant la preuve de programmes,

et produisant des obligations de preuve éventuellement déchargées vers des outils externes d'aide à la preuve. Les plus répandus concernent les annotations écrites en langage JML dans les programmes Java [23, 24, 25]. Un autre exemple est l'outil Caduceus de vérification de programmes C [26].

Dans un programme annoté, ainsi que dans une sémantique axiomatique, la forme générale des assertions est $\{P\}i\{Q\}$, signifiant que pour tout état σ satisfaisant la pré-condition P (notation $\sigma \models P$), si l'exécution de l'instruction i depuis l'état σ termine dans un état σ', alors σ' satisfait la post-condition Q (notation $\sigma' \models Q$). La description de l'évolution de l'état σ qui conférait à la sémantique précédente son caractère opérationnel a ici disparu.

Dans une sémantique axiomatique, les assertions $\{P\}i\{Q\}$ sont qualifiées d'assertions de correction partielle ou bien d'assertions de correction totale. Considérer des assertions de correction totale revient à modéliser des programmes dont l'exécution termine toujours. De telles assertions sont par exemple utilisées dans la méthode B. En sémantique axiomatique, considérer des assertions de correction partielle est plus répandu, car cela permet de s'affranchir des contraintes de terminaison des instructions. Le terme partiel signifie ici que l'exécution d'un programme peut ne pas terminer. La sémantique axiomatique que j'ai étudiée est celle d'un langage avec boucle infinie (*cf.* règle (11) de la figure 2.3), j'ai donc considéré seulement des assertions de correction partielle dans cette sémantique.

La signification d'une assertion $\{P\}i\{Q\}$ fait référence à l'exécution de l'instruction i, selon une sémantique opérationnelle. Plus précisément, une assertion $\{P\}i\{Q\}$ de correction partielle est *valide* par rapport à une sémantique par exemple à grands pas (notation $\models \{P\}i\{Q\}$) si et seulement si :

$$\forall \sigma, \sigma', \quad \textsf{si}\, \sigma \models P \textsf{ et } \sigma \vdash i \Rightarrow \sigma' \,\textsf{alors}\, \sigma' \models Q.$$

Prouver un programme directement à l'aide de cette définition n'est pas commode. Il est plus aisé d'utiliser les règles dites de Hoare (ou logique de Floyd-Hoare [27, 28]) qui facilitent la construction mécanisée de la preuve de validité d'une assertion. Les règles (10) et (11) de la figure 2.3 montrent les règles de Hoare pour les instructions des figures 2.1 et 2.3. C'est souvent l'ensemble de ces règles qui est appelé sémantique axiomatique. La règle de la boucle n'indique pas comment est exécutée la boucle ; elle utilise un invariant de boucle. Comme la boucle de la règle (11) est infinie, la post-condition de la boucle ne sera jamais satisfaite (et vaut faux dans la règle).

Les règles de Hoare considèrent un programme comme un transformateur de formules logiques portant entre autres sur l'état de la mémoire. L'évaluation des expressions ne fait pas partie de la sémantique axiomatique à proprement

$$\frac{\{P\}\, i_1\, \{R\} \quad \{R\}\, i_2\, \{Q\}}{\{P\}\, i_1; i_2\, \{Q\}}\ (10) \qquad \frac{\{\mathit{Inv}\}\, i\, \{\mathit{Inv}\}}{\{\mathit{Inv}\}\, \texttt{loop}(i)\, \{\mathbf{false}\}}\ (11)$$

$$\frac{P \Rightarrow P' \quad \{P'\}\, i\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, i\, \{Q\}}\ (12)$$

FIG. 2.3 – Exemple de règles de sémantique axiomatique

parler. Elle est utilisée dans les assertions et est en général formalisée dans un style opérationnel.

Les formules logiques permettent d'exprimer des propriétés attendues du programme. Différents formalismes logiques plus ou moins expressifs permettent de définir les assertions en logique du premier ordre. L'un des plus complets est celui de la logique de séparation, qui est dédiée aux langages avec pointeurs, et qui permet d'observer finement la mémoire et donc de décrire aisément des propriétés usuellement recherchées sur des pointeurs [29] : par exemple, le non-chevauchement (*i.e.* la séparation) entre zones de la mémoire, ou encore l'absence de cycle dans une structure de données chaînée.

Une sémantique axiomatique comprend également une règle de renforcement des pré-conditions, ainsi qu'une règle d'affaiblissement des post-conditions. Ces deux règles sont parfois rassemblées en un règle dite règle de conséquence (c.f. la règle (12) de la figure 2.3). L'ensemble des règles d'une sémantique axiomatique permet de vérifier aisément (*i.e.* de façon plus ou moins automatique) la validité d'une assertion donnée. Par contre, raisonner de façon générale sur l'exécution d'un programme est parfois difficile en sémantique axiomatique. Par exemple, la règle définissant le comportement d'une boucle fait référence à un invariant de boucle, mais n'indique pas comment trouver cet invariant.

Par ailleurs, une assertion $\{P\}i\{Q\}$ est prouvable (notation $\vdash \{P\}i\{Q\}$) s'il existe un arbre de preuve construit à partir des règles de Hoare. La cohérence (en anglais *soundness*) des règles établit qu'elles produisent des assertions valides, ce qui permet alors de conclure au final que toute assertion prouvable est valide (c'est-à-dire que si $\vdash \{P\}i\{Q\}$, alors $\models \{P\}i\{Q\}$).

Les règles de Hoare ne sont pas complètes. Il est cependant possible de définir une notion de complétude relative, non considérée dans ce mémoire.

2.1.4 Sémantique dénotationnelle

Une sémantique dénotationnelle associe une fonction à chaque élément syntaxique, représentant l'effet de cet élément sur l'exécution du programme. Il s'agit de définir d'une façon la plus abstraite possible la sémantique formelle d'un langage, indépendamment de l'implémentation de ce langage. Aussi, une sémantique dénotationnelle utilise des concepts mathématiques qui sont difficiles à manipuler dans un assistant à la preuve.

La figure 2.4 montre une version dénotationnelle de la sémantique donnée dans la figure 2.1. À chaque expression e est associée une fonction $\mathcal{E}[\![e]\!]$ évaluant cette expression (et appelée dénotation de e). De même, à chaque instruction i est associée une fonction $\mathcal{I}[\![i]\!]$. La règle (14) d'exécution d'une séquence d'instructions illustre le caractère compositionnel d'une sémantique dénotationnelle. La règle (15) définit le comportement d'une boucle comme étant la solution d'une équation de point fixe. Même si le théorème de Knaster-Tarski établit l'existence de ce point fixe, sa construction n'est pas aisée, rendant ainsi la sémantique dénotationnelle d'une boucle difficile à manipuler en tant que telle.

$\mathcal{E} : \mathbf{Exp} \rightarrow \Sigma \rightarrow \mathbf{Val}$ $\qquad$ $\mathcal{I} : \mathbf{Inst} \rightarrow \Sigma \rightarrow \Sigma$

$$\mathcal{E}[\![e_1 \; op \; e_2]\!]\sigma = \mathtt{eval_op_bin}(op, \mathcal{E}[\![e_1]\!]\sigma, \mathcal{E}[\![e_2]\!]\sigma,) \quad (13)$$

$$\mathcal{I}[\![i_1; i_2]\!] = \mathcal{I}[\![i_2]\!] \circ \mathcal{I}[\![i_1]\!] \quad (14) \qquad \mathcal{I}[\![\mathtt{loop}(i)]\!] = \mathit{fix}\, \mathcal{I}[\![i]\!] \quad (15)$$

FIG. 2.4 – Exemple de règles de sémantique dénotationnelle

L'équivalence entre les styles dénotationnel et opérationnel (par exemple à grands pas) est la propriété suivante (par exemple pour les instructions) :

$$\forall i, \sigma, \sigma', \quad \mathcal{I}[\![i]\!](\sigma) = \sigma' \Leftrightarrow \sigma \vdash i \Rightarrow \sigma'$$

La preuve de cette propriété se fait en deux étapes : la sémantique opérationnelle est correcte vis-à-vis de la dénotationnelle (sens de la gauche vers la droite), la sémantique dénotationnelle est adéquate par rapport à la sémantique opérationnelle (sens inverse).

2.2 Correction des transformations de programmes

Les transformations de programmes dont il est question dans ce mémoire préservent la sémantique des programmes initiaux. Une transformation de

programme préservant la sémantique désigne soit une transformation de programme sans changement de langage, soit une traduction de programme.

2.2.1 Équivalence observationnelle

Valider une transformation de programmes consiste à établir une propriété de préservation sémantique, c'est-à-dire à vérifier que tout programme transformé se comporte comme prescrit par la sémantique du programme source l'ayant engendré. La solution choisie dans ce mémoire consiste à abstraire le comportement d'un programme en son comportement observable, et considérer que deux programmes ont la même sémantique s'il n'y a pas de différence observable entre leur exécution. Les deux programmes sont ainsi équivalents s'ils effectuent les mêmes calculs observables. Il s'agit d'équivalence observationnelle [30].

La propriété exprimant l'équivalence observationnelle entre deux programmes est la suivante.

Propriété de préservation sémantique.
Pour tout programme source S,
si S est transformé en un programme cible C, sans signaler d'erreur,
et si S a une sémantique bien définie,
alors C à la même sémantique que S à équivalence observationnelle près.

D'après la propriété énoncée, la transformation de programme peut échouer, auquel cas rien n'est garanti. De même, un programme dont le comportement n'est pas défini peut être transformé en un programme dont le comportement est défini, auquel cas rien n'est garanti non plus. Cette indépendance entre sémantique et transformation de programme facilite grandement la vérification formelle de la propriété de préservation sémantique.

Dans mes expériences en évaluation partielle, les événements observés sont les valeurs des variables calculées lors de l'exécution d'un programme. Davantage d'événements sont observés dans mes expériences en compilation : les traces des entrées et sorties effectuées durant l'exécution d'un programme sont observées en plus de ces valeurs.

Lorsque les sémantiques sont définies selon un style opérationnel, prouver une propriété de préservation sémantique revient à prouver la commutativité du premier diagramme commutatif de la figure 2.5 [31]. Dans ce diagramme, S représente un programme source écrit dans un langage L et C un programme transformé écrit dans un langage L'. La relation $\simeq$ est une relation d'équivalence entre calculs observables (i.e. $\mathsf{Obs} \simeq \mathsf{Obs}'$ dans le diagramme). La commutativité du premier diagramme se décompose en deux propriétés de

complétude et de validité illustrées par les deux derniers diagrammes, dans lesquels le trait en pointillés indique la propriété à démontrer. La commutativité de chacun de ces deux diagrammes se montre par induction sur la relation d'évaluation concernée.

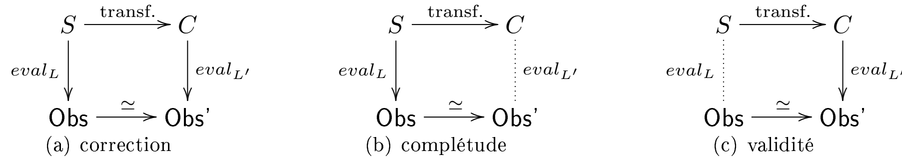


FIG. 2.5 – Diagramme commutatif exprimant la propriété de préservation sémantique

Aussi, la preuve de préservation de la sémantique se décompose en autant de diagrammes commutatifs (ou lemmes de simulation) qu'il y a de cas d'évaluation définis dans la relation d'évaluation. De plus, les états apparaissant dans la relation d'évaluation sont également représentés dans ces diagrammes. Par exemple, pour un jugement d'évaluation (pour simplifier, ce jugement est le même dans les langages L et L') de la forme $\sigma \vdash i \Rightarrow \mathsf{Obs}, \sigma'$, il est nécessaire de prouver pour chaque cas d'exécution d'une instruction i le diagramme de la figure 2.6.

D'une façon générale, dans les transformations étudiées, lorsque i est transformé en i', l'état σ est également transformé en σ'. Par exemple, les états de la mémoire changent car certaines variables ne sont pas traduites. De plus, comme l'évaluation considérée modifie les états, le diagramme commutatif exprime également que l'évaluation préserve (ou plus généralement étend) un invariant, noté $\sigma \rightarrowtail \sigma'$, reliant les états avant et après exécution des instructions. Cet invariant n'est pas nécessairement le même dans le langage source (c.f. $\sigma \rightarrowtail \sigma'$) et dans le langage cible (c.f. $\sigma_1 \rightarrowtail^{\cdot} \sigma_1'$).

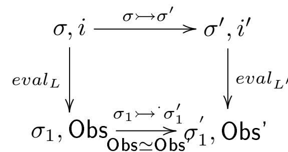


FIG. 2.6 – Diagramme commutatif (avec état)

2.2.2 Vérification formelle

Vérifier formellement une transformation de programme consiste à utiliser un assistant à la preuve pour prouver une propriété de préservation sémantique. Cette propriété peut être établie une fois pour toutes (i.e. quel que soit le programme considéré) ou bien au cas par cas, *a posteriori* pour chaque programme à transformer.

L'avantage de la première solution est que la preuve n'est faite qu'une seule fois, indépendamment du programme à transformer. C'est pourquoi cette solution a été choisie pour la plupart des transformations de programmes présentées dans ce mémoire. La seconde solution est intéressante lorsque la preuve à effectuer est beaucoup plus simple que la preuve à effectuer dans le cas de la première solution. Elle permet de plus de valider des transformations de programmes vues comme des boîtes noires (car par exemple, elles sont écrites dans un autre langage que le langage de spécifications). Cette solution a été choisie pour une transformation de programme présentée dans le chapitre 8.

2.3 Outils pour formaliser des sémantiques

Cette section introduit les deux outils Centaur et Coq que j'ai utilisés pour spécifier des sémantiques formelles et des transformations de programmes. Centaur a été développé avant Coq et il n'est guère utilisé aujourd'hui.

2.3.1 Centaur

Centaur est un environnement générique de programmation paramétré par une syntaxe et une sémantique d'un langage de programmation [32]. Cet environnement générique a été développé dans les années quatre-vingt à quatre-vingt-dix. J'ai réutilisé une syntaxe très complète de Fortran définie par la société Simulog (faisant aujourd'hui partie du groupe astek) développant des outils d'assurance qualité et de rétro-ingéniérie d'applications Fortran, utilisant Centaur [33]. Cette collaboration m'a permis de disposer d'un environnement Centaur/Fortran minimal, que j'ai amélioré pour mettre au point un environnement dédié à la compréhension de programmes Fortran (c.f. chapitre 3).

Dans Centaur, un programme est représenté par un arbre de syntaxe abstraite. Centaur propose un formalisme appelé Typol pour spécifier des sémantiques formelles selon un style à grands pas, et plus généralement des relations sémantiques concernant le langage étudié [32, 34]. Les spécifications Typol sont compilées par Centaur en Prolog. Lorsque ces spécifications sont

exécutées, Prolog sert de moteur du système de déduction. Ainsi, un interprète écrit en Prolog et implémentant les sémantiques ayant été spécifiées est généré par Centaur. Afin de faciliter l'écriture de sémantiques et de mettre en évidence les différents calculs sémantiques effectués lors de l'exécution des programmes, Centaur fournit de plus des bibliothèques écrites en Lisp dédiées à la construction d'environnements graphiques, ainsi qu'un débogueur Typol.

2.3.2 Coq

Coq est un assistant à la preuve fondé sur la Calcul des Constructions Inductives (CCI), une théorie des types d'ordre supérieur [35, 36]. Son langage de spécification est une forme de lambda-calcul typé. Le CCI comprend des constructions inductives [37]. Définir des prédicats inductifs en Coq (en particulier des relations sémantiques) se fait ainsi d'une manière naturelle. De plus, Coq engendre automatiquement le principe d'induction associé à une définition inductive et facilite ainsi le raisonnement par induction, qui est la principale technique de preuve de propriétés sémantiques des langages [38].

Une preuve en Coq est réalisée de façon interactive en exécutant des *tactiques* (i.e. des commandes) du langage de preuves. Coq fournit également des tacticiels définissant comment enchaîner les tactiques. Un langage de définition de tactiques permet de plus à l'utilisateur de Coq de définir ses propres tactiques.

En Coq, une sémantique peut également être définie selon un style fonctionnel, c'est-à-dire comme une fonction Coq. Une fonction Coq est nécessairement totale, et le moyen le plus courant pour représenter en Coq une fonction partielle f de type $A \rightarrow B$ est de définir une fonction totale de type $A \rightarrow \texttt{option}\, B$ associant à toute valeur a de type A soit la valeur $None$ (abrégée en $\perp$ dans ce mémoire) lorsque a n'a pas d'image par f, soit une valeur $\texttt{Some}(v)$ (abrégée en $\lfloor v \rfloor$ dans ce mémoire), avec v de type B le cas échéant. Cette représentation a été adoptée pour définir certaines sémantiques et transformations de programmes décrites dans ce mémoire. En effet, comme ces sémantiques ne distinguent pas différentes sortes d'erreurs d'exécution des programmes, les cas non décrits dans les règles de sémantiques correspondent aux cas d'erreurs. Il en est de même pour les transformations de programmes.

Les styles inductifs et fonctionnels de Coq ont chacun leurs avantages et inconvénients, ainsi que leurs tactiques associées. Les expériences décrites dans ce mémoire ont été l'occasion d'expérimenter ces deux styles. En particulier, les travaux autour de la sémantique du langage Cminor décrits dans les chapitres 5 et 7 ont permis de les comparer.

Coq permet de synthétiser des programmes certifiés à partir de preuves constructives de leurs spécifications (lorsque ces dernières sont écrites selon un style fonctionnel). Enfin, les spécifications Coq peuvent être structurées en modules Coq, similaires à ceux de Caml [39]. Le chapitre 6 décrit une expérience d'utilisation des modules de Coq.

3 Évaluation partielle de programmes Fortran 90

Ce chapitre commente une partie du matériau publié dans [1, 2, 3, 4, 5, 6]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Philippe Facon. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Frédéric Paumier, Hubert Parisot, Nathalie Dubois, Pousith Sayarath et Romain Vassallo [stage 13, stage 14, stage15]. Ces références sont celles de la liste détaillée des étudiants que j'ai encadrés (page 117).

L'ÉVALUATION PARTIELLE transforme un programme en un programme spécialisé, en fonction de valeurs d'une partie des variables d'entrée. Pour ce faire, l'évaluation partielle exploite les valeurs de variables connues dès la compilation afin d'évaluer symboliquement ce qui est connu dans le programme. Cette évaluation symbolique permet de simplifier certaines expressions et instructions du programme. Par exemple, une conditionnelle peut être simplifiée en l'une de ses branches, cette branche étant elle-même simplifiée.

La figure 3.1 définit schématiquement l'évaluation partielle. Étant donnés un programme P et ses variables d'entrée $x_1 \dots x_n, y_1 \dots y_m$, l'évaluation partielle de P pour les valeurs respectives $c_1 \dots c_n$ des $x_1 \dots x_n$ fournit un programme *P'* qui se comporte comme le programme initial P : quelles que soient les valeurs respectives $v_1 \dots v_m$ des variables $y_1 \dots y_m$, l'exécution de P pour les valeurs $x_1 = c_1 \dots x_n = c_n, y_1 = v_1 \dots y_m = v_m$ et l'exécution de *P'* pour les valeurs $y_1 = v_1 \dots y_m = v_m$ calculent les mêmes résultats.

$$\forall y_1 \dots y_m, P(x_1, \dots x_n, y_1 \dots y_m) \xrightarrow[x_i = c_i, \forall\, i=1 \dots n]{\text{évaluation partielle}} P'(y_1 \dots y_m)$$

$$\text{avec } exec\,(P(c_1, \dots c_n, v_1 \dots v_m)) = exec\,(P'(v_1 \dots v_m))$$

FIG. 3.1 – Évaluation partielle

L'évaluation partielle est une transformation de programme qui procède

soit en une seule étape (évaluation partielle « on-line »), soit en deux étapes principales, une analyse de temps de liaison, suivie d'une spécialisation de programme (évaluation partielle « off-line »). Un évaluateur partiel « on-line » peut être vu comme un interprète non standard, alors qu'un évaluateur partiel « off-line » procède à l'image d'un compilateur. L'évaluation partielle « on-line » est une analyse de programme beaucoup plus précise (et donc beaucoup plus coûteuse) que l'évaluation partielle « off-line ». Les stratégies « off-line » sont surtout utilisées pour générer du code efficace.

L'évaluation partielle a surtout été utilisée en compilation pour générer des compilateurs à partir d'interprètes ou encore pour générer du code efficace (dans différents domaines tels que l'optimisation de programmes numériques, le graphisme, les systèmes d'exploitation). De nombreux travaux existent sur les langages fonctionnels ; quelques travaux portent sur les langages logiques et les langages impératifs [40, 41, 42, 43].

3.1 Positionnement par rapport à mes travaux de thèse de doctorat

Mes travaux autour de l'évaluation partielle ont été effectués dans le cadre d'une collaboration avec EDF. Ils ont consisté à adapter l'évaluation partielle afin de pouvoir l'utiliser pour améliorer la compréhension et la maintenance de programmes écrits en Fortran [1, 3, 4, 5]. Le choix de cette démarche et son expérimentation sur des applications scientifiques industrielles écrites dans un petit sous-ensemble de Fortran 77 (sans appel de sous-programmes) fait l'objet de ma thèse de doctorat [44]. J'ai ainsi développé un outil qui, à partir d'un programme Fortran 77 initial et de valeurs particulières de certaines données d'entrée, fournit entièrement automatiquement un programme simplifié, qui se comporte comme le programme initial pour les valeurs particulières. Les simplifications proviennent de la propagation des informations connues à l'exécution : simplifications d'expressions et réductions d'alternatives à une de leurs branches.

L'évaluation partielle que j'ai proposée a pour objectif de faciliter la maintenabilité du code. Aussi, elle repose sur une stratégie « on-line » d'évaluation partielle, produisant du code peu efficace car proche du code source. L'évaluation partielle a été expérimentée sur une application scientifique de grande taille développée à EDF et modélisant un écoulement de fluide. L'évaluation partielle a permis de réduire sensiblement le code à comprendre car dans les programmes générés par évaluation partielle, beaucoup de variables auxiliaires qui ne sont pas significatives pour les personnes en charge de la maintenance de l'application ont disparu. De plus, ces programmes contiennent

beaucoup moins d'instructions que les programmes initiaux. En effet, dans ces programmes, de nombreuses conditionnelles ont été remplacées par l'une de leurs branches, et cette branche a elle-même été simplifiée.

L'évaluation partielle a fourni de bons résultats sur l'application que j'ai étudiée, mais elle ne peut être appliquée à n'importe quelle application. En effet, dans l'application que j'ai étudiée, la seule connaissance commune à toutes les personnes en charge de la maintenance s'exprimait précisément à l'aide d'égalités entre des variables d'entrée et des valeurs. De plus, cette connaissance commune se traduit au niveau des programmes par de nombreuses alternatives aiguillant sur différentes variantes d'un même algorithme en fonction des valeurs de ces mêmes variables d'entrée. Ainsi, l'application que j'ai étudiée modélise des écoulements de fluides dans différents contextes (par exemple écoulement d'un fluide contenant un polluant particulier à travers un volume pouvant être poreux avec un coefficient particulier de porosité), et ces contextes sont décrits par des égalités entre paramètres (par exemple un indice de porosité) et valeurs.

La figure 3.2 montre un exemple de programme spécialisé obtenu par évaluation partielle. Dans cet exemple, les instructions barrées dans le programme initial sont celles que l'évaluation partielle supprime. Les expressions soulignées sont celles qui sont évaluées totalement durant l'évaluation partielle.

Ces travaux ont été réalisés dans l'environnement Centaur. Le langage étudié durant ma thèse de doctorat était un petit sous-ensemble de Fortran 77 (similaire au langage *Imp* défini dans [20]), correspondant à un noyau de langage impératif, sans appel de fonctions et avec des types simples. Après ma thèse, de 1994 à 2000, ce langage a été étendu à un sous-ensemble de Fortran 90 [45] prenant en compte les appels de sous-programmes et les effets de bords dus aux pointeurs. J'ai également défini une analyse d'alias interprocédurale munie d'une stratégie de réutilisation des versions spécialisées des sous-programmes.

En effet, chaque simplification d'un appel de sous-programme p produit une version de p spécialisée par rapport aux valeurs initiales des données d'entrée utilisées dans p. J'ai défini un ordre entre versions spécialisés permettant de choisir la version la plus spécialisée lors de la réutilisation des versions. Par exemple, cet ordre détermine que si p possède trois paramètres formels x, y et z, alors la version de p spécialisée par rapport aux valeurs $x = 2$ et $y = 3$ est plus spécialisée que la version de p spécialisée par rapport à la valeur $x = 2$ (en supposant de plus que la valeur des autres données n'est pas connue lors de la spécialisation). D'autres critères sont appliqués lorsque la prise en compte des valeurs des données d'entrée ne suffit pas (par exemple, le nombre

FIG. 3.2 – Un exemple d'évaluation partielle de programme.

d'instructions de la version spécialisée).

Par ailleurs, j'ai développé une interface graphique dédiée à la compréhension de programmes, présentant dans plusieurs fenêtres reliées entre elles par des liens hypertextes et selon différentes fontes les différentes informations calculées par l'évaluation partielle (en particulier les versions spécialisées pouvant être réutilisées pour une version donnée). Cette interface a été programmée en Lisp, à partir d'une bibliothèque de Centaur. Elle n'est pas décrite dans ce mémoire.

Enfin, la preuve de préservation sémantique de l'évaluation partielle faite

sur papier seulement dans ma thèse de doctorat a également été améliorée. Dans le cadre d'une étude de cas d'un couplage entre Centaur et Coq, Yves Bertot et Ranan Fraer ont formellement vérifié en Coq la correction sémantique de certaines des transformations élémentaires effectuées par mon évaluation partielle [46]. Ce travail, qui n'a pas été poursuivi, était une application des travaux de thèse de Delphine Terrasse [47] dont le but était de générer automatiquement des spécifications sémantiques écrites en Typol en Coq. Plusieurs années plus tard, ma première expérience en Coq a été la formalisation de l'évaluation partielle que j'avais étudiée dans ma thèse de doctorat [6].

3.2 Le langage Fortran 90

Cette section précise le sous-ensemble de Fortran 90 dont j'ai défini une sémantique formelle. Ce sous-ensemble a permis de tester mon évaluation partielle sur une application réelle développée à EDF. Certaines instructions de Fortran ne font pas partie du sous-ensemble considéré car elles étaient proscrites dans les applications développées à EDF.

3.2.1 Particularités

J'ai ajouté au sous-ensemble de Fortran étudié pendant ma thèse de doctorat les traits interprocéduraux de Fortran 77 (qui sont aussi ceux de Fortran 90) ainsi que l'allocation dynamique de mémoire, les enregistrements et les pointeurs de Fortran 90 (qui n'existent pas en Fortran 77).

Pointeurs

En C, un pointeur est une valeur représentant l'adresse d'une cellule de la mémoire. Par contre, en Fortran 90, un pointeur représente un alias. Un pointeur en Fortran 90 peut être vu comme étant une abstraction d'un pointeur en C. En Fortran 90, il n'y a donc pas de manipulation directe d'adresse (en particulier, il n'y a pas d'arithmétique de pointeur), et le déréférencement d'un pointeur est automatique. De plus, en Fortran 90, un pointeur ne peut pointer que sur un objet cible, c'est-à-dire sur un objet ayant été déclaré en tant qu'objet sur lequel il est possible de pointer. Il existe deux affectations entre pointeurs : une affectation entre cibles de pointeurs, et une affectation entre pointeurs. Enfin, les enregistrements de Fortran 90 sont similaires à ceux du C.

Traits interprocéduraux

Un programme Fortran est composé d'unités de programmes compilées de façon indépendante, et communiquant entre elles. Les sous-programmes (i.e. fonctions et procédures) ainsi que les blocs de données sont les unités de programme les plus courantes. Les traits interprocéduraux comprennent les appels de sous-programmes, ainsi que des instructions dédiées à la transmission de blocs de données entre unités de programmes.

Concernant les appels de sous-programmes, le mode de passage des paramètres est soit par référence, soit par valeur-résultat. Le standard Fortran autorise ces deux modes. Leur point commun est la prise en compte des effets de bord dus aux paramètres effectifs (i.e. les valeurs finales des paramètres formels sont transmises aux paramètres réels correspondants). La différence entre les deux modes réside dans le traitement de l'aliasing. J'ai choisi de modéliser le mode de passage par valeur-résultat, qui évite de considérer l'aliasing entre paramètres formels et paramètres réels.

Concernant les blocs de données, j'ai considéré les instructions `COMMON`, `DATA` et `SAVE`, et volontairement laissé de côté d'autres instructions (par exemple `EQUIVALENCE`) qui étaient proscrites (car jugées dangereuses ou obsolètes) dans l'application que j'ai étudiée.

L'instruction `COMMON` définit des zones de mémoire pouvant être partagées pendant l'exécution d'unités de programmes, et rend possible les effets de bord dans les programmes. L'instruction `COMMON` regroupe des variables au sein d'un bloc nommé. Par rapport à un langage tel que C, ces variables ont un statut intermédiaire entre des variables locales et des variables globales. Elles ont une portée plus grande que les variables locales à une unité de programme. Par contre, elles ont une portée moins étendue que des variables globales (qui n'existent pas en Fortran) car leur portée se limite à l'ensemble des unités de programmes appelées depuis l'unité de programme dans laquelle est exécutée l'instruction `COMMON` courante. Cette portée est restreinte lorsqu'un même bloc de `COMMON` BC est redéclaré dans une unité de programme appartenant à la portée de BC.

Les variables d'un bloc de `COMMON` sont partagées entre unités de programmes (par défaut, leurs valeurs sont héritées dans les unités de programmes appelées) mais leur nom peut varier d'une unité de programme à l'autre. De plus, il existe des restrictions d'usage des blocs de `COMMON`. Par exemple, une même variable ne peut pas apparaître dans deux blocs de noms différents, ou encore la taille (i.e. le nombre de ces variables) d'un bloc nommé doit être la même dans chaque unité de programme le référençant.

L'instruction `DATA` initialise une variable (pouvant faire partie d'un bloc de `COMMON`). Dans ce cas, la valeur de la variable n'est pas modifiée pendant l'exécution du programme (même si une instruction du programme modifie cette valeur). L'instruction `SAVE` permet de prolonger la durée de vie d'une variable (pouvant faire partie d'un bloc de `COMMON`) à celle de l'exécution du programme. La variable est alors qualifiée de permanente (terminologie Fortran) ou encore rémanente ou statique (terminologie C).

Définir une sémantique formelle de Fortran 90 a nécessité de modéliser la durée de vie des identificateurs de variables et de blocs de `COMMON` (i.e. le temps pendant lequel ils existent en mémoire) ainsi que leur visibilité. Un identificateur est visible s'il existe en mémoire et est accessible. En Fortran, il peut exister mais être masqué par un autre de même nom. Les jugements d'évaluation présentés dans la section suivante intègrent cette modélisation.

3.2.2 Sémantique formelle

Dans un souci d'unification avec les travaux présentés dans les chapitres suivants, les jugements d'évaluation présentés dans cette section ont été modifiés par rapport à ceux présentés dans les articles publiés. Les jugements d'évaluation de la sémantique formelle de Fortran 90 sont définis dans la figure 3.3. Ils utilisent :

- Un environnement global (noté G) enregistrant les sous-programmes du programme courant. Un sous-programme est constitué de paramètres formels, de blocs de `COMMON`, de variables locales et d'instructions.
- Un ensemble d'identificateurs de blocs de `COMMON` (noté CI) dont le sous-programme courant hérite.
- Un état (noté S), constitué d'un environnement E associant à chaque valeur gauche son adresse en mémoire, d'un état mémoire M, ainsi que d'un état mémoire MC dédié au stockage des valeurs de blocs de `COMMON`. La distinction entre M et MC permet de modéliser le partage des valeurs de variables d'un même bloc de `COMMON` C, lorsque ces variables occupent la même position dans C. Les variables dont la valeur est inconnue ne sont pas représentées dans M et MC.

Dans les jugements d'évaluation, il est nécessaire de représenter MC et CI car pour chaque bloc de `COMMON`, les liens d'héritage entre blocs de `COMMON` ne se calculent pas uniquement à partir du graphe d'appel du programme (mais aussi à partir des déclarations locales de blocs de `COMMON`). Aussi, plutôt que refaire ce calcul chaque fois qu'il est nécessaire, j'ai préféré représenter MC et CI dans les jugements d'évaluation.

De même, l'environnement E enregistre l'adresse de chaque valeur gauche (et pas seulement de chaque variable). En effet, la sémantique d'une affectation entre deux pointeurs utilise une relation « pointe sur » calculée à partir de l'environnement E et de la mémoire M. Ce calcul repose sur une analyse de pointeur (également appelée analyse d'alias) sensible au contexte d'évaluation (i.e. qui tient compte des appels de sous-programmes effectués depuis le point de programme courant). Cette analyse calcule en chaque point de programme, et pour chaque pointeur, ce vers quoi doit pointer ce pointeur. Cette analyse très précise (mais peu efficace) est inspirée d'une analyse proposée pour C [48].

Catégories syntaxiques :

id	:	identificateur de valeur gauche ou de bloc de `COMMON`
def_fn	:	sous-programme (paramètres, `COMMON`, variables locales, instructions)
v	:	valeur
n	:	entier repérant la position d'une variable dans un bloc de `COMMON`
adr	:	adresse en mémoire

Environnement global :

$G ::= id \mapsto def_fn$ sous-programmes

Blocs de `COMMON` hérités :

$CI ::= \{id_1; \dots id_k\}$ ensemble d'identificateurs de blocs de `COMMON`

État :

$$
\begin{array}{lcl}
S & ::= & (E, M, MC) \\
E & ::= & id \mapsto adr \\
M & ::= & adr \mapsto v \\
MC & ::= & \{id_1 \mapsto \{n_1 \mapsto v_1; \dots n_{p_1} \mapsto v_{p_1}\}; \dots id_k \mapsto \{n_1 \mapsto v_1; \dots n_{p_k} \mapsto v_{p_k}\}\}
\end{array}
$$

Jugements d'évaluation :

$$
\begin{array}{rcll}
G, CI & \vdash & a, S \Rightarrow v, S' & \text{(expressions)} \\
G, CI & \vdash & a^*, S \Rightarrow v^*, S' & \text{(liste d'expressions)} \\
G, CI & \vdash & i, S \Rightarrow S' & \text{(instructions)} \\
 & \vdash & p \Rightarrow v & \text{(programmes)}
\end{array}
$$

Fig. 3.3 – Sémantique dynamique de Fortran 90

Fortran possède peu de types arithmétiques : un type entier et deux types flottants (simple et double précision). De plus, un bloc de `COMMON` peut être découpé différemment d'une unité de programme à l'autre, c'est-à-dire référencé à l'aide de variables de types différents. Du point de vue de la mémoire, il est donc par exemple possible d'écrire en mémoire une valeur de type entier,

puis de lire cette valeur comme étant de type flottant. Parce qu'il existe peu de contraintes de typage en Fortran, et parce que l'évaluation partielle que j'ai étudiée est un outil d'aide à la maintenance, qui s'applique à des programmes ayant été préalablement compilés, les types ne sont pas modélisés dans la sémantique formelle que j'ai définie. De même, le modèle mémoire est très simple et associe des valeurs à des adresses, sans considération de type.

Enfin, dans la sémantique formelle, la règle d'inférence d'un appel de sous-programme calcule différentes informations modélisant la transmission des valeurs entre sous-programmes appelant et appelé. Ces calculs sont exprimés à l'aide d'opérateurs relationnels empruntés aux langages B et VDM [49, 50], ce qui a permis d'abstraire la sémantique formelle et donc de la rendre plus lisible. Par exemple, la correspondance entre variables et valeurs d'un bloc de `COMMON` s'exprime aisément à l'aide de ces opérateurs.

La sémantique formelle d'un sous-ensemble de Fortran 90 a été définie en langage Typol dans l'environnement Centaur. Un interprète en Prolog a été généré automatiquement à partir des règles d'inférence en Typol. Des listes de couples ont été définies directement en Prolog afin d'implémenter la mémoire et les environnements (i.e. E, G et MC). Les opérateurs associés ont également été écrits directement en Prolog.

3.3 Une stratégie d'évaluation partielle « on-line »

J'ai défini l'évaluation partielle au moyen de règles d'inférence décrivant comment sont transformées les expressions et instructions. L'exécution (ou évaluation totale) d'un programme est un cas particulier d'évaluation partielle de ce programme dans lequel toutes les valeurs d'entrée sont connues. Aussi, j'ai défini la relation d'évaluation partielle par des règles d'inférence généralisant la relation d'exécution. Cette généralisation est similaire à celle définie dans [51] pour une autre forme d'évaluation symbolique (i.e. le découpage de programme).

La figure 3.4 définit les jugements d'évaluation partielle des instructions. Comme dans ma thèse de doctorat, la relation d'évaluation partielle (notée $\Rightarrow$ dans la figure) comprend une relation de propagation des valeurs connues (notée :) et une relation de simplification des instructions (notée $\hookrightarrow$). La relation de propagation ressemble à la sémantique formelle. La différence est que seules certaines valeurs sont propagées dans la relation de propagation (puisque certaines valeurs des variables d'entrée ne sont pas connues).

Par rapport à la sémantique formelle de Fortran 90, l'évaluation partielle gère les versions spécialisées des sous-programmes, dans le but de les réutiliser

le plus possible lors des appels de sous-programmes. Le jugement d'évaluation partielle comprend donc une table de versions spécialisées (notée *Version*), qui est mise à jour lors de la simplification des instructions. Cette table associe à chaque nom de sous-programme *sp* et chaque ensemble de valeurs initiales, un ensemble de valeurs finales et une version spécialisée de *sp*.

$$
\begin{array}{rcll}
G, CI & \vdash & i, S : S' & \text{(propagation)} \\
G, CI, Version & \vdash & i, S \hookrightarrow i', Version' & \text{(simplification)} \\
G, CI, Version & \vdash & i, S \Rightarrow i', S', Version' & \text{(évaluation partielle)}
\end{array}
$$

$$
\frac{G, CI \vdash i, S : S' \qquad G, CI, Version \vdash i, S \hookrightarrow i', Version'}{G, CI, Version \vdash i, S \Rightarrow i', S', Version'}
$$

Fig. 3.4 – Évaluation partielle d'une instruction i

$\vdash p, S_0 \mapsto p', S', Version'$ désigne l'évaluation partielle d'un programme p en un programme p', étant donné un ensemble de valeurs de certaines données d'entrée (contenu dans S_0). $S_0 \cup S$ désigne l'état S_0 auquel ont été ajoutées les valeurs des variables d'entrée inconnues dans S_0. La correction de l'évaluation partielle est la propriété suivante. Si $\vdash p, S_0 \mapsto p', S', Version'$, alors les séquents $\vdash p, S_0 \cup S \Rightarrow S''$ et $\vdash p', S_0 \cup S \Rightarrow S''$ sont équivalents, quelles que soient les valeurs des variables d'entrée non connues lors de l'évaluation partielle (et contenues dans S). Autrement dit, les exécutions de p et p' fournissent les mêmes résultats. Cette propriété a été prouvée sur papier seulement.

3.4 Une première expérience de vérification formelle en Coq

À l'issue du développement en Centaur d'un environnement graphique dédié à l'évaluation partielle, je me suis intéressée à la vérification formelle de l'évaluation partielle. Mes premières expériences en Coq ont concerné l'évaluation partielle étudiée pendant ma thèse. La version de Coq utilisée alors est la 7.3. J'ai redéfini sous la forme de relations inductives Coq les règles d'inférence de sémantique et d'évaluation partielle, que j'avais écrites en Typol. Le but du passage à Coq n'était pas d'adapter à peu de frais les règles Typol, ce qui aurait fourni une spécification Coq de trop bas niveau. Au contraire, j'ai spécifié de façon la plus abstraite possible le langage analysé ainsi que l'évaluation partielle.

La principale différence réside dans la représentation du modèle mémoire, qui permet de conserver en tout point de programme les valeurs connues des variables, sachant que durant une évaluation partielle, la valeur de certaines variables est inconnue. Soit p un programme évalué partiellement en un programme p' par rapport à des valeurs d'entrée S_0. Lorsque p et p' sont exécutés par rapport à des valeurs d'entrée $S_0 \cup S$, alors les valeurs stockées en mémoire durant l'exécution de p sont les mêmes que celles stockées durant l'exécution de p'. Par contre, l'ordre des allocations et libérations en mémoire n'est pas le même (car l'exécution de p' a lieu après l'évaluation partielle de p).

La formalisation en Coq a donc nécessité la définition d'une relation d'équivalence entre mémoires. Les valeurs des variables en tout point de programme n'étant pas toutes connues durant l'évaluation partielle ou l'exécution d'un programme, une mémoire est une fonction partielle. Les deux représentations de la mémoire les plus usuelles utilisent soit une liste de couples, soit une fonction totale. L'inconvénient d'une liste est qu'elle nécessite de gérer l'unicité des variables, puisque toute variable possède au plus une valeur. L'inconvénient d'une fonction totale est qu'elle nécessite l'introduction d'une valeur particulière $\perp$ pour les variables dont les valeurs ne sont pas connues, et donc la manipulation explicite de cette valeur $\perp$.

Afin de pallier ces inconvénients, j'ai défini en Coq une couche de bas niveau permettant de représenter la mémoire par la notion générale de table d'association définie dans [52]. Ce choix a simplifié la preuve de correction de l'évaluation partielle. Il a par exemple simplifié les preuves concernant le déterminisme de l'évaluation (partielle ou totale) des instructions. Dans [52], une table générique est paramétrée par les types des clés (A) et des informations associées (B). C'est un couple formé du domaine de la fonction partielle sous-jacente et d'une fonction totale de type $A \to B$. L'accès à la table, c'est-à-dire à la fonction encapsulée, est conditionné par la vérification que l'élément clé utilisé appartient au domaine de la table.

J'ai utilisé le type *table* comme un type abstrait algébrique. Pour les besoins de l'évaluation partielle, j'ai défini de nouvelles opérations sur les tables (par exemple, suppression d'un élément du domaine, intersection de deux tables) et vérifié formellement de nouveaux lemmes concernant ces opérations. J'ai également défini une relation d'inclusion entre tables fondée sur l'inclusion de leurs domaines, ainsi qu'une relation d'équivalence fondée sur la relation d'inclusion.

De plus, des propriétés mêlant par exemple l'évaluation (totale) d'une expression étant donné une mémoire m à l'évaluation partielle de cette expression étant donné une mémoire m_0 incluse dans m a nécessité de raisonner sur la

relation d'inclusion entre mémoires.

La principale difficulté dans la preuve de correction de l'évaluation partielle a été de raisonner sur des mémoires équivalentes. Cette difficulté n'avait pas été soulevée dans la preuve sur papier. Par exemple, lors de la preuve de correction de l'instruction séquence, il est nécessaire d'introduire des états de mémoire intermédiaires relatifs au point de programme situé après l'exécution de la première instruction, et ceci pour l'exécution comme pour l'évaluation partielle. On obtient ainsi différents états mémoire dont on doit montrer l'équivalence. D'autres lemmes d'existence ont nécessité d'exhiber différents états mémoire équivalents.

Aussi, j'ai défini le type des tables comme un setoïde, c'est-à-dire un type muni d'une relation d'équivalence. La notion de setoïde venait alors d'être introduite dans Coq [53]. L'intérêt des setoïdes est que grâce à des fonctions particulières entre setoïdes, qualifiées de morphismes, une équivalence (entre setoïdes) se manipule aussi aisément qu'une égalité (en permettant de réécrire des éléments équivalents).

Du fait des limitations existant alors dans Coq (qui n'existent plus aujourd'hui) et portant sur l'utilisation des setoïdes, je n'ai pas pu définir autant de morphismes que je l'aurais souhaité. J'ai de plus étudié une autre solution permettant d'internaliser la notion d'équivalence, et utilisant la notion de type inductif quotient définie dans [54]. Cette solution nécessite de manipuler exclusivement des tables normalisées, et donc de normaliser les tables en chaque point de programme, ce qui alourdit la preuve de correction et demande de revoir la définition des tables. Aussi, cette solution a été abandonnée.

3.5 Bilan

Ce chapitre a présenté une évaluation partielle de programmes Fortran 90. J'ai défini une sémantique formelle d'un sous-ensemble de ce langage, dans un style à grands pas, et en langage Typol. La relation d'évaluation partielle généralise la relation d'exécution.

L'évaluation partielle évalue symboliquement un programme, en ne connaissant qu'une partie de ses valeurs d'entrée. Aussi, par rapport à un compilateur qui génère un programme dont toutes les valeurs d'entrée sont connues à l'exécution du programme, lors d'une évaluation partielle, il est nécessaire de calculer par analyse statique davantage d'informations.

J'ai également spécifié en Coq (et dans un style relationnel) un sous-ensemble de l'évaluation partielle ayant été spécifiée en Typol, et vérifié formellement sa correction. Ceci a nécessité la définition d'un modèle mémoire

plus abstrait que celui défini en Typol. Ce modèle mémoire a été défini à partir d'un type représentant des tables d'association. Un setoïde a été défini sur ces tables afin de faciliter le raisonnement sur des mémoires équivalentes.

La formalisation en Coq d'un modèle mémoire plus général et plus complet que celui décrit dans ce chapitre fait l'objet du chapitre 6. Le modèle du chapitre 6 utilise la bibliothèque `FMaps` de Coq qui a été développée dans le cadre du projet CompCert quelques années après le travail présenté dans ce chapitre [55].

4 Le compilateur certifié CompCert

Un compilateur traduit un langage de haut niveau (i.e. compréhensible par un être humain) en un langage de bas niveau (i.e. exécutable efficacement par une machine). Un compilateur effectue une succession de passes de transformation de programmes. Une transformation désigne soit une traduction vers un langage de plus bas niveau, soit une optimisation de programmes (générant du code efficace). La figure 4 présente schématiquement la compilation d'un programme source P en un programme P'. Par rapport à l'évaluation partielle (*cf.* figure 3.1), les valeurs des variables d'entrée ne sont connues qu'à l'exécution des programmes.

Même si certaines passes de transformations sont employées en évaluation partielle et en compilation (par exemple la propagation de constantes), la compilation est un processus beaucoup plus complexe que l'évaluation partielle présentée dans le chapitre précédent, principalement parce que celle-ci est une transformation de programme à langage constant. En particulier, l'évaluateur partiel décrit dans le chapitre 3 peut être considéré comme un interprète non standard, alors que le compilateur que je présente dans ce chapitre accomplit 11 passes de compilation, transformant progressivement (via 7 langages intermédiaires) des programmes C en code assembleur.

$$\forall y_1 \ldots y_m, P(y_1 \ldots y_m) \xrightarrow{\text{compilation}} P'(y_1 \ldots y_m)$$

$$\text{avec } exec\,(P(v_1 \ldots v_m)) = exec\,(P'(v_1 \ldots v_m))$$

Fig. 4.1 – Compilation

Depuis janvier 2003, dans le cadre de l'ARC (Action de Recherche Collaborative financée par l'INRIA) Concert coordonnée par Yves Bertot et actuellement dans le cadre du projet ANR CompCert coordonné par Xavier Leroy, je participe au développement d'un compilateur réaliste, utilisable pour le lo-

giciel embarqué critique, et appelé CompCert [19]. Il s'agit d'un compilateur d'un large sous-ensemble du langage C (dans lequel il n'y a pas d'instructions de sauts) qui produit du code assembleur pour le processeur Power PC (qui est largement répandu dans le domaine embarqué) et effectue diverses optimisations (principalement, propagation de constantes, élimination des sous-expressions communes et réordonnancement d'instructions) afin de produire du code raisonnablement compact et efficace [56, 7]. Ce compilateur est formellement vérifié en Coq, c'est-à-dire accompagné d'une preuve Coq de préservation sémantique lors de la compilation.

La plupart des parties formellement vérifiées du compilateur sont programmées directement dans le langage de spécification de Coq, en style fonctionnel pur. Le mécanisme d'extraction de Coq produit automatiquement le code Caml du compilateur CompCert à partir des spécifications. Ce code constitue le compilateur dit certifié. Actuellement, CompCert compile des programmes C de quelques centaines de lignes, avec des performances comparables à celles des meilleurs compilateurs modérément optimisants (par exemple `gcc` utilisé au premier niveau d'optimisation). Le code Caml de CompCert est généré automatiquement à partir des spécifications. Il s'agit probablement du plus gros développement en Caml ayant été extrait de spécifications Coq.

La figure 4.2 montre l'architecture du compilateur CompCert. Un analyseur de C largement répandu appelé CIL [57], traitant la plupart des dialectes de C a été réutilisé. CIL est écrit en Caml et transforme les programmes C en arbres de syntaxe abstraite (abrégés en ASA dans la figure). CIL normalise également les programmes C. Par exemple, les initialisations implicites de tableaux sont normalisées en initialisations explicites. La syntaxe abstraite produite par CIL décrit un sous-ensemble de C appelé Clight. Nous avons modifié légèrement CIL afin d'élargir le sous-ensemble de C considéré pour l'adapter au compilateur CompCert (c.f. section 5.1). Clight est ainsi le langage d'entrée de la partie formellement vérifiée du compilateur CompCert.

La partie formellement vérifiée du compilateur CompCert traduit le langage Clight vers le langage assembleur du Power PC (PPC) ; elle comprend un front-end, un back-end, et un modèle mémoire commun aux huit langages du compilateur. J'ai contribué au développement du front-end et du modèle mémoire du compilateur. Ces travaux ont été effectués en collaboration avec Xavier Leroy ; ils sont détaillés dans les chapitres respectifs 5 et 6.

La confiance en l'analyse syntaxique de CIL résulte du fait que CIL est un analyseur de C reconnu et utilisé dans de nombreux outils opérant sur des applications réelles écrites en C (par exemple [58, 26, 59, 60, 61, 62]). Vérifier formellement un analyseur syntaxique nécessiterait de donner une sémantique

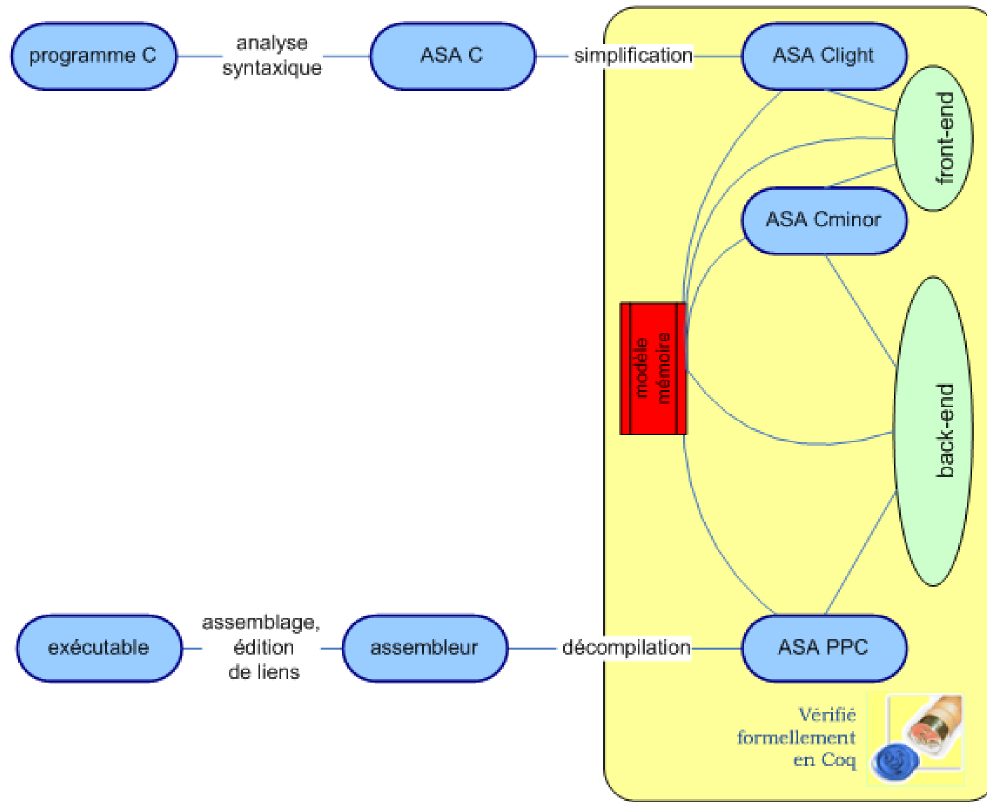


FIG. 4.2 – Architecture du compilateur CompCert

formelle à un texte (i.e. une suite de caractères). À ma connaissance, en dehors de travaux portant sur la vérification formelle de l'analyse lexicale [63], ce thème n'a pour l'instant pas été étudié.

Les langages du compilateur CompCert étant impératifs et sans gestion automatique de la mémoire, la définition d'un modèle mémoire commun à ces langages est au cœur de la formalisation des sémantiques de ces langages. Parmi les sémantiques, la définition du modèle mémoire constitue une des parties qu'il a été le plus difficile de modéliser. En plus d'une généricité permettant de s'appliquer à tous les langages du compilateur, il a été nécessaire de trouver un bon niveau d'abstraction.

En effet, un modèle trop abstrait rend impossible l'expression de propriétés telles que la chevauchement partiel de zones de la mémoire, ou encore l'aliasing

entre pointeurs. En revanche, un modèle de trop bas niveau, représentant la mémoire par un unique tableau d'octets ne permet pas nécessairement de vérifier les propriétés élémentaires requises pour les opérations de lecture et écriture en mémoire, qui doivent également être vérifiées par la sémantique du langage considéré [10, 11].

Comme le montre la figure 4.3, Cminor est le principal langage intermédiaire du compilateur CompCert. Historiquement, Cminor est le premier langage ayant été étudié dans CompCert. Des travaux en cours portent d'une part sur la traduction de mini-ML vers Cminor [64, 65] ainsi que sur la traduction d'un sous-ensemble de Java vers Cminor. Cminor a également été choisi comme langage pivot du projet en cours Concurrent Cminor (c.f. chapitre 7) qui a pour objectif à long terme de fournir un environnement dédié à la vérification formelle de programmes séquentiels et concurrents.

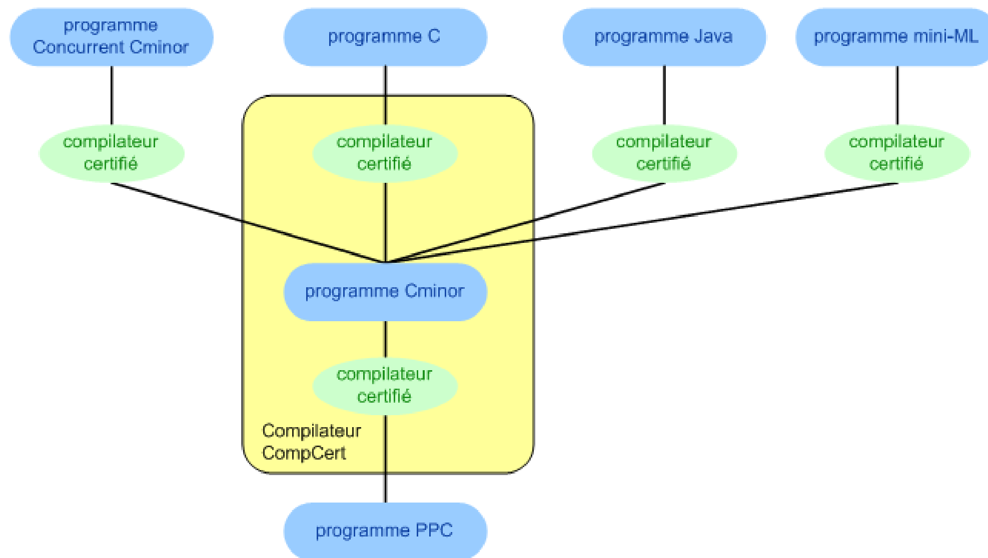


FIG. 4.3 – Le langage intermédiaire Cminor

Différents styles de sémantiques ont été étudiés pour formaliser le langage Cminor. Le style sémantique à grands pas a d'abord été choisi pour son confort. Les langages du front-end sont écrits selon ce style (alors que les langages du back-end sont définis selon un style à petits pas). Les sémantiques à grands pas du compilateur CompCert permettent d'observer les valeurs finales d'un programme, ainsi que la trace des appels des fonctions d'entrées-sorties ef-

fectués durant l'exécution du programme. L'exécution d'un programme est ainsi abstraite en le calcul des entrées-sorties et des valeurs finales du programme. D'autres styles de sémantique ont également été étudiés : différentes sémantiques à petits pas, sémantique axiomatique, sémantique coinductive et sémantique dénotationnelle.

Un résultat intéressant du projet CompCert est que la structure générale du compilateur est conditionnée non pas par les transformations de programmes, mais très fortement par le choix des langages utilisés dans le compilateur, et aussi par le style de sémantique donné à ces langages. Ainsi, les langages intermédiaires du compilateur ont été conçus dans le but de faciliter les preuves de traduction.

Souvent, lorsqu'une preuve de traduction d'un langage L_1 en un langage L_2 a nécessité de modéliser différents concepts (par exemple la mise en correspondance de zones de la mémoire, ou encore la traduction d'instructions en instructions de plus bas niveau), rendant ainsi ces preuves difficiles à faire et à maintenir, un langage intermédiaire L_i entre L_1 et L_2 a été défini. Vérifier séparément la correction des deux traductions vers et depuis L_i s'est avéré beaucoup plus aisé que vérifier la correction de la traduction de L_1 vers L_2. La facilité à prouver une traduction a donc été le principal critère choisi pour valider les sémantiques des langages considérés. Cette approche est comparable à la conception d'étapes de raffinement dans un développement formel.

Au final, le compilateur CompCert dispose de 7 langages intermédiaires. Cette approche n'est pas usuelle en compilation. Par exemple, jusqu'à une version récente du compilateur `gcc`, celui-ci ne disposait que d'un seul langage intermédiaire (le langage RTL, langage à transfert de registres, également utilisé dans CompCert). Une sémantique formelle a été définie pour chacun des 9 langages du compilateur CompCert.

La figure 4.4 montre l'ensemble des langages du compilateur CompCert. Le front-end traduit un programme Clight en un programme Cminor via un langage intermédiaire appelé Csharpminor (c.f. section 5.5.2 page 55). Le back-end traduit des programmes Cminor en programmes RTL, puis en programmes PPC via 5 langages intermédiaires.

Récemment, je me suis également intéressée à la passe d'allocation de registres qui opère sur le langage RTL, le principal langage intermédiaire du back-end du compilateur CompCert. Contrairement au langage Cminor qui a été inventé pour CompCert, le langage RTL est un langage intermédiaire de référence, souvent utilisé par les compilateurs [66].

L'allocation de registres détermine où sont stockées les variables d'un programme à tout moment de son exécution (soit en registres si ces derniers sont

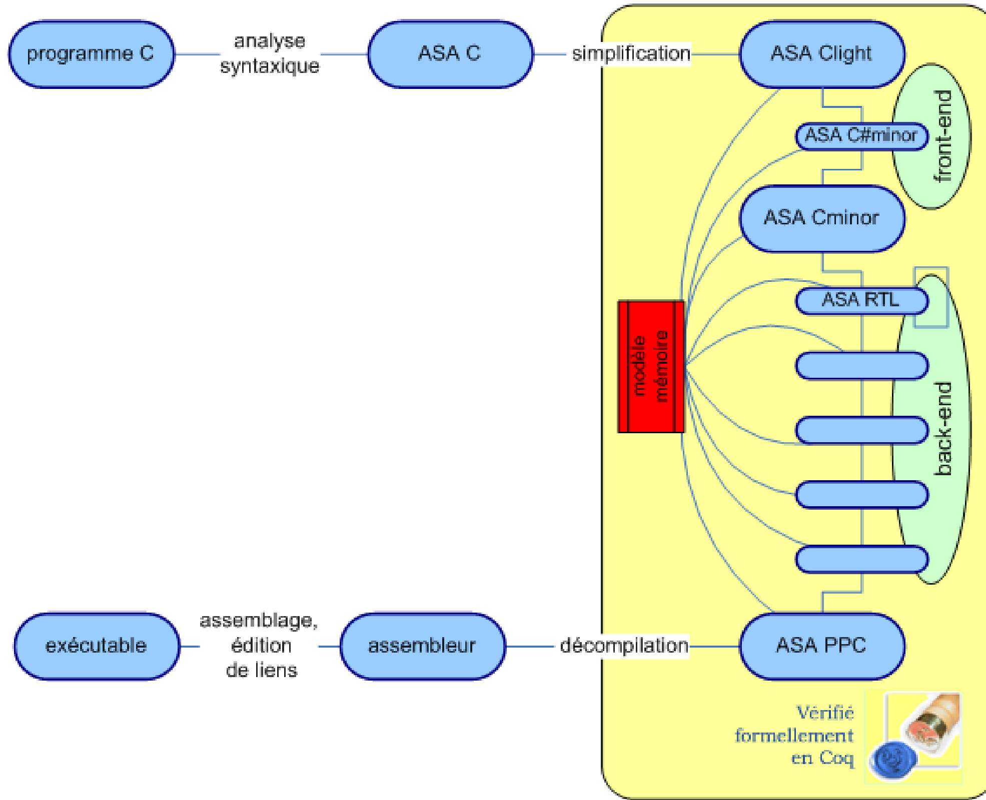


FIG. 4.4 – Le front-end et le back-end du compilateur CompCert

disponibles, soit en mémoire le cas échéant), et modifie les instructions du programme en fonction des affectations des variables. La difficulté est de proposer une affectation optimale des registres. Il est par exemple souvent nécessaire de choisir entre conserver une variable v dans un même registre R pendant l'exécution complète d'un programme (ce qui rend R inutilisable pour stocker d'autres variables), et réutiliser R lorsque la valeur de v n'a plus besoin d'être conservée en vue d'utilisations futures (ce qui nécessite de transférer en mémoire la valeur de v). J'ai proposé une allocation de registres différente de l'heuristique utilisée dans CompCert. Ces travaux sont présentés dans le chapitre 8.

5 Un front-end pour le compilateur CompCert

Ce chapitre commente une partie du matériau publié dans [7, 8, 9]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Xavier Leroy. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Zaynah Dargaye et Thomas Moniot [stage 4, stage 6].

Le front-end du compilateur CompCert traduit des programmes Clight en programmes Cminor. Ce chapitre décrit des spécifications concernant des sémantiques formelles et des traductions de programmes opérant sur ces sémantiques. Comme expliqué dans le chapitre 2, les spécifications ont été définies avec une volonté d'indépendance entre sémantiques et traductions. En particulier, l'exécution d'un programme peut échouer à cause de l'évaluation d'une valeur erronée (du point de vue de la sémantique), mais la traduction de ce programme peut réussir car la traduction de la valeur en question réussit (parce que par exemple cette valeur est stockée en mémoire).

Initialement, les sémantiques formelles ne modélisaient que les programmes dont l'exécution termine. Les définitions inductives des sémantiques formelles ont été étendues par Xavier Leroy à des définitions coinductives modélisant des programmes dont l'exécution diverge [67]. L'exécution d'un programme qui diverge est abstraite en une trace infinie d'entrées-sorties. Le théorème de préservation sémantique a ainsi été étendu aux programmes qui ne terminent pas. Il établit que le programme compilé correspondant diverge également.

La difficulté de conception du front-end a été de trouver le bon niveau d'abstraction, entre un niveau trop abstrait ne permettant pas de modéliser par exemple certaines violations du standard C couramment utilisées et un niveau trop concret rendant les preuves difficiles à faire.

Lors d'une première expérience, j'ai formalisé la sémantique d'un sous-ensemble restreint de C (sans `struct`, sans bloc, sans `switch`), selon un style à grands pas. Ce style a été choisi du fait de sa simplicité et de la facilité à raisonner sur une sémantique à grands pas. Ces critères ont été prépondérants pour un langage aussi vaste que C. De plus, parce que les compilateurs C considèrent que l'évaluation des expressions est déterministe, dans cette sémantique l'évaluation des expressions est déterministe, contrairement à la première for-

malisation sur machine d'une sémantique de C, due à Michael Norrish [68]. Cette première expérience a permis de mieux comprendre les mécanismes de conversions implicites du C tels que la promotion intégrale et le passage des valeurs entre fonctions appelante à appelée, et plus généralement le rôle des types en C [69]. De plus, elle a permis de définir les environnements d'évaluation intervenant dans la sémantique. Enfin, elle a également été l'occasion de valider une première ébauche du modèle mémoire du compilateur CompCert (c.f. chapitre 6). Grâce au modèle mémoire, la sémantique de Clight impose des accès sûrs à la mémoire, principalement en lecture et écriture.

Ce chapitre décrit les différentes parties du front-end du compilateur CompCert. D'abord, il explique brièvement comment CIL a été utilisé. Ensuite, il détaille la syntaxe abstraite et la sémantique formelle du langage Clight. Puis, il présente le langage Cminor, ainsi que la traduction de Clight vers Cminor.

5.1 CIL

Lors d'une seconde expérience, j'ai défini une sémantique de Clight (i.e. d'un sous-ensemble de C beaucoup plus vaste, grâce à l'utilisation de l'analyseur statique CIL [57]). Cette sémantique est celle du compilateur CompCert actuel. Nous avons dû adapter CIL pour CompCert. En effet, CIL transforme des programmes C en arbres de syntaxe abstraite copieusement annotés et suffisamment généraux pour être exploitables par différents outils d'analyse statique. Ainsi, CIL normalise diverses constructions C. Certaines normalisations ont été conservées dans CompCert (par exemple la transformation d'expressions en expressions sans effets de bord, l'explicitation des conversions implicites, la suppression des blocs d'instructions).

Ces transformations sont justifiées pour le compilateur CompCert car elles facilitent les preuves, notamment en évitant des inductions multiples (même si lors de la première expérience de formalisation de C, certaines de ces preuves ont été faites sans bénéficier de ces transformations). D'autres simplifications ont été supprimées. Ce sont essentiellement celles produisant des instructions de saut `goto`. Par exemple, CIL normalise toute boucle C, produisant ainsi une représentation uniforme des trois types de boucles C. Au contraire, dans CompCert, chaque forme de boucle est conservée, afin de laisser la possibilité d'utiliser dans le futur des optimisations spécifiques à certaines formes de boucles (aux boucles `for` par exemple).

Un autre exemple est la transformation par CIL des alternatives `switch` en cascades de `if` (et de `goto`). Dans CompCert, nous avons préféré garder une construction `switch`. Celle que nous avons définie est moins générale que celle

de C. Enfin, le traitement des valeurs de retour par défaut des fonctions a été modifié, pour des raisons de cohérence avec le traitement des autres valeurs de retour dans notre sémantique.

5.2 Syntaxe abstraite de Clight

La syntaxe abstraite de Clight est détaillée dans les figures 5.1 et 5.2. Les types de Clight sont ceux de C : types intégraux, pointeurs (dont les pointeurs sur fonctions), type fonction, types composés tableaux, `union` et `struct`. Contrairement au C où la taille des types intégraux n'est que partiellement standardisée (le standard imposant seulement une taille minimale et une magnitude minimale), chaque type intégral de Clight spécifie une taille et indique si le type est signé.

En C, les types `struct` et `union` peuvent être récursifs. Plus précisément, un champ d'un tel type τ peut être du type pointeur sur τ, mais pas du type τ. Du fait de la récursion, le type de ce champ ne peut pas être le type `pointeur` du langage. Il est d'un type spécifique (appelé `comp_ptr`, qui a été rajouté à la syntaxe du langage Clight pour pouvoir modéliser les types récursifs tels que τ). C'est pourquoi un type `struct` ou `union` est représenté sous la forme d'un opérateur μ de point fixe.

Au niveau des expressions, tous les opérateurs de C peuvent être exprimés en Clight. Certains opérateurs de C sont du sucre syntaxique et ne sont pas représentés dans la syntaxe absrtaite de Clight. Par exemple, l'expression C d'accès à un tableau $a_1[a_2]$ s'écrit en Clight $*(a_1 + a_2)$. Les affectations et les appels de fonction sont des instructions Clight, et non pas des expressions comme en C. Les expressions sont annotées par leur type statique. Ces types facilitent l'écriture de la sémantique des expressions contenant des opérateurs dépendant des types tels que les opérateurs arithmétiques (qui en C sont surchargés).

Au niveau des instructions, toutes les instructions de contrôle structurées du C sont définies en Clight. Par contre, les instructions de saut n'existent pas en Clight. L'instruction `switch` de Clight est restreinte par rapport à celle de C : elle ne permet pas de représenter des boucles optimisées (car partiellement dépliées et utilisant l'entrelacement possible des instructions dans un `switch`) telles que l'exemple connu sous le nom de Duff's device [70]. De tels codes étant proscrits dans le domaine embarqué, nous avons préféré garder cette représentation des `switch`, plutôt que chercher à spécifier les instructions de sauts dans notre sémantique. Les blocs de C n'existent pas en Clight car ils sont supprimés par CIL.

Types :

signedness	::=	`Signed` | `Unsigned`
intsize	::=	`I8` | `I16` | `I32`
floatsize	::=	`F32` | `F64`
τ	::=	`int`(*intsize*, *signedness*)
	|	`float`(*floatsize*)
	|	`void`
	|	`array`(τ, n)
	|	`pointer`(τ)
	|	`struct`($id, (id, \tau)^*$)
	|	`union`($id, (id, \tau)^*$)
	|	`comp_ptr`(id)
	|	`function`(τ^*, τ)

Expressions annotées par des types :

a	::=	$\underline{a}^\tau$

Expressions non annotées :

$\underline{a}$	::=	id	identificateur de variable
	|	n	constante entière
	|	f	constante réelle
	|	`sizeof`(τ)	taille d'un type
	|	$op_1\ a$	opération arithmétique unaire
	|	$a_1\ op_2\ a_2$	opération aithmétique binaire
	|	`*`a	déréférencement
	|	$a.\,id$	accès à un champ
	|	`&`a	prise d'addresse
	|	$(\tau)a$	cast (conversion explicite)
	|	a_1 `?` a_2 `:` a_3	condition
op_2	::=	`+` | `-` | `*` | `/` | `%`	opérateurs arithmétiques
	|	`<<` | `>>` | `&` | `|` | `^`	operateurs bit à bit
	|	`<` | `<=` | `>` | `>=` | `==` | `!=`	opérateurs relationnels
op_1	::=	`-` | `~` | `!`	opérateurs unaires

FIG. 5.1 – Syntaxe abstraite de Clight (types et expressions). a^* dénote 0, 1 ou plusieurs occurrences de la catégorie syntaxique a.

Les fonctions et programmes Clight ont la même syntaxe que les programmes C. Une fonction est soit interne à un programme (c'est-à-dire définie dans ce programme), soit externe (c'est-à-dire définie dans une bibliothèque externe). Une fonction externe est spécifiée par un identifiant et une signature. Un programme est défini comme étant une collection de déclarations de variables globales, une collection de fonctions et un identifiant de la première fonction appelée dans le programme.

5.3 Sémantique formelle de Clight

La sémantique statique est très limitée. Une seule vérification est nécessaire à la preuve de préservation sémantique du compilateur. Il est ainsi vérifié que tout type annotant une référence à une variable correspond à celui déclaré pour cette variable.

La formalisation de la sémantique dynamique de Clight est un plongement profond en Coq. Par exemple, les entiers machine ont été formalisés en Coq, sans recourir aux entiers de Coq.

Valeurs. Toutes les sémantiques des langages du compilateur CompCert utilisent la définition suivante des valeurs. Une valeur est soit un entier machine, soit un nombre flottant, soit la valeur indéfinie `undef`, soit la valeur d'un pointeur, c'est-à-dire une adresse en mémoire (i.e. une paire constituée d'un bloc et d'un décalage dans ce bloc, c.f. chapitre 6).

Environnements d'évaluation. Les environnements d'évaluation de la sémantique formelle de clight sont présentés dans la figure 5.3. L'environnement global (noté G) enregistre d'une part les adresses des variables globales et des identificateurs de fonctions et d'autre part les fonctions d'un programme. G est utilisé par toutes les sémantiques des langages de CompCert. Un environnement local E associe à chaque variable locale son adresse en mémoire. Les valeurs des variables sont stockées dans la mémoire (notée M dans ce chapitre) qui associe des valeurs à des adresses, et à laquelle le chapitre 6 est consacré.

Traces. Dans les sémantiques formelles des langages du compilateur CompCert, une trace représente les interactions entre un programme et son environnement extérieur (*i.e.* les entrées et sorties). Une trace est une liste d'événements. Un événement est produit lors de l'exécution d'un appel à une fonction externe. Lors de l'exécution d'un programme, les traces sont propagées ou

Instructions :

i	$::=$	`skip`	instruction vide
		$\mid a_1 = a_2$	affectation
		$\mid a = a_{fun}(a^*_{args})$	appel de fonction
		$\mid a_{fun}(a^*_{args})$	appel de fonction (sans valeur de retour)
		$\mid i_1; i_2$	séquence
		$\mid$ `if`(a) i_1 `else` i_2	conditionnelle
		$\mid$ `switch`$(a) li$	switch
		$\mid$ `while`(a) i	boucle “while”
		$\mid$ `do` i `while`(a)	boucle“do”
		$\mid$ `for`(i_1, a, i_2) i	boucle “for”
		$\mid$ `break`	sortie de la boucle courante
		$\mid$ `continue`	itération suivante de la boucle courante
		$\mid$ `return` $a^?$	retour de la fonction courante

Branches de switch :

li	$::=$	`default`(i)	cas par défaut
		$\mid$ `case`(n, i, li)	cas étiqueté

Fonctions :

fn	$::=$	$(\dots id_i : \tau_i \dots) : \tau$	en-tête (`fn_params`(fn))
		$\{\dots \tau_j\ id_j; \dots$	variables locales (`fn_vars`(fn))
		$i\ \}$	corps de fonction (`fn_body`(fn))

Signatures de fonction :

σ	$::=$	$\dots \tau_i \dots \tau$	

Définitions de fonction :

def_fn	$::=$	fn	fonction interne Clight
		$\mid < id\ \sigma >$	fonction externe (appel système)

Programmes complets :

$init$	$::=$	$\dots$	
p	$::=$	$\dots (id_i : \tau_i = init_i)^* \dots$	variables globales (noms, données initialisées types) (`prog_vars`(p))
		$\dots id_j =\ def_fn_j \dots$	fonctions (noms et définitions) (`prog_funct`(p))
		id_{main}	entry point (`prog_main`(p))

FIG. 5.2 – Syntaxe abstraite de Clight (instructions, fonctions et programmes). $a^?$ dénote une occurrence optionnelle de la catégorie a.

Valeurs :

v	$::=$ `int`(n)	entier machine (32 bits)
	$\mid$ `float`(f)	flottant (64 bits)
	$\mid$ `ptr`(b, ofs)	pointeur
	$\mid$ `undef`	
b	$\in \mathbb{Z}$	identificateur de bloc
ofs	$::= n$	décalage (en octets) dans un bloc

Environnement local :

E	$::= \mathit{id} \mapsto b$	adresses des variables locales

Environnement global :

G	$::= (\mathit{id} \mapsto b)$	adresses des variables globales
	$\times (b \mapsto \mathit{def_fn})$	et adresses des fonctions

Environnement local :

E	$::= \mathit{id} \mapsto b$	adresses des variables locales

Sortie d'instruction :

out	$::=$ `Out_normal`	aller à la prochaine instruction
	$\mid$ `Out_continue`	aller à la prochaine itération de la boucle courante
	$\mid$ `Out_break`	sortie de la boucle courante
	$\mid$ `Out_return`	sortie de fonction
	$\mid$ `Out_return`(v)	sortie de fonction, avec valeur renvoyée v

FIG. 5.3 – Valeurs, sorties d'instructions et environnement d'évaluation

concaténées (par un opérateur noté @), en fonction des instructions exécutées.

Jugements d'évaluation. La sémantique de Clight est représentée en sémantique naturelle à grands pas avec traces par les 7 jugements d'évaluation de la figure 5.4. Par exemple, l'évaluation d'une expression en position de valeur gauche calcule une adresse (d'après la définition du modèle mémoire, c'est un couple composé d'un identifiant de bloc b et d'un décalage ofs dans ce bloc), alors que l'évaluation d'une expression en position de valeur droite calcule la valeur de l'expression. Comme les expressions de Clight sont sans effet de bord, leur évaluation ne modifie pas la mémoire. L'exécution d'une instruction calcule une trace d'événements d'entrées-sorties et modifie éventuellement le contenu de la mémoire courante.

Afin de prendre en compte le flot de contrôle non trivial des programmes C

G, E	$\vdash$	$a, M \Rightarrow (b, ofs)$	(expressions en position de valeur gauche)
G, E	$\vdash$	$a, M \Rightarrow v$	(expressions en position de valeur droite)
G, E	$\vdash$	$a^*, M \Rightarrow v^*$	(liste d'expressions)
G, E	$\vdash$	$i, M \Rightarrow out, t, M'$	(instructions)
G, E	$\vdash$	$li, M \Rightarrow out, t, M'$	(instructions étiquetées d'un switch)
G	$\vdash$	$f(v^*), M \Rightarrow v, t, M'$	(invocations de fonctions)
	$\vdash$	$p \Rightarrow v, t$	(programmes)

FIG. 5.4 – Jugements de la sémantique de Clight.

(dû à la présence d'instructions telles que `break`, `continue` et `return` dans les boucles ou les fonctions), le résultat de l'exécution d'une instruction produit une indication (notée *out* et définie dans la figure 5.3) modélisant chacune de ces sorties abruptes.

Les jugements d'évaluation ont été transcrits en Coq en tant que prédicats mutuellement inductifs. Au total, les 48 règles d'inférence de la sémantique de Clight correspondent à autant de cas inductifs dans les prédicats.

La figure 5.5 détaille les règles d'exécution d'une séquence d'instructions (règles (16) et (17)) et d'une boucle `while` (règles (18) à (20)). L'exécution d'une séquence de deux instructions concatène les traces résultant de l'exécution des deux instructions (règle (16)). La deuxième instruction n'est exécutée que si la première instruction ne contient pas d'instruction `break`, `continue` ou `return` (règle (17)). L'exécution d'une instruction `break` ou `return` dans le corps d'une boucle provoque la sortie abrupte de la boucle (règle (19)). L'exécution d'une instruction `continue` dans le corps d'une boucle interrompt l'exécution du corps de la boucle et provoque la prochaine itération de la boucle (règle (20)).

Accès à la mémoire. En C, une variable de type `struct`, `union` ou `void` ne peut pas être une valeur gauche. Plus précisément, lorsqu'une variable de `struct` ou `union` a été initialisée (lors de sa déclaration), le seul moyen de modifier sa valeur est de modifier ses champs (ou les champs d'un champ s'il s'agit d'un type composé). De plus, le standard C autorise le passage en paramètre d'une variable de type `struct` ou `union`, mais le passage de paramètres en C étant par valeur, il est nécessaire de passer un pointeur sur le `struct` ou l'`union` pour modifier ce dernier. L'emploi de paramètres de type `struct` ou `union` n'est pas un usage répandu en C. Aussi, dans Clight, le passage d'un paramètre de type `struct` ou `union` n'est pas autorisé.

Séquence d'instructions

$$\frac{G, E \vdash i_1, M \Rightarrow \texttt{Out_normal}, t_1, M_1 \qquad G, E \vdash i_2, M_1 \Rightarrow out, t_2, M_2}{G, E \vdash (i_1; i_2), M \Rightarrow out, t_1@t_2, M_2} \quad (16)$$

$$\frac{G, E \vdash i_1, M \Rightarrow out, t, M' \qquad out \neq \texttt{Out_normal}}{G, E \vdash (i_1; i_2), M \Rightarrow out, t, M'} \quad (17)$$

Boucle `while`

$$\texttt{Out_break} \overset{loop}{\rightsquigarrow} \texttt{Out_normal}$$
$$\texttt{Out_return}(v) \overset{loop}{\rightsquigarrow} \texttt{Out_return}(v)$$

$$\frac{G, E \vdash a, M \Rightarrow v \qquad \texttt{is_false}(v)}{G, E \vdash (\texttt{while}(a)\ i), M \Rightarrow \texttt{Out_normal}, \emptyset, M} \quad (18)$$

$$\frac{\begin{array}{c}G, E \vdash a, M \Rightarrow v \qquad \texttt{is_true}(v) \qquad G, E \vdash i, M \Rightarrow out, t, M' \\ out \overset{loop}{\rightsquigarrow} out'\end{array}}{G, E \vdash (\texttt{while}(a)\ i), M \Rightarrow out', t, M'} \quad (19)$$

$$\frac{\begin{array}{c}G, E \vdash a, M \Rightarrow v \qquad \texttt{is_true}(v) \\ G, E \vdash i, M \Rightarrow out, t_1, M_1 \qquad out \in \{\texttt{Out_normal}, \texttt{Out_continue}\} \\ G, E \vdash (\texttt{while}(a)\ i), M_1 \Rightarrow out', t_2, M_2\end{array}}{G, E \vdash (\texttt{while}(a)\ i), M \Rightarrow out', t_1@t_2, M_2} \quad (20)$$

FIG. 5.5 – Exemples de règles de sémantique de Clight

Afin de prendre en compte tous les types de C, différents modes d'accès à la mémoire ont ainsi été définis : par valeur (pour les entiers, flottants et pointeurs), par référence (pour les tableaux et fonctions), et un dernier accès adapté aux variables de type `struct`, `union` et `void` indiquant que l'accès en mémoire n'est pas possible. Dans la sémantique de Clight, ces modes d'accès servent à aiguiller les fonctions de lecture et d'écriture de la sémantique vers les fonctions correspondantes dans le modèle mémoire (présentées dans la figure 6.3). En effet, les fonctions de lecture et d'écriture de la sémantique n'appellent pas nécessairement celles du modèle mémoire, et elles sont plus générales que ces dernières. Par exemple, étant donnée une adresse `adr` en mémoire, la fonction de la sémantique lisant un flottant (avec accès par valeur) lit effectivement en mémoire ce flottant. Par contre, la fonction de lecture

d'un tableau (avec accès par référence) n'accède pas à la mémoire (i.e. ne lit pas une case de ce tableau) mais renvoie la valeur `adr`.

5.4 Bilan sur Clight

Les difficultés rencontrées lors de la définition d'une sémantique de C adaptée à la preuve de transformations de programmes effectuées par un compilateur sont au nombre de trois.

La première a été de trouver une frontière entre les comportements à proscrire et les comportements à modéliser. En effet, même si par le passé, plusieurs travaux ont porté sur la définition d'une sémantique formelle pour C ([68, 71, 72, 73, 74], ainsi qu'une contribution Coq proposée par Dassault pour traiter des programmes compilés C depuis Lustre), nous n'avons pas pu les réutiliser. Principalement car ils concernent un sous-ensemble trop restreint de C dans lequel le modèle mémoire est imprécis car trop abstrait, et car ils sont pas adaptés à la formalisation sur machine ou à la vérification formelle de transformations de programmes. La sémantique la plus proche de la nôtre est celle de Michael Norrish [68], mais elle se focalise sur une évaluation non déterministe des expressions (que nous ne considérons pas), et n'est adaptée à la preuve de préservation sémantique. Aussi, nous n'avions à notre disposition que la manuel de C et le standard, dans lesquels nous avons dû interpréter des définitions parfois imprécises ou contradictoires.

La deuxième difficulté a été de trouver une frontière entre spécifications et preuves. Il a fallu résister à la tentation d'améliorer la sémantique en garantissant davantage de propriétés au niveau de la sémantique (et du modèle mémoire), permettant de proscrire davantage de violations de C. En effet, une sémantique trop précise peut rendre impossible la preuve de préservation sémantique de transformations de programmes assez complexes (car elle observe des événements qui n'ont pas d'équivalent dans le programme transformé). Or, le compilateur CompCert effectue actuellement quelques passes d'optimisation, et nous souhaiterions le faire évoluer en ajoutant des passes d'optimisation de plus en plus sophistiquées.

Enfin, la troisième difficulté a été de trouver une frontière entre syntaxe et sémantique. Le recours à CIL a aidé, mais nous avons dû adapter CIL le moins possible.

5.5 Cminor

Cminor est un langage généraliste de bas niveau, structuré comme C en expressions, instructions et fonctions. Les différences entre Clight et Cminor sont les suivantes. Au niveau des expressions, les opérateurs arithmétiques ne sont pas surchargés en Cminor et leur comportement ne dépend pas des types de leurs opérandes. Les opérateurs Cminor sur les entiers diffèrent de ceux sur les flottants, et les conversions entre entiers et flottants sont explicites. Cminor ne dispose que d'un type entier (32 bits) et un type flottant (64 bits). Des conversions explicites sont nécessaires afin de gérer des entiers et flottants de taille plus petite. De plus, les calculs d'adresses sont explicites (i.e. il n'y a pas de type tableau en Cminor), ainsi que l'utilisation des fonctions `load` et `store` de lecture et d'écriture en mémoire.

Au niveau des instructions, Cminor ne dispose que de quatre structures de contrôle de plus bas niveau que celles de Clight : instruction conditionnelle `if-then-else`, boucle infinie sans test d'arrêt, blocs d'instructions (i.e. instructions `block` et `exit`), et instruction `return`. Ces structures sont suffisantes pour représenter des graphes de contrôle réductibles (i.e. modélisant des programmes dont les boucles sont imbriquées de façon structurée). Pour arrêter une boucle, celle-ci doit être placée dans un bloc nommé (en utilisant l'instruction `block`), dont la sortie abrupte est rendue possible par l'instruction `exit`. Les instructions `block` et `exit` permettent d'exprimer simplement les instructions `break` et `continue` ainsi que les boucles de C.

Enfin, au niveau des fonctions, seules les variables scalaires (i.e. de type entier, flottant ou pointeur) peuvent être des variables locales. Ceci facilite l'allocation de registres ainsi que certaines optimisations du compilateur CompCert. En Cminor, une variable locale ne réside pas en mémoire, ce qui rend impossible la prise d'adresse d'une variable locale. Aussi, une variable locale Clight avec prise d'adresse (par exemple `&x` ou encore `t[i]`) n'est pas une variable locale Cminor mais un emplacement du bloc de pile courant (qui est donc alloué à l'entrée de la fonction courante et libéré à la sortie de cette fonction). L'accès à une telle variable nécessite l'emploi des instructions `load` et `store` de Cminor, ainsi que des calculs explicites d'adresse.

5.5.1 Formalisation en Coq de la sémantique de Cminor

En 2003, une des premières tâches auxquelles j'ai participé dans le projet CompCert a été la définition en Coq de sémantiques formelles pour Cminor, selon les styles opérationnel à grands pas et dénotationnel. Le but de ce tra-

vail était de choisir un style sémantique adapté à la vérification formelle de propriétés sémantiques.

Le style à grands pas est le plus conventionnel pour définir inductivement une sémantique. Par contre, lorsqu'une sémantique est définie par des relations inductives, la preuve de propriétés de cette sémantique nécessite de simuler l'exécution d'un programme et peut devenir fastidieuse (par exemple, quand Coq n'effectue pas toutes les unifications rendues possibles à partir d'hypothèses, car ces unifications nécessitent de déplier des définitions). De plus, une sémantique définie par des relations inductives ne permet pas de raisonner aisément sur des propriétés contenant des expressions inconnues.

Au contraire, le style dénotationnel permet de simuler aisément l'exécution d'un programme. D'où l'idée présentée par Yves Bertot dans [75] d'encoder une sémantique dénotationnelle par une fonction Coq, afin de prouver ensuite par réflexion des propriétés de cette sémantique. Les notions mathématiques usuellement utilisées en sémantique dénotationnelle n'étant pas modélisées dans cette formalisation en Coq, Yves Bertot qualifie cette sémantique de fonctionnelle. De plus, une définition fonctionnelle d'une sémantique permet de générer automatiquement (grâce au mécanisme d'extraction de Coq) un interprète écrit en Caml, ce que ne permet pas une définition inductive.

L'inconvénient du style fonctionnel disponible à l'époque dans Coq (i.e. celui lié au mot-clé `Fixpoint`) est qu'il est plus facile à utiliser lorsque la fonction à définir est totale et récursive structurelle. En effet, en 2003, il n'existait pas d'autre style fonctionnel en Coq. La définition de la sémantique de Cminor était trop complexe pour pouvoir réutiliser les travaux d'Antonia Balaa et Yves Bertot [76][1]. Ces contraintes ne permettent pas de définir la sémantique formelle d'un langage tel que Cminor, avec instructions de boucle. Cminor étant un langage non trivial à concevoir, il était primordial de pouvoir raisonner aisément sur ce langage, mais aussi de pouvoir interpréter des exemples de programmes. Aussi, avec Laurence Rideau, nous avons étudié le passage à l'échelle (i.e. au langage Cminor) de l'approche présentée dans [75].

Ainsi, nous avons formalisé en Coq une sémantique naturelle définie inductivement, ainsi qu'une sémantique fonctionnelle de Cminor, puis vérifié formellement l'équivalence sémantique entre ces deux styles. La définition fonctionnelle a nécessité d'écrire une fonction d'évaluation plus difficile à manipuler que la relation d'évaluation définie inductivement. En effet, il est nécessaire de décrire non pas l'évaluation à proprement parler (comme le fait la relation

[1]Ces travaux ont évolué et ont été intégrés à un nouveau style fonctionnel de Coq (i.e. celui lié au mot clé `Function`, proposant un principe d'induction associé à une fonction) plusieurs années plus tard.

d'évaluation), mais la fonction du deuxième ordre dont la fonction d'évaluation est le plus petit point fixe.

Le langage Cminor alors défini était un sous-ensemble du langage Cminor actuel. En particulier, les valeurs et les types, ainsi que le modèle mémoire étaient laissés abstraits (et non définis). Cette première définition d'un langage pour le compilateur CompCert a permis d'expérimenter quelques vérifications formelles de propriétés sémantiques. De plus, ces travaux ont mis en évidence des difficultés qu'il a été nécessaire de surmonter pour définir les sémantiques formelles des langages du compilateur CompCert : représentation en Coq des fonctions partielles, lourdeur de la définition de la sémantique liée à la taille du langage considéré (par exemple variété des opérateurs arithmétiques, utilisation de fonctions d'évaluation mutuellement récursives), rôle du modèle mémoire, prise en compte des cas d'erreur dans la sémantique. Enfin, ces travaux ont de plus permis de tester les premières possibilités d'extraction en Coq à partir de définitions inductives [77].

Actuellement, la sémantique formelle de Cminor est définie dans un style à grands pas, de façon similaire à celle de Clight. Ses cinq jugements d'évaluation sont détaillés dans la figure 5.6. Les environnements G et E ainsi que la mémoire M sont ceux de Clight. Par rapport aux jugements d'évaluation de Clight (c.f. figure 5.4), l'environnement E associe des valeurs à des variables locales, puisqu'en Cminor les variables locales ne résident pas en mémoire. Aussi, E est modifié durant l'exécution des instructions. Le paramètre sp désigne le pointeur de pile de la fonction courante (c.f. figure 6.1). sp ne fait pas partie des jugements d'évaluation de la sémantique de Clight car dans cette sémantique, il est moins utilisé et donc calculé si besoin.

Les instructions de Csharpminor sont celles de Cminor. Aussi, les sorties d'instruction dans la sémantique de Csharpminor sont adaptées en conséquence.

5.5.2 Un front-end pour Compcert

Une première traduction de Clight (il s'agissait de la première version de Clight, plus simple que le langage actuel car n'utilisant pas CIL) vers Cminor a été spécifiée et prouvée en Coq. Ce travail a fait l'objet du stage de Zayanah Dargaye. Cette preuve a été jugée particulièrement difficile, et un nouveau langage intermédiaire entre Clight et Cminor, appelé Csharpminor a donc été conçu dans le but de simplifier la preuve précédente. Ses instructions sont celles de Cminor. La différence entre Csharpminor et Cminor concerne le traitement des variables locales. La vérification formelle de la préservation

Sortie d'instruction :

out ::=	`Out_normal`	aller à la prochaine instruction
|	`Out_exit`(n)	sortie du bloc de niveau d'imbrication $n+1$
|	`Out_return`	sortie de fonction
|	`Out_return`(v)	sortie de fonction, avec valeur renvoyée v

G, E, sp	$\vdash$	$a, M \Rightarrow v$	(expressions)
G, E, sp	$\vdash$	$a^*, M \Rightarrow v^*$	(liste d'expressions)
G, E, sp	$\vdash$	$i, M \Rightarrow out, t, E', M'$	(instructions)
G	$\vdash$	$f(v^*), M \stackrel{f}{\Rightarrow} v, t, M'$	(invocations de fonctions)
	$\vdash$	$p \Rightarrow v, t$	(programmes)

FIG. 5.6 – Jugements de la sémantique de Cminor.

sémantique de la traduction de Clight vers Csharpminor (et l'adaptation de CIL pour CompCert) a fait l'objet du stage de Thomas Moniot. La traduction de Csharpminor vers Cminor a été traitée par Xavier Leroy.

De Clight à Csharpminor

La figure 5.7 donne les grandes lignes de la traduction de Clight vers Csharpminor. Dans les expressions, les opérations qui dépendent des types sont traduites. C'est lors de cette étape qu'est résolue la surcharge des opérateurs (c.f. (1) dans la figure). Tout opérateur arithmétique de Clight est traduit en un opérateur équivalent mais particularisé par le type des opérandes (qui est nécessairement le même dans le cas de deux opérandes, puisque toutes les conversions implicites ont été insérées préalablement par CIL). Des calculs d'adresses sont également effectués afin de traduire les accès aux tableaux (c.f. (2)).

Les variables de type tableau ou enregistrement résident en mémoire en Csharpminor. Aussi, un accès à une case de tableau (resp. à un champ d'un enregistrement) est transformé en un calcul de l'adresse de la case du tableau (resp. de la cellule stockant le champ), et ce calcul est utilisé par l'instruction d'accès à la mémoire (c.f. (3)).

Cette passe traduit également les structures de contrôle de Clight en structures de contrôle Cminor (c.f. (4)). Les boucles à la C sont transformées en boucles infinies avec blocs et sortie abrupte. Les constructions `if` et `switch` de Clight sont transformées en constructions `if` et `switch` de Cminor.

• (1) Résolution de la surcharge des opérateurs arithmétiques

`int x, y ; ... x + y ...` $\xrightarrow{traduction}$ `addint(x, y)`

`double x, y ; ... x + y ...` $\xrightarrow{traduction}$ `addfloat(x, y)`

• (2) Calculs d'adresses

`int * p, i ; ... p[i] ...` $\xrightarrow{traduction}$ `addint(p, mulint(i, 4))`

`struct intlist * s ; ... s->tl ...` $\xrightarrow{traduction}$ `addint(s, 4)`

• (3) Insertion de `load` et `store` lors des accès aux valeurs gauches

`int p[2][2] ; p[0][0] = 42 ;` $\xrightarrow{traduction}$ `store(int32, p, 42)`

`int **p ; p[0][0]=42 ;` $\xrightarrow{traduction}$ `store(int32, load(int32,p),42)`

• (4) Exemple de codage des structures de contrôle de Clight dans celles de Csharpminor

```
for (s1 ; e ; s2) {



  ...
  exit ;
  ...
  continue ;
  ...



}
```

```
s1 ;
block { loop {
  if (!e) exit 0 ;
  block {
    ...
    exit 1 ;
    ...
    exit 0 ;
    ...
  }
  s2 ;
} }
```

FIG. 5.7 – Traduction de Clight à Csharpminor

La propriété de préservation sémantique de cette traduction est illustrée par le diagramme commutatif de la figure 5.8, dans lequel la colonne de gauche (resp. droite) représente l'exécution d'une instruction Clight (resp. Csharpminor). Ce diagramme est une instance de celui présenté dans la figure 2.6 (c.f. page 19). La relation $E \rightarrowtail E'$ exprime la correspondance entre l'environnement local E de Clight et E', l'environnement local de Csharpminor. Cette relation précise en particulier que toute variable de E est également une variable de E'. La relation $\texttt{out} \simeq_M^{M'} \texttt{out}'$ exprime la correspondance entre une sortie d'instructions `out` de Clight et une sortie d'instructions `out`' de Csharpminor. Dans le cas d'une sortie de fonction, il est nécessaire d'évaluer et de comparer les valeurs renvoyées par la fonction lors des exécutions en Clight et en Csharpminor. Aussi, la correspondance utilise les états mémoire M et M'. Cette correspondance illustre la traduction des instructions Clight `break` et

`continue` en instructions Csharpminor `exit`.

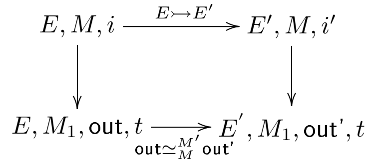


Fig. 5.8 – Préservation sémantique de la traduction de Clight vers Csharpminor

De Csharpminor à Cminor

En plus de traduire les programmes Clight en programmes de plus bas niveau, le deuxième objectif de la traduction vers Cminor est de modifier la représentation des variables afin de faciliter les passes ultérieures de compilation (certaines optimisations ainsi que l'allocation de registres). Or, conformément au standard C, un bloc de mémoire est alloué pour chaque variable Clight, et l'adresse de chaque variable peut être évaluée (au moyen de l'opérateur `&` du C, ou de l'opérateur `[ ]` d'accès à une case de tableau). Aussi, la traduction de Csharpminor à Cminor isole les variables locales dont l'adresse est prise (i.e. celles pouvant créer de l'aliasing) des autres. Ainsi, en Cminor, ces variables sont allouées explicitement en pile.

La figure 5.9 montre un exemple de traduction, dans lequel la variable i est placée en pile dans un bloc de 4 octets (d'où la déclaration `stack 4 ;`). L'identifiant i disparaît donc et l'accès à i se fait en calculant son adresse dans la pile (ici 0 car i est la première variable allouée en pile).

```
{                          {
    int i ;                   stack 4 ;
    int j ;                    var j ;
    f (&i) ;                   f (stackaddr(0)) ;
    i = ... ;                  store (int32, stack(0), ...) ;
    ... = ... i ... ;          ... = ...load (int32, stack(0))... ;
    j = ... ;                  j = ... ;
    ... = ... j ... ;          ... = ... j ... ;
}                           }
```

Fig. 5.9 – Traduction de Csharpminor à Cminor

La propriété de préservation sémantique de cette traduction a nécessité de mettre en correspondance des états mémoire particuliers. Cette correspondance repose sur la notion d'injection mémoire présentée dans le chapitre suivant (c.f. page 69).

6 Un modèle mémoire pour le compilateur CompCert

Ce chapitre commente une partie du matériau publié dans [10, 11]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Xavier Leroy. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Thibaud Tourneur et François Armand [stage 7, stage 9].

Un modèle mémoire décrit la géographie de la mémoire, ainsi que les opérations d'accès à celle-ci : allocation de mémoire, libération de mémoire, lecture d'une valeur en mémoire et écriture d'une valeur en mémoire.

La géographie de la mémoire précise les différentes zones de la mémoire, ainsi que l'organisation de chaque zone en zones plus élémentaires et en cellules. Traditionnellement, la mémoire vue depuis un processus est partagée en cinq zones : la zone stockant les instructions du programme (qui n'est pas modifiable au cours de l'exécution d'un programme), celle stockant les variables globales, la *pile* stockant les variables locales et les paramètres de fonctions qui ne sont pas dans des registres, le *tas* stockant les valeurs des objets ayant été alloués dynamiquement, et la zone rassemblant les cellules n'ayant pas encore été allouées. Au cours de l'exécution d'un programme, seules les tailles du tas, de la pile et donc de la zone non allouée varient. Classiquement, la mémoire est représentée par un unique tableau, dans lequel la zone non allouée figure entre la pile et le tas, comme le montre la figure 6.1. Les indices de ce tableau sont des adresses et chaque case contient un mot mémoire. Chaque zone est repérée par un pointeur. Par exemple, le pointeur de pile *sp* permet d'accéder à la pile.

Cette représentation de la mémoire est de trop bas niveau pour pouvoir exprimer des propriétés attendues du modèle mémoire, et donc pour pouvoir raisonner sur ce modèle. En effet, la sémantique d'un langage tel que C utilise les opérations d'accès à la mémoire (par exemple, déclarer une variable C nécessite une allocation en mémoire, ou encore une affectation de variable effectue au moins une lecture en mémoire, suivie d'une écriture en mémoire). Une telle sémantique exige que les opérations d'accès à la mémoire vérifient de nombreuses propriétés. Par exemple, il est nécessaire d'assurer que toute

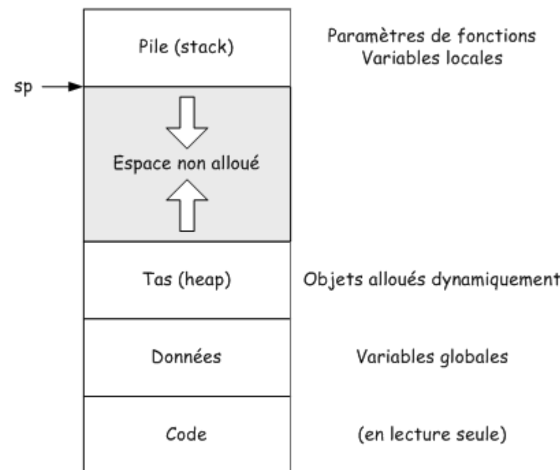


FIG. 6.1 – La mémoire d'un compilateur vue depuis un processus

allocation en mémoire ne modifie aucune des valeurs pouvant être lues auparavant.

Au contraire, le standard C considère une représentation de plus haut niveau de la mémoire, dans laquelle la mémoire est une collection de blocs, dont la taille varie en fonction du type de la valeur à stocker. Une représentation trop abstraite de la mémoire, dans laquelle la mémoire est seulement une collection de blocs disjoints n'est pas non plus exploitable par une sémantique d'un langage tel que C. En effet, afin de prendre en compte l'aliasing entre pointeurs ainsi que l'arithmétique de pointeurs du C, il est nécessaire de modéliser des propriétés de chevauchement partiel entre zones de la mémoire.

Plus généralement, quel que soit le langage étudié, les propriétés attendues de la géographie d'un modèle mémoire sont souvent classifiées en des propriétés de séparation, d'adjacence ou de confinement [78]. Différents niveaux de granularité et donc d'exigence existent. La granularité la plus fine s'applique à des propriétés relatives aux cellules de la mémoire. Les propriétés peuvent être moins précises et s'appliquer à des régions plus ou moins étendues de la mémoire. Enfin, des propriétés peuvent être antagonistes dans un modèle donné. Par exemple, établir des garanties fortes de séparation entre les cellules de la mémoire (comme le font less modèles à la Burstall-Bornat [79]) peut se faire

au détriment des propriétés d'adjacence (qui deviennent alors fausses). Notre modèle mémoire possède de nombreuses propriétés relevant des trois familles.

La difficulté de formalisation a donc consisté à trouver le bon niveau d'abstraction pour définir un modèle mémoire adapté aux sémantiques des langages du compilateur CompCert, et suffisamment précis pour exprimer des propriétés de bas niveau modélisant les pointeurs du C (et leur arithmétique). L'existence de comportements indéfinis dans le standard C a accru cette difficulté. Il a été nécessaire d'interpréter le standard C et de décider d'interdire certains comportements indéfinis. Par exemple, la conversion d'un entier en un pointeur est proscrite dans CompCert. Enfin, différentes itérations ont été nécessaires avant d'aboutir au modèle final. En outre, ce modèle étant utilisé par tous les langages du compilateur CompCert, il a été de plus nécessaire de le généraliser afin de l'instancier ensuite sur différents langages.

Au final, les principales caractéristiques de notre modèle mémoire sont les suivantes : généricité, séparation en blocs indépendants et de taille variable, prise en compte d'une valeur indéfinie (pouvant être générée pendant la compilation), tests aux bornes des blocs, adéquation à la preuve sur machine. La suite de cette section justifie ces choix et détaille les propriétés prises en compte par notre modèle.

Le modèle mémoire a d'abord été spécifié à un niveau abstrait, dans lequel des définitions sont laissées abstraites (*i.e.* un type n'est pas défini, seule la signature d'une fonction est connue), et des propriétés sont posées en axiomes. Certaines propriétés axiomatisent des notions non définies (abstraites). Ensuite, d'autres propriétés ont été prouvées à partir des précédentes. Puis, une implémentation du modèle abstrait a été définie. À ce niveau, toutes les définitions abstraites sont implémentées et tous les axiomes sont prouvés. Enfin, de nouvelles propriétés spécifiques à l'implémentation choisie ont été énoncées et prouvées.

Contrairement aux articles publiés sur le modèle mémoire qui présentent ce dernier en séparant les niveaux d'abstraction [10, 11], dans ce chapitre, je me focalise sur les propriétés du modèle mémoire. Ce chapitre présente d'abord les traits génériques du modèle mémoire. Ensuite, il détaille les propriétés abstraites requises pour décrire la sémantique de C. Puis, il précise quelques propriétés supplémentaires. Enfin, il décrit brièvement un modèle mémoire concret.

6.1 Généricité

Le modèle mémoire est paramétré par une collection de valeurs, notée `val`, pouvant être stockées en mémoire. Il existe une valeur particulière, notée `undef`, qui représente toute valeur indéfinie. Cette valeur peut être générée lors d'une phase de compilation. Elle représente par exemple la valeur d'une variable non initialisée, ou encore la valeur d'une variable ayant été lue sur une taille plus grande que la taille nécessaire pour stocker la variable. Dans le standard C, le comportement des variables non initialisées est laissé indéfini .

Un deuxième paramètre est le type `memtype` des données pouvant être stockées en mémoire. Il permet de décrire finement la nature des données stockées en mémoire (i.e. plus précisément qu'en connaissant seulement le type de la donnée). Par exemple, il peut être intéressant de distinguer dans la mémoire un entier stocké sur 4 octets d'un entier stocké sur 8 octets, même si dans le langage considéré, le type entier ne porte pas d'information de taille. Dans le cas de C, ce type correspond au type des expressions. Par contre, pour les autres langages du compilateur, il ne correspond pas au type des expressions du langage considéré.

`memtype` possède de plus une relation de compatibilité. Certaines propriétés du modèle mémoire préservent la relation de compatibilité ; elles restent vraies même si une valeur est convertie en une valeur d'un type compatible. De plus, la taille de tout type (de `memtype`) est calculable. Enfin, `memtype` sert à modéliser les `cast` implicites de C. Ce type possède une fonction de conversion modélisant les conversions de valeurs du C effectuées par exemple lors d'une séquence d'écriture et de lecture d'une même valeur.

6.2 Propriétés requises pour décrire la sémantique de C

Cette section décrit de façon informelle les propriétés du modèle mémoire qui sont utilisées dans la sémantique de Clight. Ces propriétés sont regroupées en quatre familles. Elles expriment la séparation entre blocs de mémoire, l'adjacence entre sous-blocs de la mémoire, le confinement entre sous-blocs, ainsi que les effets des opérations d'accès à la mémoire.

6.2.1 Séparation

Les propriétés de séparation sont principalement définies au niveau abstrait.

Notion de bloc

La mémoire est représentée à gros grains, comme une collection de blocs de taille variable. Un bloc de mémoire représente la quantité de mémoire allouée lors d'une allocation. Ainsi, par exemple, la déclaration d'un tableau alloue un bloc dans lequel seront stockées toutes les valeurs du tableau. Un autre exemple est la déclaration d'un enregistrement de type `struct` qui alloue un seul bloc permettant de stocker les valeurs de tous les champs de cet enregistrement.

La mémoire est une collection de blocs indépendants. Indépendant signifie qu'un accès valide à un bloc de la mémoire ne permet pas d'accéder à un autre bloc. La propriété de séparation des blocs est donc assurée par définition de la mémoire. Souvent, cette propriété est difficile à garantir dans les modèles de bas niveau. La validité d'un bloc est définie au niveau concret du modèle mémoire de la façon suivante : un bloc est valide dans une mémoire s'il a été alloué, et non encore libéré.

Fraîcheur des variables

Certaines propriétés de l'allocation en mémoire concernent la fraîcheur des blocs nouvellement alloués. Ces propriétés relèvent du modèle mémoire, mais elles sont utilisées dans la sémantique du langage Clight. En effet, celle-ci garantit que les deux zones de la mémoire correspondant à deux variables distinctes, ou encore à deux zones ayant été allouées par deux appels distincts à la fonction `malloc` sont disjointes. Dans un langage de plus haut niveau tel que Java, cette propriété n'a pas besoin d'être assurée par le modèle mémoire. Dans un langage de plus bas niveau tel qu'un langage machine, ces restrictions n'ont pas lieu d'être.

6.2.2 Adjacence

Les propriétés d'adjacence entre cellules de la mémoire ne sont définies que dans l'implémentation du modèle mémoire. Chaque bloc de mémoire se comporte comme un tableau d'octets. La figure 6.2 montre un exemple de bloc. Un bloc est composé de cellules élémentaires (indicées depuis une borne inférieure quelconque jusqu'à une borne supérieure quelconque). Une adresse en mémoire est une paire constituée d'un bloc et d'un décalage dans ce bloc (noté *ofs* dans la figure) permettant d'accéder à une cellule quelconque du bloc.

Étant donné un pointeur désignant l'adresse d'un objet de référence, l'arithmétique de pointeur définit comment accéder aux objets voisins de cet objet,

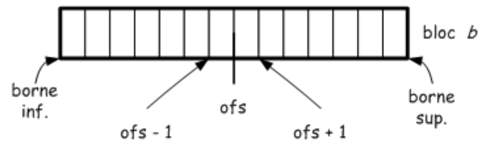


FIG. 6.2 – Un exemple de bloc de mémoire

c'est-à-dire les objets décalés (en avant ou en arrière selon la valeur de l'entier représentant ce décalage) d'une certaine position par rapport à l'adresse de l'objet de référence. Le standard C ne définit pas précisément la notion de voisin ; elle permet par exemple le débordement de tableau. Ainsi, l'arithmétique de pointeurs permet de déborder d'un tableau suite à des décalages successifs.

Notre définition des blocs facilite la modélisation de l'arithmétique de pointeurs dans la sémantique de Clight. Aussi, cette sémantique proscrit le débordement de tableau (i.e. assure que tout accès à un bloc représentant un tableau ne débordera pas de ce tableau) et l'arithmétique de pointeur n'a de sens que si elle consiste à accéder aux différents sous-blocs d'un bloc donné. Le modèle mémoire de CompCert est donc plus restrictif que le standard C.

Le modèle mémoire modélise l'alignement des données en mémoire (à l'aide de `memtype`), que certains compilateurs effectuent (sur certaines architectures) pour des raisons de performance et pour faciliter l'exécution du code. L'alignement impose qu'une donnée de taille n doit être placée en mémoire à une adresse multiple de n (sauf pour un `struct` dont la taille doit être un multiple de la taille du processeur). Ainsi, lors de l'allocation d'un `struct`, ses champs sont contenus dans un seul bloc, dans l'ordre de leur déclaration, mais ils ne sont pas adjacents car séparés par des bits de décalage rajoutés par le compilateur si ce dernier pratique l'alignement. Par contre, le compilateur CompCert actuel ne pratique pas l'alignement des données (car le processeur Power PC n'impose pas de contrainte d'alignement des données).

6.2.3 Confinement

Les propriétés de confinement sont énoncées aux niveaux abstrait et concret du modèle mémoire.

Typage faible

Un langage fortement typé garantit à la compilation qu'une valeur ayant été écrite en mémoire en tant que valeur d'un type τ sera toujours lue en tant que valeur de type τ. Cette garantie est assurée à la compilation dans le cas d'un langage statiquement typé. Elle est assurée à l'exécution dans le cas d'un langage dynamiquement typé. Le langage C et la plupart des langages intermédiaires du compilateur CompCert sont faiblement typés. Dans un langage faiblement typé, cette propriété n'est pas garantie. Aussi, en C, la valeur d'une variable `u` de type `union {short i ; double f ;}` peut être par exemple un (petit) entier (c'est la valeur de `u.i`), ou encore un (grand) réel (c'est la valeur de `u.f`).

Ainsi, en C, il est possible d'écrire en mémoire une valeur de type entier (en exécutant par exemple `u.i=2 ;`), puis de lire ensuite cette valeur comme étant de type réel (en exécutant par exemple `x=u.f ;`), donc d'un type de taille plus grande. Si cette lecture est autorisée, la valeur lue dépend du contenu précédent du bloc de mémoire stockant `u`. Le standard C précise qu'un tel comportement est indéfini. La plupart des compilateurs C acceptent un tel comportement. Par exemple, le compilateur `gcc` affiche `0.` comme valeur de `u.f`. Par contre, dans CompCert, un tel comportement n'est pas autorisé.

Ainsi, dans CompCert, toute donnée écrite en mémoire en tant que valeur d'un type τ ne peut être lue ensuite qu'en tant que valeur d'un type τ' compatible avec τ. Si cette condition n'est pas vérifiée, la lecture échoue. Ce choix nécessite de considérer dans les opérations de lecture et d'écriture un paramètre supplémentaire dénotant le type selon lequel la donnée est lue ou écrite en mémoire.

Du fait des garanties de compatibilité assurées dans les écritures suivies de lectures, notre modèle mémoire est plus strict que le standard C. Par exemple, il ne permet pas de lire une valeur de type pointeur vers τ lorsque cette valeur a été stockée comme étant de type pointeur sur τ', avec τ différent de τ'. Ce comportement est fréquent en C lorsque τ et τ' sont de même taille. Il se produit par exemple pour mettre à zéro efficacement tous les champs d'un `struct` en écrivant `struct { ... } v = { 0 } ;`, ce qui est parfois utilisé dans les programmes de systèmes d'exploitation.

6.2.4 Propriétés impliquant les opérations d'accès à la mémoire

Les quatre opérations d'accès à la mémoire (*i.e.* allocation, libération, lecture et écriture) sont également spécifiées au niveau abstrait (*i.e.* elles ne sont définies que par leur signature) et axiomatisées. Ces opérations abstraites sont

très générales et elles peuvent échouer (voir figure 6.3). Les opérations de lecture et d'écriture en mémoire sont paramétrées par le type `memtype` des données pouvant être stockées en mémoire, qui indique alors la portion de la valeur devant être lue ou écrite en mémoire. Ainsi, il est par exemple possible d'écrire une valeur dans un bloc sans nécessairement écrire dans tous les octets de ce bloc. Ceci peut se produire en C lors d'une affectation d'un champ d'un `union`.

Les opérations d'accès à la mémoire sont axiomatisées par des propriétés de bonne formation des variables (*good variables properties* [80]) exprimant le fait que les blocs de mémoire se souviennent correctement des valeurs ayant été stockées. Par exemple, la dernière propriété de la figure 6.3 exprime que l'écriture d'une valeur dans un bloc b ne modifie pas le contenu de tout bloc différent de b.

Géographie de la mémoire

Soient `mem` et `block` deux types non définis au niveau abstrait.

$$\begin{array}{rcl} \texttt{empty} & : & \texttt{mem} \\ \texttt{valid_block} & : & \texttt{mem} \times \texttt{block} \rightarrow \texttt{Prop} \\ \texttt{bounds} & : & \texttt{mem} \times \texttt{block} \rightarrow \mathbb{Z} \times \mathbb{Z} \\ \texttt{fresh_block} & : & \texttt{mem} \times \texttt{block} \rightarrow \texttt{Prop} \end{array}$$

Opérations de gestion de la mémoire

$$\begin{array}{rcl} \texttt{alloc} & : & \texttt{mem} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \texttt{option}(\texttt{block} \times \texttt{mem}) \\ \texttt{free} & : & \texttt{mem} \times \texttt{block} \rightarrow \texttt{option mem} \\ \texttt{load} & : & \texttt{memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \rightarrow \texttt{option val} \\ \texttt{store} & : & \texttt{memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \times \texttt{val} \rightarrow \texttt{option mem} \end{array}$$

Propriétés de bonne formation des variables

- Si $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ et $b' \neq b$, alors $\texttt{load}(\tau, m', b', \mathit{ofs}) = \texttt{load}(\tau, m, b', \mathit{ofs})$
- Si $\texttt{free}(m, b) = \lfloor m' \rfloor$ et $b' \neq b$, alors $\texttt{load}(\tau, m', b', \mathit{ofs}) = \texttt{load}(\tau, m, b', \mathit{ofs})$
- Si $\texttt{store}(\tau, m, b, \mathit{ofs}, v) = \lfloor m' \rfloor$ et $\tau \sim \tau'$, alors $\texttt{load}(\tau', m', b, \mathit{ofs}) = \texttt{convert}(v, \tau')$
- Si $\texttt{store}(\tau, m, b, \mathit{ofs}, v) = \lfloor m' \rfloor$ et $b' \neq b \vee \ \mathit{ofs}' + |\tau'| \leq \mathit{ofs} \vee \mathit{ofs} + |\tau| \leq \mathit{ofs}'$, alors $\texttt{load}(\tau', m', b', \mathit{ofs}') = \texttt{load}(\tau', m, b', \mathit{ofs}')$

FIG. 6.3 – Spécification abstraite la mémoire (extrait). Le type `option` et la notation $\lfloor x \rfloor$ sont définis dans le chapitre 2, page 21.

D'autres axiomes ont été définis au niveau abstrait afin de détailler comment les relations précédemment définies (par exemple la validité d'un bloc dans une mémoire) sont préservées par les opérations d'accès à la mémoire. Ces

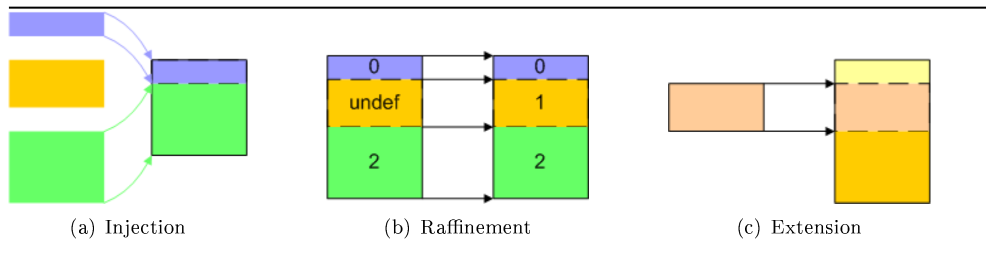
relations ont ainsi été axiomatisées. Enfin, des propriétés dérivées des axiomes ont également été prouvées au niveau abstrait. Par exemple, si m désigne une mémoire dans laquelle un bloc b est alloué (ce qui modifie m en m'), alors tout accès valide à b dans m est également valide dans m'.

6.3 Propriétés relatives aux transformations de la mémoire

Durant les différentes passes de compilation d'un programme, les blocs alloués en mémoire sont transformés. Par exemple, la déclaration d'une variable C locale à une fonction alloue un bloc en mémoire. Lors d'une des phases de compilation, cette variable pourra ensuite être stockée dans plusieurs sous-blocs d'une trame de pile, qui sera ensuite étendue afin de stocker des registres devant être vidés en mémoire, à l'issue de la phase d'allocation de registres.

Les accès à la mémoire sont également transformés. Par exemple, une lecture en mémoire peut être supprimée car la valeur lue est également stockée dans un registre. Prouver que de telles transformations préservent la sémantique (du langage source du compilateur) nécessite également de raisonner sur le modèle mémoire, et de modéliser de nouvelles relations de confinement entre mémoires. En effet, dans ces preuves, il est nécessaire de relier la mémoire utilisée par un programme initial avec la mémoire utilisée par le programme transformé correspondant, et de prouver des lemmes de simulation entre les opérations d'accès à la mémoire effectuées par les deux programmes.

Lors de la traduction de Csharpminor à Cminor, des variables sont sorties de la mémoire pour être placées dans l'environnement des variables locales (c.f. chapitre 5 page 58). Du point de vue de la mémoire, des blocs ayant été alloués lors d'une exécution en Clight ne le sont plus lors de l'exécution en Cminor. Une relation d'injection entre mémoires modélise cette situation. Elle est illustrée dans le premier schéma de la figure 6.4.



(a) Injection (b) Raffinement (c) Extension

FIG. 6.4 – Transformations de la mémoire durant la compilation

La notion d'injection entre mémoires a été généralisée par Xavier Leroy à une notion plus générale de plongement mémoire. En plus de la première passe de compilation, les plongements mémoire sont utilisés lors de deux autres passes de compilation de CompCert, l'allocation de registres et le vidage de registres en mémoire. Ces deux plongements correspondent aux deux derniers schémas de la figure 6.4.

6.4 Modèle concret

Un modèle concret implémentant le modèle abstrait précédent et adapté au compilateur CompCert a été défini. Dans la figure 6.5, la mémoire est représentée par un quadruplet (N, B, F, C). N désigne le prochain bloc qui sera alloué. Ce bloc est toujours défini car dans ce modèle concret, le nombre de blocs est infini (et donc l'allocation est déterministe et n'échoue jamais). La fonction B associe à chaque bloc ses bornes. La fonction F indique pour chaque bloc b si b a été libéré (auquel cas $F(b)$ vaut `true`) ou non (auquel cas $F(b)$ vaut `false`).

Les blocs ayant été libérés ne sont jamais réutilisés afin de faciliter la preuve de préservation sémantique. Les blocs sont identifiés par des entiers (i.e. le type `block` est $\mathbb{N}$). Les axiomes du modèle abstrait ont été prouvés, ainsi que de nouvelles propriétés, relatives aux dimensions des blocs, et exprimant la compatibilité entre les dimensions d'un bloc et le décalage considéré dans une opération de lecture ou d'écriture.

Mémoire *mem* : (N, B, F, C)

N : `block`	bloc suivant
B : `block` $\rightarrow \mathbb{Z} \times \mathbb{Z}$	bornes (inférieure et supérieure)
F : `block` $\rightarrow$ `bool`	blocs ayant été libérés
C : `block` $\rightarrow \mathbb{Z} \rightarrow$ `option(memtype` $\times$ `val)`	contenu (cellules) d'un bloc

FIG. 6.5 – Extrait de l'implémentation de la mémoire.

Bien qu'étant infini, ce modèle reflète assez fidèlement la gestion de la mémoire d'un compilateur classique. Il a été choisi après avoir étudié un modèle fini, dans lequel le nombre de blocs de chaque zone de la mémoire est limité (ainsi que la taille de chaque cellule), et dans lequel l'allocation échoue lorsqu'il n'existe plus de bloc disponible. Dans un modèle fini, la compilation d'un

programme échoue dès qu'une allocation de variable échoue. Or, dans le processus de compilation, chaque traduction génère des allocations. Il existe alors de nombreuses opportunités d'échec de la compilation.

La figure 6.4 montre les trois évolutions possibles d'un bloc durant la compilation. Lorsque les passes de compilation se succèdent, le nombre de blocs alloués diminue (par exemple dans le cas (a) de la figure 6.4), mais la taille de chaque bloc alloué augmente (c.f. figure 6.4 (c)). Par exemple, un bloc alloué initialement pour stocker des variables locales à une fonction disparaît ensuite lorsque ces variables sont stockées en pile. Enfin, certaines traductions allouent des blocs afin de stocker des valeurs temporaires (résultant de l'évaluation de longues expressions contenant plusieurs appels de fonctions et variables). Un exemple d'augmentation de la taille d'un bloc alloué est le suivant : la taille d'une trame de pile n'est connue qu'à la fin de la compilation, c'est-à-dire à l'issue de l'allocation de registres (qui décide quelles variables doivent être vidées en mémoire, et qui peut donc faire grossir chaque trame de pile).

Ainsi, les traductions ne préservent pas le contenu des blocs. Dans un modèle mémoire fini, elles peuvent donc échouer parce qu'elle traduisent des blocs en blocs plus gros. L'exécution d'un programme traduit peut donc échouer bien que l'exécution du programme source n'échoue pas. La propriété prouvée est donc plus faible que celle prouvée dans un modèle infini : si la compilation d'un programme S en le programme T n'échoue pas, si l'exécution de S termine avec une mémoire M, et si l'exécution de T termine, alors elle termine avec la même mémoire M. C'est pourquoi un modèle mémoire infini a été finalement choisi dans CompCert.

À l'avenir, plutôt que passer à un modèle fini, nous envisageons d'effectuer des analyses statiques afin de déterminer une approximation de la quantité de mémoire allouée pour un programme source donné, et donc de se ramener sans restriction à un modèle mémoire fini.

6.5 Bilan

Le modèle mémoire présenté dans ce chapitre a été développé à partir d'un premier modèle de bas niveau, plus simple que le modèle actuel, et non générique. Ce développement a été effectué par Benjamin Grégoire, dans le cadre de l'ARC Concert.

En plus du souci principal qui a été de trouver un niveau d'abstraction adapté à la preuve de transformations de programmes effectuées par un compilateur, le développement de ce modèle mémoire a été l'occasion de mener différentes expériences en Coq. Par exemple, le modèle mémoire a permis

de tester et d'améliorer l'extraction de Coq (implémentée principalement par Pierre Letouzey, membre du projet CompCert), à partir de spécifications de plusieurs milliers de lignes. Ces améliorations concernent la rapidité de génération du code extrait (la correction d'un bug a permis le passage de quelques heures à quelques secondes) ainsi que l'extraction d'enregistrements Coq vers des enregistrements Caml.

Une autre expérience concerne les modules de Coq. Un premier développement plus abstrait, très modulaire et générique, reposant sur l'utilisation de foncteurs et de structures de données de type table d'associations a d'abord été effectué. Il a été l'occasion d'expérimenter la bibliothèque `FSets` de Coq (qui à l'époque était toute récente [55]) dans le cadre de CompCert. La solution actuellement choisie dans CompCert utilise les modules de Coq uniquement pour définir le modèle mémoire à différents niveaux d'abstraction. Elle n'utilise pas les tables d'associations, qui sont par contre intensivement utilisées dans les sémantiques des langages de CompCert pour définir les environnements d'exécution (elles ne l'étaient pas à l'époque du premier développement).

7 Quelles sémantiques pour la preuve de programmes ?

Ce chapitre commente une partie du matériau publié dans [12, 13]. Sauf mention contraire, les travaux présentés dans ce chapitre ont été effectués en collaboration avec Andrew W.Appel. Sur les thèmes présentés dans ce chapitre, j'ai encadré le stage de M2 de Sonia Ksouri ([stage 5], sur le thème de la logique du « rely guarantee ») et piloté le post-doctorat de Keiko Nakata (sur le thème de la logique de séparation).

LA PREUVE DE PROGRAMME est une approche complémentaire à la compilation certifiée, qui permet de vérifier des propriétés fonctionnelles des programmes. Le compilateur certifié CompCert garantit que tout programme compilé se comporte comme le programme source l'ayant engendré. Les propriétés garanties par CompCert sont donc des propriétés sur le langage source du compilateur. Par contre, la preuve de programmes permet d'établir des propriétés exprimées dans ce langage. Ce chapitre présente des travaux ayant été effectués dans le but de relier le compilateur CompCert à des preuves de programmes Cminor vérifiées en Coq. La figure 7.1 illustre cette démarche.

Ce chapitre introduit d'abord la logique de séparation, qui est une approche dédiée à la preuve de programmes manipulant des pointeurs. Bien que le langage de programmation étudié soit Cminor, j'ai dû définir une nouvelle sémantique de ce langage, utilisant un style à petits pas. La raison principale de ce choix est de permettre d'étendre le travail réalisé dans un contexte concurrent. La deuxième partie de ce chapitre justifie ce choix et détaille cette sémantique à petits pas. Par ailleurs, la technologie sous-jacente à la preuve des programmes est l'application de règles à la Hoare. La troisième partie de chapitre décrit ces règles pour le langage Cminor, ainsi que leur correction par rapport à la sémantique à petits pas de Cminor. Enfin, ce chapitre précise une extension de ces travaux à un contexte concurrent.

7.1 Vérification formelle de programmes en logique de séparation

Étant donnée une sémantique axiomatique définissant les exécutions valides de chaque instruction du langage source dans lequel est écrit un programme,

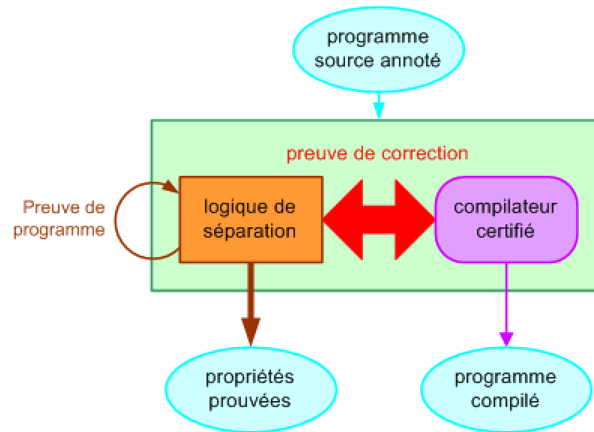


FIG. 7.1 – Preuve de programme et compilation certifiée

prouver un programme annoté par des assertions consiste à appliquer les règles de cette sémantique afin de réduire le programme annoté en un ensemble de conditions de vérification, puis à vérifier la validité de ces conditions de vérification (c.f. page 14). La preuve de programme peut être vérifiée formellement. Il s'agit alors de vérification formelle de programme.

Les assertions qu'il est nécessaire d'écrire dans les programmes sont les pré et post-conditions des fonctions, les invariants de boucle et éventuellement les conditions de terminaison des boucles. Un intérêt de la vérification formelle de programme est que les assertions sont écrites dans un langage proche du langage de programmation. Par contre, le principal inconvénient réside dans la difficulté à inventer les assertions (en particulier les invariants peuvent être difficiles à trouver, les assertions peuvent devenir difficiles à comprendre). Aussi, il est nécessaire de mécaniser le plus possible la vérification formelle des assertions et de disposer d'un langage d'assertions facilitant l'expression des propriétés attendues du programme.

Une logique de Floyd-Hoare ne permet pas de raisonner sur des programmes avec pointeurs car cette logique suppose que les variables d'un programme sont stockées à des emplacements distincts de la mémoire. La logique de séparation est une extension récente de la logique de Floyd-Hoare, adaptée aux langages impératifs manipulant des pointeurs [81, 82, 29]. La logique de séparation définit un langage d'assertions permettant d'observer finement les données

dynamiquement allouées en mémoire dans le tas et donc de décrire aisément des propriétés usuellement recherchées sur des pointeurs : par exemple, le non-chevauchement (i.e. la séparation) entre zones de la mémoire, ou encore l'absence de cycle dans une structure de données chaînée.

En logique de séparation, une propriété relative à une instruction ne concerne que l'*empreinte mémoire* de l'instruction, c'est-à-dire les zones de la mémoire utilisées lors de l'exécution de l'instruction. Contrairement à la logique de Hoare, la logique de séparation permet de raisonner localement sur l'empreinte mémoire d'une instruction, et garantit que les autres zones de la mémoire ne sont pas modifiées par l'exécution de cette instruction, ce qui facilite grandement la vérification des programmes.

Ce raisonnement local est rendu possible d'une part grâce à l'emploi de connecteurs particuliers dans les assertions, et d'autre part grâce à une règle d'inférence supplémentaire (par rapport à une sémantique axiomatique). En effet, la logique de séparation étend la logique classique par des connecteurs permettant d'exprimer aisément des propriétés de séparation de zones de la mémoire (et donc des conditions de non aliasing). Par exemple, la conjonction séparante notée $A * B$ exprime que les assertions A et B sont vraies sur deux parties disjointes de la mémoire. La logique de séparation étend également la logique de Floyd-Hoare par la règle d'inférence de la figure 7.1 dite d'encadrement (« frame rule ») permettant de restreindre les assertions à vérifier.

$$\frac{\{P\}\, i\, \{Q\}}{\{A * P\}\, i\, \{A * Q\}}$$

FIG. 7.2 – Règle d'encadrement de la logique de séparation

Actuellement, les outils automatisant le raisonnement en logique de séparation opèrent sur de petits langages impératifs [83, 84]. De plus, la logique de séparation commence à être utilisée afin d'effectuer des analyses statiques sophistiquées de pointeurs, en particulier des analyses de forme [85, 86, 87]. Enfin, la logique de séparation commence également à être employée pour prouver des propriétés relatives à des programmes concurrents [88, 89].

7.2 Une sémantique à petits pas pour Cminor

Cette section présente brièvement le projet Concurrent Cminor pour lequel j'ai défini une sémantique à petits pas de Cminor. Elle explique ensuite les

liens et les différences entre cette sémantique et celle à grands pas du compilateur CompCert. La sémantique à petits pas de Cminor est une sémantique à continuations. Ses continuations sont détaillées dans une troisième partie. Enfin, cette section détaille la sémantique à petits pas de Cminor.

7.2.1 Concurrent Cminor

L'objectif à long terme du projet Concurrent Cminor [90] coordonné par Andrew W. Appel est de fournir un environnement dédié à la vérification formelle de programmes écrits dans différents langages de programmation. En particulier, il s'agit de vérifier formellement des applications constituées de programmes séquentiels et de programmes concurrents, en utilisant la logique de séparation. Afin de gagner davantage de confiance dans ces programmes, il s'agit également de vérifier formellement leur compilation.

Cminor est le principal langage intermédiaire du compilateur CompCert. Il a été choisi comme langage pivot du projet Concurrent Cminor. C'est vers ce langage que seront compilés les différents langages source considérés (comme le montre la figure 4.3 de la page 40). L'idée de ce projet est donc d'une part de bénéficier du compilateur CompCert, et d'autre part d'utiliser Cminor comme langage sur lequel prouver des programmes ayant été écrits dans différents langages de programmation puis compilés en langage Cminor.

Du point de vue du compilateur CompCert, l'intérêt de ce travail est d'utiliser Cminor dans un contexte autre que celui de la compilation, et donc de valider davantage sa sémantique formelle. En effet, afin de doter le langage Cminor de traits concurrents, j'ai d'abord défini une sémantique à petits pas pour Cminor, dont j'ai vérifié formellement l'équivalence avec celle à grands pas du compilateur CompCert (c.f. section 5.3). Ceci a donc été l'occasion de valider encore un peu plus la sémantique formelle du langage compilé par CompCert.

Avec Andrew W. Appel, nous avons défini une logique de séparation pour Cminor et nous avons prouvé la validité de cette logique par rapport à une sémantique à petits pas de Cminor. Ce travail a été étendu par Andrew W. Appel, Aquinas Hobor et Francesco Zappa Nardelli afin de définir une logique de séparation pour une extension concurrente de Cminor.

Le langage Concurrent Cminor étend Cminor par l'ajout des instructions de prise de verrou `lock` et de libération de verrou `unlock` permettant la modélisation de processus légers (« threads ») et de sections critiques. C'est pourquoi la sémantique des instructions de Cminor est dans un style à petits pas. Par contre, la sémantique des expressions est celle à grands pas du compilateur

CompCert. En effet, décrire l'évaluation des expressions dans un style à petits pas n'est pas nécessaire pour définir la sémantique de Concurrent Cminor.

7.2.2 De CompCert à Concurrent Cminor

Indépendamment du style à petits pas choisi pour Cminor, le langage Cminor a dû être modifié afin d'une part de faciliter le raisonnement en logique de séparation et d'autre part de servir de langage pivot pour un compilateur qui à l'avenir sera utilisable pour différentes architectures cible. Ces adaptations concernent la syntaxe (suppression des effets de bord dans l'évaluation des expressions, suppression de deux catégories syntaxiques dans l'évaluation des expressions, suppression des dépendances vis-à-vis du processeur Power PC) et la sémantique (ajout de contraintes imposant d'une part que toute entrée dans un bloc se termine par une instruction `exit` de sortie de ce bloc, et d'autre part que toute fonction appelée calcule une valeur de retour) de Cminor. Elles ont été effectuées par Xavier Leroy dans le compilateur CompCert. La sémantique de Cminor que j'ai présentée dans la chapitre précédent est celle qui résulte de ces modifications.

Par ailleurs, il existe deux différences principales entre la sémantique de Cminor de CompCert et celle de Concurrent Cminor. La première réside dans la modélisation des environnements d'évaluation et permet de raisonner en logique de séparation. La seconde réside dans la formalisation en Coq. Alors que la sémantique de Cminor de CompCert est définie par des relations inductives en Coq, la sémantique de Concurrent Cminor est écrite dans un style fonctionnel, dans le but de faciliter la conception du langage Concurrent Cminor. Cette modélisation fonctionnelle est radicalement différente de celle présentée précédemment (c.f. section 5.5.1). Il s'agit d'une écriture fonctionnelle de la sémantique inductive, selon un style monadique. Lors du passage à Concurrent Cminor, cette écriture fonctionnelle a finalement été abandonnée au profit d'une écriture inductive, afin de faciliter le raisonnement sur la sémantique de Concurrent Cminor.

Environnement d'évaluation des expressions

Pour pouvoir raisonner en logique de séparation, il est nécessaire de modéliser dans les environnements d'évaluation la notion d'empreinte mémoire (c.f. page 75). Nous avons défini une empreinte mémoire φ par une table associant des permissions (de lecture ainsi que d'écriture) à des adresses en mémoire. Les propriétés de ces permissions sont similaires à celles décrites dans [91, 92]. Cette formalisation des permissions a été étendue pour être adaptée au lan-

gage Concurrent Cminor. La somme disjointe de deux empreintes (notée $\oplus$) est également définie, ainsi que certaines de ses propriétés (par exemple associativité, commutativité). Cet opérateur sert à définir l'opérateur de conjonction séparante $*$ de logique de séparation.

Ainsi, de façon similaire à la modélisation de la mémoire présentée pour l'évaluation partielle (i.e. à l'aide d'une table d'associations et d'un domaine), en logique de séparation, le tas est modélisé par un couple constitué d'une mémoire (i.e. celle définie dans le chapitre 6) et d'une empreinte φ (représentant le domaine de la mémoire).

Le jugement d'évaluation des expressions fait référence à φ ainsi qu'aux éléments du jugement de la sémantique à grands pas de Cminor (c.f. figure 5.6 page 56). Il s'écrit $G, (sp; E; \varphi; M) \vdash a \Rightarrow v$. Afin de simplifier l'écriture de la sémantique à petits pas de Cminor, un état σ est défini comme étant un quadruplet $(sp; E; \varphi; M)$.

Écriture fonctionnelle

La sémantique de l'évaluation des expressions de Cminor a été réécrite dans un style fonctionnel, d'une part dans le but de faciliter la conception du langage Concurrent Cminor (en disposant grâce au mécanisme d'extraction automatique de Coq d'un interprète permettant de tester l'exécution de programmes), et d'autre part dans le but de faciliter les preuves en logique de séparation. En effet, alors qu'une définition inductive d'une sémantique (représentée par une fonction f) nécessite de prouver des propriétés de la forme $f\ x\ y \ \Rightarrow\ f\ x\ y \ \Rightarrow\ y = z$, avec une notation fonctionnelle ces propriétés s'écrivent plus simplement, de la forme $f(x) = f(x)$.

Deux arguments supplémentaires ont milité en faveur d'une nouvelle écriture fonctionnelle. D'une part, la mise à disposition de la construction `Function` dans Coq, permettant (sous certaines conditions) de générer un principe d'induction associé à une écriture fonctionnelle. D'autre part, la définition en Coq (par Xavier Leroy) de monades facilitant l'écriture de la sémantique, de façon similaire à ce qui avait déjà été proposé en Isabelle [93]. L'utilisation d'une monade d'erreur et d'une tactique d'inversion monadique prenant en charge la propagation des erreurs a en effet permis de s'affranchir des inconvénients rencontrés lors de la première tentative d'écriture fonctionnelle d'une sémantique (c.f. page 53), et de se rapprocher d'une écriture inductive. Finalement, il n'a pas été nécessaire d'utiliser la construction `Function`, principalement à cause de l'utilisation de continuations. La construction `Function` permettrait probablement de simplifier les preuves ayant déjà été effectuées, mais nous

n'avons pas suffisamment exploré cette piste.

7.2.3 Quelles continuations pour Cminor ?

Cminor dispose de structures de contrôle non locales (i.e. les instructions `return` et `exit`) qui compliquent les preuves de préservation sémantique. En effet, une preuve par induction sur l'exécution des instructions Cminor nécessite également de raisonner par cas afin de distinguer les flots de contrôle non séquentiels (i.e. sortie abrupte suite à l'exécution d'une instruction `return` ou d'une instruction `exit`). Ceci rend difficile la définition des principes d'induction associés.

Ainsi, raisonner à la fois par induction et par cas sur la sémantique de Cminor a nécessité de définir manuellement (i.e. sans l'aide de Coq) des principes d'induction, dont la formalisation en Coq n'est pas intuitive. Par exemple, dans un principe d'induction, choisir entre un quantificateur existentiel et un quantificateur universel n'est pas toujours facile. C'est seulement lors de la preuve de propriétés sémantiques qu'une erreur dans la définition du principe d'induction est mise en évidence. Il est alors nécessaire de modifier le principe d'induction, puis de refaire les preuves déjà effectuées. Plusieurs itérations de ce processus peuvent être nécessaires afin de converger vers la bonne définition.

Une alternative à ce procédé consiste à définir des types inductifs adaptés, de manière à bénéficier des principes d'induction générés automatiquement par Coq, et donc de raisonner de façon plus assistée. Aussi, la difficulté dans la formalisation de la sémantique à petits pas de Cminor a été de trouver ces types inductifs. Je n'ai pas rencontré cette difficulté lors des preuves de propriétés relatives à des sémantiques à grands pas car une sémantique à grands pas décrit moins précisément l'exécution des instructions. Aussi, j'ai formalisé plusieurs sémantiques à petits pas de Cminor, et j'ai finalement choisi une sémantique à continuations, car les continuations permettent de raisonner de façon uniforme par induction sur les différents cas d'exécution des programmes.

Les continuations permettent de rendre explicites les différents aspects d'un flot de contrôle et d'un flot de données (i.e. l'ordre d'évaluation des arguments d'une fonction, des valeurs intermédiaires, les valeurs renvoyées par une fonction). Les continuations sont essentiellement utilisées dans les langages fonctionnels en tant que technique de programmation, en sémantique dénotationnelle (dans le style dit par passage de continuations). Les continuations sont également utilisées dans les représentations intermédiaires de certains compilateurs, afin de faciliter l'écriture de certaines optimisations [94, 95].

Étant donné un programme p et une instruction i de p, la continuation de i représente la suite d'instructions de p qu'il reste à exécuter une fois i exécutée, afin de terminer d'exécuter p. Disposer du reste d'un programme permet en particulier de l'ignorer (par exemple en cas de sortie abrupte d'une boucle) et de le sauvegarder dans un environnement dans le but de le restaurer ultérieurement. Une continuation peut être vue comme une fonction associant au résultat de l'exécution de i le résultat de l'exécution de p.

Dans la sémantique à continuations de Cminor, une continuation est définie comme une paire constituée d'un état σ et d'une pile de contrôle κ qui modélise le flot de contrôle qu'il reste à exécuter. Il y a quatre opérateurs de contrôle : Kstop représentant une exécution sûre (c.f. page 12) d'un programme (et plus généralement la terminaison sûre d'un calcul), un opérateur de séquence (noté $\cdot$) représentant un flot de contrôle local, Kblock représentant un flot de contrôle intraprocédural (i.e. l'entrée ou la sortie dans un bloc Cminor) et Kcall représentant un flot de contrôle interprocédural. L'opérateur Kcall comprend en plus d'une pile de contrôle κ, la variable x dans laquelle est stockée la valeur de retour de la fonction f (cette variable fit partie de l'instruction d'appel à f), la fonction f ainsi que des informations relatives à la fonction appelante (i.e. son pointeur de pile sp et son environnement local E). Ces dernières informations permettent de restaurer l'environnement de la fonction appelante, une fois que la fonction appelée a été exécutée.

$$
\begin{array}{rcl}
\kappa : \mathsf{control} & ::= & \mathsf{Kstop} \;\mid\; s \cdot \kappa \;\mid\; \mathsf{Kblock}\, \kappa \;\mid\; \mathsf{Kcall}\, x\, f\, sp\, E\, \kappa \\
k : \mathsf{continuation} & ::= & (\sigma, \kappa)
\end{array}
$$

Dans la sémantique de Concurrent Cminor, davantage d'opérateurs de contrôle ont été définis afin de prendre en compte le flot de contrôle de programmes concurrents.

7.2.4 Une sémantique à continuations pour Cminor

Le jugement d'exécution des instructions de la sémantique à petits pas de Cminor est $G \vdash k \longmapsto k'$. La figure 7.3 montre un extrait de cette sémantique et exprime comment une continuation k est transformée en une continuation k'. Étant donnée une pile de contrôle κ représentant des instructions à exécuter à l'issue d'une séquence d'instructions $i_1; i_2$, un pas d'exécution dans cette séquence consiste à exécuter i_1 et à insérer i_2 en tête de κ (règle 21). Un pas d'exécution d'une boucle infinie déplie cette boucle (règle 22). Étant donnée une pile de contrôle κ représentant des instructions à exécuter à l'issue d'un

bloc $\mathtt{block}(i)$, un pas d'exécution dans ce bloc marque κ d'un connecteur Kblock et empile i (règle 23).

Séquence d'instructions

$$G \vdash (\sigma,\ (i_1; i_2) \cdot \kappa) \ \longmapsto\ (\sigma, i_1 \cdot i_2 \cdot \kappa) \ \ (21)$$

Boucle infinie

$$G \vdash (\sigma,\ (\mathtt{loop}\, i) \cdot \kappa) \ \longmapsto\ (\sigma, i \cdot \mathtt{loop}\, i \cdot \kappa) \ \ (22)$$

Entrée de bloc

$$G \vdash (\sigma, (\mathtt{block}\, i) \cdot \kappa) \ \longmapsto\ (\sigma, i \cdot \mathsf{Kblock}\, \kappa) \ \ (23)$$

FIG. 7.3 – Extrait de la sémantique à continuations de Cminor

Par définition, une continuation $k = (\sigma, \kappa)$ est bloquante si κ représente un flot de contrôle non vide ($\kappa \neq \mathsf{Kstop}$) et si aucune transition n'est possible à partir de k ($\nexists k'$tel que $G \vdash k \longmapsto k'$). Une continuation k est sûre (notation $G \vdash \mathsf{safe}(k)$) si aucune continuation bloquante n'est atteignable (en utilisant $\mapsto^*$, la fermeture réflexive et transitive de $\mapsto$) à partir de k.

J'ai prouvé en Coq l'équivalence entre cette sémantique à continuations et celle à grands pas définie dans le compilateur CompCert, pour des programmes dont l'exécution termine. Cette preuve nécessite de gérer la correspondance entre instructions et résultats calculés pas les sémantiques. Plus précisément, l'exécution d'une instruction i selon la sémantique à grands pas calcule une sortie d'instruction out qui correspond à une continuation. De plus, étant donnée une pile de contrôle κ, l'exécution de i selon la sémantique à continuations calcule une nouvelle pile de contrôle κ'. Lors du calcul de out, seuls certains effets de $i \cdot \kappa$ sont pris en compte. Les effets restant (notés $i \# \kappa$) doivent également être modélisés dans la preuve d'équivalence.

La preuve d'équivalence sémantique se décompose en deux lemmes réciproques. Classiquement, le plus simple est le lemme établissant que toute évaluation à grands pas est également une évaluation à petits pas. Le lemme réciproque nécessite de définir une sémantique à grands pas avec continuations, dont le jugement est $G \vdash (\sigma, \kappa) \Rightarrow out, \sigma'$. Cette sémantique est définie à l'aide d'une relation exprimant comment une pile de contrôle consomme une sortie d'instruction out. En plus de relier le compilateur CompCert et la preuve de programme en logique de séparation, cette preuve a été utile pour mettre au point la sémantique à petits pas de Cminor.

7.3 Une logique de séparation pour Cminor

Cette section présente un langage d'assertions, ainsi qu'une sémantique axiomatique pour Cminor. Puis, elle explique comment la validité de cette sémantique axiomatique par rapport à la sémantique à continuations de Cminor a été formellement vérifiée. Enfin, elle précise comment améliorer le raisonnement en logique de séparation.

7.3.1 Langage d'assertions

Les connecteurs de logique de séparation comprennent les opérateurs de la logique classique ainsi que des opérateurs spécifiques, parmi lesquels la conjonction séparante (notée $*$) et l'opérateur « pointe sur » (noté $\mapsto$). Ces derniers permettent d'exprimer des propriétés concernant les pointeurs. Par exemple, $x \mapsto 5 * y \mapsto 5$ signifie que les deux variables x et y pointent chacune sur une cellule du tas contenant la valeur 5, et de plus ces cellules sont disjointes.

Une formule P de logique de séparation (de type `formule`) est définie en Coq comme une fonction opérant sur un état σ et un environnement global G (i.e. $P\,G\,\sigma$ est une proposition logique, donc du type `Prop` des propositions logiques de Coq). Par définition, un état σ satisfait une formule $P * Q$ si son empreinte φ_σ (qui est un composant du quadruplet σ) peut être partitionnée en deux empreintes φ_1 et φ_2 (notation $\varphi_\sigma = \varphi_1 \oplus \varphi_2$) telles que les restrictions de σ à φ_1 (notation $\sigma[:= \varphi_1]$) et φ_2 (notation $\sigma[:= \varphi_2]$) satisfont P et Q respectivement.

$$P * Q =_{\text{def}} \ \lambda G\sigma.\ \exists\varphi_1.\exists\varphi_2.\ \ \varphi_\sigma = \varphi_1 \oplus \varphi_2 \ \wedge\ P(\sigma[:= \varphi_1]) \ \wedge\ Q(\sigma[:= \varphi_2])$$

Les opérateurs de la logique classique sont définis en logique de séparation à partir de ceux de Coq (plongement superficiel). Par exemple, l'opérateur $\vee$ est défini ainsi :

$$P \vee Q =_{\text{def}} \ \lambda G\sigma.\ P\sigma \ \vee\ Q\sigma$$

.

Les formules de logique de séparation qu'il est possible d'écrire sont plus générales que celles usuellement écrites en logique de séparation. En effet, nos formules peuvent contenir des jugements d'évaluation d'expressions, ainsi que des propositions logiques quelconques. De plus, nos formules peuvent utiliser des variables définies en dehors des programmes considérés (afin par exemple de relier une pré-condition à une post-condition), ainsi que des expressions pouvant lire des valeurs dans le tas. Dans la sémantique axiomatique, nous

distinguons les expressions pures, n'effectuant pas de lecture dans le tas, des autres.

7.3.2 Sémantique axiomatique

Afin de prendre en compte les structures de contrôle non locales de Cminor (i.e. les instructions `exit` et `return`), nous avons étendu les triplets de Hoare à des sextuplets de la forme $G; R; B \vdash \{P\}s\{Q\}$, où G (de type `formule`) désigne les formules relatives à l'environnement global, et R l'environnement permettant de gérer les appels et retours de fonctions. Plus précisément, R est de type `valeur` $\rightarrow$ `formule` et associe à chaque valeur renvoyée par une instruction `return` rencontrée, la post-condition de la fonction à laquelle cette instruction correspond. Enfin, B (de type `nat` $\rightarrow$ `formule`, avec `nat` désignant le type des entiers de Coq) est l'environnement permettant de gérer les blocs de Cminor ; il représente les conditions de sortie dans chaque bloc en cours d'exécution. B associe à chaque entier représentant le niveau d'imbrication d'un bloc en cours d'exécution la condition de sortie de ce bloc. Ces environnements compliquent également l'écriture de la règle de cadrage (c.f. figure 7.1).

Les deux premières règles de la figure 7.4 sont similaires à celles de la figure 2.3. L'entrée dans un bloc met à jour l'environnement B. Dans la règle (26) d'entrée dans un bloc, $Q \cdot B$ désigne B dans lequel le niveau d'imbrication de chaque bloc en cours d'exécution est augmenté d'un, auquel est ajoutée la formule Q associée au niveau d'imbrication le plus faible (i.e. le niveau 0). De manière similaire à la sortie d'une boucle, le seul moyen de sortir d'un bloc est d'exécuter une instruction `exit`, ce qui explique la post-condition **false** en prémisse de la règle d'entrée dans un bloc.

Les expressions apparaissant dans les règles de sémantique axiomatique sont pures. Des règles supplémentaires ont été écrites afin de traiter les expressions qui ne sont pas pures (et de se ramener à des expressions pures).

7.3.3 De la sémantique à continuations à la sémantique axiomatique

Les règles de sémantique axiomatique sont valides par rapport à la sémantique à continuations. Plutôt que définir une sémantique axiomatique et prouver ensuite la validité de cette sémantique par rapport à la sémantique à continuations, nous avons interprété les assertions (i.e. les sextuplets) à partir de la sémantique à continuations. Ainsi, chaque règle de sémantique axiomatique est un théorème ayant été vérifié formellement en Coq.

Séquence d'instructions

$$\frac{G; R; B \vdash \{P\} i_1 \{P'\} \qquad G; R; B \vdash \{P'\} i_2 \{Q\}}{G; R; B \vdash \{P\} i_1 ; i_2 \{Q\}} \ (24)$$

Boucle infinie

$$\frac{G; R; B \vdash \{\mathit{Inv}\} i \{\mathit{Inv}\}}{G; R; B \vdash \{\mathit{Inv}\} \mathtt{loop}\, i \{\mathbf{false}\}} \ (25)$$

Entrée de bloc

$$\frac{\Gamma; R; Q \cdot B \vdash \{P\} i \{\mathbf{false}\}}{\Gamma; R; B \vdash \{P\} \mathtt{block}\, i \{Q\}} \ (26)$$

FIG. 7.4 – Extrait de la sémantique axiomatique de Cminor

Interprétation des assertions

L'interprétation des assertions utilise une relation d'équivalence entre états (notée $\cong$). Étant donné un état σ, l'exécution d'une pile de contrôle κ est sûre (notation $\mathbf{safe}\ \kappa$) lorsque tout état σ' équivalent à σ est tel que la continuation (σ', κ) est sûre. L'interprétation des assertions utilise également un opérateur de garde d'une pile de contrôle κ par une formule de logique de séparation P, étant donné un *cadre* (i.e. une formule close de logique de séparation, c'est-à-dire ne contenant pas de variable Cminor) A. Par définition, P garde κ étant donné A (notation $P \,\square_A\, \kappa$) lorsque si $A * P$ est vrai, alors l'exécution de κ est sûre.

$$P \,\square_A\, \kappa \ =_{\text{def}} \ A * P \ \Rightarrow \ \mathbf{safe}\ \kappa.$$

Deux extensions de cet opérateur de garde ont également été définies pour traiter les formules des environnements de retour de fonction (notation $R \boxed{\scriptscriptstyle R}_A\, \kappa$) et de sortie de bloc (notation $B \boxed{\scriptscriptstyle B}_A\, \kappa$).

Un sextuplet $G; R; B \vdash \{P\}\ i\ \{Q\}$ est défini ainsi :

$$\forall A, \kappa.\ R \boxed{\scriptscriptstyle R}_{\mathsf{cadre}(G,A,i)}\, \kappa \wedge B \boxed{\scriptscriptstyle B}_{\mathsf{cadre}(G,A,i)}\, \kappa \wedge Q \,\square_{\mathsf{cadre}(G,A,i)}\, \kappa \ \Rightarrow \ P \,\square_{\mathsf{cadre}(G,A,i)}\, i \cdot \kappa$$

Dans la définition, le cadre $\mathsf{cadre}(G, A, i)$ est commun aux différentes expressions gardées. Il restreint la formule A aux formules séparées de l'environnement global G et closes par rapport aux variables modifiées par l'instruction

i. En omettant de considérer les blocs et les fonctions, la définition d'un sextuplet exprime que si une post-condition garde une pile de contrôle κ (autrement dit si on peut exécuter κ), alors la pré-condition correspondante garde $i \cdot \kappa$ (autrement dit on peut exécuter i puis κ). La formulation « à l'envers » de cette définition (par rapport à la définition de la validité donnée page 15) est liée à l'utilisation de continuations.

Validité de la sémantique axiomatique

La preuve de la validité de la sémantique axiomatique par rapport à la sémantique à continuations est la preuve que chaque règle de sémantique axiomatique (en particulier celles de la figure 7.4) se déduit de la définition des sextuplets.

La règle la plus difficile à vérifier formellement est la règle de la boucle infinie. Une boucle infinie s'exécute jusqu'à ce qu'une instruction `exit` ou `return` du corps de la boucle soit exécutée. Raisonner sur le nombre de pas d'exécution dans la sémantique à continuations n'est pas suffisant. Après n pas d'exécution (représentés par $\longmapsto^n$) dans une boucle, il est également nécessaire de connaître combien d'itérations de la boucle ont été effectuées. De plus, dans une boucle l'exécution d'une instruction `exit` ne provoque pas nécessairement la sortie de la boucle (par exemple, elle peut seulement provoquer la sortie d'un bloc imbriqué dans le corps de la boucle). Aussi, il a été également nécessaire de modéliser la notion de sortie de boucle.

Cette modélisation repose sur la notion d'*absorption* de pas d'exécution par une instruction. Étant donné un état σ, une instruction i absorbe n pas d'exécution si l'exécution de $j \leq n$ pas d'exécution partant de la pile de contrôle $i \cdot \kappa$ ne consomme pas κ :

$$\forall j \leq n.\, \exists \kappa_{\text{prefix}}.\exists \sigma'.\ \forall \kappa.\ G \vdash\ (\sigma, i \cdot \kappa) \longmapsto^j (\sigma', \kappa_{\text{prefix}} \circ \kappa),$$

où $\circ$ modélise la concaténation de piles de contrôle.

7.3.4 Des tactiques pour la logique de séparation

Avant de définir une logique de séparation pour Cminor, Andrew W. Appel avait proposé de réutiliser un développement en Coq effectué dans le cadre de la thèse de Nicolas Marti et portant sur un langage jouet [96], dans le but d'étudier l'opportunité de raisonner en logique de séparation en Coq sur de petits programmes. Raisonner en logique de séparation nécessite de raisonner sur les formules écrites dans les assertions, ainsi que sur la sémantique du langage

considéré. Les premières preuves en logique de séparation dans cet environnement ont mis en évidence des propriétés récurrentes, dont Andrew W. Appel a automatisé la preuve à l'aide de tactiques Coq [97]. L'exécution d'une instruction élémentaire, ou encore la substitution d'une variable par une valeur dans une assertion sont deux exemples de ces tactiques.

L'objectif des auteurs du développement formel que nous avons réutilisé était de proposer un environnement dédié à un mini-langage de programmation, dans le but de vérifier en logique de séparation des programmes « système » [98]. Après discussion avec ces auteurs, nous avons décidé de ne plus utiliser leur développement en Coq, et de choisir Cminor comme langage sur lequel définir une logique de séparation. Les tactiques précédemment développées ont été réutilisées et ainsi certaines preuves déjà réalisées ont été refaites à peu de frais. Ces premières preuves ont donné davantage de confiance dans la sémantique de Cminor.

L'automatisation du raisonnement en logique de séparation fait l'objet du post-doctorat de Keiko Nakata. Des expériences de preuves de programmes plus conséquents montrent que certaines automatisations ne peuvent pas être écrites simplement à l'aide du langage de tactiques de Coq. Aussi, une seconde solution actuellement étudiée est l'utilisation du prouveur automatique Alt-Ergo [99]. Ce prouveur est dédié à la vérification de programmes et il est utilisable depuis Coq (mais pas pour traduire notre logique de séparation, pour l'instant). Il permet de prouver sans assistance humaine certaines propriétés utiles en logique de séparation. Les premiers résultats montrent que ces deux approches sont complémentaires. Davantage de travail est nécessaire pour valider ces deux approches.

7.4 Une autre expérience en preuve de programmes

L'environnement que nous avons développé en Coq pour prouver des programmes en logique de séparation a été conçu pour être étendu à la preuve de programmes concurrents. Prouver un programme concurrent est difficile car il est nécessaire de raisonner sur l'ensemble des interférences (i.e. le partage de variables) entre tous les processus légers s'exécutant simultanément. Plusieurs solutions à ce problème existent, et elles ont rarement été vérifiées formellement. Andrew W. Appel et d'autres ont étudié comment étendre le langage Cminor et la logique de séparation à un contexte concurrent, en s'inspirant des premiers travaux récents sur la logique de séparation concurrente [89]. De mon côté, avec Marc Shapiro, nous avons étudié une approche complémentaire, la

combinaison de la logique de séparation avec la logique du « rely guarantee », proposée également récemment [100].

La logique du « rely guarantee » a été proposée en 1981 pour prouver des programmes concurrents [101, 102]. Il s'agit d'une extension de la logique de Floyd-Hoare qui permet de décrire les interférences entre processus légers. Dans cette logique, les assertions sont de la forme $\{P, R\}i\{G, Q\}$. Pour chaque processus léger *proc*, le prédicat R (appelé condition de « rely ») modélise les interférences que subit *proc* par les autres processus légers ; le prédicat G (appelé condition de « guarantee ») modélise les interférences que fait subir *proc* aux autres processus légers. Par rapport à une logique de Floyd-Hoare, prouver un programme nécessite de plus de prouver que d'une part si chaque condition R de chaque processus léger est satisfaite, alors le processus léger satisfait sa condition G, et d'autre part, chaque condition G de chaque processus léger implique les conditions G des autres processus légers.

La logique du « rely guarantee » a été formalisée en Isabelle/HOL pour un mini-langage impératif, et la validité de cette logique par rapport à une sémantique opérationnelle a été vérifiée formellement [103, 104]. La validité de la logique du « rely guarantee » a également été récemment prouvé sur papier [105].

La logique du « rely guarantee » est adaptée à la description des interférences entre processus. Par contre, cette logique n'est pas compositionnelle : la spécification des interférences est globale, et doit être vérifiée à chaque modification de l'état. La logique de séparation permet au contraire un raisonnement modulaire, grâce à l'existence d'une règle de cadrage et d'un opérateur de séparation. Récemment, [100] a proposé une combinaison de ces deux logiques, que nous avons formalisée en Coq.

Par souci de simplification, le langage de programmation considéré a été le mini-langage impératif de [100] usuellement étudié en logique de séparation. Les modifications apportées dans notre environnement de logique de séparation ont été les suivantes. Dans les environnements d'évaluation, l'état comprend deux composantes disjointes : un état partagé par tous les processus et un état local accessible à un seul processus seulement. Dans le langage d'assertions, les formules de logique de séparation sont également soit partagées, soit locales. De plus, deux instructions concurrentes ont été ajoutées (l'exécution parallèle de deux instructions, et une instruction de synchronisation).

Enfin, les interférences entre processus sont représentées par des actions décrivant les modifications de l'état partagé. La sémantique à petits pas que nous avons définie comprend donc deux types de transitions : celles représentant l'exécution d'une instruction (qui étaient les seules transitions possibles

dans la sémantique de Cminor), et celles représentant une action. Enfin, afin de modéliser les contraintes de la logique du « rely guarantee », il a également été nécessaire de définir une relation de stabilité d'une assertion par rapport à une action.

Au final, le raisonnement sur cette sémantique s'est avéré difficile et a nécessité de définir à la main des principes d'induction. Une difficulté supplémentaire provient du modèle de cohérence des données considéré. En effet, conformément à l'approche décrite dans [106], nous avons considéré un modèle à cohérence atomique (i.e. à cohérence forte) [107]. Ce modèle est très contraint et a nécessité de découvrir et modéliser des propriétés supplémentaires qui sont implicites dans les preuves sur papier de programmes concurrents. La définition d'une logique de séparation concurrente pour une extension concurrente de Cminor a finalement été préférée [108]. Cette solution repose sur un modèle à cohérence faible [109], plus simple et davantage adapté à la preuve de programmes concurrents en Coq.

7.5 Bilan

Ce chapitre a présenté une sémantique à continuations pour le langage Cminor, ainsi qu'une logique de séparation pour Cminor. Cette logique de séparation opère sur un langage de programmation plus vaste que les langages de programmation usuellement considérés en logique de séparation. De plus, elle permet d'écrire des contraintes plus générales dans les assertions (utilisant des variables définies en dehors des programmes étudiés, et des expressions pouvant lire des valeurs en mémoire).

La figure 7.5 présente dans l'ordre chronologique et sur un schéma les différentes sémantiques de Cminor présentées dans ce mémoire.

La sémantique à continuations et la logique de séparation pour Cminor ont été étendues par Andrew W. Appel & al. à un langage concurrent appelé Concurrent Cminor [108]. Cette extension conservative de Cminor réutilise non seulement la syntaxe et la sémantique de Cminor, mais aussi les propriétés de cette sémantique ; elle a nécessité de séparer dans des modules Coq la partie séquentielle de la partie concurrente du développement. La figure 7.6 schématise cette extension et la démarche décrite dans ce chapitre.

Actuellement, chacun des deux projets CompCert et Concurrent Cminor évolue séparément, et vise des objectifs différents. Aussi, chaque projet dispose de sa propre version de Cminor. Ceci permet à chaque projet de faire évoluer son langage à sa guise. En particulier, Concurrent Cminor propose une vision très modulaire de la sémantique de Cminor (qui définit donc de nombreux

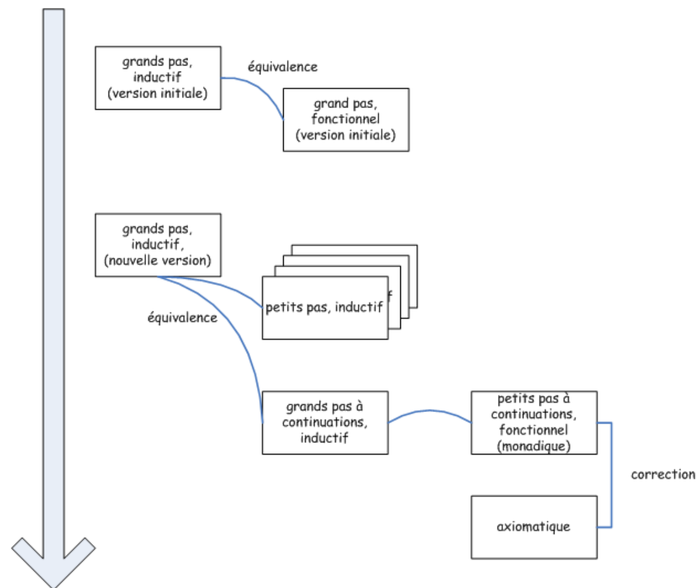


FIG. 7.5 – Chronologie des différentes sémantiques de Cminor

modules en Coq) qui n'est pour l'instant pas souhaitable dans CompCert (car ceci nécessiterait une refonte complète des preuves de CompCert, dans le but de généraliser le langage Cminor, ce qui *a priori* n'est pas un objectif du projet CompCert). Quelquefois cependant, des évolutions de la partie séquentielle du langage Concurrent Cminor deviennent également des évolutions du langage Cminor du compilateur CompCert.

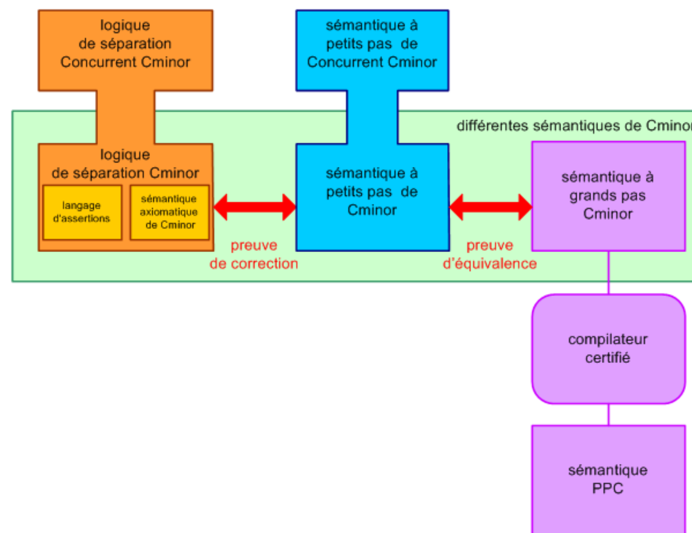


Fig. 7.6 – De CompCert à Concurrent Cminor

8 Autres expériences

Les chapitres précédents présentent des spécifications de sémantiques formelles et de transformations de programmes opérant sur ces sémantiques, ainsi que la vérification formelle de la préservation sémantique de ces transformations. Ce chapitre décrit trois expériences n'entrant pas dans ce cadre. La première expérience concerne la vérification formelle en Coq d'algorithmes d'allocation de registres. Ces algorithmes opèrent sur le langage intermédiaire RTL du compilateur CompCert. La deuxième expérience porte sur la spécification formelle en langage B d'un langage de réutilisation de composants de spécifications formelles. Enfin, la troisième expérience est une formalisation en Coq d'un langage déclaratif de description de glossaires.

8.1 Allocation de registres

Ce chapitre commente une partie du matériau publié dans [110, 111, 112]. Les travaux présentés dans cette section ont été effectués en collaboration avec Éric Soutif. J'ai encadré le stage de M2 de Benoît Robillard [stage 1], qui s'est poursuivi par une thèse que je co-encadre actuellement.

Le but de l'allocation de registres est de déterminer où sont stockées les variables d'un programme à tout moment de son exécution : soit en registres si ces derniers sont disponibles, soit en mémoire le cas échéant. La difficulté est de proposer une affectation optimale des registres. Il est par exemple souvent nécessaire de choisir entre conserver une variable v dans un même registre R pendant toute l'exécution d'un programme (ce qui rend R inutilisable pour stocker d'autres variables), et réutiliser R lorsque la valeur de v n'a plus besoin d'être conservée en vue d'utilisations futures (ce qui nécessite de transférer en mémoire la valeur de v).

L'allocation de registres est la passe de compilation la plus étudiée et la plus difficile à mettre en œuvre dans un compilateur. La qualité du code compilé dépend en effet de la qualité de l'allocation de registres. Les deux tâches

principales de l'allocation de registres sont le *vidage* de registres en mémoire, et la *fusion* de registres. Le vidage décide quelles variables seront stockées soit en registres, soit en mémoire, et minimise le coût des accès aux variables vidées en mémoire. La fusion tient compte le plus possible des préférences entre variables, afin de minimiser les transferts entre registres. Les préférences proviennent d'affectations entre variables. Par exemple, une affectation [1] entre variables `mv` $x\,y$ entraîne une préférence entre les variables x et y correspondant à la contrainte qu'au vu de cette instruction, il serait préférable de stocker x et y dans le même registre.

L'allocation de registres est également une transformation de programmes qui :

- insère des instructions de lecture et écriture en mémoire, pour chaque variable à vider en mémoire,
- supprime des instructions de transferts entre registres lorsqu'elles concernent des variables stockées dans des registres ayant été fusionnés,
- renomme des variables du programme.

L'allocation de registres consiste à minimiser le nombre d'accès à la mémoire, étant donné un nombre fixe de registres (qui dépend du processeur considéré). Classiquement, l'allocation des registres d'un programme se ramène à un problème de coloration du graphe d'interférences du programme, construit à l'issue d'une analyse de vivacité du programme exprimé en langage RTL. Ce problème étant NP-difficile, les compilateurs le résolvent à l'aide d'heuristiques [113, 114, 115, 116, 117] proposant différentes combinaisons des deux phases de vidage et de fusion. Les heuristiques les plus simples effectuent une phase de vidage suivie d'une phase de fusion. D'autres heuristiques plus sophistiquées effectuent simultanément vidage et fusion. Les heuristiques d'allocation de registres évoluent aujourd'hui encore, par exemple afin de tenir compte de la rapidité croissante des processeurs ainsi que du coût croissant des accès à la mémoire.

Pour un compilateur d'un langage tel que C, l'heuristique de coloriage de graphe la plus efficace à l'heure actuelle est celle d'Appel et George [116]. C'est celle qui a été choisie dans le compilateur CompCert. Dans CompCert, cette heuristique n'est pas écrite en Coq, principalement car elle est peu adaptée à une écriture fonctionnelle, et donc l'effort nécessaire pour spécifier en Coq et prouver ensuite la préservation sémantique de l'allocation de registres a été jugé trop important. L'allocation de registres de CompCert est écrite en Caml puis validée *a posteriori* en Coq : pour chaque programme à compiler, il

[1] `mv` désigne l'instruction `move` de transfert entre registres du langage RTL.

est vérifié formellement que la solution calculée par l'allocation de registres est correcte. L'intérêt de cette approche est que la preuve à effectuer est beaucoup plus facile, puisqu'il s'agit de vérifier que l'allocation de registres a bien renvoyé une coloration correcte du graphe d'interférences (i.e. les couleurs de tout couple de sommets du graphe reliés par un arc sont distinctes). En outre, cette technique permet de valider une transformation de programme qui n'a pas été écrite en Coq.

Le but de mes travaux (et de la thèse de Benoît Robillard, commencée en septembre 2007) est de proposer une allocation de registres optimale. Il s'agit d'une part de proposer des algorithmes exacts opérant sur des familles de graphes à définir et utilisables par le compilateur CompCert, et d'autre part de modéliser l'allocation de registres par un (ou plusieurs) programme linéaire en nombres entiers, et dont la résolution par un solveur externe fournit une solution optimale, lorsqu'il en existe une.

La modélisation de l'allocation de registres par des programmes linéaires en nombres entiers (i.e. dont les variables sont à valeurs dans $\{0; 1\}$ et dont les contraintes portant sur les variables sont linéaires) a déjà été étudiée [118, 119, 120], et connaît un regain d'intérêt récent. Les premiers travaux ont concerné des architectures CISC (i.e. avec de nombreuses instructions et peu de registres), différentes de l'architecture RISC (i.e. avec peu d'instructions et de nombreux registres) de CompCert. En particulier, les contraintes sur les variables diffèrent d'une architecture à l'autre.

La phase de vidage est étudiée dans [119]. La phase de fusion est étudiée dans [120]. Des résultats récents ont montré que dans le cas général le vidage et la fusion sont des problèmes NP-difficiles [121, 122]. Seul [118] a étudié la résolution conjointe (i.e. par un unique programme linéaire en nombres entiers) des deux phases de vidage et de fusion. À l'époque, en 1996, les temps de résolution des solveurs étaient trop élevés pour que cette approche passe à l'échelle. Depuis, les performances des machines se sont améliorées et de plus, des techniques sophistiquées de résolution ont été intégrées aux solveurs. Aussi, il nous a semblé opportun de reconsidérer cette résolution, en l'étudiant dans un contexte plus simple (i.e. l'architecture RISC), mais en définissant des algorithmes simplifiant les graphes d'interférences, dans le but d'améliorer ensuite la résolution du programme linéaire.

Cette année, nous avons étudié l'influence d'une optimisation nécessaire à la résolution optimale de la phase de vidage (i.e. le « live-range splitting ») sur la phase de fusion. Cette optimisation est considérée comme néfaste pour la phase de fusion car elle transforme les graphes d'interférences en graphes beaucoup plus gros, dans lesquels sont ajoutés de nombreux arcs de préférence

qui rendent la fusion beaucoup plus difficile. Aussi, Benoît Robillard a proposé deux algorithmes simplifiant un graphe d'interférences, et permettant ainsi d'améliorer sensiblement la résolution optimale de la phase de fusion proposée dans [120]. Ces résultats ont été testés sur un banc de tests de 474 graphes de référence [123] et sont meilleurs que les résultats connus sur ces graphes. En particulier, pour la première fois, toutes les solutions optimales ont été obtenues sur ces graphes.

De plus, nous avons considéré les graphes d'interférences ayant la propriété d'être *triangulés*, c'est-à-dire ne possédant aucun cycle induit sans corde de longueur supérieure ou égale à quatre. Des résultats récents ont montré que la très grande majorité des graphes d'interférences de programmes impératifs ont cette propriété [124, 117]. Nous avons spécifié en Coq un algorithme de coloration gourmande. Cet algorithme calcule une coloration optimale (i.e. un nombre de couleurs minimal) lorsque le graphe d'interférences est triangulé [125]. Nous avons vérifié formellement ce résultat. Ceci a nécessité de modéliser plusieurs structures de données en Coq : des graphes utilisant la bibliothèque `FMaps` de Coq et sur lesquels il a été nécessaire de définir à la main des principes d'induction, des ordres lexicographiques, ainsi que des notions de théorie des graphes (en particulier les notions de clique et d'ordre d'élimination simplicial).

8.2 Un langage pour la réutilisation de composants de spécifications

Ce chapitre commente une partie du matériau publié dans [126]. Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Régine Laleau. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Frédéric Gervais, Nathalie Ly et Mourad Naouari [stage 8, stage 10, stage 11].

Avant de participer au projet CompCert, je me suis intéressée à la formalisation de processus de réutilisation dans les phases amont du cycle de développement de logiciels. Les travaux de Régine Laleau portent sur la conception formelle de systèmes d'information, à l'aide de la méthode B, dans le but de vérifier formellement des propriétés des systèmes d'information, propriétés qui ne sont pas assurées par les modélisations en UML par exemple. Il s'agit de traduire des schémas de conception exprimés en UML en machines B, afin de raisonner ensuite sur ces machines B.

Mon travail a porté sur la réutilisation de patrons de conception [127, 128]. Ces patrons de conception sont exprimés en UML et sont fréquemment employés pour réutiliser des schémas de conception UML. Cette réutilisation

est informelle et il existe de nombreuses manières d'appliquer ces patrons de conception. Nous avons adapté la réutilisation de patrons de conception au cadre formel proposé par Régine Laleau. Aussi, nous avons formalisé en langage B la notion de patron de conception ainsi que différents mécanismes de réutilisation de patrons de conception que nous avons définis : instanciation, extension et trois niveaux de composition dépendant des liens existant entre les patrons composés (que nous avons qualifiés de juxtaposition, composition et unification).

Nos patrons de conception opèrent sur des spécifications B. Ces spécifications sont obtenues à partir de diagrammes UML, et sont qualifiées de composants de spécification (par analogie avec les composants sur lesquels opèrent les patrons de conception). Certains de nos mécanismes de réutilisation procèdent par raffinement de spécifications. Nous avons défini un prototype en Caml automatisant la construction complète des composants réutilisés. En particulier, ce prototype génère les obligations de preuve liées à la composition (i.e. les propriétés que la méthode B impose de vérifier formellement) des composants générés. Ainsi, les obligations de preuve sont des propriétés du patron de conception ; elles ne sont vérifiées formellement qu'une seule fois et réutilisées pour chaque composant construit par réutilisation.

Enfin, afin d'appliquer ces mécanismes de réutilisation à un domaine particulier, nous nous sommes également intéressées à des transformations de programmes SQL effectuant des mises à jour de bases de données. Ces transformations ont pour but de faire de la rétro-ingéniérie de programmes, afin de produire des spécifications en B.

8.3 Un langage pour la description de glossaires

Les travaux présentés dans ce chapitre ont été effectués en collaboration avec Philippe Michelin de la société Bfd et Marc Frappier. Les étudiants de niveau M2 ou équivalent que j'ai encadrés sur les thèmes présentés dans ce chapitre sont Agnès Gonnet et Maxime Bargiel. Ces travaux n'ont pas été publiés car ils ont fait l'objet d'une clause de confidentialité.

Définir une sémantique formelle d'un langage permet de mieux comprendre ce langage. Cela s'avère utile pour un langage de programmation tel que C, mais aussi pour concevoir de nouveaux langages. J'ai ainsi formalisé en Coq la sémantique d'un langage de description de glossaires utilisés dans le domaine bancaire, et utilisé le mécanisme d'extraction pour générer automatiquement un interprète de ce langage, évaluant des questions simples posées par un

utilisateur. Un glossaire définit une liste de termes dont la définition varie d'une banque à l'autre. C'est un document contractuel servant à élaborer un cahier des charges à destination des clients de la banque. Ce langage de description de glossaires est utilisé par les consultants de la société Bfd dirigée par Philippe Michelin.

Afin de clarifier les définitions contenues dans les glossaires, Philippe Michelin et Marc Frappier ont défini de façon informelle un langage de description de glossaires. Il s'agit d'un langage logique (à la Prolog), inspiré de la logique de Spencer-Brown, une logique utilisée en sciences cognitives [129]. Cette logique est très générale (la logique classique en est une interprétation possible) et définie de façon informelle. C'est une logique trivaluée, dont les deux principaux opérateurs sont l'opérateur « est un » et la distinction binaire entre deux termes. Des logiques similaires, relevant de la logique de description sont actuellement très étudiées pour représenter des connaissances, et en particulier pour décrire des ontologies [130].

Disposer d'une sémantique formelle a permis de mieux comprendre le langage étudié. En particulier, une partie de la syntaxe de ce langage est constituée d'opérateurs ayant été rajoutés pour pouvoir décrire des glossaires (conformément à la démarche préconisée par la logique de Spencer-Brown). La sémantique formelle a mis en évidence des imprécisions dans l'emploi de ces opérateurs.

De plus, le langage de description de glossaires considéré est déclaratif et il autorise des définitions multiples d'un même concept. J'ai donc défini une notion de validité de glossaire et intégré cette notion de validité à la sémantique formelle, afin d'assurer que tout glossaire évaluable soit valide.

Une première condition pour qu'un glossaire soit valide est qu'il ne contienne aucune définition multiple. Cette vérification n'est pas facile à effectuer à la main car la description d'un glossaire comprend deux niveaux, conformément à la logique de Spencer-Brown. Le premier niveau est un niveau « méta » dans lequel de nouveaux opérateurs peuvent être définis (dans le but de ne pas contraindre l'utilisateur par l'emploi d'une syntaxe rigide ou de trop bas niveau), ainsi que des faits généraux (i.e. des propriétés quantifiées universellement) des glossaires. Le second niveau regroupe des définitions et propriétés opérant sur des instances des termes définis au niveau précédent.

La deuxième condition de validité d'un glossaire concerne la cohérence de ses faits. Lorsqu'un fait F décrit au niveau « méta » est réflexif ou transitif, alors le glossaire est cohérent si aucune des instances de F définies au second niveau ne contredit le caractère réflexif ou transitif de F. Seules la réflexivité et la transitivité ont été considérées car ce sont les propriétés des principaux

opérateurs de la logique de Spencer-Brown. J'ai également défini une propriété de subsomption (i.e. permettant de décider si une définition est plus générale qu'une autre), mais sa vérification formelle n'a pas été effectuée car elle repose sur des calculs de chemins dans des graphes de dépendances entre les définitions d'un glossaire (et ces graphes n'ont pas été définis en Coq).

Cette expérience a permis de fournir à la société Bfd un évaluateur de glossaires écrit en Caml (et automatiquement extrait des spécifications Coq), et plus complet (car effectuant quelques vérifications de validité) qu'un évaluateur précédemment écrit en SML par un étudiant de Marc Frappier. Certaines personnes de la société Bfd ont été convaincues de l'intérêt des méthodes formelles pour définir un nouveau langage. Malheureusement, au bout de deux ans, pour des raisons politiques, la société Bfd a choisi de n'utiliser que le langage C# pour développer ses outils, ce qui a mis fin à cette expérience.

9 Conclusion

Les méthodes formelles, et en particulier les assistants à la preuve sont suffisamment mûrs pour aider à concevoir de véritables langages de programmation. Les travaux présentés dans ce mémoire s'inscrivent dans la thématique plus générale de la vérification formelle des outils de production et de validation de code.

Les principaux styles de sémantique présentés dans ce mémoire sont les styles opérationnel (à grands pas et à petits pas) et axiomatique. Ces styles sont complémentaires. Choisir un style n'est pas aisé. Ainsi, le langage Cminor est actuellement défini selon une sémantique coinductive à grands pas dans le projet CompCert, alors qu'il est défini selon une sémantique à petits pas dans le projet Concurrent Cminor. L'ajout d'une instruction de saut de type `goto` au langage Cminor est actuellement étudié. Deux solutions sont envisagées : une sémantique à petits pas, et une sémantique axiomatique dont la représentation des sauts est inspirée de travaux effectués sur un mini-langage [131].

Des transformations de programmes ont été étudiées dans ce mémoire. La vérification formelle de la préservation sémantique de ces transformations a montré que les styles opérationnels (à petits pas et à grands pas) sont adaptés à cette tâche. Le choix d'un style dépend également de la transformation de programme considérée. De plus, vérifier formellement la correction d'une transformation de programmes est aussi l'occasion de valider la sémantique des langages sur lesquels cette transformation opère.

Le projet CompCert arrive à sa fin. Il montre qu'il est désormais possible de vérifier formellement un compilateur réaliste du langage C. Ce projet a permis de mettre au point une ingéniérie de preuve adaptée à la compilation certifiée. Davantage de travail est nécessaire pour améliorer le compilateur CompCert.

Une première perspective de recherche concerne l'amélioration de la sémantique de C et du modèle mémoire, dans le but de disposer de différents niveaux d'abstraction pour les décrire. Je souhaiterais définir des relations de raffinement (à l'aide des modules de Coq) entre ces niveaux, et vérifier formellement la correction de ces raffinements. En particulier, la sémantique de

Clight actuelle est déterministe, alors qu'en C l'ordre d'évaluation des expressions est quelconque. Même si tous les compilateurs actuels de C considèrent un seul ordre d'évaluation des expressions, il serait intéressant de définir une sémantique formelle plus abstraite que celle existant actuellement, ainsi qu'un raffinement entre ces deux sémantiques. Nous disposerions ainsi d'une « véritable » sémantique de C, plus fidèle au standard C que l'est celle de Clight, en plus d'une sémantique de plus bas niveau utile au compilateur CompCert.

Le modèle mémoire actuel est défini à deux niveaux d'abstraction et adapté au compilateur CompCert. Il est également suffisamment générique pour être réutilisé indépendamment du compilateur. En particulier, il serait intéressant d'étudier le raffinement d'un modèle plus abstrait à la Burstall-Bornat, utilisé dans les outils automatiques de preuve de programmes, vers notre modèle. En effet, le modèle mémoire pris isolément permet de prouver de petits programmes (dont les instructions sont construites à partir des fonctions d'accès à la mémoire), indépendamment de l'environnement de preuve de programmes en logique de séparation introduit dans ce mémoire.

De plus, un modèle mémoire de plus bas niveau que le modèle actuel fournirait une vision davantage matérielle (i.e. « hardware ») de la mémoire, et modéliserait plus finement dans la sémantique de C des erreurs de bas niveau, ce qui permettrait de compiler davantage de programmes largement répandus. En effet, le compilateur actuel ne modélise pas précisément ces erreurs ; il rejette donc des programmes contenant ces erreurs ainsi que d'autres programmes exempts de ces erreurs. Par exemple, les `cast` arbitraires entre pointeurs ne sont pas autorisés dans la sémantique actuelle. Ces `cast` ne sont pas utilisés dans les programmes du domaine embarqué. Par contre, ils sont souvent employés dans des applications « système ».

Par ailleurs, une deuxième perspective de recherche concerne la formalisation de nouveaux langages source et cible. Le seul langage cible du compilateur est l'assembleur du Power PC, et le passage à l'échelle du compilateur nécessite de considérer d'autres langages cible. En plus des travaux en cours sur de nouvelles traductions vers Cminor (depuis les langages Java, Concurrent Cminor et mini-ML), il serait également intéressant de vérifier formellement des traductions vers le langage C (par exemple depuis un langage synchrone).

Une troisième perspective de recherche concerne le lien entre compilation certifiée et preuve de programmes. En particulier, il serait intéressant d'étudier l'apport du style axiomatique au compilateur certifié CompCert. Il s'agit de trouver des propriétés utiles au compilateur certifié, qu'il est plus aisé de vérifier formellement à partir d'une sémantique axiomatique. Cette perspective de recherche concerne également la logique de séparation.

En effet, récemment, la logique de séparation a été utilisée pour définir des analyses statiques dites de forme (« shape analysis ») qui permettent de découvrir des structures de données chaînées utilisées dans les programmes. Ces analyses permettent de découvrir (i.e. générer) des invariants dans les programmes, et donc d'automatiser davantage le raisonnement en preuve de programmes.

Ces analyses n'ont pas encore été vérifiées formellement. Elles sont difficiles à mettre en œuvre et elles reposent sur des techniques sophistiquées d'interprétation abstraite. Il serait ainsi intéressant de bénéficier de l'environnement de logique de séparation présenté dans ce mémoire pour formaliser des analyses de forme et les vérifier formellement *a posteriori*, c'est-à-dire sans formaliser les notions d'interprétation abstraite (de treillis et de correspondances de Galois) [132]. De plus, il serait alors intéressant d'exploiter ces analyses dans le compilateur CompCert, afin de le munir de davantage d'optimisations.

Enfin, une dernière perspective de recherche concerne la vérification formelle d'algorithmes de recherche opérationnelle utiles à la compilation. Ce travail nécessite la formalisation de bibliothèques de structures de données de graphes, qui pour l'instant ne sont pas disponibles en Coq.

Bibliographie

[1] Sandrine Blazy. Partial evaluation for the understanding of Fortran programs. *Journal of Software Engineering and Knowledge Engineering*, 4(4) :535–559, 1994.

[2] Sandrine Blazy and Philippe Facon. Formal specification and prototyping of a program specializer. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95 : Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 666–680, Aarhus, May 1995. Springer-Verlag.

[3] Sandrine Blazy and Philippe Facon. An automatic interprocedural analysis for the understanding of scientific application programs. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *International seminar on partial evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 1–16, Dagstuhl castle, February 1996. Springer-Verlag.

[4] Sandrine Blazy and Philippe Facon. Partial evaluation for program understanding. *ACM Computing Surveys*, 30es(4), September 1998. Symposium on partial evaluation.

[5] Sandrine Blazy. Specifying and automatically generating a specialization tool for Fortran 90. *Journal of Automated Software Engineering*, 7(4) :345–376, December 2000.

[6] Sandrine Blazy. Transformations certifiées de programmes impératifs. Technical report 398, CEDRIC, December 2002.

[7] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006 : 14th Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[8] Sandrine Blazy. Experiments in validating formal semantics for C. In *Proceedings of the C/C++ Verification Workshop*, pages 95–102. Technical report ICIS-R07015, Radboud University Nijmegen, 2007.

[9] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. Soumis à la revue "Journal on Automated Reasonning" le 1er octobre 2008, 24 pages.

[10] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In Kung-Kiu Lau and Richard Banach, editors, *7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, November 2005.

[11] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal on Automated Reasoning*, 41(1) :1–31, July 2008.

[12] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[13] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor (extended version). Technical Report RR 6138, INRIA, March 2007. https ://hal.inria.fr/inria-00134699.

[14] Maulik A. Dave. Compiler verification : a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6) :2, 2003.

[15] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy*, pages 1–13. ACM, 2004.

[16] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler : Code generation and implementation correctness. In *IEEE Conference on Software Engineering and Formal Methods (SEFM'05)*, 2005.

[17] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4) :619–695, 2006.

[18] Adam J. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 54–65, 2007.

[19] Projet ANR-05-SSIA-019 CompCert. `http://compcert.inria.fr`, march 2008. CompCert C compiler, version 1.2.

[20] Glynn Winskel. *The formal semantics of programming languages - an introduction*. MIT Press, 1993.

[21] Gilles Kahn. Natural Semantics. In *Proc. of the Symp. on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 237–257. Springer, 1987.

[22] Zohar Manna Aaron R. Bradley. *The calculus of computation*. Springer-Verlag, 2007. 366 pages.

[23] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML : A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[24] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 7(3) :212–232, 2005.

[25] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml. *Journal of Logic and Algebraic Programming*, 58(1-2) :89–106, 2004.

[26] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *6th Int. Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[27] Robert W. Floyd. Assigning meanings to programs. In American Mathematical Society, editor, *Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[29] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.

[30] Andrew M. Pitts and P. Dybjer, editors. *Semantics and Logics of Computation*. Cambridge University Press, New York, NY, USA, 1997.

[31] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proc. of the Symposium on Logic in Computer Science*, Cambridge, USA, June 1986.

[32] P. Borras, D. Clement, Th. Despeyrouz, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR : The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24, pages 14–24, New York, NY, 1989. ACM Press.

[33] Foresys. `http://www.simulog.fr/eis/fore1.htm`.

[34] Thierry Despeyroux. Typol : a formalism to implement natural semantics. Technical Report RT-0094, INRIA, march 1988.

[35] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988.

[36] Coq development team. The Coq proof assistant. Software and documentation available at `coq.inria.fr`, 1989-2008.

[37] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[38] Tobias Nipkow. Winskel is (almost) right : Towards a mechanized semantics. *Formal Aspects of Computing*, 10(2) :171–186, 1998.

[39] J. Chrząszcz. *Modules in Type Theory with Generative Definitions*. PhD thesis, Warsaw Univerity and University of Paris-Sud, January 2004.

[40] Neil D. Jones, C.K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[41] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. of Principles Of Progamming Languages symposium (POPL)*, pages 493–501, 1993.

[42] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *International seminar on partial evaluation*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl castle, February 1996. Springer-Verlag.

[43] ACM. *Symposium on partial evaluation*, number 4 in ACM Computing Surveys, December 1998.

[44] Sandrine Blazy. *La spécialisation de programmes pour l'aide à la maintenance du logiciel*. PhD thesis, CNAM, December 1993.

[45] ANSI, New York. *Programming language Fortran 90*, 1992. ANSI X3.198-1992 and ISO/IEC 1539-1991(E).

[46] Y. Bertot and R. Fraer. Reasoning with executable specifications. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95 : Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 531–45, Aarhus, May 1995. Springer-Verlag.

[47] Delphine Terrasse. Encoding natural semantics in Coq. In V. S. Alagar, editor, *Proc. of the 4th Conf. on Algebraic Methodology and Software Technology*, volume 936 of *Lecture Notes in Computer Science*, pages 230–244, Montreal, Canada, 1995. Springer-Verlag.

[48] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[49] Jean Raymond Abrial. *The B-Book assigning programs to meanings*. Cambridge University Press, 1996.

[50] Cliff B. Jones. *Systematic development using VDM*. Prentice-Hall, 1990.

[51] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Proc. of Principles Of Progamming Languages symposium (POPL)*, pages 379–392, San Francisco, USA, 1995.

[52] Olivier Boite and Catherine Dubois. Proving Type Soundness of a Simply Typed ML-like Language with References. In R. Boulton and P. Jackson, editors, *Supplemental Proc. of TPHOL'01, Informatics Research Report EDI-INF-RR-0046 of University of Edinburgh*, pages 69–84, 2001.

[53] Clément Renard. Un peu d'extensionnalité en Coq. Mémoire de DEA, September 2001. Université Paris VI.

[54] Pierre Courtieu. Normalized types. In *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 554–569. Springer, 2001.

[55] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[56] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[57] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL : Intermediate language and tools for analysis and transformation of C programs. In *CC '02 : Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, 2002.

[58] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, New York, NY, USA, 2003. ACM Press.

[59] Koushik Sen, Darko Marinov, and Gul Agha. Cute : a concolic unit testing engine for c. In *ESEC/FSE-13 : Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[60] CEA LIST. FRAMA-C : Framework for modular analysis of C. `frama-c.cea.fr`, 2008.

[61] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE '07 : Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2007. ACM.

[62] Ben Hardekopf and Calvin Lin. The ant and the grasshopper : fast and accurate pointer analysis for millions of lines of code. *SIGPLAN Not.*, 42(6) :290–299, 2007.

[63] Tobias Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998. Invited talk.

[64] Zaynah Dargaye. Décurryfication certifiée. In *Journées Francophones des Langages Applicatifs (JFLA'07)*, pages 119–134. INRIA, 2007.

[65] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 211–225. Springer, 2007.

[66] Andrew W.Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[67] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 2007. Accepted for publication, to appear in the special issue on Structural Operational Semantics.

[68] Michael Norrish. *C formalised in HOL.* PhD thesis, University of Cambridge, December 1998.

[69] Brian W.Kernighan and Dennis M.Ritchie. *The C programming language.* software series. Prentice-Hall, second edition, 1988.

[70] Tom Duff, 1983. `www.lysator.liu.se/c/duffs-device.html`.

[71] N.S. Papaspyrou. *A formal semantics for the C programming language.* PhD thesis, National Technical University of Athens, February 1998.

[72] V. A. Nepomniaschy, Igor S. Anureev, I. N. Mikhailov, and Alexey V. Promsky. Towards verification of c programs. c-light language and its formal semantics. *Programming and Computer Software*, 28(6) :314–323, 2002.

[73] E. Börger, N.G. Fruja, V.Gervasi, and R.F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2-3) :235–284, 2005.

[74] Y. Gurevich and J.K. Huggins. The semantics of the C programming language. In *Proc. of CSL'92 (Computer Science Logic)*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer Verlag, 1993.

[75] Yves Bertot, Venanzio Capretta, and Kuntal Das Barman. Type-theoretic functional semantics. In *Theorem Proving in Higher Order*

Logics, 20th International Conference, TPHOLs 2002, Proceedings, volume 2410 of *Lecture Notes in Computer Science*, pages 83–98, 2002.

[76] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, August 2000.

[77] David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. Extracting purely functional contents from logical inductive types. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 70–85, 2007.

[78] David Walker. Stacks, heaps and regions : one logic to bind them. In *Second workshop on semantics, program anlysis and computing analysis for memory management (SPACE)*, Venice, Italy, January 2004. invited talk.

[79] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[80] R.D. Tennent and D.R. Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2) :119–129, 2000.

[81] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS-02))*, pages 55–74, Los Alamitos, July 22–25 2002. IEEE Computer Society.

[82] Ishtiaq and O'Hearn. BI as an assertion language for mutable data structures. In *POPL : 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

[83] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot : Modular automatic assertion checking with separation logic. In Frank S. de Boer and, editor, *4th International Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[84] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In George C.

Necula and Philip Wadler, editors, *Proc. of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 87–99. ACM, 2008.

[85] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2007.

[86] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis : A shape analysis that discovers preconditions. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 402–418. Springer, 2007.

[87] Élodie Jane Sims. *Pointer analysis and separation logic*. PhD thesis, École Polytechnique, Décembre 2007.

[88] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2007.

[89] Stephen Brookes. A semantics for concurrent separation logic. *TCS : Theoretical Computer Science*, 375 :227–270, 2007.

[90] Concurrent Cminor project. `http://www.cs.princeton.edu/~appel/cminor/`.

[91] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL'05*, pages 259–270, 2005.

[92] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[93] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified : Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23 :299–318, 1999.

[94] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *POPL*, pages 293–302, 1989.

[95] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[96] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Verification of the heap manager of an operating system using separation logic. In *3rd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2006), Charleston SC, USA, January 14, 2006*, pages 61–72, Jan. 2006.

[97] Andrew W. Appel. Tactics for separation logic. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`, jan 2006.

[98] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and He Jifeng, editors, *8th International Conference on Formal Engineering Methods (ICFEM 2006), Macao SAR, China*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer-Verlag, Oct. 2006.

[99] Sylvain Conchon, Evelyne Contejean, Johannes Kannig, and Stéphane Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In John Rushby and N. Shankar, editors, *AFM07 (Automated Formal Methods)*, 2007.

[100] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[101] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as : Programming Research Group, Technical Monograph 25.

[102] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4) :596–619, 1983.

[103] Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

[104] Leonor Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

[105] J. W. Coleman and C. B. Jones. Guaranteeing the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4) :807–841, 2007.

[106] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (11th PPOPP'2006), ACM SIGPLAN Notices*, pages 129–136, New York, New York, USA, March 2006. ACM SIGPLAN 2006. Published as Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (11th PPOPP'2006), ACM SIGPLAN Notices, volume 41, number 3.

[107] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, New York, NY, USA, 1987. ACM.

[108] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[109] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9) :690–691, September 1979.

[110] Éric Soutif Sandrine Blazy, Benoît Robillard. Vérification formelle d'un algorithme d'allocation de registres par coloration de graphes. In *Journées Francophones des Langages Applicatifs (JFLA'08)*, Étretat, France, January 2008. Accepted for publication, to appear.

[111] Sandrine Blazy, Benoît Robillard, and Éric Soutif. Coloration avec préférences : complexité, inégalités valides et vérification formelle. In *ROADEF'08, 9e congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision*, 2008.

[112] Sandrine Blazy and Benoît Robillard. Live-range unsplitting for faster optimal coalescing. Soumis à la conférence CC'09 le 11 octobre 2008, 15 pages.

[113] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6) :98 – 105, 1982.

[114] David Bernstein, Dina Q. Goldin, Martin Charles Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the conference on Programming language design and implementation (PLDI)*, pages 258–263, 1989.

[115] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :428 – 455, 1994.

[116] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3) :300–324, 1996.

[117] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. *Programming Languages and Systems, 3rd Asian Symp., APLAS 2005, Japan, November, 2005, Proc.*, 3780 :315–329, 2005.

[118] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8) :929–965, 1996.

[119] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–253, 2001.

[120] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In Shriram Krishnamurthi and Martin Odersky, editors, *Compiler Construction, 16th International Conference, CC 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 26-30, Proceedings*, volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[121] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *LCTES '07 : Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 103–112, New York, NY, USA, 2007. ACM.

[122] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *International symposium on code generation and optimization (CGO'07)*, pages 102–114, San Jose, USA, mar 2007. IEEE Computer Society.

[123] Andrew W. Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09 – `http://www.cs.princeton.edu/~appel/graphdata/`, 2005.

[124] Christian Andersson. Register allocation by optimal graph coloring. In *Compiler Construction (CC)*, pages 33–45, 2003.

[125] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1 :180–187, 1972.

[126] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of specification patterns with the B method. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB 2003 : Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pages 40–57. Springer, 2003.

[127] Martin Fowler. *Analysis patterns : reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[128] Erich Gamma, Richard Helm, Ralph Johnson, and John M.Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995.

[129] George Spencer-Brown. *Laws of form*. Cognizer Press, 4e edition, 1994.

[130] *The Description Logic Handbook*. Cambridge University Press, 2003.

[131] Gang Tan and Andrew W. Appel. A compositional logic for control flow. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proc.*, volume 3855 of *Lecture Notes in Computer Science*, pages 80–94, 2006.

[132] William Mansky. Automating separation logic for Concurrent Cminor. Master's thesis, Princeton University, mai 2008.

Encadrement d'étudiants

Stages de niveau M2 ou assimilés

[stage 1] **Benoît Robillard**. *Vérification formelle d'un algorithme d'allocation de registres.*
Stage de 3e année de l'ENSIIE et du master M2 STIC (parcours RO), CNAM, mars à septembre 2007, co-encadrement à 70% avec Éric Soutif, financé par le CEDRIC.

[stage 2] **Maxime Bargiel**. *Étude et extension du procédé @L-is pour la vérification et l'enrichissement automatique de glossaires formels.*
Stage de maîtrise de l'Université de Sherbrooke (M.Sc canadien), septembre 2006 à juin 2008, co-encadrement à 30% avec Marc Frappier, financé par la société Bfd.

[stage 3] **Sonia Ksouri**. *Logique de séparation et logique du rely/guarantee.*
Stage du master M2 STIC (parcours LS), CNAM-Paris VI, avril à septembre 2007, co-encadrement à 80% avec Marc Shapiro (LIP6), financé par le LIP6.

[stage 4] **Thomas Moniot**. *Sémantique formelle d'un sous-ensemble réaliste du langage C.*
Stage de 3e année de l'ENSIIE et du master M2 MOPS, ENSIIE-INT-Université d'Évry, avril à septembre 2006, financé par le projet ANR CompCert.

[stage 5] **Agnès Gonnet**. *Validation de glossaires en Coq.*
Stage du DESS Développement de Logiciels Surs, CNAM-Paris VI, mars à septembre 2005, co-encadrement à 80% avec Philippe Michelin (société Bfd), financé par Bfd.

[stage 6] **Zaynah Dargaye**. *Sémantique formelle et pré-compilation d'un sous-ensemble du langage C.*
Stage du Master Parisien de Recherche en Informatique, mars à juillet 2005, co-encadrement à 50% avec Xavier Leroy.

[stage 7] **François Armand**. *Certification en Coq d'un compilateur.*

Stage de 3e année de l'IIE et du DEA d'informatique IIE-INT-Université d'Évry, février à juin 2004, financé par l'ARC Concert.

[stage 8] **Mourad Naouari**. *Rétro-conception de transactions de bases de données.*
Stage du DEA d'informatique IIE-INT-Université d'Évry, février à juin 2004, co-encadrement à 50% avec Régine Laleau (LACL).

[stage 9] **Thibaut Tourneur**. *Analyses de flots de données certifiées en Coq.*
Stage de 3e année de l'IIE, février à juin 2003.

[stage 10] **Nathalie Ly**. *Spécification et implémentation de mécanismes de réutilisation de composants de spécifications.*
Stage du DESS Développement de Logiciels Surs, CNAM-Paris VI, février à septembre 2003, co-encadrement à 50% avec Régine Laleau (LACL).

[stage 11] **Frédéric Gervais**. *Réutilisation de composants de spécifications en B.*
Stage du DEA d'informatique IIE-INT-Université d'Évry, février à septembre 2002, co-encadrement à 50% avec Régine Laleau (LACL).

[stage 12] **Audrey Moulin**. *Automatisation de traitements de migration de données.*
Stage de 3e année de l'IIE, janvier à juin 1998, financé par la société Stéria et co-encadré à 50% avec une personne de cette société.

[stage 13] **Romain Vassallo**. *Ergonomie et évolution d'un outil de compréhension de programmes.*
Stage de 3e année de l'IIE, janvier à juin 1995.

[stage 14] **Nathalie Dubois, Pousith Sayarath**. *Aide à la compréhension et à la maintenance du logiciel : les pointeurs en spécialisation de programmes.*
Stages de 3e année de l'IIE, janvier à juin 1995.

[stage 15] **Frédéric Paumier, Hubert Parisot**. *Calculs interprocéduraux pour la spécialisation de programmes Fortran.*
Stages de 3e année de l'IIE, janvier à juin 1994, financés par EDF.

[stage 16] **Skander Korrab**. *Représentation de programmes dans Centaur.*
Stage de 3e année de l'IIE, janvier à septembre 1993, financé par EDF.

Direction de thèse

Depuis octobre 2007, je co-encadre (à 60%, avec Éric Soutif) la thèse de Benoît Robillard intitulée « Vérification formelle d'algorithmes pour compila-

teurs optimisants ». Cette thèse est financée par une bourse MENRT de l'école doctorale EDITE.

Divers

D'avril 2007 à septembre 2008, j'ai piloté le travail de post-doctorat de Keiko Nakata, qui a été recrutée sur un financement du projet ANR CompCert.

10 Curriculum vitae

Situation

Depuis septembre 1994, je suis maître de conférences en informatique à l'ENSIIE [1]. J'effectue mes recherches au laboratoire CEDRIC (Centre d'Études et de Recherches en Informatique du CNAM), au sein de l'équipe Conception et Programmation Raisonnées (CPR) dirigée actuellement par Catherine Dubois et par Philippe Facon puis Véronique Donzeau-Gouge auparavant.

Dans le cadre d'un congé pour recherches, puis d'une délégation de l'INRIA, j'ai passé deux années universitaires (de 2004 à 2006) à l'INRIA Paris-Rocquencourt dans l'équipe projet Gallium (anciennement Cristal).

Enseignement

Mon activité d'enseignement se déroule à l'ENSIIE, qui est une école d'ingénieurs recrutant principalement sur le concours commun de Centrale-Supélec. Les promotions sont constituées actuellement de 150 étudiants environ, présents pendant 3 ans, répartis en 07/08 en 5 groupes de TDs, et 5 ou 10 groupes de TPs selon les matières. Mes cours sont dispensés soit aux étudiants de l'ENSIIE, soit aux stagiaires de la formation continue en alternance d'ingénieurs en informatique de l'ENSIIE (FIP) diplômant des promotions d'une dizaine de stagiaires, présents pendant 2 ans. J'interviens également dans le master M2-R MOPS de l'Université d'Évry (ex-DEA d'informatique). Depuis 1994, presque chaque année, j'ai dispensé des enseignements à ces 3 publics.

Les enseignements que je dispense depuis 1994 sont précisés dans le tableau suivant. J'en indique la durée moyenne et la promotion concernée. Pour chaque enseignement, figure la période durant laquelle je l'ai effectué. Dans le tableau, les enseignements sont triés par niveau d'étude : L3 (ENSIIE et FIP $1^{ère}$

[1] Par le passé, l'ENSIIE (École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise) dépendait du CNAM (Conservatoire National des Arts et Métiers, situé à Paris) et s'appelait alors IIE (Institut d'Informatique d'Entreprise). Le 1er août 2006, l'IIE est devenue une école autonome (EPA, art.43), rattachée à l'Université d'Évry.

matière	période	cours	td/tp	année	effectif	commentaires
algorithmique et programmation impérative (Pascal puis C)	1994-2004 1994-2004	12h 10h	32h 20h	1A FIP 1	130 12	polycopié, exercices corrigés, devoir, tp noté, projet individuel, soutien, coordination de 8 chargés de tp, recrutement de vacataires
programmation fonctionnelle	depuis 1996		31h		30	projet indivuduel
structures de données avancées	1994-1998 1994-1998	12h 6h	32h 20h	1A FIP 1	130 12	polycopié, ex. corrigés, devoir, tp polycopié, ex. corrigés, devoir
projet informatique 1	1994-2000		50h	1A	130	sujets de projet
spécifications formelles (td) spécifications formelles (cours, td) spécifications formelles (cours) spécifications formelles (projet)	depuis 1994 2002-2004 2002 1994-1998, 2008	 7h30 12h 7h30	9h 9h	2A 2A M2 FIP 2	30 130 20 12	ex. corrigés polycopié, ex. corrigés, devoir polycopié sujets de projet
langage Prolog	1994-1997		8h	2A	30	projet indivuduel
compilation	depuis 2006 2000-2004	12h 10h	10h30 10h	2A FIP2	130 12	polycopié, ex. corrigés, devoir, tp
test du logiciel	2000-2007		7h30	2A	30	
interfaces homme-machine	1995-1998	1h30	12h	2A	120	polycopié, tp noté
système d'exploitation (noyau Linux)	1998	10h	20h	2A	15	
projet informatique 2	1994-2000		50h	2A	130	sujets de projet
programmation mathématique	2007		22h	2A	130	sujets de projet
analyse statique par interprétation abstraite	depuis 2002	18h	3h	M2	20	polycopié, devoir, tp noté
sémantique des langages	1994-1997	3 à 9h		3A M2	20 20	polycopié, devoir
maintenance du logiciel	1994-1998 1998	7h30 6h		FIP 2 Hull M1	12 40	polycopié polycopié (en anglais)
techniques de preuve formelle	2002-2004	3h	12h	3A	20	tp noté
projet d'option	depuis 2002		24h	3A	15	sujets de projet

Tab. 9.1 – Synthèse des activités d'enseignement

années), M1 (ENSIIE et FIP 2e années, Université de Hull en Angleterre) et M2 (master M2-R MOPS et option de 3e année de l'ENSIIE).

Depuis 1994, j'ai eu la responsabilité de cours (indiqués en gris clair dans le tableau) que j'enseigne à l'ENSIIE, en FIP (algorithmique et programmation, structures de données avancées, compilation, spécifications formelles), et depuis 2000 en master M2 (analyse statique par interprétation abstraite).

Par ailleurs, j'ai mis en place de nouveaux enseignements (en analyse statique et en compilation) et profondément remanié le cours d'algorithmique-programmtion en langage C (qui ne comportait ni TPs, ni enseignement du C). En 2002, avec Catherine Dubois, nous avons créé et mis en place une option de troisième année intitulée programmation raisonnée.

Depuis 1994, j'ai eu la responsabilité de matières consistant en du suivi de projet (en gris foncé dans le tableau : projet informatique de 1ère et 2e années à l'ENSIIE, projet de spécifications formelles en FIP, projet de l'option programmation raisonnée de 3e année). Dans chacune de ces matières, le suivi de projet n'existait pas auparavant. En 1997, j'ai mis en place une nouvelle organisation des projets d'informatique de l'ENSIIE (planification de séances régulières de suivi, remise de trois rapports d'avancement). Nous étions alors trois enseignants à suivre l'ensemble des binômes de 1ère et 2e années.

Responsabilités collectives

Membre titulaire de la commission de spécialistes du CNAM durant 6 ans (98/01, 04/07).
Membre titulaire extérieur de la commission de spécialistes de l'Université d'Évry (01/04).
Membre du conseil d'école de l'ENSIIE depuis sa création en 1998.
Membre du conseil d'administration de l'ENSIIE depuis sa création en 2006.
Depuis 2007, je représente l'ENSIIE auprès de l'incubateur d'entreprise de l'INT.

Direction des études du MSBI de l'IIE. En 2001, avec Catherine Dubois, nous avons créé un mastère spécialisé en bioinformatique à l'IIE (diplôme agréé par la Conférence des Grandes Écoles), et destiné à des informaticiens titulaires d'un diplôme de niveau bac+5. Jusqu'à l'obtention de mon congé pour recherches en 2004, j'ai eu la responsabilité de la direction des études de ce mastère.

Responsabilité d'échanges avec des universités étrangères Depuis 1994, j'ai été responsable de différentes coopérations avec des universités ou laboratoires étrangers (Université de Dortmund en Allemagne, laboratoire de recherche de Toshiba à Tokyo au Japon, universités de Hull et Oxford en Angleterre, université de Sherbrooke au Canada), dans lesquels des étudiants ont accompli soit une année complète, soit un stage.

Recrutement des étudiants issus d'IUT. Outre le recrutement sur le concours Centrale-Supélec, l'ENSIIE accepte une dizaine d'étudiants titulaires d'un DUT en $1^{ère}$ année. De 1994 à 2000, j'ai participé aux oraux et jurys de sélection qui étaient soit internes à l'ENSIIE, soit organisés à Cergy-Pontoise par l'ENSEA.

Formation

- **Décembre 1993** Thèse de doctorat au Conservatoire National des Arts et Métiers (CNAM) sous la direction de Philippe Facon : *La spécialisation de programmes pour l'aide à la maintenance du logiciel.*
- **Juin 1990** DEA d'informatique, Université Paris 6.
- **Juin 1990** Diplôme d'ingénieur en informatique, Institut d'Informatique d'Entreprise (IIE, CNAM).

Publications

Revues internationales

- Xavier Leroy, Sandrine Blazy. *Formal verification of a C-like memory model and its uses for verifying program transformations.*
  Journal on Automated Reasoning. Vol.41, numéro 1. Juillet 2008, pp. 1-31.
- Sandrine Blazy. *Specifying and Automatically Generating a Specialization Tool for Fortran 90.*
  Journal of Automated Software Engineering. Vol. 7, numéro 4. Décembre 2000, pp. 345-376.
- Sandrine Blazy, Philippe Facon. *Partial evaluation for program comprehension.*
  ACM Computing Surveys, Symposium on Partial Evaluation, Vol. 30, numéro 3 es. Septembre 1998.
- Sandrine Blazy, Philippe Facon. *Partial Evaluation for the understanding of Fortran programs (extended version).*

Journal of Software Engineering and Knowledge Engineering. Vol. 4, numéro 4, 1994, pp. 535-559.

Revue nationale

– Sandrine Blazy. *Comment gagner confiance en C ?*
Technique et Science Informatique (TSI), numéro spécial « Langages applicatifs », Lavoisier, Volume 26, numéro 9, 2007, pp.1195-1200 (chronique).

Édition d'actes

– Sandrine Blazy. *Actes de la conférence JFLA2008 (Journées Francophones des Langages Applicatifs).* ISBN 2-7261-1295-1, INRIA, 2008, 173 pages.

Conférences internationales avec comité de sélection

– Andrew W. Appel, Sandrine Blazy. *Separation logic for small-step Cminor.*
20e conf. "Theorem Proving in Higher Order Logics" (TPHOLs 2007), Kaiserslautern, Allemagne. LNCS n°4732, 2007, pp.5-21.
– Sandrine Blazy, Zaynah Dargaye, Xavier Leroy. *Formal verification of a C compiler front-end.*
14^{e} "Symposium on Formal Methods" (FM'06), Hamilton, Canada. LNCS n°4085, 2006, pp.460-475.
– Sandrine Blazy, Xavier Leroy. *Formal verification of a memory model for C-like imperative languages.*
7e "International Conference on Formal Engineering Methods" (ICFEM'05), Manchester, Royaume-Uni. LNCS n°3785, 2005, pp.280-299.
– Sandrine Blazy, Frédéric Gervais, Régine Laleau. *Reuse of Specification Patterns with the B Method.*
3e Conf. of B and Z Users (ZB'2003), Turku, Finlande. LNCS 2651, 2003.
– Stéphanie Bocs, Sandrine Blazy, Philippe Glaser, Claudine Médigue, *An automatic detection of prokaryotic coDing sequences combining several independant methods.*
Genome 2000, Int. Conf. on microbial and model genomes, American Society for Microbiology and Institut Pasteur, Paris, 2000.
– Sandrine Blazy, Philippe Facon. *Application of formal methods to the development of a software maintenance tool.*
IEEE Automated Software Engineering Conf. (ASE'97), Lake Tahoe,

États-Unis, 1997, pp. 162-171.
Article primé parmi les 6 meilleurs articles présentés lors de la conférence.

- Sandrine Blazy, Philippe Facon. *An automatic interprocedural analysis for the understanding of scientific application programs.*
  International seminar on partial evaluation, Dagstuhl castle, Saarbrücken, Allemagne, LNCS 1110, 1996, pp. 1-16.
- Sandrine Blazy, Philippe Facon. *Formal specification and prototyping of a program specializer.*
  Theory and Practice of Software Development (TAPSOFT'95), Aarhus, Danemark, LNCS 915, 1995, pp. 666-680.
- Sandrine Blazy, Philippe Facon. *Partial evaluation for the understanding of Fortran programs.*
  Software Engineering and Knowledge Engineering Conf. (SEKE'93), San Francisco, États-Unis, 1993, pp. 517-525.
  Prix du meilleur article étudiant.
- Sandrine Blazy, Philippe Facon. *Partial evaluation and symbolic computation for the understanding of Fortran programs.*
  Conf. on Advanced Information Systems Engineering (CAISE'93), Paris, pp. 184-198. LNCS 685, 1993.
- Marc Haziza, Jean-François Voidrot, Earl Minor, Lech Pofelski, Sandrine Blazy. *Software maintenance : an analysis of industrial needs and constraints.*
  IEEE Conf. on Software Maintenance (CSM'92), Orlando, États-Unis, 1992, pp. 18-26.

Workshops internationaux avec comité de sélection

- Sandrine Blazy. *Which C semantics to embed in the front-end of a formally verified compiler ?*
  Tools and techniques for the Verification of System Infrastructure (TTVSI), conférence en l'honneur du professeur Michael Gordon à l'occasion de ses 60 ans, Londres, Royaume-Uni, 2008. Sélection sur résumé, présentation d'un poster.
- Sandrine Blazy. *Experiments in validating formal semantics for C.*
  C/C++ Verification Workshop, Oxford, Royaume-Uni, 2007. Raboud University Nijmegen report ICIS-R07015, pp.95-102.
- Sandrine Blazy, Philippe Facon. *Interprocedural analysis for program comprehension by specialization.*

IEEE Workshop on Program Comprehension (WPC'96), Berlin, Allemagne, pp. 133-141.
- Sandrine Blazy, Philippe Facon. *SFAC : a tool for program comprehension by specialization.*
  IEEE Workshop on Program Comprehension (WPC'94), Washington D.C., États-Unis, pp. 162-167.
- Sandrine Blazy, Philippe Facon. *Partial evaluation as an aid to the comprehension of Fortran programs.*
  IEEE Workshop on Program Comprehension (WPC'93), Capri, Italie, pp. 46-54.
- Sandrine Blazy, Philippe Facon. *Évaluation partielle pour la compréhension de programmes Fortran.*
  Conf. Génie logiciel et ses applications, Toulouse, 1992, pp. 411-417.

Conférences nationales avec comité de sélection

- Benoît Robillard, Sandrine Blazy, Éric Soutif. *Coloration avec préférences : complexité, inégalités valides et vérification formelle.*
  9e congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'08), 2008, pp.123-138.
- Benoît Robillard, Sandrine Blazy, Éric Soutif. *Vérification formelle d'un algorithme d'allocation de registres par coloriage de graphes.*
  19e Journées Francophones des Langages Applicatifs (JFLA'08), 2008, pp.31-46.
- Sandrine Blazy, Benoît Robillard, Éric Soutif. *Coloration avec préférences dans les graphes triangulés.*
  Journées Graphes et algorithmes (JGA'07), 2007, p.32 (résumé d'une page).
- Sandrine Blazy, Frédéric Gervais, Régine Laleau. *Une démarche outillée pour spécifier formellement des patrons de conception réutilisables.*
  Workshop Objets, Composants et Modèles dans l'ingéniérie des systèmes d'information (OCM-SI 2003), 2003, p.5-9.

Rapports techniques

- Andrew W.Appel, Sandrine Blazy. *Separation logic for small-step Cminor (extended version)*, rapport de recherche INRIA, RR-6138, mars 2007, 29 pages.
- Sandrine Blazy. *Transformations certifiées de programmes impératifs*, rapport CEDRIC numéro 398, 2002.

- Sandrine Blazy, Frédéric Gervais, Régine Laleau. *Un exemple de réutilisation de patterns de spécification avec la méthode B*, rapport CEDRIC numéro 395, 2002.
- Sandrine Blazy, Earl Minor, Jean-Pierre Queille, Amaury Simon. *Software maintenance survey of activities*, rapport CEDRIC, avril 1992.
- Équipe EPSOM. *Software maintenance : an analysis of industrial needs and constraints*, rapport EPSOM, projet ESF-SP205, délivrable D1.2, décembre 1991, 121 pages.
- Équipe EPSOM. *State on the art on software maintenance*, rapport EPSOM, projet ESF-SP205, délivrable D1.1, août 1991, 75 pages.