# Sémantiques formelles

Sandrine Blazy

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 figures

# 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 $\sigma \vdash \mathit{synt} \Rightarrow \mathit{sem}, \sigma'$. Ce jugement d'évaluation relie un état initial $\sigma$ 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 $\sigma'$. L'état $\sigma'$ est omis lorsque l'évaluation ne modifie jamais $\sigma$. 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 $\sigma$ 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. $\forall \sigma, i, \sigma', \quad \sigma \vdash i \Rightarrow \sigma' \quad \Leftrightarrow \quad \langle i, \sigma \rangle \mapsto^* \langle \texttt{Skip}, \sigma' \rangle$

Schéma de la preuve :

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

---

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 $\sigma$ satisfaisant la pré-condition $P$ (notation $\sigma \models P$), si l'exécution de l'instruction $i$ depuis l'état $\sigma$ termine dans un état $\sigma'$, alors $\sigma'$ satisfait la post-condition $Q$ (notation $\sigma' \models Q$). La description de l'évolution de l'état $\sigma$ 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 :

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

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

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

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

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.

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

$$\mathcal{E}[\![e_1\ op\ e_2]\!]\sigma = \texttt{eval\_op\_bin}(op, \mathcal{E}[\![e_1]\!]\sigma, \mathcal{E}[\![e_2]\!]\sigma, ) \quad (13)$$

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

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.

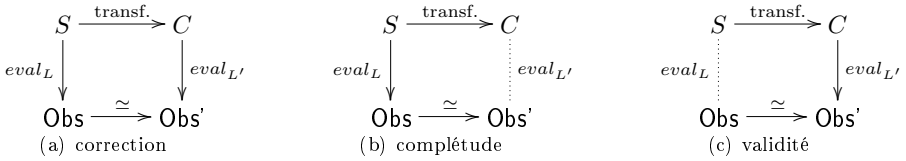



FIG. 2.5 – Diagramme commutatif exprimant la propriété de préservation sémantique

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 $\sigma$ est également transformé en $\sigma'$. 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. $\sigma_1 \rightarrowtail^{\cdot} \sigma_1'$).

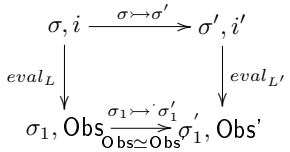



FIG. 2.6 – Diagramme commutatif (avec état)

### 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 $A \to B$ est de définir une fonction totale de type $A \to \texttt{option}\, B$ associant à toute valeur $a$ de type $A$ soit la valeur $None$ (abrégée en $\bot$ 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 :

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

Jugements d'évaluation :

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

---

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*.

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

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

FIG. 3.4 – Évaluation partielle d'une instruction $i$

$\vdash p, S_0 \mapsto p', S', \mathit{Version}'$ 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 $\vdash p, S_0 \mapsto p', S', \mathit{Version}'$, 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 $A \rightarrow B$. 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.

$$\forall y_1 \dots y_m, P(y_1 \dots y_m) \xrightarrow{\text{compilation}} P'(y_1 \dots y_m)$$
$$\text{avec } exec(P(v_1 \dots v_m)) = exec(P'(v_1 \dots v_m))$$

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

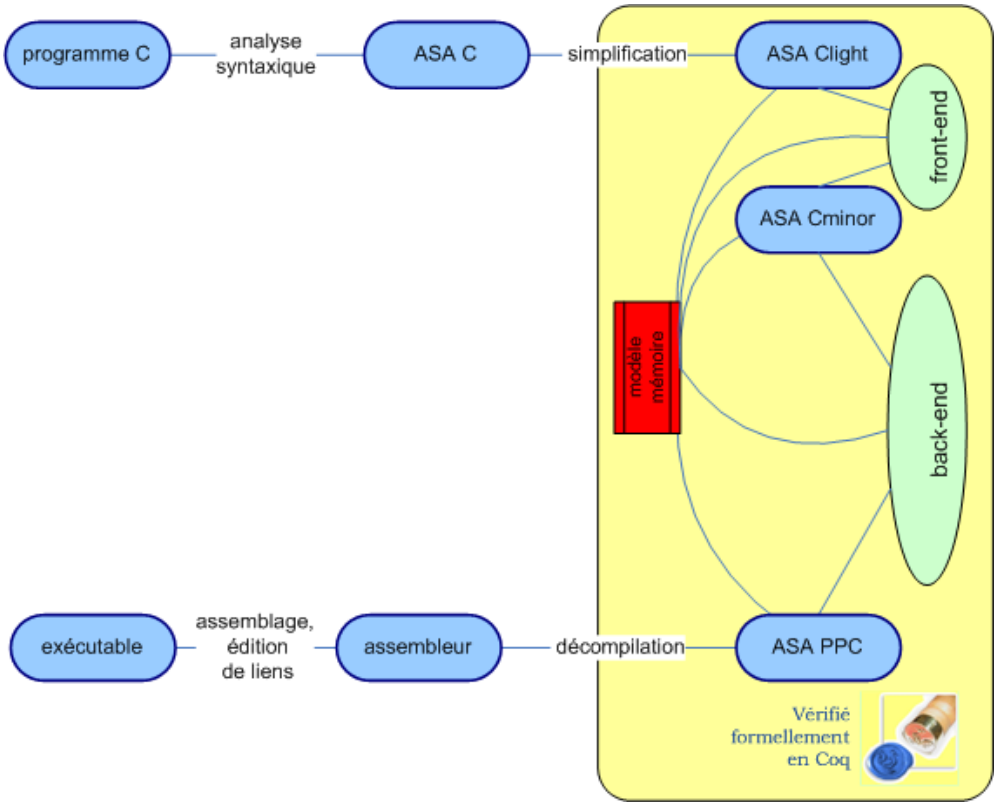



FIG. 4.2 – Architecture du compilateur CompCert

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.

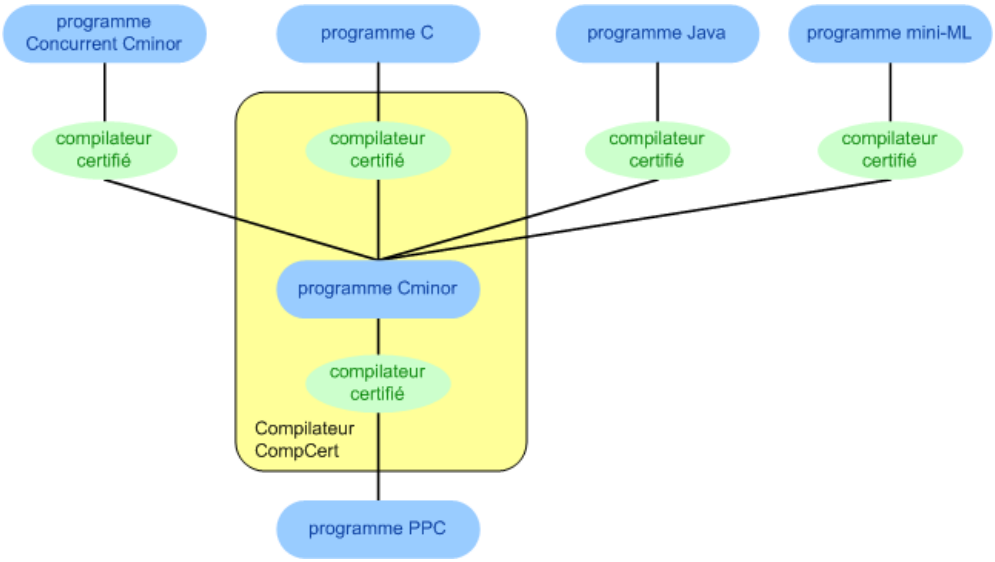



FIG. 4.3 – Le langage intermédiaire Cminor

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

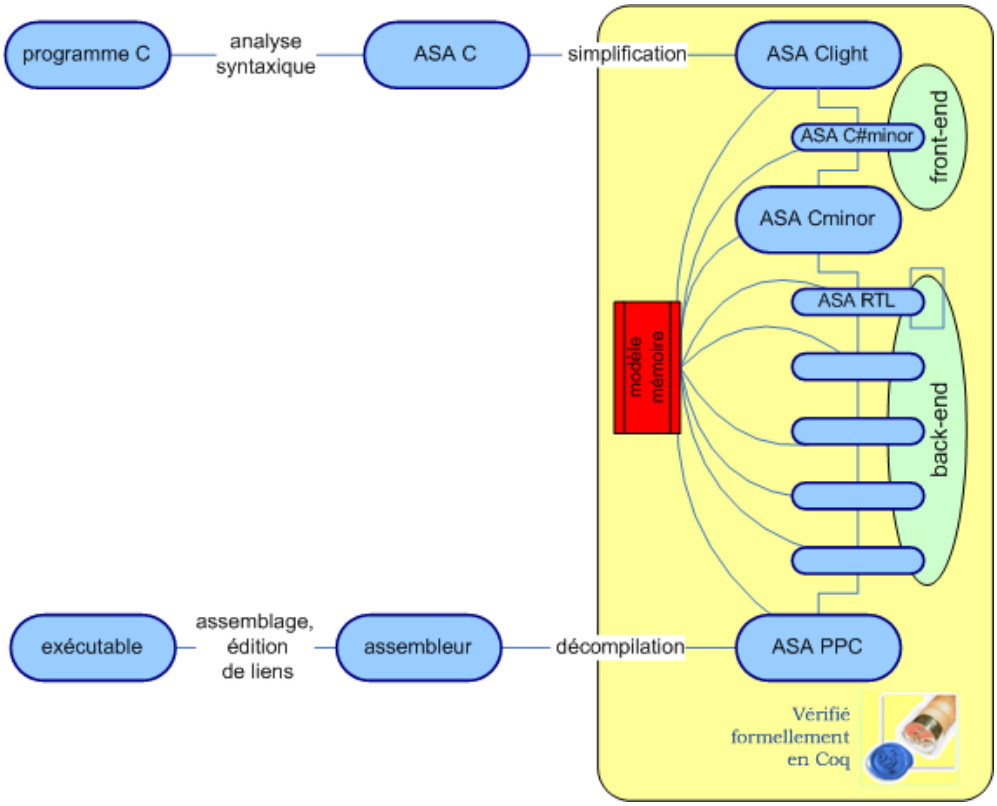



FIG. 4.4 – Le front-end et le back-end du compilateur CompCert

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 $\tau$ peut être du type pointeur sur $\tau$, mais pas du type $\tau$. 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 $\tau$). C'est pourquoi un type `struct` ou `union` est représenté sous la forme d'un opérateur $\mu$ 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` | |
| $\tau$ | ::= | `int`$(intsize, signedness)$ | |
| | \| | `float`$(floatsize)$ | |
| | \| | `void` | |
| | \| | `array`$(\tau, n)$ | |
| | \| | `pointer`$(\tau)$ | |
| | \| | `struct`$(id, (id, \tau)^*)$ | |
| | \| | `union`$(id, (id, \tau)^*)$ | |
| | \| | `comp_ptr`$(id)$ | |
| | \| | `function`$(\tau^*, \tau)$ | |

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`$(\tau)$ | 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 :

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

Signatures de fonction :

| | | |
|---|---|---|
| $\sigma$ | ::= | $\ldots \tau_i \ldots \tau$ |

Définitions de fonction :

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

Programmes complets :

| | | | |
|---|---|---|---|
| *init* | ::= | $\ldots$ | |
| $p$ | ::= | $\ldots (id_i : \tau_i = init_i)^* \ldots$ | variables globales (noms, données initialisées types) (`prog_vars`$(p)$) |
| | | $\ldots id_j = \ def\_fn_j \ldots$ | 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 :

| | | | |
|---|---|---|---|
| $v$ | $::=$ | $\texttt{int}(n)$ | entier machine (32 bits) |
| | $\mid$ | $\texttt{float}(f)$ | flottant (64 bits) |
| | $\mid$ | $\texttt{ptr}(b, \mathit{ofs})$ | pointeur |
| | $\mid$ | $\texttt{undef}$ | |
| $b$ | $\in$ | $\mathbb{Z}$ | identificateur de bloc |
| $\mathit{ofs}$ | $::=$ | $n$ | 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 :

| | | | |
|---|---|---|---|
| $\mathit{out}$ | $::=$ | $\texttt{Out\_normal}$ | aller à la prochaine instruction |
| | $\mid$ | $\texttt{Out\_continue}$ | aller à la prochaine itération de la boucle courante |
| | $\mid$ | $\texttt{Out\_break}$ | sortie de la boucle courante |
| | $\mid$ | $\texttt{Out\_return}$ | sortie de fonction |
| | $\mid$ | $\texttt{Out\_return}(v)$ | 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 $\mathit{ofs}$ 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`

$$\begin{array}{rcl} \texttt{Out\_break} & \overset{loop}{\rightsquigarrow} & \texttt{Out\_normal} \\ \texttt{Out\_return}(v) & \overset{loop}{\rightsquigarrow} & \texttt{Out\_return}(v) \end{array}$$

$$\frac{G, E \vdash a, M \Rightarrow v \qquad \texttt{is\_false}(v)}{G, E \vdash (\texttt{while}(a)\ i), M \Rightarrow \texttt{Out\_normal}, \emptyset, M} \quad (18)$$

$$\frac{G, E \vdash a, M \Rightarrow v \qquad \texttt{is\_true}(v) \qquad G, E \vdash i, M \Rightarrow out, t, M' \qquad out \overset{loop}{\rightsquigarrow} out'}{G, E \vdash (\texttt{while}(a)\ i), M \Rightarrow out', t, M'} \quad (19)$$

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

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 :

$$
\begin{array}{rll}
out ::= & \texttt{Out\_normal} & \text{aller à la prochaine instruction} \\
\mid & \texttt{Out\_exit}(n) & \text{sortie du bloc de niveau d'imbrication } n+1 \\
\mid & \texttt{Out\_return} & \text{sortie de fonction} \\
\mid & \texttt{Out\_return}(v) & \text{sortie de fonction, avec valeur renvoyée } v
\end{array}
$$

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

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 ...    --traduction-->  addint(x, y)
double x, y; ... x + y ... --traduction-->  addfloat(x, y)
```

- (2) Calculs d'adresses

```
int * p, i; ... p[i] ...          --traduction-->  addint(p, mulint(i, 4))
struct intlist * s; ... s->tl ... --traduction-->  addint(s, 4)
```

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

```
int p[2][2] ; p[0][0] = 42 ;  --traduction-->  store(int32, p, 42)
int **p ; p[0][0]=42 ;        --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) {                 s1 ;
                                    block { loop {
                                      if (!e) exit 0 ;
                                      block {
  ...                                   ...
  exit ;                                exit 1 ;
  ...                                   ...
  continue ;                            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`.

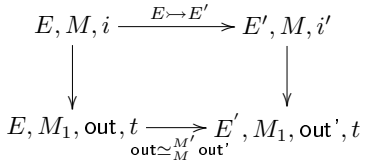



FIG. 5.8 – Préservation sémantique de la traduction de Clight vers Csharpminor

### 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

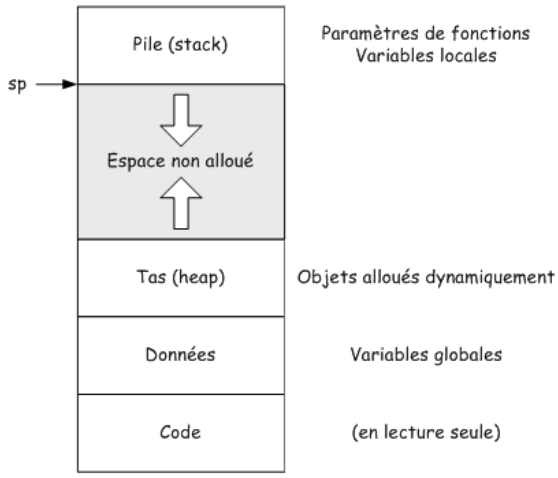



FIG. 6.1 – La mémoire d'un compilateur vue depuis un processus

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,

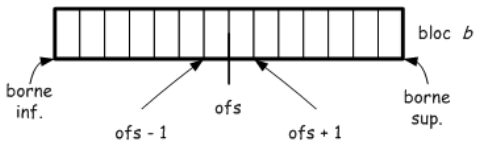



FIG. 6.2 – Un exemple de bloc de mémoire

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 $\tau$ sera toujours lue en tant que valeur de type $\tau$. 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 $\tau$ ne peut être lue ensuite qu'en tant que valeur d'un type $\tau'$ compatible avec $\tau$. 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 $\tau$ lorsque cette valeur a été stockée comme étant de type pointeur sur $\tau'$, avec $\tau$ différent de $\tau'$. Ce comportement est fréquent en C lorsque $\tau$ et $\tau'$ 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.

$$\begin{aligned} \texttt{empty} &: \texttt{mem} \\ \texttt{valid\_block} &: \texttt{mem} \times \texttt{block} \to \texttt{Prop} \\ \texttt{bounds} &: \texttt{mem} \times \texttt{block} \to \mathbb{Z} \times \mathbb{Z} \\ \texttt{fresh\_block} &: \texttt{mem} \times \texttt{block} \to \texttt{Prop} \end{aligned}$$

**Opérations de gestion de la mémoire**

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

**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

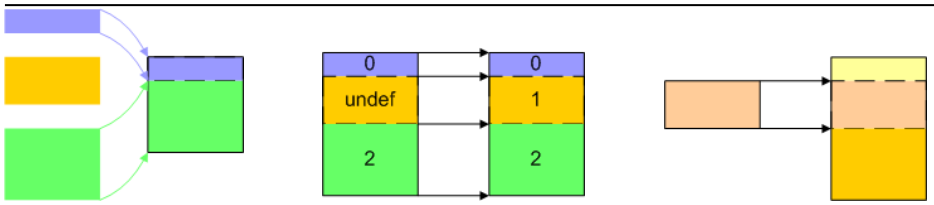
relations ont ainsi été axiomatisées. Enfin, des propriétés dérivées des axiomes ont également été prouvées au niveau abstrait. Par exemple, si $m$ désigne une mémoire dans laquelle un bloc $b$ est alloué (ce qui modifie $m$ en $m'$), alors tout accès valide à $b$ dans $m$ est également valide dans $m'$.

## 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` $\to \mathbb{Z} \times \mathbb{Z}$ | bornes (inférieure et supérieure) |
| $F$ : `block` $\to$ `bool` | blocs ayant été libérés |
| $C$ : `block` $\to \mathbb{Z} \to$ `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,

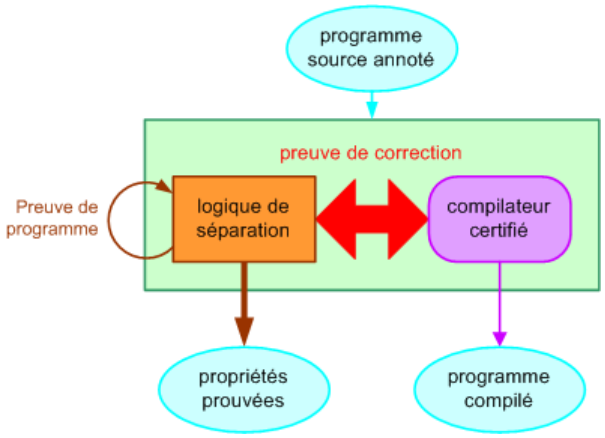



FIG. 7.1 – Preuve de programme et compilation certifiée

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 $\varphi$ 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 $\varphi$ (représentant le domaine de la mémoire).

Le jugement d'évaluation des expressions fait référence à $\varphi$ 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 $\sigma$ 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 $\sigma$ et d'une pile de contrôle $\kappa$ 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 $\kappa$, 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 $\kappa$ 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 $\kappa$ (règle 21). Un pas d'exécution d'une boucle infinie déplie cette boucle (règle 22). Étant donnée une pile de contrôle $\kappa$ représentant des instructions à exécuter à l'issue d'un

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

Séquence d'instructions

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

Boucle infinie

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

Entrée de bloc

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

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

Par définition, une continuation $k = (\sigma, \kappa)$ est bloquante si $\kappa$ 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 $\kappa$, l'exécution de $i$ selon la sémantique à continuations calcule une nouvelle pile de contrôle $\kappa'$. 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 $\sigma$ 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 $\sigma$ satisfait une formule $P * Q$ si son empreinte $\varphi_\sigma$ (qui est un composant du quadruplet $\sigma$) peut être partitionnée en deux empreintes $\varphi_1$ et $\varphi_2$ (notation $\varphi_\sigma = \varphi_1 \oplus \varphi_2$) telles que les restrictions de $\sigma$ à $\varphi_1$ (notation $\sigma[:= \varphi_1]$) et $\varphi_2$ (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

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

Boucle infinie

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

Entrée de bloc

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

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 $\sigma$, l'exécution d'une pile de contrôle $\kappa$ est sûre (notation $\mathbf{safe}\ \kappa$) lorsque tout état $\sigma'$ équivalent à $\sigma$ est tel que la continuation $(\sigma', \kappa)$ est sûre. L'interprétation des assertions utilise également un opérateur de garde d'une pile de contrôle $\kappa$ 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 $\kappa$ étant donné $A$ (notation $P \,\square_A\, \kappa$) lorsque si $A * P$ est vrai, alors l'exécution de $\kappa$ 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 $R \,\square\!\!\!_{\mathrm{R}}\, _A\, \kappa$) et de sortie de bloc (notation $B \,\square\!\!\!_{\mathrm{B}}\, _A\, \kappa$).

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

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

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 $\kappa$ (autrement dit si on peut exécuter $\kappa$), alors la pré-condition correspondante garde $i \cdot \kappa$ (autrement dit on peut exécuter $i$ puis $\kappa$). 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 $\sigma$, 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 $\kappa$ :

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

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

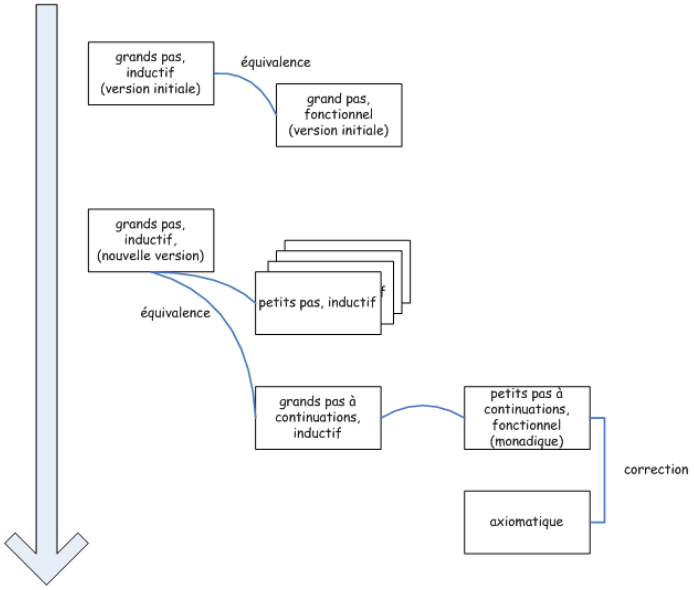



FIG. 7.5 – Chronologie des différentes sémantiques de Cminor

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.

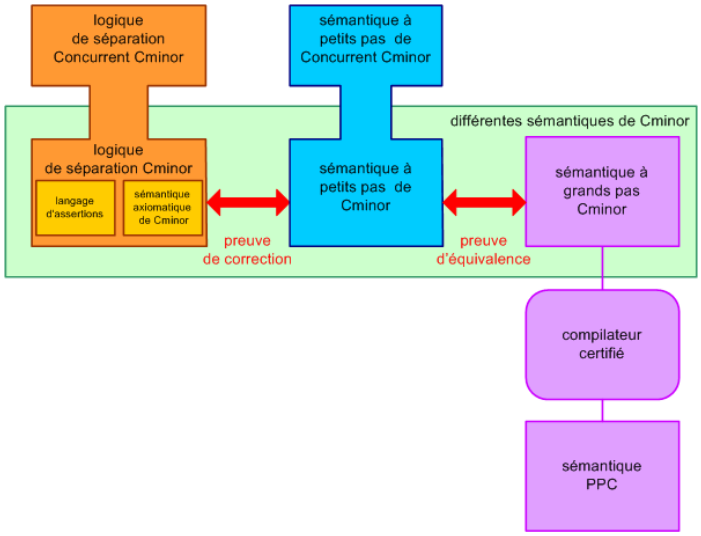



FIG. 7.6 – De CompCert à Concurrent Cminor

# 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 $1^{er}$ 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 3$^e$ 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 3$^e$ 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 3$^e$ 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 3$^e$ 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 3$^e$ 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 3$^e$ 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 3$^e$ année de l'IIE, janvier à juin 1995.

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

[stage 16] **Skander Korrab**. *Représentation de programmes dans Centaur.*
Stage de 3$^e$ 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 $1^{er}$ 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<br>1994-2004 | 12h<br>10h | 32h<br>20h | 1A<br>FIP 1 | 130<br>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<br>1994-1998 | 12h<br>6h | 32h<br>20h | 1A<br>FIP 1 | 130<br>12 | polycopié, ex. corrigés, devoir, tp<br>polycopié, ex. corrigés, devoir |
| projet informatique 1 | 1994-2000 | | 50h | 1A | 130 | sujets de projet |
| spécifications formelles (td) | depuis 1994 | | 9h | 2A | 30 | ex. corrigés |
| spécifications formelles (cours, td) | 2002-2004 | 7h30 | 9h | 2A | 130 | polycopié, ex. corrigés, devoir |
| spécifications formelles (cours) | 2002 | 12h | | M2 | 20 | polycopié |
| spécifications formelles (projet) | 1994-1998, 2008 | 7h30 | | FIP 2 | 12 | sujets de projet |
| langage Prolog | 1994-1997 | | 8h | 2A | 30 | projet indivuduel |
| compilation | depuis 2006<br>2000-2004 | 12h<br>10h | 10h30<br>10h | 2A<br>FIP2 | 130<br>12 | polycopié, ex. corrigés,<br>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,<br>devoir, tp noté |
| sémantique des langages | 1994-1997 | 3 à 9h | | 3A<br>M2 | 20<br>20 | polycopié, devoir |
| maintenance du logiciel | 1994-1998<br>1998 | 7h30<br>6h | | FIP 2<br>Hull M1 | 12<br>40 | polycopié<br>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 $2^e$ années, Université de Hull en Angleterre) et M2 (master M2-R MOPS et option de $3^e$ 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 $1^{ère}$ et $2^e$ années à l'ENSIIE, projet de spécifications formelles en FIP, projet de l'option programmation raisonnée de $3^e$ 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 $1^{ère}$ et $2^e$ 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.*
20$^e$ 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.*
7$^e$ "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.*
3$^e$ 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.*
  9$^e$ 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.*
  19$^e$ 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.

## Principales publications

Ce document rassemble mes 5 principales publications, présentées dans l'ordre des chapitres avec leur numéro de référence dans la bibliographie.

(chapitre 3)

[5] Sandrine Blazy, Philippe Facon. *Partial Evaluation for the understanding of Fortran programs.*
Journal of Software Engineering and Knowledge Engineering. Vol. 4, numéro 4, 1994, pp. 535-559.

[1] 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.

(chapitre 5)

[7] Sandrine Blazy, Zaynah Dargaye, Xavier Leroy. *Formal verification of a C compiler front-end.*
$14^e$ "Symposium on Formal Methods" (FM'06), Hamilton, Canada, 23 au 25 août 2006. Lecture Notes in Computer Science (LNCS) n°4085, Springer Verlag, pp.460-475.

(chapitre 6)

[10] Xavier Leroy, Sandrine Blazy. *Formal verification of a C-like memory model and its uses for verifying program transformations.*
Journal of Automated Reasoning. Vol.41, numéro 1, juillet 2008, pp. 1-31.

(chapitre 7)

[11] Andrew W. Appel, Sandrine Blazy. *Separation logic for small-step Cminor.*
$20^e$ conférence internationale "Theorem Proving in Higher Order Logics" (TPHOLs 2007), Kaiserslautern, Allemagne, 10 au 13 septembre 2007. Lecture Notes in Computer Science (LNCS) n°4732, Springer Verlag, pp.5-21.

International Journal of Software Engineering and Knowledge Engineering
Vol. 4 No. 4 (1994) 535–559


# PARTIAL EVALUATION FOR THE UNDERSTANDING OF FORTRAN PROGRAMS*

SANDRINE BLAZY† and PHILIPPE FACON
*CEDRIC IIE, 18 allée Jean Rostand, 91025 Evry Cedex, France*
*E-mail:* {*blazy, facon*} *@iie cnam fr*

Received 15 October 1993
Revised 20 June 1994
Accepted 20 September 1994

This paper describes a technique and a tool that support partial evaluation of FORTRAN programs, i e., their specialization for specific values of their input variables The authors' aim is to understand old programs, which have become very complex due to numerous extensions. From a given FORTRAN program and these values of its input variables, the tool provides a simplified program, which behaves like the initial program for the specific values This tool mainly uses constant propagation and simplification of alternatives to one of their branches. The tool is specified in terms of inference rules and operates by induction on the FORTRAN abstract syntax These rules are compiled into Prolog by the Centaur/FORTRAN programming environment The completeness and soundness of these rules are proven using rule induction

*Keywords*: FORTRAN, software maintenance, program understanding, program specialization, partial evaluation, proof of completeness and soundness, Centaur.

## 1. Introduction

Program understanding is the most expensive phase of the software life-cycle. It is said that 40% of the maintenance effort is spent trying to understand how existing software works [23] All maintenance problems do not require complete program understanding, but each problem requires at least a limited understanding of how the source code works, and how it is related to the external functions of the application. There exists now a wide range of tools to support program understanding [24].

Program slicing is a technique for restricting the behavior of a program to some specified subset of interest. The slice of a program $P$ on a subset $V$ of the variables of $P$ at location $i$ is the set of all the statements that might affect the values of the variables in $V$ at $i$. This is an executable program that is obtained by data flow

*This paper is an extended version of a paper that received the student paper award at the SEKE'93 conference
†Sandrine Blazy was previously at the EDF Research Center, Clamart, France

analysis. Program slicing can be used to help maintainers understand and debug code [11]

We have developed a complementary technique: reduction of programs for specific values of their input variables. It aims at understanding old programs, which have become very complex due to extensive modifications. For a given FORTRAN program and some form of restriction of its usage (e.g., the knowledge of some specific values of its input variables), the tool provides a simplified program, which behaves like the initial program when used according to the restriction. *This approach is particularly well adapted to programs which have evolved as their application domains increase continuously.*

Partial evaluation is an optimization technique used in compilation to specialize a program for some of its input variables. Partial evaluation of a subject program $P$ with respect to input variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ for the values $x_1 = c_1, \ldots, x_m = c_m$ gives a residual program $P'$, whose input variables are $y_1, \ldots, y_n$, and the executions of $P(c_1, \ldots, c_m, y_1, \ldots, y_n)$ and $P'(y_1, \ldots, y_n)$ produce the same results [19]. Such a program is obtained by replacing variables by their constant values, by propagating constant values, and by simplifying statements; for instance, replacing each alternative whose condition simplifies to a constant value (true or false) by the corresponding branch.

Partial evaluation has been applied to program optimization and compiler generation from interpreters (by partially evaluating the interpreter for a given program) [15]. In this context, previous work has primarily dealt with functional [3] and declarative languages [22]. The structure of the program may be modified (using loop expansion, and subroutine expansion and renaming [9]) in order to optimize the residual code.

As far as imperative languages are concerned, partial evaluation has been used for software reuse improvement by restructuring software components to improve their efficiency [4]. Partial evaluation has been applied to numerical computation to provide performance improvements for a large class of numerical programs, by eliminating data abstractions and procedure calls [5].

Our goal is different. We remove groups of statements that are never used in the given context, but we do not expand statements. This does not change the original structure of the code. We transform general-purpose programs into shorter and easier-to-understand special-purpose programs. This transformational approach aims at improving a given program without disturbing its correctness when used in a given restricted and stable context. However, unlike [17], we do not aim at improving a program according to a performance criterion (e.g., memory), but at improving the readability of programs.

What are the interesting program simplifications in that context? We believe that to remove useless code is always beneficial to program understanding. In that case, the objective is compatible with that of program optimization (dead code elimination [2]), but this is certainly not the case in general. On the other hand, the replacement of (occurrences of) variables by their values is not so obvious.

The benefit depends on what these variables mean for the user: Variables like PI, TAX_RATE, etc., are likely to be kept in the code; on the contrary, intermediate variables used only to decompose some computations may be not so meaningful for the user, and he may prefer to have them removed.

Replacing variables by their values may lead to dead code (by making the assignments to these variables useless) and thus gives more opportunities to remove code. However, this is certainly not a sufficient reason to do systematic replacement. Of course, even when there is no replacement, the known value of a variable is kept in the environment of our simplification rules, as it can give opportunities to remove useless code, for instance, if the condition of an alternative may be evaluated thanks to that knowledge (and thus a branch may be removed).

The benefit of replacement depends not only on the kind of variable but also on the kind of user: A user who knows the application program well may prefer to keep the variables, the meaning of which is already known to him; a user trying to understand an unfamiliar application program may prefer to see as few variables as possible. In fact, our experiments have shown that the system must be very flexible in that respect. Thus, our system works as follows. There are three options: no replacement, systematic replacement, and each replacement depending on the user.

This paper is organized as follows. First, we justify our interest in scientific applications written in FORTRAN in Sec. 2. Next, we present in Sec. 3 the two main tasks of our partial evaluator: constant propagation and simplification. In Sec. 4, we specify our partial evaluation as a set of inference rules, and we show how these rules combine constant propagation and simplification rules. Section 5 presents proofs of soundness and completeness of our partial evaluation rules with respect to the dynamic semantics of FORTRAN. Section 6 explains how we implement our partial evaluator and gives some quantitative results. Section 7 presents conclusions and future work.

## 2. Scientific Programming

Many scientific application programs, written in FORTRAN for decades, are still vital in various domains (management of nuclear power plants, telecommunication satellites, etc.), even though more modern languages are used to implement their user interfaces. It is not unusual to spend several months to understand such application programs before being able to maintain them. For example, understanding an application program of 120,000 lines of FORTRAN code took nine months [12]. So, providing the maintainer with a tool, which finds parts of lost code semantics, allows him to reduce this period of adaptation.

### 2.1. *Characteristics*

One of the peculiarities of scientific applications is that the technological level of scientific knowledge (linear systems resolution, turbulence simulaton, etc.) is higher than the knowledge usually necessary for data processing (memory allocation, data

representations). The discrepancy is increased by the widespread use of FORTRAN, which is an old-fashioned language. Furthermore, for large scientific applications at EDF[a], FORTRAN 77 [1], which is quite an old version of the language, is used exclusively to guarantee the portability of the applications between different machines (mainframes, workstations, vector computers).

### 2.2. *General purpose applications*

Our study has highlighted common characteristics in FORTRAN programming at EDF. These scientific applications have been developed a decade ago. During their evolution, they had to be reusable in new various contexts. For example, the same thermohydraulic code implements both general design surveys for a nuclear power plant component (core, reactor, steam generator, etc.) and subsequent improvements in electricity production models. The result of this encapsulation of several models in a single large application domain increases program complexity, and thus amplifies the lack of structures in the FORTRAN programming language.

This generality is implemented by FORTRAN input variables whose value does not vary in the context of the given application. We distinguish two classes of such variables:

- *data about geometry*, which describes the modeled domain. They appear most frequently in assignment statements (equations that model the problem).
- *data taking a small number of values*, which are either *filters* necessary to switch the computation in terms of the context (modeled domain), or *tags* allowing us to minimize risks due to precision error in the output values. They appear in particular in conditions of alternatives or loops.

Figure 1 shows an example of program reduction. The code section in Fig. 1(a) is extracted from one of the application programs we have studied [20]. The partial evaluation of this code section according to the reduction criteria in Fig. 1(b) yields the code section in Fig. 1(c). In order to show how the reduced code has been obtained from the initial one, some links between both codes are shown. The initial code which is left unchanged in the simplified code is italicized. Expressions which have been replaced by their value and which appear in the reduced code are written in bold. The rest of the initial code is the code that has been removed in the reduced code. When using the tool, such links can be displayed in different colors.

A maintenance team is used to update a specific version of the application. These people know some filter properties (IC = 0 and IREX = 1) as well as data about geometry (DXLU = 0.5). Furthermore, IM is a tag whose value is 20. The knowledge of these values of input variables reduces the code (as shown in Fig. 1(c)). Because of the truth of the relation IREX = 1, two alternatives are simplified (1). The first alternative is simplified to its then-branch. In this branch, the variable

[a] EDF is the national French company that provides and distributes electricity to the whole country.

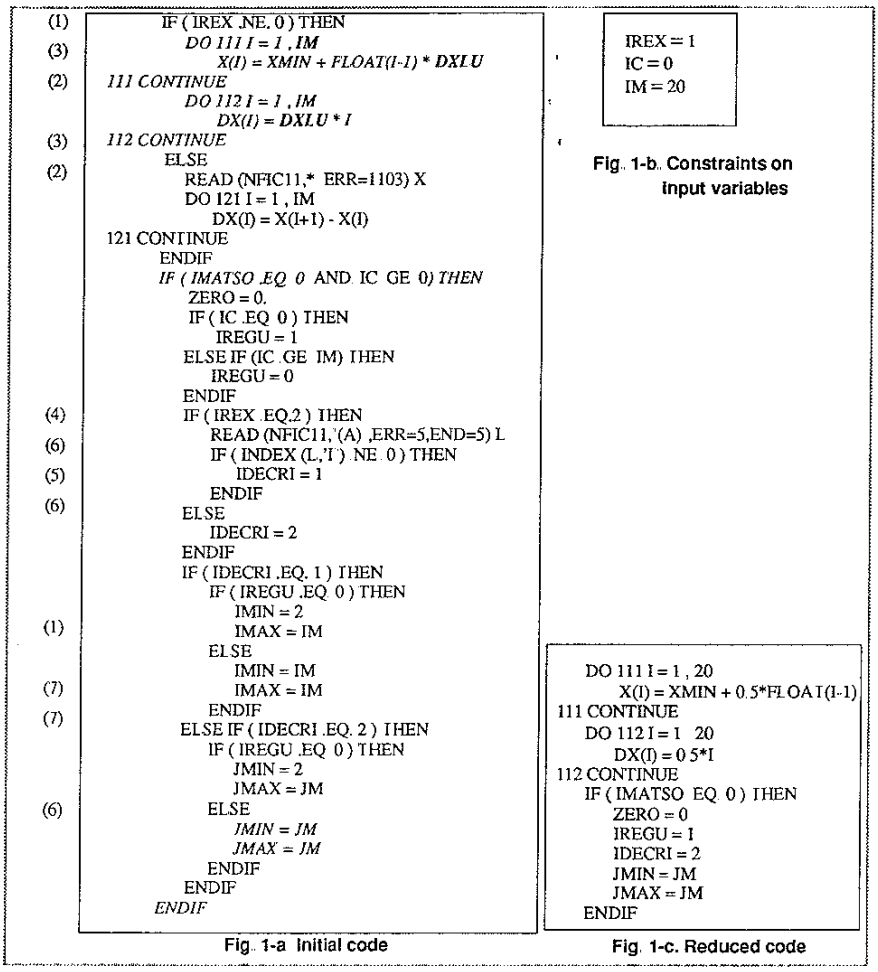



**Fig. 1.** An example of program reduction

DXLU is replaced by its value, which modifies the variables X(I) and DX(I) (2) The variable IM is also replaced by its value (3).

The condition of the next alternative is simplified (4) since the value of IC is known Furthermore, the relation IC = 0 simplifies the following alternative to its then-branch (5), which allows us to compute the value of the variables ZERO, IREGU, and IDECRI in the branch (6). Because the values of these three variables are constant values, the three corresponding assignments are removed from the code only if all possible uses of these occurrences of variables do not use another value of the variables (as discussed later) Then other alternatives are simplified (7).

## 3. Two Aspects of Partial Evaluation Applied to Imperative Programs

Our partial evaluator performs two main tasks: constant propagation through the code and simplification of statements. The tool can give the result of one of the two independently of the other. As we mentioned in the introduction, the simplification is up to the user. For instance, if he is a physicist who is familiar with the equations implemented in the code, he may wish to locate in FORTRAN statements these equations and their variables as they appear in the formulae of these equations (with the removal of unused statements but without the replacement of variables by their values). But if the user is a maintainer who does not know the application well, he would rather visualize the code simplified as much as possible. However, for optimal partial evaluation, the tool performs both tasks.

### 3.1. *Constant propagation*

Constant propagation is a well-known global flow analysis technique used by compilers. It aims at discovering values that are constant on all possible executions of a program and to propagate forward these constant values as far as possible through the program. Some algorithms now exist to perform fast and powerful constant propagation [25].

We describe in this section our constant propagation process, a version derived from [25]. It modifies most expressions by replacing some variable occurrences by their values and by normalizing all expressions through symbolic computation. Presently, our tool propagates only *equalities (and some specific inequalities) between variables and constants*. Of course, that limits the results of the analysis.

***Substitution.*** Before running the partial evaluator, the user specifies numerical values for some input variables of the program (based on his personal knowledge of the application). Constant propagation spreads this initial knowledge supplied by the user. In the first stage, the partial evaluator replaces each specified variable by its value. Then, expressions whose operands are all constant values are evaluated and these resulting values are propagated forward through the whole program. This technique allows us to remove from the code all occurrences of variable identifiers that are no longer meaningful. The substituted values may be displayed differently from other values (in bold as in the previous example or in a different color).

Furthermore, the user can specify some variables for which the substitution will not be displayed. For instance, he can indicate that the variable PI will not be replaced by 3.1416 (of course, that value will nevertheless be used to possibly simplify statements).

***Normalization.*** For any given numerical expression, we have to recognize if it reduces to a constant value (e.g., $x + 3 - x$ reduces to 3). In the same way, for any given logical expression, we have to recognize if it reduces to a conjunction of equalities or to a disjunction of inequalities. In the first case (respectively the second case), we will be able to propagate equalities in the then- (respectively else-) branch of alternatives. For example, if the condition of an alternative reduces to

($x = 1$ AND $y = 4$), it is propagated in the then-branch of the alternative. To do this, we perform constant propagation. To propagate constant values, our system normalizes each expression into a canonical form: a polynomial form for numerical expressions and a conjunctive normal form for logical expressions.

In a polynomial form, expressions are simplified by computing the values of the coefficients of the polynomial. Polynomial forms are written according to the decreasing powers order. When some terms of a polynomial have the same degree (e.g., $z^2$, $x_3^2$, and $t * u$), they are sorted according to the lexicographical order (e.g., $t * u < x^2 < z^2$). The canonical form of a relational expression is obtained from the canonical forms of its two numerical subexpressions. In normalized relational expressions, all variables and values occur on only one side of the operator (e.g., $x = y + 5$ is normalized to $x - y - 5 = 0$).

Because of these modifications of expressions, overflow, underflow, or round-off errors may happen. Therefore, the normalization of expressions may cause run-time errors. Conversely, some run-time errors may vanish thanks to the partial evaluation. As do most partial evaluation systems [18], our tool ignores such problems. In this regard alone, the tool does not yield a program which behaves like the initial one.

### 3.2. *Simplification*

Simplification is the second task of the partial evaluator. First, the expressions are simplified during the propagation as explained above. Then, the partial evaluator reduces the size of the code by removing statements which are never used for the specified values. This simplification includes the elimination of redundant tests and, in particular, the simplification of alternatives to one case thanks to the evaluation of their conditions. To simplify a statement means to remove it or to simplify its components. This section defines the simplification for each statement.

A *write* statement is simplified by simplifying those of its parameters that are expressions. A *read* statement is simplified by removing parameters whose values are known input values, and by adding an assignment for each parameter before the read statement. If all its parameters have known values, the read statement is removed.

An *assignment* simplification consists of simplifying the assigned expression.

An *alternative* is reduced to one of its branches when its condition has been evaluated to either true or false, and the branch is simplified. Otherwise, the statements of both branches are simplified. In this case a branch may become the empty statement.

*Loops* that are never entered are removed. When infinite loops are discovered, a comment is added at the beginning of the loop and its body is simplified. The only statements of a loop which are simplified by the knowledge of variable values are those statements whose expressions are invariant. Thus, we do not expand loops because we want to keep the original structure of the code. Since some FORTRAN

77 loops are implemented using labels and "go to" statements, when such a loop is removed, its label statement (at the entry of the loop) is kept while other statements contain "go to" statements to such labels (the label is left unchanged and the statement is replaced by an empty statement)

A *call* statement simplification consists of adding an assignment for each actual parameter of the call before the call statement. The identifier of the called subroutine is left unchanged in the current program (subroutines are not necessarily specialized) and the user has to run the partial evaluator on this subroutine code if it is also to be simplified. In this case, the tool provides him with the list of variables whose value is known at the entry point of the call.

The partial evaluation does not simplify other statements. This subset of FORTRAN being general enough, such a partial evaluation could be easily applied to other imperative languages. Note that our tool is as yet incapable of dealing with either inequalities (except for those appearing in conditions of alternatives) or literal values.

## 4. Inference Rules for Partial Evaluation

To specify the partial evaluation, we use inference rules operating on the FORTRAN abstract syntax and expressed in the natural semantics formalism [16], augmented with some VDM [14] operators. This section first presents rules defining both the constant propagation process and the simplification process. Then, it details the rules for partial evaluation of statements. These new rules combine the propagation rules and the simplification rules. Note that these techniques are not new, but we specify and use them in a novel way.

### 4.1. *Propagation and simplification rules*

In the following, we use sequents such as $H \overset{propag}{\vdash} I : H'$ (propagation), $H \overset{simpl}{\vdash} I \rightarrow I'$ (simplification), and a combination of both $H \overset{PE}{\vdash} I \longrightarrow I', H'$ (propagation and simplification). In these sequents:

- H is the environment associating values to variables whose values are known before executing I. It is modeled by a VDM-like map [14], shown as a collection of pairs contained in set braces such as {variable → constant, ...}, where no two pairs have the same first elements. Our system initializes such maps by the list of variables and their initial values, supplied by the user.
- I is a FORTRAN statement (expressed in a linear form of the FORTRAN abstract syntax)
- I' is the simplified statement under the hypothesis H.
- H' is H which has been modified by the execution of I.
- The superscript of the turnstile, such as *propag*, *simpl*, or *PE* denotes the set of rules the sequent belongs to.

In the sequents, we use the map operators *dom*, $\cup$, $\cap$, $=$, $\dagger$, $\lhd$, and $\lhd\!\!\!-$.

- The domain operator *dom* yields the set of first elements of the pairs in the map
- The union operator $\cup$ yields the union of maps whose domains are disjoint (in VDM, this operator is undefined if the domains overlap).
- The intersection operator $\cap$ of two maps yields the pair common to both maps.
- The equality operator $=$ of two maps yields true if and only if each pair of one map is a pair of the other map (and reciprocally)
- The map override operator $\dagger$ whose operands are two maps, yields a map which contains all of the pairs from the second map and those pairs of the first map whose first elements are not in the domain of the second map.
- The map restriction operator $\lhd$ is defined with a first operand which is a set and a second operand which is a map; the result is all of those pairs in the map whose first elements are in the set.
- When applied to a set and a map, the map deletion operator $\lhd$ yields those pairs in the map whose first elements are not in the set

The examples in Fig. 2 illustrate these definitions of map operators

| | |
|---|---|
| $m = \{X \to 5, B \to true\}$ | $dom(m) = \{X, B\}$ |
| | $m \cup \{Y \to 7\} = \{Y \to 7, X \to 5, B \to true\}$ |
| | $\{X, Z\} \lhd m = \{X \to 5\}$ |
| | $\{B\} \lhd m = \{X \to 5\}$ |
| | $m \cap \{X \to 5, B \to false\} = \{X \to 5\}$ |
| $n = \{C \to false, X \to 8\}$ | $m \dagger n = \{X \to 8, B \to true, C \to false\}$ |
| | $n \dagger m = \{X \to 5, B \to true, C \to false\}$ |

**Fig. 2.** Some map operators

We have written some rules to explain how sequents are obtained from other sequents. A rule has two parts, separated by a horizontal bar. Above the bar is a set of sequents, which are the premises of the rule. This set may be empty (in which case no bar is drawn). Below the bar is a single sequent, the conclusion of the rule. If the premises hold, then the conclusion holds. The subscript on a turnstile is omitted when the type of the sequent is evident from the context.

The rules we present in Fig. 3 express the simplification of logical or numerical expressions. They belong to the *eval* system, which is a subsystem of the simplification system *simpl*. The first rule has no premise. It specifies that a variable $X$ which belongs to the environment is simplified into a constant which is equal to its value $C$. Otherwise (second rule), the variable is not modified

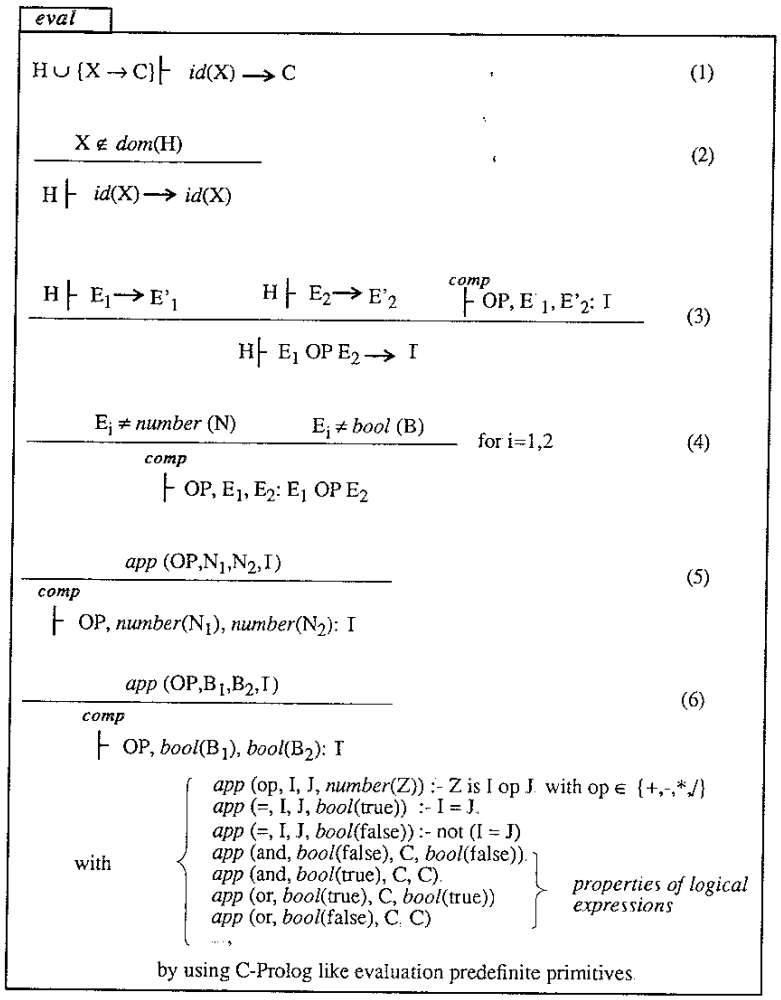



**Fig. 3.** Simplification of expressions

To evaluate an expression $E_1$ OP $E_2$ to the value $T$, its two operands $E_1$ and $E_2$ must have been evaluated to $E'_1$ and $E'_2$ respectively, and the value $T$ is the result of the computation of $E'_1$ OP $E'_2$ (through the *comp* system that performs numerical computation) (third rule). If $E'_1$ and $E'_2$ are both constants (respectively $N_1$ and $N_2$), the computation of $T$ is processed by the application of the *app* primitive to the operator OP and to its two operands $N_1$ and $N_2$ (fifth and sixth rules). Otherwise, the result of the computation is $E'_1$ OP $E'_2$ (fourth rule). Some classical properties of logical expressions have been enclosed in the *app* primitive to perform a deeper evaluation of expressions. The *app* primitive presumes correct typing of programs.

The rule in Fig 4(a) expresses the propagation through a sequence of statements. To propagate the environment H through the sequence of statements $I_1$; $I_2$, H is propagated through the statement $I_1$, which updates H to $H_1$. This new environment $H_1$ is propagated through $I_2$, which updates $H_1$ to $H_2$.

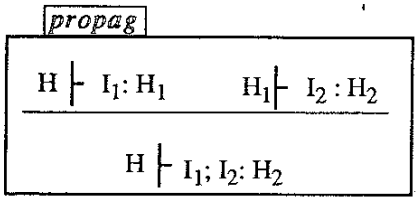



**Fig. 4(a).** Propagation through a sequence of statements

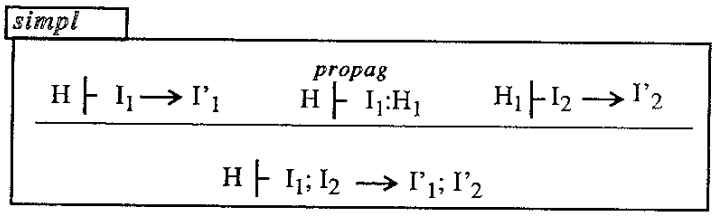



**Fig. 4(b).** Simplification of a sequence of statements

The rule in Fig. 4(b) expresses the simplification of such a sequence. Given an environment H, to simplify a sequence of statements $I_1$; $I_2$, the first statement $I_1$ is simplified to $I'_1$, and the environment H is propagated through $I_1$. In this new environment $H_1$, the second statement $I_2$ is then simplified to $I'_2$. $I'_1$; $I'_2$ is the simplified sequence of statements.

Figure 5 presents six of the eight simplification and propagation rules for alternatives. If the condition $C$ of an alternative evaluates to true, then:

- the environment H' resulting from the propagation of H through the alternative is obtained by propagating H through the statements $I_1$ of the then-branch (first rule: propagation),
- the simplification of the alternative is the simplification of its then-branch (second rule: simplification).

In the same way, there are two rules for an alternative whose condition evaluates to false (in these rules "true" becomes "false", "$I_1$" becomes "$I_2$", and "then" becomes "else"). Since these rules are very similar to the first two rules, they do not appear in Fig. 5. They are shown with partial evaluation rules in Fig. 8.

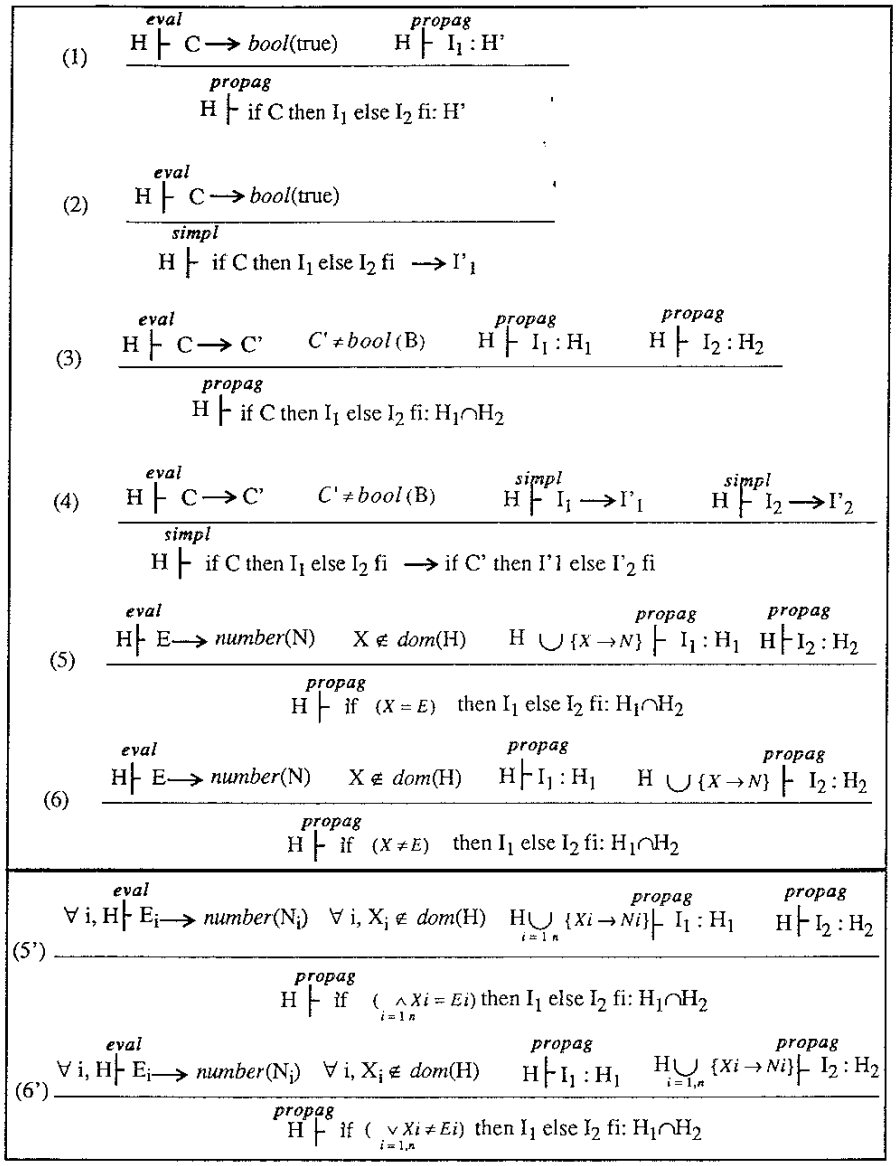



**Fig. 5.** Propagation and simplification rules for alternatives

If the condition $C$ of an alternative is only partially evaluated to $C'$, the propagation and simplification proceed along both branches of the alternative:

- the propagation of H through the then-branch $I_1$ (respectively the else-branch $I_2$) leads to an environment $H_1$ (respectively $H_2$). The intersection of both en-

vironments is the final environment: If a variable has the same value in both environments $H_1$ and $H_2$, that value is kept in the final environment, otherwise it is removed from the final environment (third rule: propagation).

- the simplification of the alternative yields the alternative whose condition is the partially evaluated condition $C'$ and whose branches are the simplified branches of the initial alternative (fourth rule: simplification)

The fifth rule in Fig. 5 is a propagation rule. It shows that information can sometimes be derived from the equality tests that control alternatives. If the condition of an alternative is expressed as an equality such as $X = E$, where X is a variable that does not belong to the domain of the environment H and E evaluates to a constant N, then the pair $(X, N)$ is added to the environment related to the then-branch.

Since the statements "*if* $X \neq E$ *then* $I_1$ *else* $I_2$ *fi*" and "*if* $X = E$ *then* $I_2$ *else* $I_1$ *fi*" are semantically equivalent, there is a corresponding rule (sixth rule) for a condition of an alternative expressed as an inequality such as $X \neq E$. In that case, the pair $(X, N)$ is added to the environment related to the else-branch. Rules 5 and 6 express that only equalities between variables and constants can be added to the environment. Thus, if other information is expressed in the condition, it is not taken into account by the partial evaluator.

Rules 5 and 6 have been generalized to conditions of alternatives expressed as conjunctions of equalities and disjunctions of inequalities (rules 5' and 6') In these rules, we have used generalized AND (noted $\underset{i=1,n}{\wedge}$ and OR (noted $\underset{i=1,n}{\vee}$).

By adding the unfolding of loops, the dynamic semantics would be a special case for our system: If the initial environment associates values to all input variables, the final environment would give (among others) the values of all output variables, with the executable part of the program simplified to the empty statement. Thus, the very special use of the implementation of our system is to see it as a standard interpreter.

Since the simplification is performed in the context of the propagation, and the propagation uses the simplification of expressions, we have chosen to combine propagation and simplification in our rules.

### 4.2. *Combined Rules*

For every FORTRAN statement, we have written rules that describe the combination of the propagation and simplification systems The combination $\overset{PE}{\longrightarrow}$ of these two systems is defined by:

$$H \vdash I \overset{PE}{\longrightarrow} I', H' \quad \text{iff} \quad H \overset{propag}{\vdash} I : H' \quad \text{and} \quad H \overset{simpl}{\vdash} I \rightarrow I'$$

From this rule, we may define inductively the $\overset{PE}{\longrightarrow}$ relation. For instance, Fig. 6 shows the rule for a sequence of statements. A sequence is evaluated from left to right: The partial evaluation of a sequence of two statements $I_1$ and $I_2$ consists

of simplifying $I_1$ to $I'_1$ and then updating (adding, deleting, or modifying) the environment H. In this new environment $H_1$, $I_2$ is simplified to $I'_2$ and $H_1$ is modified to $H_2$.

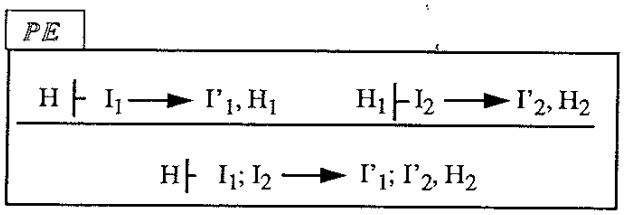



**Fig. 6.** Partial evaluation of a sequence of statements.

Figure 7 specifies the rules for partial evaluation of assignments. The *eval* notation refers to the formal system of rules which simplifies the expressions that we have previously presented in Fig. 3.

If the expression E evaluates to a numerical constant N, the environment H is modified: the value of X is N whether X already has a value in H or not. With the kind of propagation performed, the assignment X := E can be removed only if all possible uses of that occurrence of X do not use another value of X. For instance, in the sequence

$$X := 2;\ \text{if CODE} \neq 5 \text{ then } X := X + 1 \text{ fi};\ Y := X,$$

the value 2 of X is propagated in the expression X + 1, but the assignment X := 2 cannot be removed because in the assignment Y := X, X comes from either from X := 2 (value 2) or from X := X + 1 (value 3). Thus, that sequence is only simplified into X := 2; if CODE ≠ 5 then X := 3 fi; Y := X.

To eliminate assignments that become useless after the partial evaluation, we use classical dead code elimination algorithms [2]. Thus, elimination of redundant assignments is performed in a separate optimization phase.

If E is only partially evaluated to E', the expression E is modified as part of the assignment X := E and the variable E is removed from the environment if it was in it, because its value has become unknown.

The following examples illustrate these two cases. In *Ex. 1*, as the value of the variable A is known, the new value of the assigned variable C is introduced into the environment. We suppose that the assignment C := A + 1 can be removed from the reduced program. In *Ex. 2*, after the partial evaluation of the expression A + B, the value of C becomes unknown. Such a case only happens when A and B do not have both constant values.

*Ex. 1* $\{A \to 1, C \to 4\} \vdash C := A + 1 \longrightarrow \text{skip}, \{A \to 1, C \to 2\}$

*Ex. 2* $\{A \to 1, C \to 2\} \vdash C := A + B \longrightarrow C := 1 + B, \{A \to 1\}$

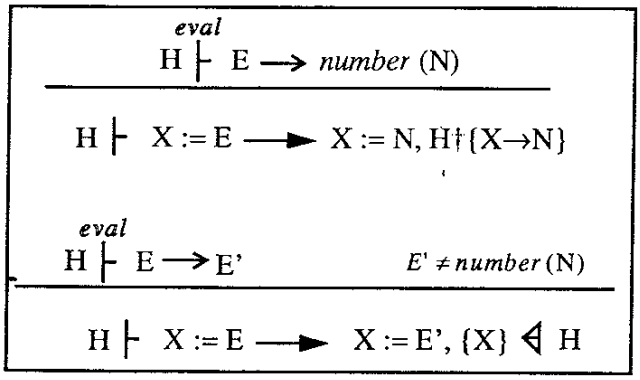



**Fig. 7.** Partial evaluation of assignments

The rules for partial evaluation of alternatives are defined in Fig. 8. If the condition C of an alternative evaluates to a logical constant, this alternative can be simplified to the corresponding simplified branch. If C is only partially evaluated to C', the partial evaluation proceeds along both branches of the alternative, and the final environment is the intersection of the two environments resulting from the simplification of both branches (as explained above).

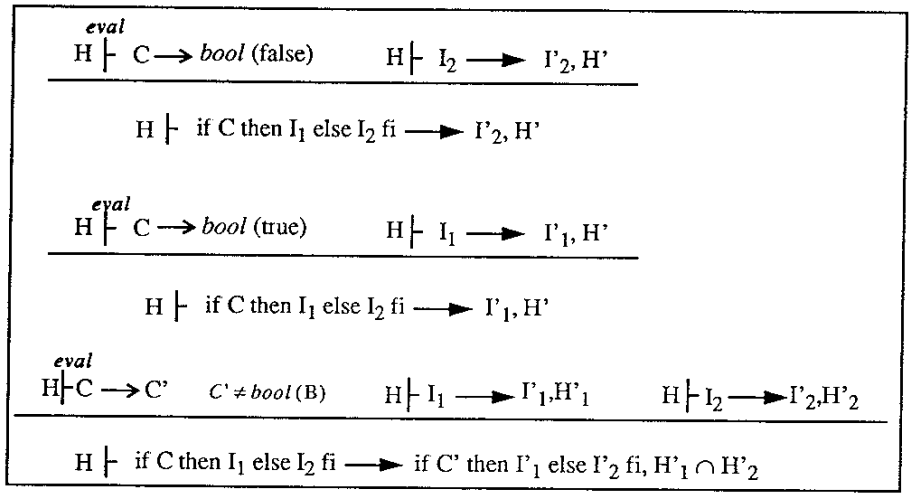



**Fig. 8.** Partial evaluation of alternatives

As for alternatives, the rules for partial evaluation of loops, presented in Fig. 9, depend on the ability to evaluate the truth or falsity of the condition C in the current environment H. The first rule specifies that if the loop is not entered, it is removed from the code. There is no specific rule for the case where C evaluates to true, because we do not expand loops.

In the second rule, if C evaluates to C' (and C' differs from false), the statements I of the loop can be simplified, given H is restricted to a loop invariant Inv(I). Inv(I) is a pessimistic estimation of the variables that are not modified by the loop body. It is calculated by the partial evaluator and consists of a list of variables whose values are neither on the left-hand side of an assignment, nor a modified parameter of a call or a parameter of a read statement. Thus, performing the propagation through I will not modify that restricted environment.

Some infinite loops are treated by the second rule: a loop is infinite when $\mathrm{Inv(I)} \lhd \mathrm{H} \overset{eval}{\vdash} \mathrm{C} \rightarrow \mathrm{bool(true)}$; in this case a comment is added at the beginning of the loop. Figure 9 shows the rules for while-statements, but similar rules exist for repeat-statements.

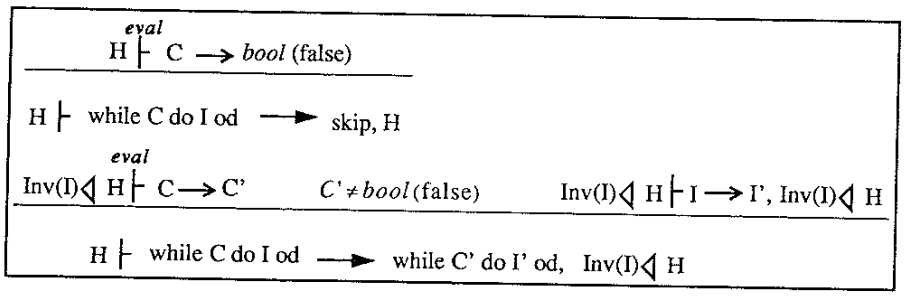



**Fig. 9.** Partial evaluation of while-loops

Some empty statements may be located in an initial program, for example, in an alternative with no then-branch. Figure 10 expresses that the partial evolution of an empty statement never affects the current environment H.

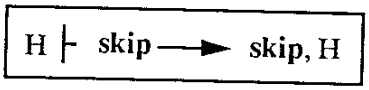



**Fig. 10.** Partial evaluation of an empty statement

Some partial evaluation rules have not been presented in this section. These are either rules allowing us to reach statements from an initial program (containing

other syntactical objects we do not analyze, such as data declarations) or rules performing the normalization of expressions using mainly associativity, distributivity, and reduction to the common denominator.

## 5. Soundness and Completeness of the Partial Evaluation

Our aim in this section is to show how to prove that the simplification presented above is correct with respect to the dynamic semantics of FORTRAN, given in the natural semantics formalism. We recall that partial evaluation of a program P with respect to input variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ for the values $x_1 = c_1, \ldots, x_m = c_m$ must give a residual program $P'$, whose input variables are $y_1, \ldots, y_n$ and the executions of $P(c_1, \ldots, c_m, y_1, \ldots, y_n)$ and $P'(y_1, \ldots, y_n)$ produce exactly the same results.

We will show that this is expressed by two inference rules, one expressing soundness (each result of $P'$ is correct with respect to $P$) and one expressing completeness (each correct result is computed by $P'$ too). As $P$ and $P'$ are deterministic, we could have only one rule using equality, but the demonstration of our two rules is less complicated and more general (being also applicable for nondeterministic programs). Examples of proofs for some FORTRAN statements are detailed in this section.

### 5.1. *Rules proving soundness and completeness*

To prove the simplification, we need a formal dynamic semantics of FORTRAN and we must prove the soundness and completeness of the simplification rules with respect to that dynamic semantics.

To express the dynamic semantics of FORTRAN, we use the same formalism (natural semantics [16]) as for simplification. Natural semantics has its origin in the work of Plotkin [13, 21]. Under the name "structured operational semantics", he gives inference rules as a direct formalization of an intuitive operational semantics: His rules define inductively the transitions of an abstract interpreter. Natural semantics extends that work by applying the same idea (use of a formal system) to different kinds of semantic analysis (not only interpretation, but also typing, translation, etc.). Thus, the semantic rules we give have to generate theorems of the form

$$\mathrm{H} \overset{sem}{\vdash} \mathrm{I} : \mathrm{H'},$$

meaning that, in environment H, the execution of statement I leads to the environment H' (or the evaluation of expression I gives value H'). These rules are themselves not proved: they are supposed to define ex nihilo the semantics of FORTRAN, as Plotkin [21] and Kahn [16] did for other languages like ML.

To prove these rules would mean having another formal semantics (e.g., a denotational one) and to prove that the rules are sound and complete with respect to it. But there is no official semantics for FORTRAN. Thus, that proof would have to be proof of consistency between two dynamic semantics we give. That is outside

the scope of our work. We want to prove consistency between simplification and dynamic semantics, not between two dynamic semantics.

Now how can we prove that the simplification system is sound and complete with respect to the dynamic semantics system? Instead of the usual situation, that is a formal system and an intended model, we have two formal systems: the simplification system (noted *PE*) and the dynamic semantics system (noted *sem*) A program P is simplified to P' under hypothesis $H_0$ on some input variables if and only if

$$H_0 \vdash P \rightarrow P'$$

is a theorem of the simplification system

Let us call H the environment containing the values of the remaining input variables Thus, $H_0 \cup H$ is the environment containing the values of all input variables With that initial environment, P' (respectively P) evaluates to H' if and only if

$$H_0 \cup H \overset{sem}{\vdash} P'\text{: } H' \text{ (respectively } H_0 \cup H \overset{sem}{\vdash} P\text{:}H')$$

is a theorem of the dynamic semantics (*sem*) system.

Now, soundness of simplification with respect to dynamic semantics means that each result computed by the residual program is computed by the initial program That is, for each P, P', $H_0$, H, H', if P is simplified to P' under hypothesis $H_0$ and P' executes to H' under hypothesis $H_0 \cup H$, then P executes to H' under hypothesis $H_0 \cup H$. Thus, soundness of simplification with respect to dynamic semantics is formally expressed by the first rule in Fig 11.

Completeness of simplification with respect to dynamic semantics means that each result computed by the initial program P is computed by the residual program P'. Thus, it is expressed by the second inference rule in Fig. 11. In fact, our approach to prove simplification is very close to the approach of [8] to prove the correctness of translators. In that paper, dynamic semantics and translation are both given by formal systems, and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules (that are proved by induction on the length of the proof; here we will use rule induction instead)

Note that both rules are not the most restricting rules (for instance, their initial environment is $H_0 \cup H$ and not only H, to allow partial simplification).

To prove both rules concerning programs in Fig. 11, we prove that they hold for any statement we partially evaluate (remember that we do not analyze data declarations). Thus, we have to prove that both rules in Fig. 12 hold. In these rules, I denotes a FORTRAN statement and I' denotes the corresponding simplified statement

The dynamic semantics of FORTRAN formalized by the *sem* system is shown in Fig. 13. Some dynamic semantics rules (such as the rules for alternatives) are propagation rules For that reason, in the *sem* system, we have overloaded the

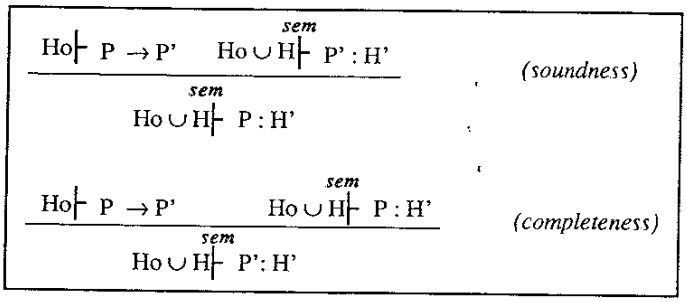



**Fig. 11.** Soundness and completeness of the program simplification

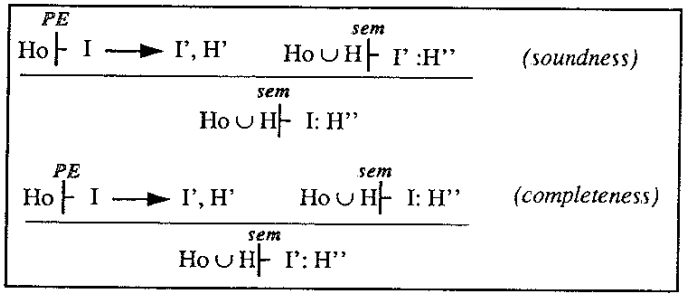



**Fig. 12.** Soundness and completeness of the statements partial evaluation

":" symbol, representing the system *propag*, instead of using a new symbol. The dynamic semantics rules for loop statements differ from the corresponding propagation rules since we do not expand loops while performing the partial evaluation. The interpretation of loop statements is made here by unfolding.

To prove the validity of the completeness and soundness rules, we use rule induction on the partial evaluation and on the dynamic semantics. Indeed, the *PE* and *sem* systems have been defined inductively.

Our inductive hypothesis for soundness is the following property $\Pi_s$, which is defined as follows: $\Pi_s\ (H_0, I, I', H') \Leftrightarrow$

$$(\forall\, H_0,\, I, I', H' \mid H_0 \overset{PE}{\vdash} I \longrightarrow I', H' : (\forall\, H, H''\ H_0 \cup H \overset{sem}{\vdash} I' : H'' \Rightarrow H_0 \cup H \overset{sem}{\vdash} I : H''))$$

The inductive hypothesis $\Pi_c$ for completeness is defined in a similar way. The rule induction on *PE* (respectively *sem*) states that quadruples are only obtained by the rules belonging to the *PE* (respectively *sem*) system.

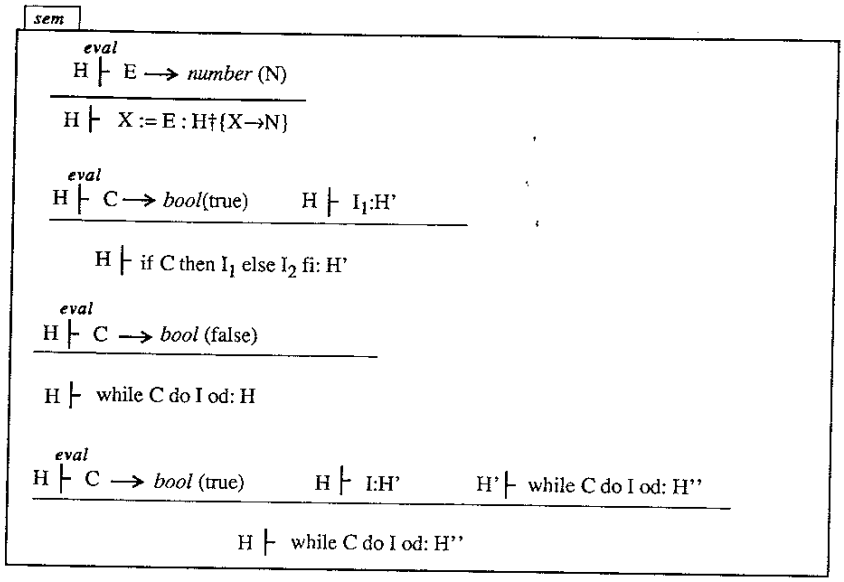
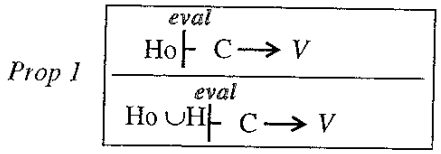



**Fig 13.** Some FORTRAN dynamic semantics rules.

To construct our proof trees we use property *Prop 1*, which states that if some (variable, value) pairs are added to an environment $H_0$, what had already been proved in this environment $H_0$ still holds in the new environment $H_0 \cup H$.




### 5.2. *Examples of proofs of soundness*

The following examples deal only with proofs of soundness. Proofs of completeness are similar. We start with treating simple statements, which are not composed of other statements. They form the basic cases of proof. Figure 14 shows a proof of such a statement. The possible removal of assignment does not appear in the proof tree, since it is performed during a dead code elimination phase, subsequent to the evaluation of the expression of the assignment.

Once simple statements have been proved, we have to prove that the soundness rule holds for composite statements. Figure 15 shows a proof of soundness for an alternative whose condition evaluates to true. There is a similar proof for the case when the condition evaluates to false.

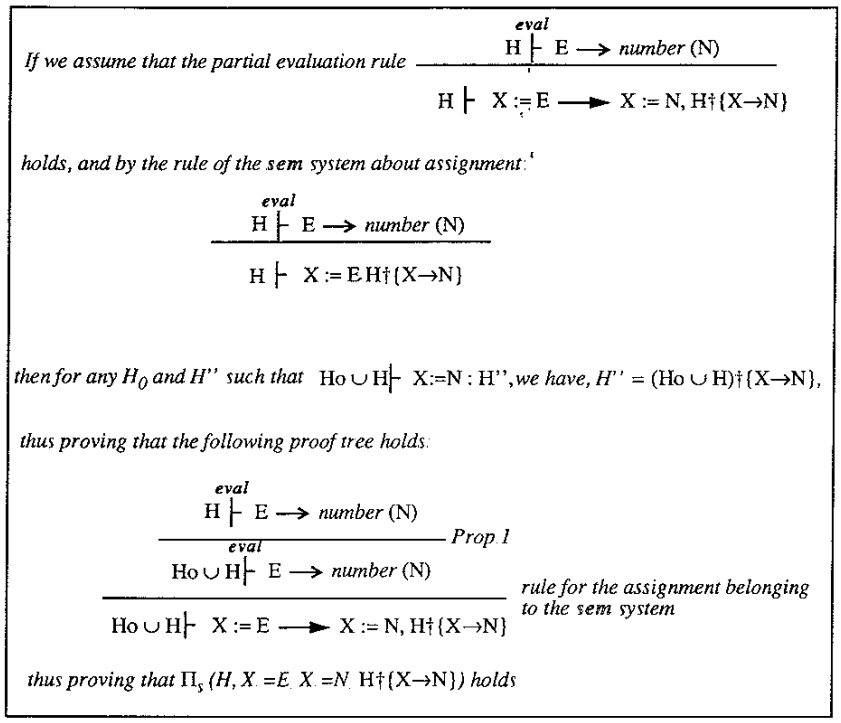



**Fig. 14.** Proof of soundness of an assignment partial evaluation

## 6. Implementation

This section describes the overall architecture of our system. Then, it gives quantitative results.

### 6.1. *Architecture of the partial evaluator*

The partial evaluation rules are very close to the ones we have implemented in the Centaur/FORTRAN environment. The Centaur system [6] is a generic programming environment parameterized by the syntax and semantics of programming languages. When provided with the description of a particular programming language, including its syntax and semantics, Centaur produces a language specific environment. The resulting environment consists of a structured editor, an interpreter/debugger, and other tools, together with a uniform graphical interface. Furthermore, in Centaur, program texts are represented by abstract syntax trees. The textual (or graphical) representation of abstract syntax tree nodes may be specified by pretty-printing rules. Centaur provides a default representation.

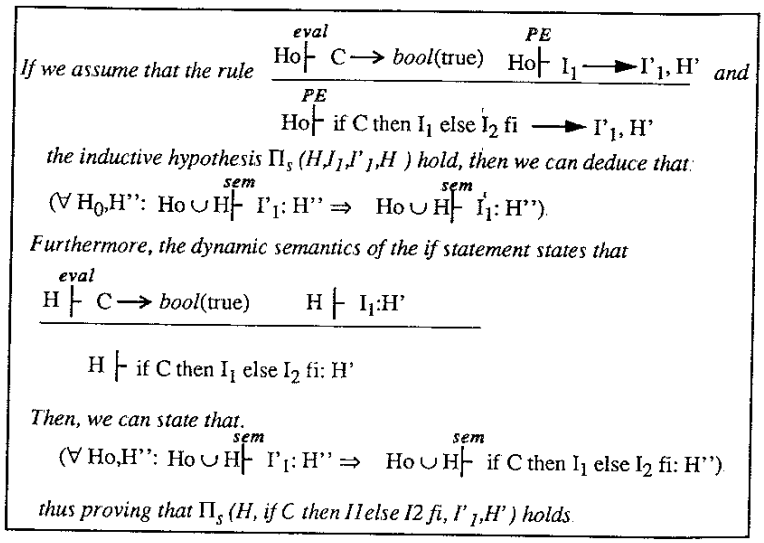



**Fig. 15.** Proof of soundness of an alternative whose condition evaluates to true

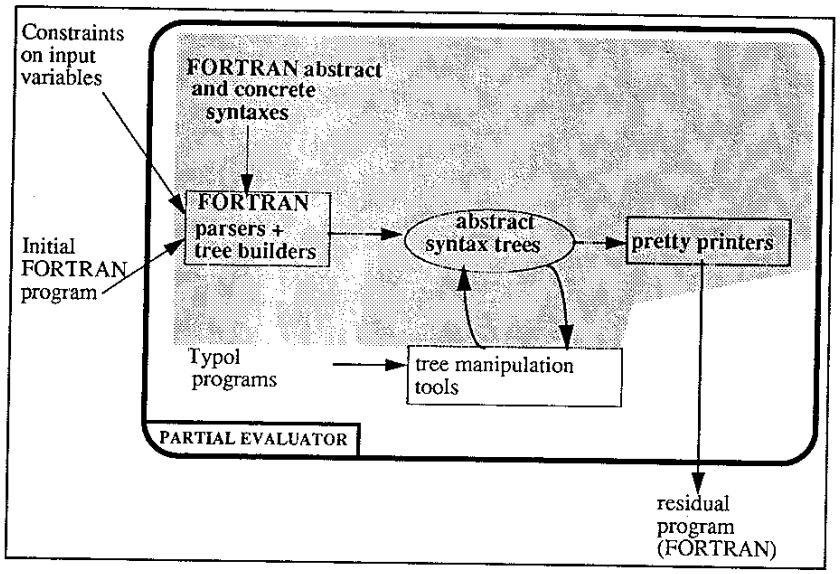



**Fig. 16.** The Centaur/FORTRAN environment

We have used such a resulting environment, Centaur/FORTRAN, to build our partial evaluator. From Centaur/FORTRAN, we have implemented an environment for partial evaluation of FORTRAN programs. Figure 16 shows the overall architecture of this environment, where Centaur/FORTRAN is represented by the grey part.

A language for specifying the semantic aspects of languages called Typol is included in Centaur, so that the system is not restricted to manipulations that are based solely on syntax. Typol is an implementation of natural semantics. It can be used to specify and implement static semantics, dynamic semantics, and translations. Typol specifications are compiled into Prolog code. When executing these specifications, Prolog is used as the engine of the deductive system.

### 6.2. *Quantitative results*

We have written about 200 Typol rules to implement our partial evaluator. Ten rules express how to reach abstract syntax nodes representing simplifiable statements. Ninety rules perform the normalization of expressions. Among the 100 rules for simplification, 60 rules implement the simplification of expressions. The 40 other rules implement the statements simplification. We have written about 25 Prolog predicates to implement the VDM operators we have used to specify the simplification. Thus, these operators are used in Typol rules as in the formal specification of the simplification.

The partial evaluator may analyze any FORTRAN program, but it simplifies only a subset of FORTRAN 77. This subset is a recommended standard at EDF. For instance, it does not analyze any "go to" statement (they are not recommended at EDF), but only "go to" statements that implement specific control structures (e.g., a while-loop).

The average initial length of programs or subroutines we have analyzed is 100 lines of FORTRAN code, which is lengthier than the recommended length at EDF (60–70 lines). The reduction rate amounts from 25% to 80% of lines of code. This reduction is especially important when there is a large number of assignments and conditionals. This is the case for most subroutines implementing mathematical algorithms. For subroutines whose main purpose is editing results or calling other subroutines, the reduction is generally not so important.

## 7. Conclusion

We have used partial evaluation for programs which are difficult to maintain because they are too general. Specialized programs for some values of their input variables are obtained by propagating these constant values (through a normalization of the expressions) and by performing simplifications on the code, for instance, alternatives are reduced to one of their branches. We have shown how to prove by rule induction the completeness and soundness of our formal system for partial evaluation, given the dynamic semantics of FORTRAN.

This technique of partial evaluation helps the maintainer to understand the program behavior in a particular context. Our experiments have given satisfactory results. The residual program is more efficient because many statements and variables have been removed, and no additional statement has been inserted. Another advantage of this technique is that it can also be applied to abstractions at a higher level than the code (e.g., it can be applied to algorithms). Note that the techniques we develop are not new, but we specify (inference rules), implement (Centaur), and use them (for program understanding) in a novel way.

Our partial evaluation system may be used in two ways: by visual display of the residual program as part of the initial program (for documentation or for debugging), or by generating this residual program as an independent (compilable) program. Our system has been tested on several examples and has given satisfactory results.

We are now focusing on the possibility for the user to supply general properties about input variables as in parameterized partial evaluation [10]. These general properties are, for instance, relational expressions composed of some literal values (e.g., $x < z + 4$) instead of only equalities to constant values. We will consequently take into account that kind of information in the conditions of alternatives and loops. We intend to apply linear resolution methods and symbolic manipulation packages for FORTRAN [7] to propagate such properties.

## References


1. *FORTRAN*, ANSI standard X3.9 (1978).
2. A. Aho, R. Sethi, and J. Ullman, *Compilers* (Addison-Wesley, 1986).
3. V. Ambriola, F. Giannotti, D. Pederschi, and F. Turini, "Symbolic semantics and program reduction", *IEEE TOSE* **11**, 8 (1985) 784–794.
4. L. O. Andersen, "C program specialization", Master's Thesis, University of Copenhagen, Denmark (May 1992).
5. A. Berlin and D. Weise, "Compiling scientific code using partial evaluation", *Computer* **23**, 12 (1990) 25–37.
6. *Centaur 1.1 Documentation*, INRIA (January 1990).
7. P. D. Coward, "Symbolic execution systems — a review", *Softw. Eng. J.* (November 1988) 229–239.
8. J. Despeyroux, "Proof of translation in natural semantics", *Symp. Logic in Computer Science*, Cambridge, USA (June 1986).
9. A. P. Ershov and B. N. Ostrovski, "Controlled mixed computation and its application to systematic development of language-oriented parsers", *Program Specification and Transformation, IFIP'87* (1987) pp. 31–48.
10. C. Consel and S. C. Khoo, "Parameterized partial evaluation", *ACM TOPLAS* **15**, 3 (1993) 463–493.
11. K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance", *IEEE TOSE* **17**, 8 (1991) 751–761.
12. M. Haziza, J. F. Voidrot, E. Minor, L. Pofelski, and S. Blazy, "Software maintenance: an analysis of industrial needs and constraints", *IEEE Conf. Software Maintenance*, Orlando, USA (November 1992) pp. 18–26.
13. M. Hennessy, *The Semantics of Programming Languages* (Wiley, 1990).

14. C. B. Jones, *Systematic Software Development Using VDM*, 2nd ed. (Prentice-Hall, 1990).
15. N. D. Jones, P. Sestoft, and H. Sondergaard, "MIX: a self-applicable partial evaluator for experiments in compiler generation", *Lisp and Symbolic Computation* **2** (1989) 9–50.
16. G. Kahn, "Natural semantics", *Proc. STACS'87, Lecture Notes in Computer Science*, Vol. 247 (March 1987).
17. V. Kasyanov, "Transformational approach to program concretization", *Theoretical Computer Science* **90** (1991) 37–46.
18. R. Kemmerer and S. Eckmann, "UNISEX: a UNIX-based symbolic executor for Pascal", *Software Practice and Experience* **15**, 5 (1985) 439–457.
19. U. Meyer, "Techniques for partial evaluation of imperative languages", in *Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, *ACM SIGPLAN Notices* **26**, 9 (1991) pp. 94–105.
20. G. Nicolas *et al.*, "A finite volume approach for 3D two phase flows in tube bundles: the THYC code", Kernforschungszentrum, Karlsruhe, Vol. 2 (1989) pp. 1247–1253.
21. G. Plotkin, "A structural approach to operational semantics", Technical Report DAIMI FN-19, University of Aarhus, Denmark (1981).
22. D. Sahlin, "An automatic partial evaluator for full Prolog", Ph.D. Thesis, SICS, Copenhagen, Denmark (March 1991).
23. T. H. Sneed, "The myth of 'top-down' software development and its consequences", *IEEE Conf. Software Maintenance*, Miami, USA (October 1989) pp. 22–29.
24. H. J. Van Zuylen, "Understanding in reverse engineering", *The REDO Handbook* (Wiley, 1992).
25. M. N. Wegman and K. Zadeck, "Constant propagation with conditional branches", *ACM TOPLAS* **13**, 2 (1991) 181–210.

Automated Software Engineering, 7, 345–376, 2000


# Specifying and Automatically Generating a Specialization Tool for Fortran 90

SANDRINE BLAZY blazy@iie.cnam.fr
*CEDRIC IIE, 18 allee Jean Rostand, 91 025 Evry Cedex, France*

**Abstract.** Partial evaluation is an optimization technique traditionally used in compilation. We have adapted this technique to the understanding of scientific application programs during their maintenance. We have implemented a tool that analyzes Fortran 90 application programs and performs an interprocedural pointer analysis. This paper presents a dynamic semantics of Fortran 90 and manually derives a partial evaluator from this semantics. The tool implementing the specifications is also detailed. The partial evaluator has been implemented in a generic programming environment and a graphical interface has been developed to visualize the information computed during the partial evaluation (values of variables, already analyzed procedures, scope of variables, removed statements, etc.).

**Keywords:** program understanding, partial evaluation, dynamic semantics, formal specification, interprocedural analysis, alias analysis, proof of correctness

## 1. Introduction

A wide range of software maintenance tools analyze existing application programs in order to transform them (Baxter et al., 1998; Sellink and Verhoef, 2000; van den Brand et al., 1996; Yank et al., 1997). Some of these transformations aim at facilitating the understanding of programs. Furthermore, these transformations are based on rather complex analyses. As software maintenance tools, these tools must introduce absolutely no unforeseen changes in programs.

To ensure that the transformation preserves the meaning of programs, we have used formal specifications to develop a software maintenance tool. In our framework, a formal specification yields:

- A basis for expressing precisely which transformations are performed. Formal concepts are powerful enough to clarify concepts of programming languages and to model complex transformations. The formal specification can be seen as a reference document between specifiers and end-users. In our context, end-users are software maintainers who have a strong background in mathematics. Thus, they are disposed to understand our formal specifications.
- A mathematical formalism for proving and validating properties of program transformations.
- A framework for simplifying the implementation of a tool. The formal specification has been refined to obtain an implementation.

Our tool aims at improving the understanding of scientific application programs. Such programs are difficult to maintain mainly because they were developed a few decades ago by experts in physics and mathematics, and they have become very complex due to extensive modifications. For a maintenance team working on a specific application program, one of the most time consuming steps is to extract by hand in the code the statements corresponding to the specific context of the maintenance team. This context is very well known by all the people belonging to the maintenance team; this is their minimum knowledge concerning the data of their application program. This context is described by equalities between specific variables and values (Blazy and Facon, 1998).

### *1.1. Motivations*

Partial evaluation is an optimization technique, also known as program specialization. When given a program and known values of some input data, a partial evaluator produces a so-called residual or specialized program. Running the residual program on the remaining input data will yield the same result as running the original program on all of its input data (Jones et al., 1993). The residual program is more optimized than the original program. Partial evaluation has been applied to generate compilers from interpreters (by partially evaluating the interpreter for a given program). In this context, previous work (ACM, 1998; Danvy et al., 1996) has primarily dealt with functional and declarative languages.

Partial evaluation has also been applied to improve speedups of functional, declarative and imperative programs (Andersen, 1994; Baier et al., 1994; Marlet et al., 1997). Usually, the chief motivation for doing partial evaluation is speed. The residual program is faster than the initial one because expensive calculations have been eliminated. This is done with sophisticated techniques such as binding-time analysis. The basic partial evaluation techniques are procedure inlining (calls to procedures are replaced with copies of their bodies) and loop unfolding (loops are replaced with several copies of their bodies). Both techniques replace statements by faster statements, but they increase the size of the code and generate new variables or rename variables. These partial evaluation techniques conflict with our goal, so we do not use them.

Our goal is to generate programs that are easier to understand. Thus, our transformations do not modifiy the structure of the code. In like manner, our partial evaluator neither generates new variables nor renames variables. Our partial evaluation is based on constant propagation and statements simplification. The programs we generate are easier to understand because many statements and variables have been removed and no additional statement or variable has been inserted. Furthermore, the known values of variables like PI or TAX RATE are propagated during partial evaluation but these variables are likely to be kept in the code (e.g. in the code 2*PI + 1 should be easier to understand than 7.28). The benefit of replacing variables by values depends also on the kind of user (see (Blazy and Facon, 1994) for details about our specialization strategy).

Figure 1 briefly illustrates how source code is specialized with respect to constraints on input variables. The source code that has been removed is striked out (e.g. in figure 1, the first `if` statement is removed and replaced by its specialized `then` branch). The source code that is not striked out corresponds to the specialized code. In the specialized code,

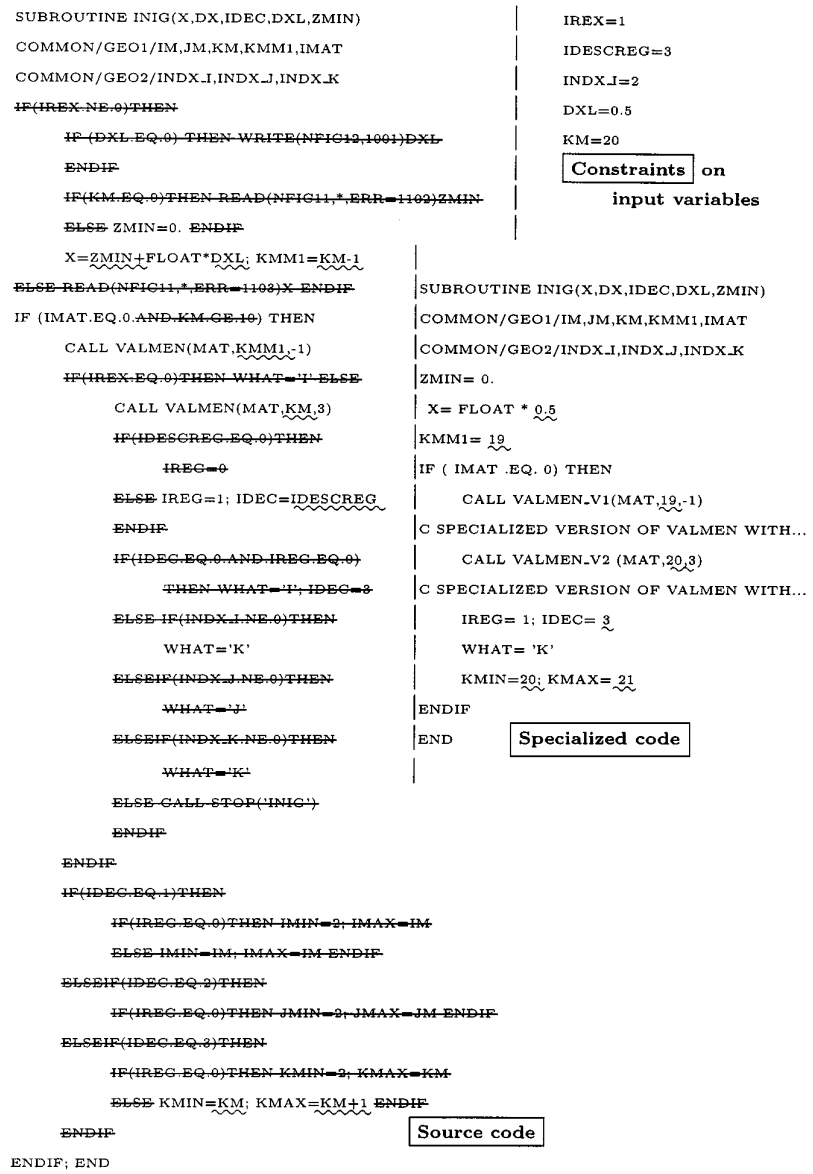



*Figure 1.* An example of program specialization.

expressions are also simplified. In source and specialized codes, simplified expressions are underwaved. Known values of variables are propagated in called procedures. Called procedures have been replaced by their specialized versions and a comment recalls the name of the called procedure and its initial known values. Other information is computed and displayed during the partial evaluation (e.g. final values of some variables). It is not shown in figure 1 to simplify the presentation of the figure.

### *1.2. Technical overview of our interprocedural alias analysis*

Our partial evaluator implements a flow-sensitive and context-sensitive interprocedural alias analysis (Emami et al., 1994; Hasti and Horwitz, 1998; Landi and Ryder, 1992; Liang and Harrold, 1999; Wilson and Lam, 1995). In a flow-sensitive analysis, aliasing information depends on control flow: the analysis takes into account the order in which statements are executed. In a context-sensitive analysis, aliasing information is differentiated among call sites: the analysis takes into account the fact that a function must return to the site of the most recent call.

Our analysis computes *must* alias information (i.e. alias information that must occur on all paths through the flowgraph). For each pointer and each program point, our analysis computes a conservative approximation of the memory locations to which that pointer must point.

Flow-sensitive analyses generally take more time and space than flow-insensitive analyses; however, the results are usually more precise (Hasti and Horwitz, 1998). Our analysis is more sensitive than other alias analysis that are more efficient (Andersen, 1994; Carini and Hind, 1995; Sagiv et al., 1997; Steensgaard, 1996), but our analysis does not take prohibitive time or space. The reason is that our partial evaluation propagates only equalities between variables and values. This means that when an if statement or a loop is analyzed, some information is lost at the end of the analysis of the statement:

- when two branches of an if statement are analyzed, the analysis propagates only the equalities between variables and values that hold in both branches,
- in a similar way, when a loop is analyzed, only the equalities between variables and values that belong to the invariant of the loop are propagated outside the loop.

Our analysis does not support recursive calls (the application programs we have analyzed are not recursive) but it handles return constants. The framework of our analysis is similar to the more general one described by Chase and Zadeck in Chase and Zadeck (1990): for each procedure, information that describes the effects of that procedure are propagated through the call graph. This graph represents the structure of the analyzed program.

### *1.3. A formalism for specifying program transformations*

We use natural semantics rules to specify and implement our relations. These are inference rules that are made of sequents defining inductively our relations. The rules operate on

abstract syntax trees. Each rule expresses how to deduce sequents (the denominator of the rule) from other sequents (the numerator of the rule) (Kahn, 1987). While natural semantics rules provide a simple means for specifying our program transformation, they use very simple data structures for representing programs and other information, and they use only very simple formulas and inference rules for expressing program transformations. Hannan (1993) explains that this simplicity has the advantage of yielding straightforward and efficient implementations of the rules, but it also has the obvious disadvantage of yielding specifications that contain primitive encoding of program transformations.

In our interprocedural alias, we need to specify side-effects on global variables and pointers, and side-effects accomplished through parameter passing. Thus, we have extended the natural semantics formalism. To simplify the presentation of the rules:

- We have used various set and relational operators to specify our program transformations. The computations using these operators are not defined in the rules themselves, but before the rules in a section called "definitions" (e.g. see figure 6).
- The links between nodes of abstract syntax trees (representing pieces of programs) and all other data associated with specifications have been modeled in object diagrams. Instances of the objects are data appearing in the rules, and arrows between objects are functions applied to data appearing in the rules (e.g. see figure 4). Only the instances of some objects (those that consist of other objects) of the diagram occur in the rules. The instances of other objects and the arrows between objects are only defined in the diagrams.

Thus, in this paper, a specification consists of definitions of data, a natural semantics rule (with data defined in the definition part) and an object diagram showing composition links between the data of the rule. As there is no formal description of the semantics of Fortran 90, we have first specified a dynamic semantics of Fortran 90. We have then specified our partial evaluation and this specification is based on the specification of the dynamic semantics.

### *1.4. Overview of the paper*

The aim of this paper is to show how we have specified, proven and implemented our partial evaluator, and especially our interprocedural alias analysis. Compared to our previous work (Blazy and Facon, 1994; 1995; 1996):

- Our program analyses take into account an interprocedural alias analysis.
- The definition of our partial evaluation is based on a definition of the dynamic semantics of Fortran 90.
- We have extended the natural semantics formalism to present higher-level specifications.
- We have implemented a graphical interface.

The rest of this paper is structured as follows. First, Section 2 recalls some concepts of Fortran 90 and gives some definitions that are useful to understand our specifications. Then, Section 3 specifies the dynamic semantics of call statements and pointers. Section 4

specifies the partial evaluation of call statements. It shows how the propagation relation has been derived from the dynamic semantics relation, and it presents the general framework for proving the correctness of the partial evaluation. Section 5 is devoted to the implementation of our tool.

## 2. Background and notations

This section gives a brief overview of the features of Fortran 90: procedures, common blocks, structures, pointers and targets. Then, it defines set and relational opertors that will be used in the specifications.

### *2.1. Fortran 90*

The most interesting aspects of Fortran 90 for partial evaluation are the treatment of parameters, variables and structures, when they are passed between procedures (i.e. subroutines or functions). In Fortran 90, procedures cannot be nested. Parameters may be modified in procedures. Certain side effects are prohibited by the Fortran standard (e.g. the statement `X = F(I) + I` is not allowed if the function `F` changes the value of `I`). Other side effects are permitted, but the result of using them is not defined (i.e. it is processor dependent). Some Fortran implementations use a call-by-value-result semantics. Other implementations use a call-by-reference semantics (Aho et al., 1988). Both semantics are allowed by the Fortran standard, and we have chosen to specify a call-by-value-result semantics.

Fortran 90 has no global storage other than variables in common blocks. Common blocks represent contiguous areas of memory. The variables of common blocks are shared across procedures. Common blocks are unusual for modern programming languages in that they associate variables by location, rather than by name. Although the names of the common blocks themselves are global, none of the variables within the common blocks are global. Common blocks may also be inherited across procedure calls. Given a common block *CB* declared in a procedure $P$, the rule of dynamic scoping for *CB* states that *CB* can be referred to, not only in $P$, but also in any procedure called from within $P$, even if *CB* is not declared in such a procedure. If a common block is neither declared in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block are undefined.

A structure consists of a list of fields, each of some particular type. The field types may include pointers to structures of the type being defined, of a type previously defined, or of a type yet to be defined. A pointer may point to any object that has the `TARGET` attribute, which may be any dynamically allocated object or a named object declared with that attribute. When the statement `NULLIFY(`$p$`)` is executed, the pointer $p$ becomes `NULL`. In Fortran 90, a pointer should be thought of as a variable aliased to another data object where the data is actually stored—the target (ANSI, 1992; Muchnick, 1997). There is no pointer arithmetic in Fortran 90. There is no notation for representing pointed variables (dereferencing is automatic in Fortran 90). We will then use a C-notation when needed for clarity (e.g. *$*(*p.next)$* in figures 2 and 3).

```
TYPE node
   REAL :: ident                  ! data field
   TYPE(node), POINTER :: next    ! pointer field
END TYPE node
 ...
 q => p%next    ! q points to (i.e. is an alias of) *(*p.next)
 p%ident = 3.4  ! the value 3.4 is assigned to the field ident of p
 q%ident = 6.2
```

*Figure 2.* An example of Fortran 90 code.

Figure 2 shows a Fortran 90 program that defines a new type called `node`. The `node` type contains two fields: `ident` containing the value associated with the node, and `next` pointing to the next element of the list. Once two variables $p$ and $q$ of type pointer to `node` have been declared, the values 3.4 and 6.2 are inserted in the list of `nodes` in that order. The insertion in the list modifies the three variables $q$, $p\%ident$ and $q\%ident$.

The syntax of identifiers is specified by the abstract syntax of figure 3. An identifier is either a simple identifier (e.g. $v$), or a compound left-hand side (e.g. ${}^{*}p.next$), or a pointer dereference (e.g. ${}^{*}p$, ${}^{*}({}^{*}p.next)$). The set of simple variables identifiers of a procedure is denoted by *VarName*. The set of left-hand sides is denoted by *Lhs*. Among left-hand sides are simple variables. Thus, $VarName \subseteq Lhs$. The example in this figure shows the connection between some concrete Fortran 90 variables and the corresponding abstract syntax notations.

Abstract syntax

$$VarName \subseteq Lhs$$

$$\mathbf{deref} : Lhs \to Lhs$$
$$\mathbf{field\_lhs} : Lhs \times VarName \to Lhs$$

Examples

| concrete syntax | abstract syntax |
|---|---|
| $v$ | $VarName(v)$ |
| $person\%address\%town$ | $\mathbf{field\_lhs}(\mathbf{field\_lhs}(person, address), town)$ |
| $*((*p).next)$ | $\mathbf{deref}(\mathbf{field\_lhs}(\mathbf{deref}(p), next))$ |

*Figure 3.* Abstract syntax rules and examples of links with concrete syntax.

### 2.2. *Definitions*

Before going into the details of how to specify the dynamic semantics and the partial evaluation, we define in this section some notation, especially set operators, that we use in our specifications. We first introduce useful set operators, similar to those defined in the formal specification languages B (Abrial, 1996) and VDM (Jones, 1990): mainly inverse ($^{-1}$), domain (**dom**), range (**ran**), union ($\cup$), override ($\dagger$), restriction ($\rhd$, $\lhd$ and $\lhd\!\!\!\!-$), relation composition ($;$ [1]) and direct product ($\otimes$). In the following definitions $s$ denotes a set, $r$, $p$ and $q$ denote binary relations, $m$ and $n$ denote maps (specific binary relations where each element is related to at most one element). A binary relation is a set of pairs. Thus, classical set operators such as union can also be applied to binary relations. For each operator that we define, Table 1 shows through an example of how the operator can be used.

- $r^{-1} = \{x \mapsto y \mid y \mapsto x \in r\}$
- $\mathbf{dom}(r) = \{x \mid x \mapsto y \in r\}$
- $\mathbf{ran}(r) = \{y \mid x \mapsto y \in r\}$
- $r \dagger p = \{x \mapsto y \mid x \mapsto y \in p \bigvee (x \mapsto y \in r \bigwedge x \notin \mathbf{dom}(p))\}$
- $r \rhd s = \{x \mapsto y \in r \mid y \in s\}$
- $s \lhd r = \{x \mapsto y \in r \mid x \in s\}$
- $s \lhd\!\!\!\!- r = \{x \mapsto y \in r \mid x \notin s\}$
- $r; p = \{x \mapsto z \mid \exists y \cdot x \mapsto y \in r \bigwedge y \mapsto z \in p\}$
- $r \otimes p = \{x \mapsto (y, z) \mid x \mapsto y \in r \bigwedge x \mapsto \dot{z} \in p\}$

When $r$ and $p$ are functions, their direct product is especially interesting when it is an injective function. In this case, when a pair of the form $x \mapsto (y,\ z)$ belongs to $r \otimes p$ then $(y,\ z)$ determines $x$ uniquely, and $x$ may be written $(r \otimes p)^{-1}(y,\ z)$. The $\otimes$ relation generalizes naturally to more arguments.

Furthermore, we have defined an operator to bind variable names of a common block to their corresponding values. Given s a set of pairs of maps (e.g. $s$ is of the form

*Table 1.* Examples of notations.

| Operator | Example |
|---|---|
| $^{-1}$ | $\{1 \mapsto a, 3 \mapsto u\}^{-1} = \{a \mapsto 1, u \mapsto 3\}$ |
| **dom** | $\mathbf{dom}(\{1 \mapsto a, 3 \mapsto u\}) = \{1, 3\}$ |
| **ran** | $\mathbf{ran}(\{1 \mapsto a, 3 \mapsto u\}) = \{a, u\}$ |
| $\dagger$ | $\{1 \mapsto a, 3 \mapsto u\} \dagger \{1, \mapsto b, 2 \mapsto d\} = \{1 \mapsto b, 2 \mapsto d, 3 \mapsto u\}$ |
| $\rhd$ | $\{1 \mapsto a, 3 \mapsto u\} \rhd \{a, b\} = \{1 \mapsto a\}$ |
| $\lhd$ | $\{1, 2\} \lhd \{1 \mapsto a, 3 \mapsto u\} = \{1, \mapsto a\}$ |
| $\lhd\!\!\!\!-$ | $\{1, 2\} \lhd\!\!\!\!- \{1 \mapsto a, 3 \mapsto u\} = \{3, \mapsto u\}$ |
| ; | $\{1 \mapsto a, 3 \mapsto u\}; \{a, \mapsto 67, b \mapsto 26\} = \{1 \mapsto 67\}$ |
| $\otimes$ | $\{1 \mapsto a, 3 \mapsto u\} \otimes \{1 \mapsto b, 2 \mapsto d, 3 \mapsto z\} = \{1 \mapsto (a, b), 3 \mapsto (u, z)\}$ |

$\{m_1 \mapsto n_1, m_2 \mapsto n_2, m_3 \mapsto n_3, \ldots\}$, where $m_1, n_1, m_2, n_2, m_3, n_3, \ldots$ are maps), we define:

$$\mathbf{Corres}(s) = \bigcup \{m^{-1}; n \mid m \mapsto n \in s\}.$$

Variables of common blocks are shared among procedures (their values are inherited in each called procedure) but their names may change in each procedure. Thus, each pair $m \mapsto n$ of $s$ corresponds to a common block. $m$ and $n$ map integers (the position in the list of declared variables of the common block) to respectively variable names and values. $m^{-1}; n$ is then a map from variable names to values and **Corres**($s$) is the union of all such maps.

In the following example, the declaration of two common blocks `B` and `C` is represented by the map *ComDecl*. At the current program point, the values of variables belonging to common blocks are given by the map *ComVal*. *ComVal* states that:

- the value of the first variable of `B` is `0` and the value of the other variable of `B` is unknown (this variable is not represented in the map *ComVal*),
- the value of the first variable of `C` is 0.5, the value of the second variable of `C` is unknown, and the value of the third variable of `C` is 6.2.

In the current procedure, the variables of `B` are `t` and `u`, and the variables of `C` are `x`, `y` and `z`. Thus, at the current program point, the map from these variables to their values is **Corres**($s$).

*EXAMPLE.*
```
COMMON B / t, u
COMMON C / x, y, z
```

$$\begin{aligned} \mathit{ComDecl} &= \{B \mapsto \{1 \mapsto t, 2 \mapsto u\}, C \mapsto \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\}\} \\ \mathit{ComVal} &= \{B \mapsto \{1 \mapsto 0\}, C \mapsto \{1 \mapsto 0.5, 3 \mapsto 6.2\}\} \\ \text{If } s &= \mathit{ComDecl}^{-1}; \ \mathit{ComVal} \quad \text{then} \end{aligned}$$

$$s = \{\{1 \mapsto t, 2 \mapsto u\} \mapsto \{1 \mapsto 0\}, \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\} \mapsto \{1 \mapsto 0.5, 3 \mapsto 6.2\}\}$$
$$\text{and} \quad \mathbf{Corres}(s) = \{t \mapsto 0, x \mapsto 0.5, z \mapsto 6.2\}$$

## 3. Specification of the dynamic semantics

This section details the specification of the dynamic semantics of two kinds of statements: call statements and assignments between pointers. The dynamic semantics of assignments between targets and pointers is specified in a similar way, but is not presented in this paper. Each specification is presented in two sections: the first section details an object diagram and the second section gives definitions and natural semantics rules. These rules use data that are defined either in the object diagram or in the definitions.

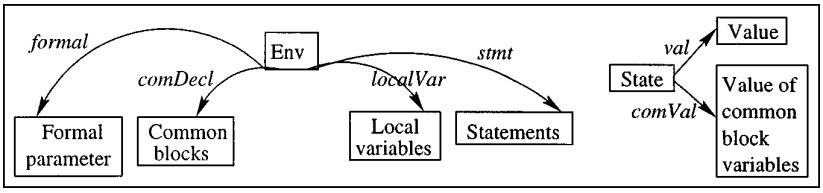



*Figure 4.* The state and the environment of a procedure.

### *3.1. Object diagram representing programs*

The dynamic semantics specifies how procedures are executed. To execute a call statement we execute the body of the called procedure. This means that we have to keep track of the association of procedures 1) with their bodies and also 2) with the values of their variables. Thus, in figure 4, we define two objects for representing a Fortran 90 procedure:

1. an environment (called Env in figure 4), that represents what does not vary during the analysis of the procedure (formal parameters, common block variables that have a scope in the current procedure, local variables and statements),
2. a state (called State in figure 4) modeling relations between variables and values at the current program point. When a procedure is analyzed, each analysis of a statement updates the state of the procedure. An object State (called S) consists of two objects: *val*(S) and *comVal*(S). *val*(S) maps variables to values and *comVal*(S) maps common blocks to the known values of their variables. Since in common blocks, values are shared between two variables simply by the fact that they occupy corresponding positions within the same common block, these values are modeled differently from the values of other variables. In figure 4 these values are represented by the object called "Value of common block variables" that is accessed from State through the *comVal* function.

The diagram of figure 5 extends the diagram of figure 4, to model the whole data that represent procedures. In figure 5, the object called Procedure denotes a called procedure SP

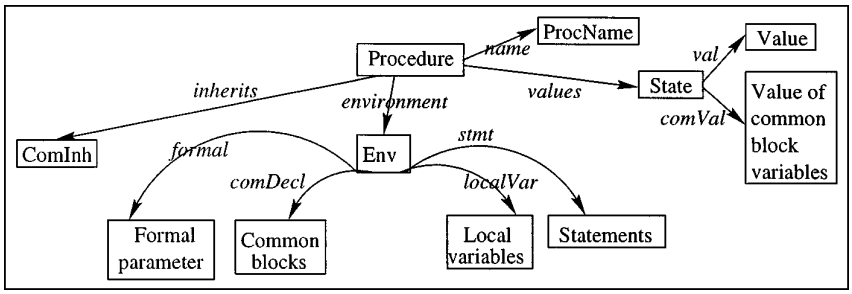



*Figure 5.* Object diagram showing data propagated during an interprocedural analysis.

at a given program point. From SP, we have access to its environment *environment*(SP), its state *values*(SP), but also to its inherited common blocks *inherits*(SP) and its name *name*(SP). This name belongs to ProcName, that denotes the set of procedure names that are used in the current application program.

An instance of the diagram of figure 5 represents information for all procedures that are called from a main program (or from one of its called procedures). It shows the bindings between procedure names and their corresponding statements. The objects of the instance are never modified by the sequents. Thus, the instance is an implicit parameter of the dynamic semantics and partial evaluation systems, to simplify the presentation of the rules.

### *3.2. Natural semantics rule for the dynamic semantics of a call statement*

The dynamic semantics of Fortran 90 is formalized by the *sem* system of inference rules, that generates sequents of the form $\mathsf{E}, \mathsf{S}, \mathsf{Cl} \overset{sem}{\vdash} \mathsf{I} : \mathsf{S}'$, meaning that given an environment E, a state S and inherited common blocks Cl, the execution of statement I leads to the values given by the state S′. E, S and Cl are the hypotheses of the sequent. Figure 6 shows the dynamic semantics rule for a call statement `CALL SP (LParam)`. SP is the name of the called procedure and LParam is a map from positions to the actual parameters of SP at the current program point. The rule specifies a call-by-value-result dynamic semantics.

The beginning of figure 6 illustrates through an example which data are propagated during the execution of statements with respect to the dynamic semantics rule of the figure. In this example, these statements are located at the program points $\alpha$, $\beta$, $\gamma$ and $\delta$. The execution of the call statement starts with a forward propagation through the call graph (from program point $\alpha$ to program point $\gamma$) followed by a backward propagation (from program point $\delta$ to program point $\beta$). The forward propagation aims at giving new values to formal parameters and common blocks variables of the called procedure SP. These values are *val*(S) (values of variables) and *comVal*(S) (values of common blocks). The backward propagation aims at giving new values to the actual parameters of the called procedure and to the variables belonging to common blocks of the caller that have a scope in the called procedure.

In the definitions part of the figure, some definitions are factorized to simplify the presentation of the dynamic semantics rule. They introduce some macros used in specifying the dynamic semantics rule. In the definitions, the $\underline{\Delta}$ symbol means "by definition". The first definition defines EnvSP, the environment of the procedure named SP. $name^{-1}$ yields an object Procedure from an object ProcName (see figure 5). Thus, since SP is a procedure name, $name^{-1}$(SP) represents the procedure whose name is SP and EnvSP is defined as the environment of this procedure.

In the rest of Section 3.2, we describe each of the macros in more detail. To describe the macros, we follow the execution order of the statements. The first five macros compute the values that are transmitted to the called procedure SP. The general idea is that the macros update the initial state S and inherited common blocks C so that when executing the statements of EnvSP, the data declared in SP will be available and the side-effects will be taken into account. The macros detailed in the last subsection compute the values that SP transmits back to the call site that invoked it.

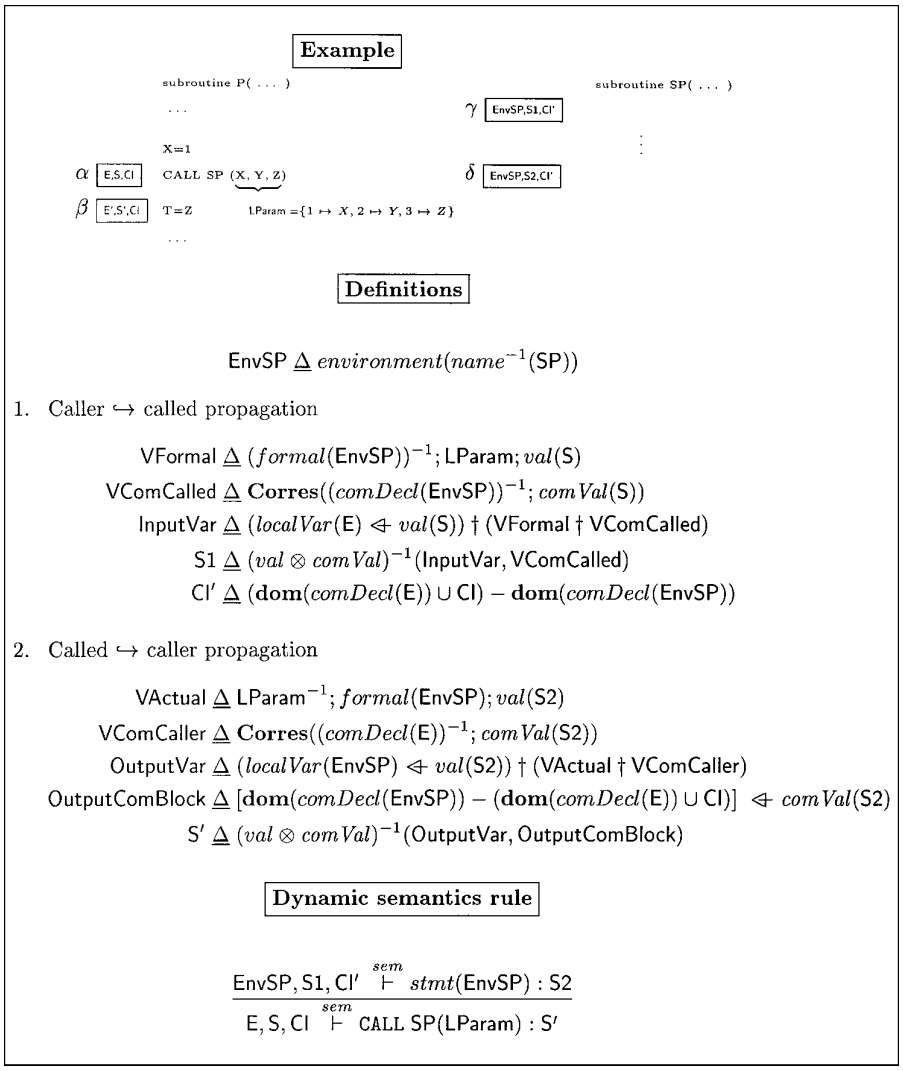



*Figure 6.* Dynamic semantics of a call statement.

***3.2.1. Computation of VFormal (values of formal parameters at program point $\gamma$).*** Formal (resp. actual) parameters are represented by maps between positions and names (resp. values) of the parameters. In figure 6, these maps are respectively *formal*(EnvSP) and LParam.

- ( *formal*(EnvSP))$^{-1}$ maps the names of the formal parameters of SP to their positions,
- LParam; *val*(S) is the result of the evaluation of the actual parameters at the program point $\alpha$. It maps the names of the actual parameters to their current values.

- Thus, $(formal(\mathsf{EnvSP}))^{-1}$; LParam; $val(\mathsf{S})$ maps the names of the formal parameters of SP to their initial values, i.e. the values of actual parameters at the program point $\alpha$.

***3.2.2. Computation of VComCalled (values of common blocks at program point $\gamma$).*** Variables belonging to a common block are represented by a map between positions in the common block and variable names. The values of the variables that belong to the common blocks of the calling procedure are also transmitted to the corresponding variables (they share a same position in the common blocks). Thus,

$$\mathsf{VComCalled} \underline{\triangle} \mathbf{Corres}((comDecl(\mathsf{EnvSP}))^{-1}; comVal(\mathsf{S}))$$

is a list of pairs (variable of a common block declared in SP, value) (see the example of Section 2.2), and it is an instance of the object called "Value of common blocks variables" in figure 5.

***3.2.3. Computation of InputVar (values of variables at program point $\gamma$).*** The variables of SP (and their initial values) are:

- variables of the calling procedure P that have a scope in SP. These are the current static variables $val(\mathsf{S})$.
- formal parameters and variables of common blocks that are declared in SP (i.e. VFormal†VComCalled): in SP, if a formal parameter is also declared in a common block, then its value is the value given by the common block.

The restriction between the two maps VFormal†VComCalled and $localVar(\mathsf{E}) \lhd val(\mathsf{S})$ models scope rules between procedures. The resulting map InputVar belongs to the object called Value in figure 5.

***3.2.4. Computation of S1 (known values at program point $\gamma$).*** $val \otimes comVal$ is an injective function:

- Any state State represents at least one program point, where values of some variables are known. These values are precisely given by the pair of maps $(val \otimes comVal)(\mathsf{State})$. Thus, $val \otimes comVal$ is a total function.
- Given two states S1 and S2, $(val \otimes comVal)(\mathsf{S1}) = (val \otimes comVal)(\mathsf{S2})$ means by definition of a pair that $val(\mathsf{S1}) = val(\mathsf{S2})$ and $comVal(\mathsf{S1}) = comVal(\mathsf{S2})$. This means that all the values of static variables are the same, and that the static variables are also the same in S1 and S2. Thus, S1 and S2 denote the same state.

We can then write $(val \otimes comVal)^{-1}(\mathsf{InputVar}, \mathsf{VComCalled})$ to denote a State object for SP, as explained previously (see Section 2.2). We call this state S1.

***3.2.5. Computation of Cl′ (names of inherited common blocks at program point $\gamma$).*** The common blocks Cl′ that are inherited by SP are those that have a scope in its caller except those that are also declared in SP (these are **dom**$(comDecl(\mathsf{EnvSP}))$, where the domain of the map $comDecl(\mathsf{EnvSP})$ yields the names of common blocks). The common blocks

that have a scope in the caller are its declared common blocks **dom**(*comDecl*(E)) and its inherited common blocks Cl.

***3.2.6. Premise of the inference rule.*** Once S1 and Cl′ have been computed, values have been transmitted to SP and the forward propagation ends. The current program point is $\gamma$. Then, the premise of the dynamic semantics rule is triggered. As indicated by this premise, the statements of SP are executed given the environment of SP (EnvSP), its state (S1) and inherited common blocks (Cl′), yielding a new State object S2 for SP. S2 represents the new values (for common blocks and parameters of SP) that need to be transmitted to the calling procedure.

***3.2.7. Computations of the backward propagation (second part of definitions in figure 6).*** Once the premise of the rule has been triggered, the analysis has reached program point $\delta$. The known values are then transmitted back to the calling procedure: the maps LParam and *comDecl*(E) (from actual parameters and common blocks variables to their values) are updated, yielding the final state S′ of the caller. Similarly to VFormal in the forward propagation,

$$\mathsf{VActual} \triangleq \mathsf{LParam}^{-1}; \mathit{formal}(\mathsf{EnvSP}); \mathit{val}(\mathsf{S2})$$

is a list of pairs (actual parameter, value of formal parameter). The definition of VComCaller (resp. OutputVar) is very close to the definition of VComCalled (resp. InputVar).

SP has inherited the values of common blocks from P. But, the values of some common blocks of SP are not backward propagated in P. These are the common blocks declared in SP that do not have a scope in P. Thus, the definition of S′ differs slightly from the definition of S1. The values OutputComBlock of common blocks at program point $\beta$ are the values of common blocks at program point $\delta$ (i.e. *comVal*(S2)), except for the common blocks that we call CB that are declared in SP and that do not have a scope in P. This means that OutputComBlock is defined as follows:

$$\mathsf{OutputComBlock} \triangleq \mathsf{CB} \lhd \mathit{comVal}(\mathsf{S2})$$

Among the common blocks declared in SP (i.e. **dom**(*comDecl*(EnvSP))), CB represents the common blocks that are neither declared in P, nor inherited by P (from one of its callers). Thus, CB is defined as follows:

$$\mathsf{CB} \triangleq \mathbf{dom}(\mathit{comDecl}(\mathsf{EnvSP})) - (\mathbf{dom}(\mathit{comDecl}(\mathsf{E})) \cup \mathsf{Cl})$$

and the values OutputComBlock of common blocks at program point $\beta$ are:

$$[\mathbf{dom}(\mathit{comDecl}(\mathsf{EnvSP})) - \mathbf{dom}(\mathit{comDecl}(\mathsf{E})) \cup \mathsf{Cl})] \lhd\!\!- \mathit{comVal}(\mathsf{S2}).$$

### *3.3. Object diagram representing pointers*

As Nielson and Nielson (1992), we define two kinds of maps for representing pointers: a variable environment that associates a location with each variable, and a store that associates a value with each location. The variables are represented by locations in stores. The set of

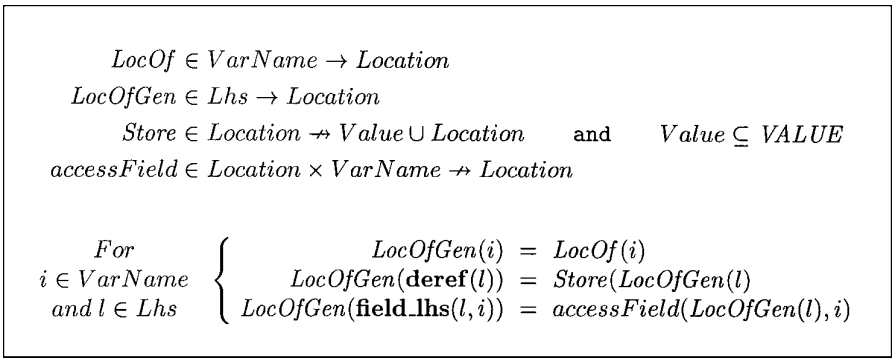



*Figure 7.* Dynamic semantics of some variables.

values of variables that have a scope in a given procedure is denoted by *Value*. It is a subset of *VALUE*, the syntactic category representing values of variables. The set of locations of a given procedure is denoted by *Location*. The dynamic semantics of pointers is modeled by the following functions, which are formally defined in figure 7:

- *LocOf* maps (simple) identifiers to their locations.
- The map *LocOfGen* extends the *LocOf* map to left-hand sides and dereferences. The location of a pointed record is the location of its first field.
- The map *Store* represents the aliases between variables and their values. These values belong either to the set *Value* or to the set *Location*. For instance:
  - If the value of a variable $i$ is 3.4, then in the store this information is modeled by a pair (location of $i$, 3.4), where 3.4 belongs to *Value*.
  - If a pointer $p$ points to a target $^*q$, then in the store this information is modeled by a pair (location of $p$, location of $q$).

  If a pointer is `NULL`, it does not point to anything, and thus it is not represented in the *Store* function. Thus, *Store* is a partial function.
- given the location of a record $r$ and a field $f$, *accessField* yields the location of $r.f$. This is a partial function since only record names with their corresponding fields may have a location.

In figure 7, the operators **deref** and **field_lhs** belong to the Fortran abstract syntax (see figure 3).

The function *val* mapping variables to their known values (see figures 4 and 5) has been split into two maps *LocOfGen* and *Store*. It is now defined as follows:

$$val \triangleq LocOfGen; Store \triangleright VALUE$$

This means that to determine the value of a variable $i$ we shall first determine the location $l = LocOfGen(i)$ associated with $i$, and then determine the value *Store*($l$) associated with

the location $l$. As *Store*($l$) is either a location or a value, the map *LocOfGen*; *Store* is restricted to possible values.

The information about the locations to which a pointer points is modeled by *pointsTo*. *pointsTo* is a map from pointers to their targets, that is defined as follows:

$$pointsTo \triangleq LocOfGen; Store; LocOfGen^{-1}$$

*Example*. If we consider a program point where a pointer $p$ points to a target $^*q$, a pointer $r$ points to a memory cell whose location is denoted by *loc* and there is no other pointer, then at this program point, we will have $pointsTo = \{p \mapsto \mathbf{deref}(q), r \mapsto loc\}$.

Figure 8 represents in diagrammatic form the linked list created by the statements of figure 2. It also shows the dynamic semantics of the corresponding statements. All pointer chaining is resolved before the last two assignments, so any node can be referred to directly by its location. Each node has been dynamically allocated. Thus, each node has a unique location, as shown in the map *LocOfGen*. The definition of this map is illustrated in the second part of figure 8, where the location of *p%next%ident* is computed. The last part of figure 8 details the maps *val* and *pointsTo*:

- *val* states that the value of $^*p$ is 3.4 and the value of **((* p).next)* is 6.2.
- *pointsTo* states that $p$ points to $^*p$ and $q$ points to **((* p).next)*.

Figure 9 models all of the data that are propagated during the execution of any statement. It extends figure 5 by showing functions modeling pointer variables. The maps *LocOfGen* and *Store* are denoted by two objects that are accessed from the object called State. Values of variables are now given by the *store* function. Thus, the object called State may be considered as a triplet *state* (set of locations, set of stores, set of values for variables of common blocks), where *state* denotes a constructor of occurrences of this object.

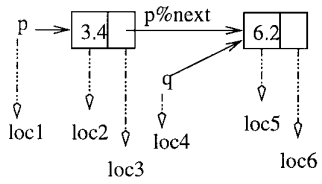
### *3.4. Natural semantics rule for the dynamic semantics of assignments between pointers*

Figure 10 shows the dynamic semantics of a pointer assignment $p \Rightarrow q$, given two pointers $p$ and $q$, an environment $\mathsf{E}$, a state $\mathsf{S}$ and inherited common blocks $\mathsf{Cl}$ (same hypotheses as in the rule for a call statement). The execution of the assignment modifies only locations and stores. This means that it updates only the current state $\mathsf{S}$. The current locations and stores are respectively $\mathsf{L} \triangleq locOfGen(\mathsf{S})$ and $\mathsf{ST} \triangleq store(\mathsf{S})$. The dynamic semantics expresses that $\mathsf{L}$ and $\mathsf{ST}$are updated by the alias introduced by that assignment.

Before the execution of the assignment $p \Rightarrow q$, one of the following three situations arises:

1. $q$ is not `NULL` and points to a target,
2. $q$ is `NULL`,
3. $q$ is not `NULL` and does not point to any target.

**Representation of a list**




$$
\begin{aligned}
LocOf &= \{p \mapsto loc_1, q \mapsto loc_4\} \\
LocOfGen &= \{\mathbf{deref}(p) \mapsto loc_2, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto loc_5\} \cup LocOf \\
Store &= \{loc_1 \mapsto loc_2, loc_3 \mapsto loc_5, loc_4 \mapsto loc_5, loc_2 \mapsto 3.4, loc_5 \mapsto 6.2\} \\
accessField &= \{(loc_2, next) \mapsto loc_3, (loc_2, ident) \mapsto loc_2, (loc_5, ident) \mapsto loc_5, \\
&\quad (loc_5, next) \mapsto loc_6\}
\end{aligned}
$$

**Location of the cell $p\%next\%ident$**

The abstract syntax of $p\%next\%ident$ is:
$\mathbf{fieldLhs}(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)), ident)$.

$$
\begin{aligned}
&LocOfGen(\mathbf{fieldLhs}(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)), ident)) = \\
&accessField(LocOfGen(\mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next))), ident) = \\
&accessField(Store[LocOfGen(\mathbf{fieldLhs}(\mathbf{deref}(p), next))], ident) = \\
&accessField(Store[accessField(LocOfGen(\mathbf{deref}(p), next))], ident) = \\
&accessField(Store[accessField(Store(LocOf(p)), next)], ident) \\
&accessField(Store[accessField(Store(loc_1), next)], ident) = \\
&accessField(Store(loc_3), ident) = loc_5
\end{aligned}
$$

**Maps *val* and *pointsTo***

$$
\begin{aligned}
LocOfGen; Store &= \{\mathbf{deref}(p) \mapsto 3.4, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto 6.2, \\
&\quad p \mapsto loc_2, q \mapsto loc_5\} \\
val &= \{\mathbf{deref}(p) \mapsto 3.4, \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next)) \mapsto 6.2\} \\
pointsTo &= \{p \mapsto \mathbf{deref}(p), q \mapsto \mathbf{deref}(\mathbf{fieldLhs}(\mathbf{deref}(p), next))\}
\end{aligned}
$$

*Figure 8.* An example of linked list.

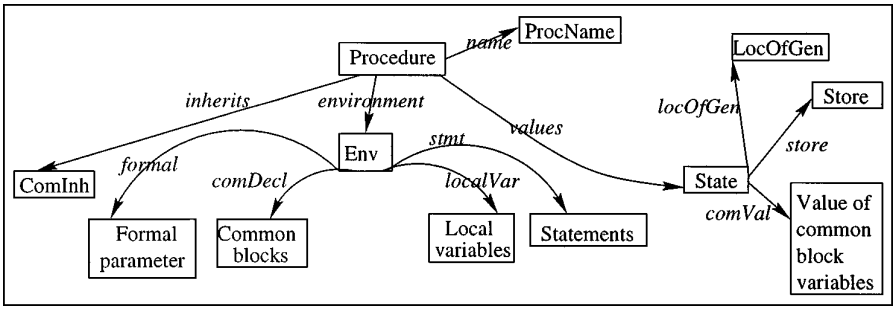



*Figure 9.* Object diagram showing data propagated during the execution of a procedure.

These three situations are modeled by the three dynamic semantics rules of figure 10. The first rule is triggered when $q$ points to a target ${}^*q$. This happens only if $q \in \mathbf{dom}(pointsTo(\mathsf{S}))$. In this case, when $p$ is assigned to the value of $q$:

- $p$ points to ${}^*q$. The value in the store at the location of $p$ (i.e. $\mathsf{ST}(\mathsf{L}(p))$) becomes the location of $q$ (i.e. $\mathsf{L}(q)$), yielding a new map of locations $\mathsf{L1}$.
- the location of ${}^*p$ becomes the location of ${}^*q$, $\mathsf{ST}(\mathsf{L}(q))$, yielding a new map of stores $\mathsf{ST1}$.

A new state $\mathsf{S1}$ is then created from the three components $\mathsf{L1}$, $\mathsf{ST1}$ and $comVal(S)$. This creation is very close to the creation of the states $\mathsf{S1}$ and $\mathsf{S'}$ in the dynamic semantics of call statements (Section 3.2).

In the second rule, $q$ has the NULL value. This means that $q \notin \mathbf{dom}(store)(\mathsf{S})$ (see Section 3.3) and thus by definition of *pointsTo*, $q \notin \mathbf{dom}(points\ To(\mathsf{S}))$. In this case, ${}^*p$ has no location anymore. ST becomes $\{L(p)\} \lhd \mathsf{ST}$. A new state S2 is created from this new store and from the two other components that have not changed.

In the third rule, there is no location pointed by $q$. Then, the maps of locations and stores are restricted in the following way:

- $p$ does not point to any location. L becomes $\{\mathbf{deref}(p)\} \lhd \mathsf{L}$.
- ${}^*p$ has no location anymore. As in the second rule, ST becomes $\{\mathsf{L}(p)\} \rhd \mathsf{ST}$.

## 4. Specification of the partial evaluation

This section shows through an example how the specification of the partial evaluation is derived from the specification of the dynamic semantics. There are two main differences between the dynamic semantics and the partial evaluation. First, in the partial evaluation, some variables have unknown values. Second, to improve the partial evaluation, specialized procedures are reused as often as possible. The example presented in this section concerns the partial evaluation of call statements.

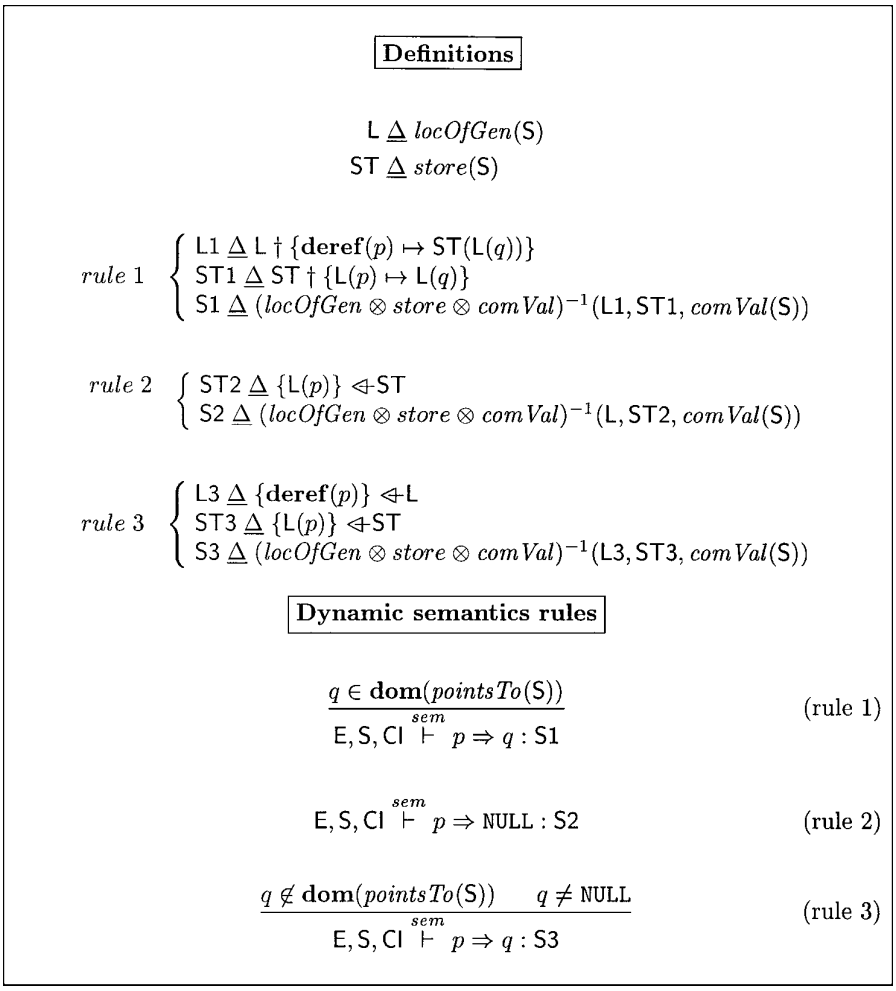



*Figure 10.* Dynamic semantics of a pointer assignment.

As in the dynamic semantics, the specification is presented in two parts: the first part details the object diagram representing specialized versions (i.e. procedures that have already been specialized), and the second part gives the natural semantics rules that specify the partial evaluation of call statements. The section ends with a sketch of a proof of correctness of the partial evaluation.

### *4.1. Object diagram representing specialized versions*

Specialization proceeds depth-first in the call-graph to preserve the order of side-effects. Thus, the specialization of a call statement first runs the specializer on the called

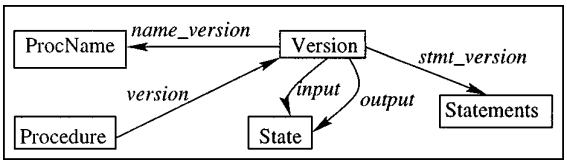



*Figure 11*. Object diagram modeling specialized versions.

procedure SP. This yields a specialized version of SP and the call statement is replaced by a call to this specialized version. A procedure is specialized with respect to specific values of some of its input data (its input static data). At the end of its specialization, the known values of variables belong to its output static data, and a new name is given to the new specialized version (if any).

The diagram of figure 11 models specialized versions. A procedure SP belongs to the object called Procedure. SP represents the code of the whole procedure (its declarations and its statements). The specialized versions of SP belong to the object called Version. They are represented by the set *version*(SP). A specialized version $v$ of SP consists of a name for this version (*name_version*($v$)), input static data (*input*($v$)), output static data (*output*($v$)) and statements (*stmt_version*($v$)). This is equivalent to saying that a version is represented by a quintuplet (name of original procedure, version name, input data, output data, statements). The version and its corresponding procedure have the same arity. Thus, the procedure and the version have the same formal parameters. The name of the version is a name that could be the name of a procedure.

To improve the specialization, specialized versions of procedures are reused. Thus, given a set of specialized procedures, when a call to a procedure SP is encountered in the current procedure (e.g. `CALL SP(x, y, z)`), if the set of input static data of SP and their values (e.g. `x` evaluates to `1`, `y` evaluates to `2` and the value of `z` is unknown):

- is the same as those of a previous call, then the corresponding version is directly reused,
- strictly includes those of a previous call (e.g. `CALL SP(a,b,c)` where a evaluates to `1` and the values of `b` and `c` are unknown), then the corresponding version is specialized yielding a new version that is added to the already specialized versions. If several versions match, the following selections are successively made:
  - most specialized versions, that is the versions with the largest set of input static data,
  - shortest version among most specialized versions.

The partial evaluation of a program is an analysis that propagates:

- the same data as the dynamic semantics (environment, state and inherited common blocks),
- specialized versions of already specialized procedures.

This means that in the sequents for partial evaluation, the data that are written at the left of the turnstile are an environment (e.g. E), a state (e.g. S), some inherited common blocks

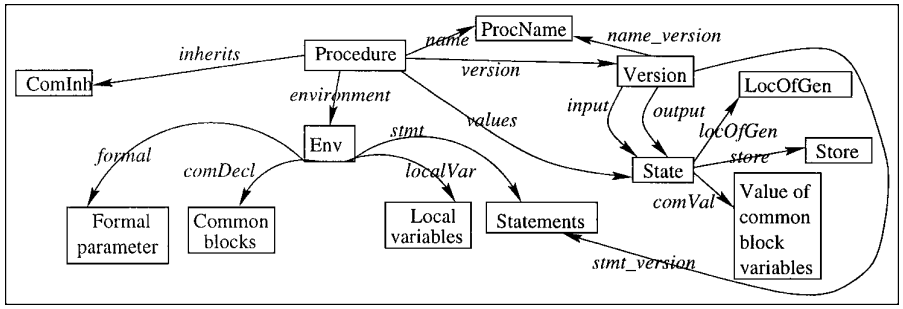



*Figure 12.* Object diagram showing data propagated during partial evaluation.

(e.g. Cl), and also a set (e.g. V) of versions. These versions are procedures that have been already called and then specialized.

Figure 12 shows data that are propagated during the partial evaluation. It extends figures 9 and 11. As in figure 5, an instance of the diagram of figure 12 (without the object Version) is an implicit parameter of the sequents belonging to the partial evaluation system, to simplify the presentation of the sequents. Except for the object Version that is treated separately and appears in the sequents, the objects of the instance are never modified by the sequents.

### *4.2. Natural semantics rule for the partial evaluation*

The inference rules for partial evaluation consist of sequents of the form

$$\mathsf{E,S,Cl,V} \overset{PE}{\vdash} \mathsf{I} \mapsto \mathsf{I}', \mathsf{S}', \mathsf{V}'$$

meaning that given the environment E, the state S, the inherited common blocks Cl and the set of specialized versions V, the specialization of the statement I yields a simplified statement I′, a new state S′ and a new set of versions V′. V′ is the union between V and versions that have been created during the simplification of I. The partial evaluation relation *PE* is the combination of the propagation (*propag*) and simplification (*simpl*) relations.

The dynamic semantic propagates all values of variables through statements. The propagation relation is close to the dynamic semantics relation except that it propagates only known values of variables (with respect to the initial values given by the user). The inference rules for propagation build the *propag* system. They consist of sequents of the form $\mathsf{E, S, Cl} \overset{propag}{\vdash} \mathsf{I : S}'$ meaning that given E, S and Cl, the execution of the statement I modifies the initial state S into the final state S′. The inference rules for simplification consist of sequents of the form $\mathsf{E, S, Cl, V} \overset{simpl}{\vdash} \mathsf{I} \hookrightarrow \mathsf{I}', \mathsf{V}'$ meaning that given E, S, Cl and V, the statement I simplifies into the statement I′ and the set of versions V becomes V′.

In the sequel of this section, we detail the rules for the propagation and the simplification of call statements. Then, we show how the rule for partial evaluation of call statements combines the propagation and the simplification rules.

***4.2.1. Propagation of call statements.*** This section discusses how we have manually derived the propagation of a call statement from its dynamic semantics. This process is similar to the process described by Field, Ramalingam and Tip in (Field et al., 1995), where program slicing algorithms have been automatically derived from semantic specifications, using term rewriting systems.

Figure 13 specifies the propagation of a call statement to a procedure SP with a list LParam of actual parameters, given an environment E, a state S and inherited common blocks Cl. E, S and Cl are propagated through the statement `CALL` SP(LParam), and the result of the propagation is a new state S′.

As in the dynamic semantics, the propagation is composed of forward propagation followed by a backward propagation. Compared to the dynamic semantics, some parameters and variables have unknown values. Formal parameters of SP that have a known value are denoted by StaticFormal. As in figure 6, the map from the formal

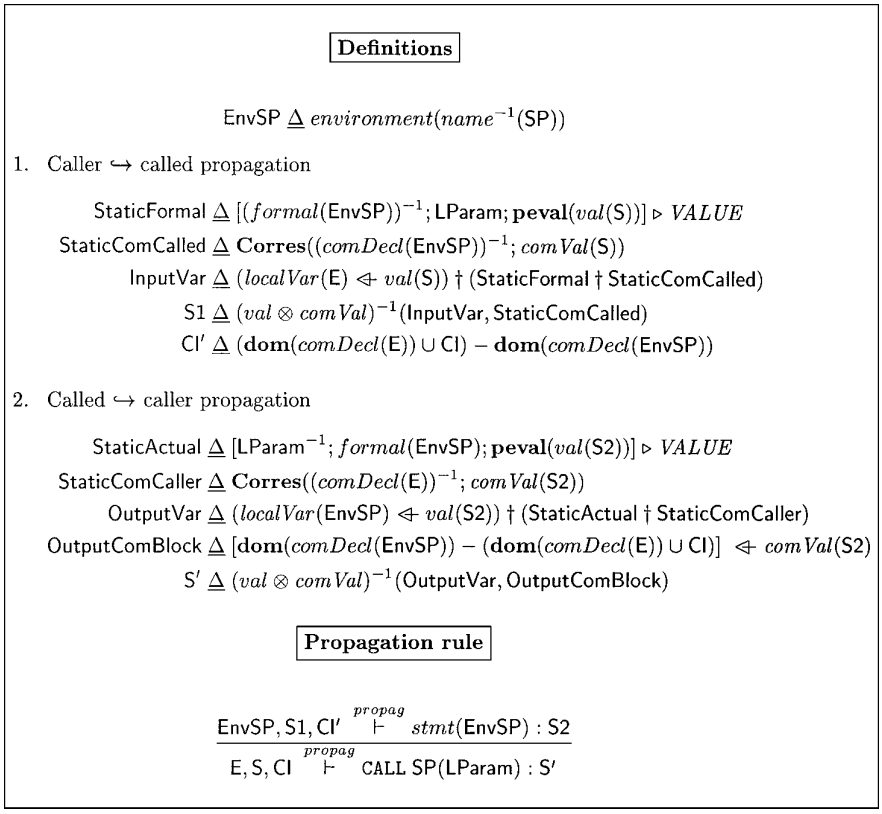



*Figure 13.* Propagation of call statements.

parameters of SP to the actual parameters of the current call statement is represented by $(formal(\mathsf{EnvSP}))^{-1}$; LParam. The **peval** function either yields the value of an expression (if it is known) or gives a residual expression (Blazy and Facon, 1994). Actual parameters are also partially evaluated and simplified (e.g. $x + 1 + y$ is simplified into the normalized expression $3 + x$ when the value of $y$ is 2 and the value of $x$ is unknown) but here evaluation yields expressions whose values may be unknown. As our partial evaluation propagates only equalities between variables and values, the resulting map is restricted to static formal parameters (i.e. formal parameters that have been totally evaluated).

In like manner, during the backward propagation process, StaticActual is computed in the following way:

1. $\mathsf{LParam}^{-1}$; $formal(\mathsf{EnvSP})$ is a map of pairs (actual parameter, corresponding formal parameter).
2. Formal parameters are evaluated with respect to the known values at the end of the propagation in SP.
3. Dynamic actual parameters (they map to an expression whose value is unknown e.g. $z = a - 2$ with $a$ unknown) are removed from the map. The resulting map is StaticActual.

Actually, the computations are the same as the dynamic semantic, except the computations of StaticFormal and StaticActual. The differences come from the evaluation of parameters. Blazy and Facon (1996) give examples of expressions propagated through called procedures.

***4.2.2. Simplification of call statements.*** Figure 14 specifies the simplification of a call statement when the name of the called procedure is SP. The simplification rule:

- simplifies the statements of the called procedure (first premise of the rule),
- propagates data through these simplified statements (second premise),
- creates a new version of the called procedure (third premise).

In figure 14, V denotes the initial set of already specialized versions. As these versions are maintained (following the strategy introduced in Section 4.1), the implementation of the simplification rule has been optimized (see (Blazy and Facon, 1996) for details) to take into account the cases when:

- the called procedure has already been specialized with respect to the same static data,
- the most specialized version of the called procedure is not as specialized as wanted.

As for the dynamic semantics, the simplification process starts with the computation of a new state S1 and new inherited common blocks Cl′ (definitions of figure 14). These definitions are given in figure 13 and thus, they are not repeated in figure 14. As explained previously in the dynamic semantics, S1 represents the set of initial static variables of SP and Cl′ represents the inherited common blocks of SP.

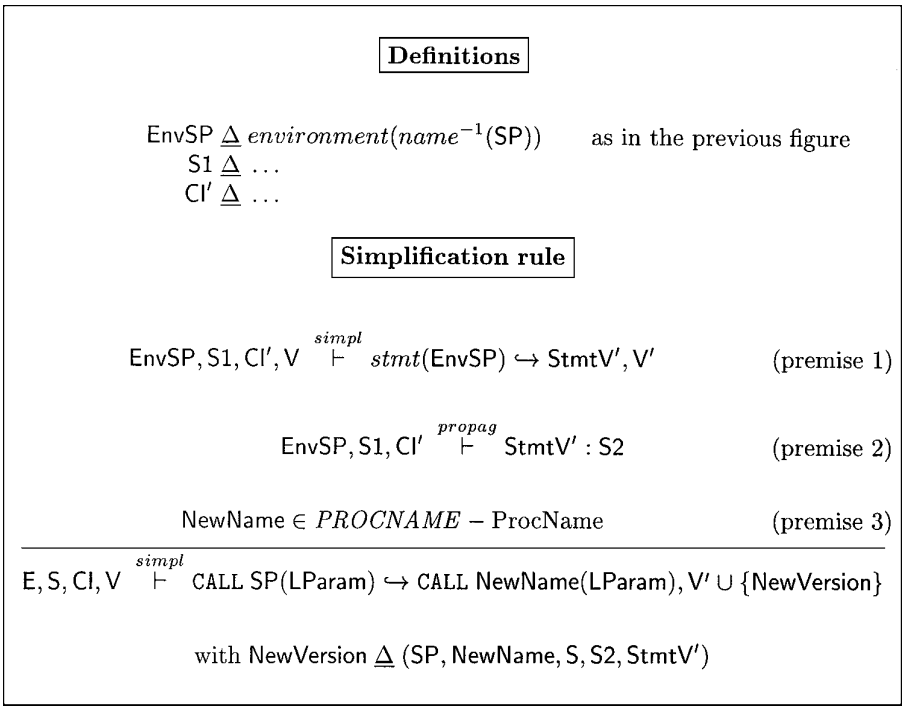



*Figure 14.* Simplification of call statements.

Then, given EnvSP, S1 and Cl′, the statements of the called procedure SP are simplified into StmtV′, yielding an updated set of specialized versions V′ (first premise). V′ is the union of V and the versions that have been created during the simplification of the statements of SP. Then, EnvSP, S1 and Cl′ are propagated through these simplified statements StmtV′, yielding a new state S2 for SP (second premise). A new name (NewName) is created for StmtV′ (third premise). This new name is a possible name that is not already a procedure name:

NewName ∈ *PROCNAME* − ProcName.

The call to SP is replaced by the call to NewName with the same parameters (no unfolding). The new specialized version NewVersion is also added among specialized versions of SP (conclusion of the rule). This new version is the quintuplet (*name of original procedure* = SP, *version name* = NewName, *input data* = S, *output data* = S2, *statements* = StmtV′).

***4.2.3. Partial evaluation.*** Given an environment E, a state S, inherited common blocks Cl, and specialized versions V, the partial evaluation of a call statement combines the

$$\frac{\mathsf{E},\mathsf{S},\mathsf{CI} \overset{propag}{\vdash} \texttt{CALL}\ (\mathsf{LParam}) : \mathsf{S}' \qquad \mathsf{E},\mathsf{S},\mathsf{CI},\mathsf{V} \overset{simpl}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \hookrightarrow \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}),\mathsf{V}'}{\mathsf{E},\mathsf{S},\mathsf{CI},\mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}),\mathsf{S}',\mathsf{V}'}$$

*Figure 15.* Partial evaluation of call statements.

propagation of E, S and CI through this statement and the simplification of this statement. Figure 15 shows how the partial evaluation of a call statement yields a new call statement (with the same actual parameters), a new state and a new set of specialized versions.

### *4.3. Correctness of the partial evaluation*

The natural semantics rules that specify partial evaluation define inductively a formal system (*PE*), that groups the propagation (*propag*) and the simplification (*simpl*) systems. Propagated data and simplified statements are built up by applying the natural semantics rules. To prove that the partial evaluation system is correct means proving that it is sound (each result of a residual program is correct with respect to the initial program) and complete (each correct result is computed by the residual program, too) with respect to the dynamic semantics of Fortran 90 given in Section 3. The derivation of the partial evaluation rules from the dynamic semantics rules has facilitated the proof of correctness. The dynamic semanatics rules are not proved: they are supposed to define *ex nihilo* the semantics of Fortran 90.

Figure 16 gives the general schema to prove the correctness of the partial evaluation. Given an initial procedure P, its environment E represents its syntax, and its inherited common blocks CI can be computed from its call graph. Initially, not a single specialized version has been created. Given some values S0 for known input data, partial evaluation of P yields a residual procedure $\mathsf{P}'$. Let a state $\mathsf{S}'$ and a set of specialized versions $\mathsf{V}'$ be such that the sequent $\mathsf{E}, \mathsf{S0}, \mathsf{CI}, [\ ] \overset{PE}{\vdash} \mathsf{P} \mapsto \mathsf{P}', \mathsf{S}', \mathsf{V}'$ holds in the partial evaluation system (*PE*). In this sequent [ ] denotes the initial and empty list of specialized versions. Given this sequent, we want to prove that the partial evaluation is sound and complete with respect to

Given the sequent $\mathsf{E},\mathsf{S0},\mathsf{CI},[\,] \overset{PE}{\vdash} \mathsf{P} \mapsto \mathsf{P}',\mathsf{S}',\mathsf{V}'$
we want to show $\mathsf{E},\mathsf{S0}\cup\mathsf{S},\mathsf{CI} \overset{sem}{\vdash} \mathsf{P} : \mathsf{S}'' \Leftrightarrow \mathsf{E},\mathsf{S0}\cup\mathsf{S},\mathsf{CI} \overset{sem}{\vdash} \mathsf{P}' : \mathsf{S}''$
for all states S and S''

*Figure 16.* Correctness of the partial evaluation.

the dynamic semantics (*sem* system). This is expressed by the following property of partial evaluation, where the implication $\Leftarrow$ (resp. $\Rightarrow$) states soundness (resp. completeness).

$$\forall \mathsf{S}, \mathsf{S1} : (\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \mathsf{P} : \mathsf{S1} \Leftrightarrow \mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \mathsf{P}' : \mathsf{S1}).$$

S denotes the values of dynamic input variables (initially, their values are unknown). Such variables are the remaining input data once S0 has been given by the user. Thus, S0 ∪ S denotes the values of the whole input variables. To prove in the *sem* system this property on programs, we prove it for any Fortran 90 statement we simplify (i.e. for any rule of the partial evaluation system). Such a proof proceeds by structural induction on the Fortran 90 abstract syntax: the proof that the partial evaluation of a compound statement such as `IF` *c* `THEN` *i*1 `ELSE` *i*2 is correct is done under the induction hypothesis that states it is correct for the substatements *i1* and *i2*. The proof for some simple simplifications of statements has been given in Blazy and Facon (1995).

Our approach to prove the correctness of partial evaluation is close to the approach of Despeyroux (1986) to prove the correctness of translators: in that paper, dynamic semantics and translation are both given by formal systems and the correctness of the translation with respect to dynamic semantics of source and object languages is also formalized by inference rules that are proved by induction on the length of the derivation of the translated terms. However, as we are concerned with only one language (Fortran 90), to simplify our proofs we do not deal with the validity of inference rules in the union of several formal systems (expressing the dynamic semantics of two langages and the translation from one language to the other). Furthermore, we do not need rule induction for all our proofs, but only structural induction and sometimes induction on derivations (to handle loops).

As an example, we give here the guidelines for proving by structural induction the soundness of the partial evaluation of a call statement `CALL` SP(LParam). The partial evaluation of this statement generates a call statement to a different procedure SP′, but SP′ is called with the same actual parameters as SP (i.e. LParam). Figure 17 gives the soundness rule to prove. This rule is proved under the induction hypothesis that the partial evaluation of the statements of SP is sound. The partial evaluation of this statement generates a call statement to a different procedure SP′, but SP′ is called with the same actual parameters as SP (see Section 4.2.3).

$$\frac{\mathsf{E}, \mathsf{S0}, \mathsf{CI}, \mathsf{V} \overset{PE}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) \mapsto \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}), \mathsf{V}', \mathsf{S}' \qquad \mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI} \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}'(\mathsf{LParam}) : \mathsf{S}''}{\mathsf{E}, \mathsf{S0} \cup \mathsf{S}, \mathsf{CI}, \overset{sem}{\vdash} \texttt{CALL}\ \mathsf{SP}(\mathsf{LParam}) : \mathsf{S}''}$$

*Figure 17.* Soundness rule for a call statement.

Given the two sequents of the premise of the rule given in figure 17, we have to prove that given the same input values $S0 \cup S$, the call statement of the source program yields the same output values $S''$. This is expressed by the conclusion of the rule. It will then prove the soundness of the partial evaluation of the call statement `CALL SP(LParam)` and of the specialized versions of SP. This is proved in two stages.

1. First, no data declaration is modified during the partial evaluation. Thus, both procedures SP and SP′ have the same formal parameters and the same data declarations. It follows that given input values $S0 \cup S$ the data declarations of SP yield the same values as the declarations of SP′.
2. Second, we assume that the induction hypothesis on the simplified statements of SP holds, and deduce that the new version and the statements of SP are sound as required.

## 5. The tool

We have implemented our partial evaluator on top of a kernel that has been generated by the Centaur system (INRIA, 1994). The Centaur system is a generic programming environment parametrized by the syntax and semantics of programming languages. When provided with the description of a particular programming language, Centaur produces a language-specific environment. The intermediate format for representing program text is the abstract syntax tree. We have merged two specific environments (one dedicated to Fortran 90 and another to a language that we have defined for expressing the scope of general constraints on variables) into an environment for partial evaluation. This environment consists of structured editors for constraints and Fortran 90 procedures (provided by Centaur), a partial evaluator, together with an uniform graphical interface. Figure 18 shows the architecture of our tool, its inputs and outputs. Given a file of constraints and an initial program represented by several

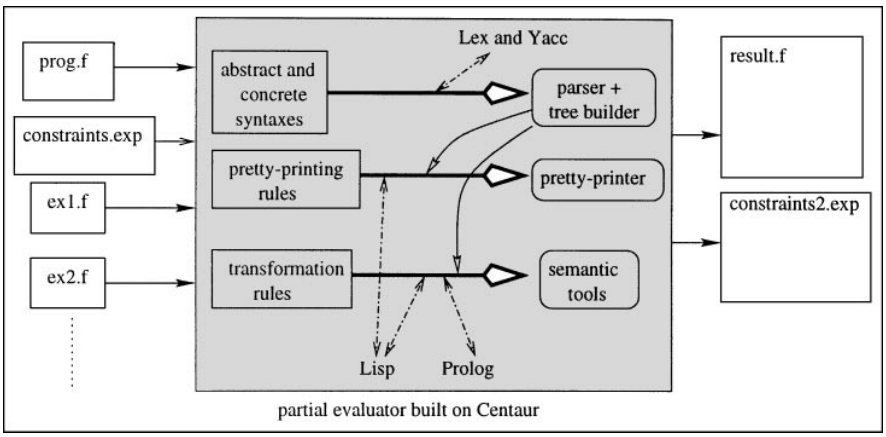



*Figure 18.* Architecture of the partial evaluator.

Fortran 90 files, the tool generates the program specialized with respect to these constraints and the corresponding final constraints.

The formal specifications have been implemented in a language provided by Centaur called Typol. Typol is an implementation of natural semantics. Typol programs are compiled into Prolog code. When executing these programs, Prolog is used as the engine of the deductive system. Set and relational operators as definitions have been written directly in Prolog in order to develop succinct and efficient Typol rules (Dubois and Sayarath, 1996). Thus, the Typol rules operate on the abstract syntax and they are close to the formal specification rules as shown in Blazy and Facon (1994). The partial evaluation process transforms an initial abstract syntax tree (representing the initial Fortran procedures and constraints) into a residual abstract syntax tree (representing the specialized code and the final constraints).

The abstract syntax used to describe Fortran 90 is general and close to the abstract syntax of any imperative language. For instance, to be more general, our specifications assume a dereferencing operator that does not exist in Fortran 90. The only peculiarities of Fortran 90 are the parameter passing (by reference only) and the use of common blocks instead of global variables. Except the corresponding specification rules, other rules are abstract enough to be the rules of any other imperative language (without recursion).

We have implemented a graphical interface to facilitate the exploration of Fortran 90 application programs (Vassallo, 1996). It has been written in Lisp, enhanced with structures for programming communication between graphical objects and processes. Different kinds of information are important for the user. At the beginning of the partial evaluation process, the user selects an initial application program (usually consisting of several Fortran files) and some constraints related to variables defined in any of the Fortran procedures. Then, the user runs the partial evaluator on his application program and constraints. Several partial evaluation processes can be run at once. Partial evaluation produces:

- the specialized application program (it has the same structure than the initial application program) and the links between this application program and the initial one,
- the specialized versions of called procedures and the links with the initial and final application programs,
- the final values of the variables whose initial values have been given by the user,
- the final values of other variables if they are known,
- information that is computed during the partial evaluation but that is not very important for most users (e.g. initial values of formal parameters before the partial evaluation of a called procedure).

The graphical interface organizes this information into different displays so that it is useful to the user. It allows the user:

- to identify specific nodes in abstract syntax trees (e.g. called procedures) as they are specifically colored,
- to display by default only the most important information,
- to trigger the display of other information when needed.

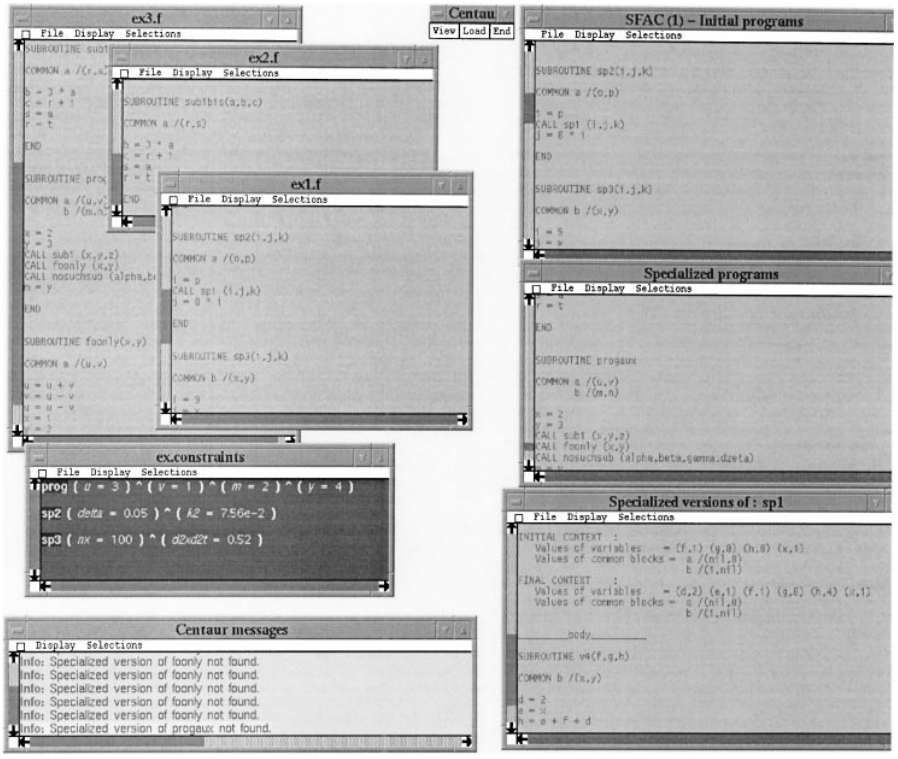



*Figure 19.* Partial evaluation of a Fortran 90 application (with reuse of specialized versions).

For instance, the specialized versions of a called procedure `P` are displayed only when the user clicks on a call statement to `P`. In the same way, the final values of propagated data are not displayed automatically.

Figure 19 presents the graphical interface. In this example:

- The three windows in the upper left corner show that the user has selected the files called `ex1.f`, `ex2.f` and `ex3.f`. They define the application program to specialize.
- The window below these three windows shows that the user has clicked in one of the three previous windows to choose the constraints related to these three windows. In figure 19 the constraints are written in the file called `ex.constraints`.
- The window in the lower left corner (called "Centaur messages") displays warning messages. In the example, the messages have been displayed when the user has asked for the display of a specialized version that does not exist. Error messages are also displayed, for instance when the user has forgotten to select constraints related to an initial program.
- The two windows in the upper right corner show that partial evaluation has been triggered (by a click in any of the three Fortran windows or in the constraints window). The first window resulting from the partial evaluation (called "Initial program" in figure 19)

displays all the procedures to specialize. The second window displays (below the previous one in the figure) all the specialized procedures.

- The lower right window displays the specialized versions of the procedure `sp1` (e.g. in the window called "Specialized versions of sp1" in figure 19). The display of this window has been triggered by a click on the call statement to `SP1` in any of the Fortran windows.

Figure 19 shows only one partial evaluator running on a constraints file and a Fortran application program made of three Fortran files. In figure 19, this partial evaluator is numbered SFAC(1) (this number is written in the title of the "Initial programs" window). But, several partial evaluations can also run in parallel. This is very useful when for instance the user needs to compare the results of the partial evaluation of an application program with several constraints files.

## 6. Conclusion

This paper has presented an approach to the understanding of application programs during their maintenance. The approach relies on partial evaluation, a technique that we have adapted to program understanding. The partial evaluation performs an interprocedural pointer analysis. We have formally specified our partial evaluation process and we have derived the propagation rules from the dynamic semantics rules, also expressed in natural semantics. In these specifications, inference rules in natural semantics show only how statements are simplified from data propagation and simplification of other statements. To clarify the presentation of the natural semantics rules, we have used set and relational operators to express outside the rules the computations on propagated data. Some of these data are not introduced directly in the dynamic semantics rules. They are accessed from other data, according to the object diagram that we have defined.

From the specifications, we have proven by induction the correctness of the partial evaluation with respect to a dynamic semantics of Fortran 90. This proof has been done by hand. We are currently investigating an automatic proof of the correctness of the partial evaluation.

A tool has been implemented from the specifications. A graphical interface has also been implemented to visualize program dependencies (mainly between variables and values and between reused versions of procedures). The tool has been tested at the EDF (the French national company that produces and distributes electricity), that provided us with scientific application programs (Blazy and Facon, 1994). The first results are very encouraging. We are planning more empirical work to validate these preliminary results: we intend to test other application programs using pointers extensively and to use metrics such as the metrics defined in Carini and Hind (1995) to evaluate our interprocedural constant propagation. Another current focus is in improving the analysis by propagating general constraints between variables instead of only equalities between variables and values. To this end, we could adapt the rules described in Bergstra et al. (1997).

Furthermore, partial evaluation is complementary to program slicing, another technique for extracting code when debugging a program. Program slicing aims at identifying the statements of the program which impact directly or indirectly on some variable values. We

believe that merging partial evaluation (a forward walk through the call graph) and program slicing (a blackward walk that can also be followed by a forward walk) would improve a lot the simplification of programs.

**Acknowledgments**

I would like to thank the anonymous reviewers for their detailed comments and suggestions, which have helped improve both the contents and presentation of this article.

**In Memoriam**

I would like to dedicate this paper to Professor Philippe Facon, who died tragically on September 1st, 1999. He was the leader of our research team and was one of the guiding forces of the project presented above.

**Note**

1. We could have used the other symbol for composition $\circ$ that is tantamount to ; since for any pair of binary relations $r$ and $p$ it is defined by $p \circ r = r; p$.

**References**

1. Abrial, J.R. 1996. *The B-Book Assigning Programs to Meanings*. Cambridge University Press, New York.
2. ACM. 1998. *Symposium on Partial Evaluation*, number 4 in ACM Computing Surveys.
3. Aho, A.V., Sethi, R., and Ullman, J.D. 1988. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.
4. Andersen, L.O., 1994. Program Analysis and Specialization for the C Programming Language. PhD Thesis, University of Copenhagen, DIKU report 94/19.
5. ANSI, 1992. New York. *Programming Language Fortran 90*, ANSI X3.198-1992 and ISO/IEC 1539–1991 (E).
6. Baier, R., Glück, R., and Zöchling, R. 1994. Partial evaluation of numerical programs in Fortran. In *Proceedings of the Partial Evaluation and Semantics Based Program Manipulation Workshop*, Melbourne, pp. 119–132. ACM SIGPLAN.
7. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. 1998. Clone detection using abstract syntax trees. In *Proc. of the Conf. on Soft. Maintenance*. IEEE.
8. Bergstra, J.A., Dinesh, T.B., Field, J., and Heering, J. 1997. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programing Languages and Systems*, 19(5):639–684.
9. Blazy, S. and Facon, P. 1994. SFAC, a tool for program comprehension by specialization. In *Proceedings of the Third Workshop on Program Comprehension*, Washington D.C., pp. 162–167. IEEE.
10. Blazy, S. and Facon, P. 1995. Formal specification and prototyping of a program specializer. In *Proceedings of the TAPSOFT Conference*, Lecture Notes in Computer Science, 915, pp. 666–680. Aarhus, Elsevier Science Publishers B.V. (North-Holland), Amsterdam.
11. Blazy, S. and Facon, P. 1996. An automatic interprocedural analysis for the understanding of scientific application programs. In Danvy et al. (1996), pp. 1–16.
12. Blazy, S. and Facon, P. 1998. Partial evaluation for program understanding. *ACM Computing Surveys—Symposium on Partial Evaluation*, 4(4).
13. Carini, R. and Hind, M. 1995. Flow-sensitive interprocedural constant propagation. In *Proc. of the Programming Languages Design and Implementation Conf.*, La Jolla, pp. 23–31. ACM SIGPLAN.

14. Chase, D.R., Wegman, M., and Zadeck, F.K. 1990. Analysis of pointers and structures. In *Proc. of the Programming Languages Design and Implementation Conf.*, White Plains, pp. 296–310. ACM SIGPLAN.
15. Danvy, O., Glück, R., and Thiemann, P. (Eds.), 1996. *International seminar on partial evaluation*, Dagstuhl castle, Elsevier Science Publishers B.V. (North-Holland), Amsterdam. Lecture Notes in Computer Science 1110.
16. Despeyroux, J. 1986. Proof of translation in natural semantics. In *Proceedings of the Symposium on Logic in Computer Science*, Cambridge, USA.
17. Dubois, N. and Sayarath, P. 1996. Aide à la compréhension et à la maintenance: Pointeurs pour la spécialisation de programmes. Master's Thesis. IIE-CNAM, in French.
18. Emami, M., Ghiya, R., and Hendren, L.J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the Programming Languages Design and Implementation Conf.*, pp. 242–256. ACM SIGPLAN.
19. Field, J., Ramalingam, G., and Tip, F. 1995. Parametric program slicing. In *Proc. of Principles of Programming Languages Conf.*, San Francisco, USA, pp. 379–392.
20. Hannan, J. 1993. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152.
21. Hasti, R. and Horwitz, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proc. of the Programming Languages Design and Implementation Conf.*, Montreal, pp. 97–105. ACM SIGPLAN.
22. INRIA, 1994. Centaur 1.2 documentation.
23. Jones, C.B. 1990. *Systematic Development Using VDM*. Prentice-Hall, Englewood Cliff.
24. Jones, N.D., Gomard, C.K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliff.
25. Kahn, G. 1987. Natural semantics. In *Proc. of the Symp. on Theoretical Aspects of Comp. Science*, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Lecture Notes in Computer Science 247, pp. 237–257.
26. Landi, W. and Ryder, B.G. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the Programming Languages Design and Implementation Conf.*, pp. 235–248. ACM SIGPLAN.
27. Liang, D. and Harrold, M.J. 1999. Efficient points-to analysis for whole-program analysis. In *Proc. of ESEC/FSE Joint Conf.*, Toulouse, France, Lecture Notes in Computer Science 1687, pp. 199–215. ACM SIGSOFT.
28. Marlet, R., Thibault, S., and Consel, C. 1997. Mapping software architectures to efficient implementation via partial evaluation. In *Proceedings of the Automated Software Engineering Conference*, pp. 183–192. IEEE.
29. Muchnick, S.S. 1997. *Advanced Compiler-Design and Implementation*. Morgan Kaufmann. Los Altos.
30. Nielson, H.R. and Nielson, F. 1992. *Semantics with Application—A Formal Introduction*. John Wiley and Sons, New York.
31. Sagiv, M., Reps, T., and Wilhelm, R. 1997. Solving shape-analysis problems in languages with destructive updating. In *Proc. of Principles of Programming Languages Conf.*
32. Sellink, M.P.A. and Verhoef, C. 2000. Scaffolding for software renovation. In *Proc. of the Conf. on Soft. Maintenance*, Zürich. IEEE.
33. Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proc. of Principles of Programming Languages Conf.*, pp. 32–41.
34. van den Brand, M.G.J., Klint, P., and Verhoef, C. 1996. Reverse engineering and system renovation: an annotated bibliography. Technical Report P9603, University of Amsterdam, Programming Research Group.
35. Vassallo, R. 1996. Ergonomie et évolution d'un outil de compréhension de programmes. Master's thesis. IIE-CNAM, in French.
36. Wilson, R.P. and Lam, M.S. 1995. Efficient context-sensitive pointer analysis for c programs. In *Proc. of the Programming Languages Design and Implementation Conf.*, pp. 1–12. ACM SIGPLAN.
37. Yank, H., Luker, P., and Chu, W. 1997. Code understanding through program transformation for reusable component identification. In *Fifth Workshop on Program Comprehension Proceedings*. IEEE.

# Formal Verification of a C Compiler Front-End

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy

INRIA Rocquencourt, 78153 Le Chesnay, France
{Sandrine.Blazy, Zaynah.Dargaye, Xavier.Leroy}@inria.fr

**Abstract.** This paper presents the formal verification of a compiler front-end that translates a subset of the C language into the Cminor intermediate language. The semantics of the source and target languages as well as the translation between them have been written in the specification language of the Coq proof assistant. The proof of observational semantic equivalence between the source and generated code has been machine-checked using Coq. An executable compiler was obtained by automatic extraction of executable Caml code from the Coq specification of the translator, combined with a certified compiler back-end generating PowerPC assembly code from Cminor, described in previous work.

## 1 Introduction

Formal methods in general and program proof in particular are increasingly being applied to safety-critical software. These applications create a strong need for on-machine formalization and verification of programming language semantics and tools such as compilers, type-checkers and static analyzers. In particular, formal operational semantics are required to validate the logic of programs (e.g. axiomatic semantics) used to reason about programs. As for tools, the formal certification of compilers—that is, a proof that the generated executable code behaves as prescribed by the semantics of the source program—is needed to ensure that the guarantees obtained by formal verification of the source program carry over to the executable code.

For high-level programming languages such as Java and functional languages, there exists a considerable body of on-machine formalizations and verifications of operational semantics and programming tools such as compilers and bytecode verifiers. Despite being more popular for writing critical embedded software, lower-level languages such as C have attracted less interest: several formal semantics for various subsets of C have been published, but few have been carried on machine. (See section 5 for a review.)

The work presented in this paper is part of an ongoing project that aims at developing a realistic compiler for the C language and formally verifying that it preserves the semantics of the programs being compiled. A previous paper [8] describes the verification, using the Coq proof assistant, of the back-end of this compiler, which generates moderately optimized PowerPC assembly code from a low-level, imperative intermediate language called Cminor. The present paper reports on the development and proof of semantic preservation in Coq of a C

J. Misra, T. Nipkow, and E. Sekerinski (Eds.): FM 2006, LNCS 4085, pp. 460–475, 2006.

front-end for this compiler: a translator from Clight, a subset of the C language, to Cminor. To conduct the verification, a precise operational semantics of Clight was formalized in Coq. Clight features all C arithmetic types and operators, as well as arrays, pointers, pointers to functions, and all C control structures except `goto` and `switch`.

From a formal methods standpoint, this work is interesting in two respects. First, compilers are complex programs that perform sophisticated symbolic computations. Their formal verification is challenging, requiring difficult proofs by induction that are beyond the reach of many program provers. Second, proving the correctness of a compiler provides an indirect but original way to validate the semantics of the source language. It is relatively easy to formalize an operational semantics, but much harder to make sure that this semantics is correct and captures the intended meaning of programs. Typically, extensive testing and manual reviews of the semantics are needed. In our experience, proving the correctness of a translator to a simpler, lower-level language detects many small errors in the semantics of the source and target languages, and therefore generates additional confidence in both.

The remainder of this paper is organized as follows. Section 2 describes the Clight language and gives an overview of its operational semantics. Section 3 presents the translation from Clight to Cminor. Section 4 outlines the proof of correctness of this translation. Related work is discussed in section 5, followed by conclusions in section 6.

## 2 The Clight Language and Its Semantics

### 2.1 Abstract Syntax

The abstract syntax of Clight is given in figure 1. In the Coq formalization, this abstract syntax is presented as inductive data types, therefore achieving a deep embedding of Clight into Coq.

At the level of types, Clight features all the integral types of C, along with array, pointer and function types; `struct`, `union` and `typedef` types are currently omitted. The integral types fully specify the bit size of integers and floats, unlike the semi-specified C types `int`, `long`, etc.

Within expressions, all C operators are supported except those related to structs and unions. Expressions may have side-effects. All expressions and their sub-expressions are annotated by their static types. We write $a^\tau$ for the expression $a$ carrying type $\tau$. These types are necessary to determine the semantics of type-dependent operators such as the overloaded arithmetic operators. Similarly, combined arithmetic-assignment operators such as `+=` carry an additional type $\sigma$ (as in $(a_1 \texttt{+=}^\sigma a_2)^\tau$) representing the result type of the arithmetic operation, which can differ from the type $\tau$ of the whole expression.

At the level of statements, all structured control statements of C (conditional, loops, `break`, `continue` and `return`) are supported, but not unstructured statements (`goto`, `switch`, `longjmp`). Two kinds of variables are allowed: global variables and local `auto` variables declared at the beginning of a function.

Types:

| | | |
|---|---|---|
| *signedness* | ::= `Signed` \| `Unsigned` | |
| *intsize* | ::= `I8` \| `I16` \| `I32` | |
| *floatsize* | ::= `F32` \| `F64` | |
| $\tau$ | ::= `Tint`$(intsize, signedness)$ \| `Tfloat`$(floatsize)$ | |
| | \| `Tarray`$(\tau, n)$ \| `Tpointer`$(\tau)$ \| `Tvoid` \| `Tfunction`$(\tau^*, \tau)$ | |

Expressions annotated with types:

| | | |
|---|---|---|
| $a$ | ::= $b^\tau$ | |

Unannotated expressions:

| | | |
|---|---|---|
| $b$ | ::= $id$ | variable identifier |
| | \| $n$ \| $f$ | integer or float constant |
| | \| `sizeof`$(\tau)$ | size of a type |
| | \| $op_u\ a$ | unary arithmetic operation |
| | \| $a_1\ op_b\ a_2$ \| $a_1\ op_r\ a_2$ | binary arithmetic operation |
| | \| `*`$a$ | dereferencing |
| | \| $a_1[a_2]$ | array indexing |
| | \| `&`$a$ | address of |
| | \| `++`$a$ \| `--`$a$ \| $a$`++` \| $a$`--` | pre/post increment/decrement |
| | \| $(\tau)a$ | cast |
| | \| $a_1 = a_2$ | assignment |
| | \| $a_1\ op_b =^\tau\ a_2$ | arithmetic with assignment |
| | \| $a_1$ `&&` $a_2$ \| $a_1$ `\|\|` $a_2$ | sequential boolean operations |
| | \| $a_1, a_2$ | sequence of expressions |
| | \| $a(a^*)$ | function call |
| | \| $a_1$ `?` $a_2$ `:` $a_3$ | conditional expression |
| $op_b$ | ::= `+` \| `-` \| `*` \| `/` \| `%` | arithmetic operators |
| | \| `<<` \| `>>` \| `&` \| `\|` \| `^` | bitwise operators |
| $op_r$ | ::= `<` \| `<=` \| `>` \| `>=` \| `==` \| `!=` | relational operators |
| $op_u$ | ::= `-` \| `~` \| `!` | unary operators |

Statements:

| | | |
|---|---|---|
| $s$ | ::= `skip` | empty statement |
| | \| $a$; | expression evaluation |
| | \| $s_1; s_2$ | sequence |
| | \| `if`$(a)$ $s_1$ `else` $s_2$ | conditional |
| | \| `while`$(a)$ $s$ | "while" loop |
| | \| `do` $s$ `while`$(a)$ | "do" loop |
| | \| `for`$(a_1^?, a_2^?, a_3^?)$ $s$ | "for" loop |
| | \| `break` | exit from the current loop |
| | \| `continue` | next iteration of the current loop |
| | \| `return` $a^?$ | return from current function |

Functions:

| | | |
|---|---|---|
| *fn* | ::= $(\ldots id_i : \tau_i \ldots) : \tau$ | declaration of type and parameters |
| | { $\ldots \tau_j\ id_j; \ldots$ | declaration of local variables |
| | $s$ } | function body |

**Fig. 1.** Abstract syntax of Clight. $a^*$ denotes 0, 1 or several occurrences of syntactic category $a$. $a^?$ denotes an optional occurrence of category $a$.

Block-scoped local variables and `static` variables are omitted, but can be emulated by pulling their declarations to function scope or global scope, respectively. Consequently, there is no block statement in Clight.

A Clight program consists of a list of function definitions, a list of global variable declarations, and an identifier naming the entry point of the program (the `main` function in C).

### 2.2 Dynamic Semantics

The dynamic semantics of Clight is specified using natural semantics, also known as big-step operational semantics. While the semantics of C is not deterministic (the evaluation order for expressions is not completely specified and compilers are free to choose between several orders), the semantics of Clight is completely deterministic and imposes a left-to-right evaluation order, consistent with the order implemented by our compiler. This choice simplifies greatly the semantics compared with, for instance, Norrish's semantics for C [10], which captures the non-determinism allowed by the ISO C specification. Our semantics can therefore be viewed as a refinement of (a subset of) the ISO C semantics, or of that of Norrish.

The semantics is defined by 7 judgements (relations):

$$
\begin{array}{rl}
G, E \vdash a, M \stackrel{l}{\Rightarrow} loc, M' & \text{(expressions in l-value position)} \\
G, E \vdash a, M \Rightarrow v, M' & \text{(expressions in r-value position)} \\
G, E \vdash a^?, M \Rightarrow v, M' & \text{(optional expressions)} \\
G, E \vdash a^*, M \Rightarrow v^*, M' & \text{(list of expressions)} \\
G, E \vdash s, M \Rightarrow out, M' & \text{(statements)} \\
G \vdash f(v^*), M \Rightarrow v, M' & \text{(function invocations)} \\
\vdash p \Rightarrow v & \text{(programs)}
\end{array}
$$

Each judgement relates a syntactic element (expression, statement, etc) and an initial memory state to the result of executing this syntactic element, as well as the final memory state at the end of execution. The various kinds of results, as well as the evaluation environments, are defined in figure 2.

For instance, executing an expression $a$ in l-value position results in a memory location $loc$ (a memory block reference and an offset within that block), while executing an expression $a$ in r-value position results in the value $v$ of the expression. Values range over 32-bit integers, 64-bit floats, memory locations (pointers), and an undefined value that represents for instance the value of uninitialized variables. The result associated with the execution of a statement $s$ is an "outcome" $out$ indicating how the execution terminated: either normally by running to completion or prematurely via a `break`, `continue` or `return` statement. The invocation of a function $f$ yields its return value $v$, and so does the execution of a program $p$.

Two evaluation environments, defined in figure 2, appear as parameters to the judgements. The local environment $E$ maps local variables to references of memory blocks containing the values of these variables. These blocks are allocated

Values:

| | | |
|---|---|---|
| $loc$ | $::= (b, n)$ | location (byte offset $n$ in block referenced by $b$) |
| $v$ | $::= \texttt{Vint}(n)$ | integer value |
| | $\mid \texttt{Vfloat}(f)$ | floating-point value |
| | $\mid \texttt{Vptr}(loc)$ | pointer value |
| | $\mid \texttt{Vundef}$ | undefined value |

Statement outcomes:

| | | |
|---|---|---|
| $out$ | $::= \texttt{Out\_normal}$ | go to the next statement |
| | $\mid \texttt{Out\_continue}$ | go to the next iteration of the current loop |
| | $\mid \texttt{Out\_break}$ | exit from the current loop |
| | $\mid \texttt{Out\_return}$ | function exit |
| | $\mid \texttt{Out\_return}(v, \tau)$ | function exit, returning the value $v$ of type $\tau$ |

Global environments:

| | | |
|---|---|---|
| $G$ | $::= (id \mapsto b)$ | map from global variables to block references |
| | $\times (b \mapsto fn)$ | and map from references to function definitions |

Local environments:

| | | |
|---|---|---|
| $E$ | $::= id \mapsto b$ | map from local variables to block references |

**Fig. 2.** Values, outcomes, and evaluation environments

at function entry and freed at function return. The global environment $G$ maps global variables and function names to memory references. It also maps some references (those corresponding to function pointers) to function definitions.

In the Coq specification, the 7 judgements of the dynamic semantics are encoded as mutually-inductive predicates. Each defining case of each predicate corresponds exactly to an inference rule in the conventional, on-paper presentation of natural semantics. We have one inference rule for each kind of expression and statement described in figure 1. We do not list all the inference rules by lack of space, but show some representative examples in figure 3.

The first two rules of figure 3 illustrate the evaluation of an expression in l-value position. A variable $x$ evaluates to the location $(E(x), 0)$. If an expression $a$ evaluates to a pointer value $\texttt{Vptr}(loc)$, then the location of the dereferencing expression $(*a)^\tau$ is $loc$.

Rule 3 evaluates an application of a binary operator $op$ to expressions $a_1$ and $a_2$. Both sub-expressions are evaluated in sequence, and their values are combined with the `eval_binary_operation` function, which takes as additional arguments the types $\tau_1$ and $\tau_2$ of the arguments, in order to resolve overloaded and type-dependent operators. This is a partial function: it can be undefined if the types and the shapes of argument values are incompatible (e.g. a floating-point addition of two pointer values). In the Coq specification, `eval_binary_operation` is a total function returning optional values: either `None` in case of failure, or $\texttt{Some}(v)$, abbreviated as $\lfloor v \rfloor$, in case of success.

Rule 4 rule shows the evaluation of an l-value expression in a r-value context. The expression is evaluated to its location $loc$, with final memory state $M'$. The value at location $loc$ in $M'$ is fetched using the `loadval` function (see section 2.3) and returned.

Expressions in l-value position:

$$\frac{E(x) = b}{G, E \vdash x^\tau, M \stackrel{l}{\Rightarrow} (b, 0), M} \quad (1) \qquad \frac{G, E \vdash a, M \Rightarrow \mathtt{Vptr}(loc), M'}{G, E \vdash (*a)^\tau, M \stackrel{l}{\Rightarrow} loc, M'} \quad (2)$$

Expressions in r-value position:

$$\frac{G, E \vdash a_1^{\tau_1}, M \Rightarrow v_1, M_1 \quad G, E \vdash a_2^{\tau_2}, M_1 \Rightarrow v_2, M_2 \quad \mathtt{eval\_binary\_operation}(op, v_1, \tau_1, v_2, \tau_2) = \lfloor v \rfloor}{G, E \vdash (a_1^{\tau_1}\, op\, a_2^{\tau_2})^\tau, M \Rightarrow v, M_2} \quad (3)$$

$$\frac{G, E \vdash a^\tau, M \stackrel{l}{\Rightarrow} loc, M' \quad \mathtt{loadval}(\tau, M', loc) = \lfloor v \rfloor}{G, E \vdash a^\tau, M \Rightarrow v, M'} \quad (4)$$

$$\frac{G, E \vdash a^\tau, M \stackrel{l}{\Rightarrow} loc, M_1 \quad G, E \vdash b^\sigma, M_1 \Rightarrow v_1, M_2 \quad \mathtt{cast}(v_1, \sigma, \tau) = \lfloor v \rfloor \quad \mathtt{storeval}(\tau, M_2, loc, v) = \lfloor M_3 \rfloor}{G, E \vdash (a^\tau \ \mathtt{=}\ b^\sigma)^\tau, M \Rightarrow v, M_3} \quad (5)$$

Statements:

$$G, E \vdash \mathtt{break}, M \Rightarrow \mathtt{Out\_break}, M \quad (6)$$

$$\frac{G, E \vdash s_1, M \Rightarrow \mathtt{Out\_normal}, M_1 \quad G, E \vdash s_2, M_1 \Rightarrow out, M_2}{G, E \vdash (s_1; s_2), M \Rightarrow out, M_2} \quad (7)$$

$$\frac{G, E \vdash s_1, M \Rightarrow out, M' \quad out \neq \mathtt{Out\_normal}}{G, E \vdash (s_1; s_2), M \Rightarrow out, M'} \quad (8)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M' \quad \mathtt{is\_false}(v)}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow \mathtt{Out\_normal}, M'} \quad (9)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M_1 \quad \mathtt{is\_true}(v) \quad G, E \vdash s, M_1 \Rightarrow \mathtt{Out\_break}, M_2}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow \mathtt{Out\_normal}, M_2} \quad (10)$$

$$\frac{G, E \vdash a, M \Rightarrow v, M_1 \quad \mathtt{is\_true}(v) \quad G, E \vdash s, M_1 \Rightarrow out, M_2 \quad out \in \{\mathtt{Out\_normal}, \mathtt{Out\_continue}\} \quad G, E \vdash (\mathtt{while}(a)\ s), M_2 \Rightarrow out', M_3}{G, E \vdash (\mathtt{while}(a)\ s), M \Rightarrow out', M_3} \quad (11)$$

**Fig. 3.** Selected rules of the dynamic semantics of Clight

Rule 5 evaluates an assignment expression. An assignment expression $a^\tau$ `=` $b^\sigma$ evaluates the l-value $a$ to a location $loc$, then the r-value $b$ to a value $v_1$. This value is cast from its natural type $\sigma$ to the expected type $\tau$ using the partial function `cast`. This function performs appropriate conversions, truncations and sign-extensions over integers and floats, and may fail for undefined casts. The result $v$ of the cast is then stored in memory at location $loc$, resulting in the

final memory state $M_3$, and returned as the value of the assignment expression.

The bottom group of rules in figure 3 are examples of statement executions. The execution of a break statement yields an `Out_break` outcome (rule 6). The execution of a sequence of two statements starts with the execution of the first statement, yielding an outcome that determines if the second statement must be executed or not (rules 7 and 8). Finally, rules 9–11 describe the execution of a `while` loop. Once the condition of a while loop is evaluated to a value $v$, the execution of the loop terminates normally if $v$ is false. If $v$ is true, the loop body is executed, yielding an outcome *out*. If *out* is `Out_break`, the loop terminates normally. If *out* is `Out_normal` or `Out_continue`, the whole loop is reexecuted in the memory state modified by the execution of the body.

### 2.3 The Memory Model of the Semantics

The memory model used in the dynamic semantics is described in [1]. It is a compromise between a low-level view of memory as an array of bytes and a high-level view as a mapping from abstract references to contents. In our model, the memory is arranged in independent blocks, identified by block references $b$. A memory state $M$ maps references $b$ to block contents, which are themselves mappings from byte offsets to values. Each block has a low bound $L(M, b)$ and a high bound $H(M, b)$, determined at allocation time and representing the interval of valid byte offsets within this block. This memory model guarantees separation properties between two distinct blocks, yet enables pointer arithmetic within a given block, as prescribed by the ISO C specification. The same memory model is common to the semantics of all intermediate languages of our certified compiler.

The memory model provides 4 basic operations:

$\texttt{alloc}(M, lo, hi) = (M', b)$
: Allocate a fresh block of bounds $[lo, hi)$. Returns extended memory $M'$ and reference $b$ to fresh block.

$\texttt{free}(M, b) = M'$
: Free (invalidate) the block $b$.

$\texttt{load}(\kappa, M, b, n) = \lfloor v \rfloor$
: Read one or several consecutive bytes (as determined by $\kappa$) at block $b$, offset $n$ in memory state $M$. If successful return the contents of these bytes as value $v$.

$\texttt{store}(\kappa, M, b, n, v) = \lfloor M' \rfloor$
: Store the value $v$ into one or several consecutive bytes (as determined by $\kappa$) at offset $n$ in block $b$ of memory state $M$. If successful, return an updated memory state $M'$.

The memory chunks $\kappa$ appearing in `load` and `store` operations describe concisely the size, type and signedness of the memory quantities involved:

| | |
|---|---|
| $\kappa ::=$ `Mint8signed` \| `Mint8unsigned` | |
| \| `Mint16signed` \| `Mint16unsigned` | small integers |
| \| `Mint32` | integers and pointers |
| \| `Mfloat32` \| `Mfloat64` | floats |

In the semantics of C, those quantities are determined by the C types of the datum being addressed. The following $\mathcal{A}$ ("access mode") function mediates between C types and the corresponding memory chunks:

$$\begin{aligned}
\mathcal{A}(\texttt{Tint}(\texttt{I8}, \texttt{Signed})) &= \texttt{By_value}(\texttt{Mint8signed}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I8}, \texttt{Unsigned})) &= \texttt{By_value}(\texttt{Mint8unsigned}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I16}, \texttt{Signed})) &= \texttt{By_value}(\texttt{Mint16signed}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I16}, \texttt{Unsigned})) &= \texttt{By_value}(\texttt{Mint16unsigned}) \\
\mathcal{A}(\texttt{Tint}(\texttt{I32}, _)) = \mathcal{A}(\texttt{Tpointer}(_)) &= \texttt{By_value}(\texttt{Mint32}) \\
\mathcal{A}(\texttt{Tarray}(_, _)) = \mathcal{A}(\texttt{Tfunction}(_, _)) &= \texttt{By_reference} \\
\mathcal{A}(\texttt{Tvoid}) &= \texttt{By_nothing}
\end{aligned}$$

Integer, float and pointer types involve an actual memory `load` when accessed, as captured by the `By_value` cases. However, accesses to arrays and functions return the location of the array or function, without any `load`; this is indicated by the `By_reference` access mode. Finally, expressions of type `void` cannot be accessed at all. This is reflected in the definitions of the `loadval` and `storeval` functions used in the dynamic semantics:

$$\begin{aligned}
\texttt{loadval}(\tau, M, (b, n)) &= \texttt{load}(\kappa, M, b, n) && \text{if } \mathcal{A}(\tau) = \texttt{By_value}(\kappa) \\
\texttt{loadval}(\tau, M, (b, n)) &= \lfloor b, n \rfloor && \text{if } \mathcal{A}(\tau) = \texttt{By_reference} \\
\texttt{loadval}(\tau, M, (b, n)) &= \texttt{None} && \text{if } \mathcal{A}(\tau) = \texttt{By_nothing} \\
\texttt{storeval}(\tau, M, (b, n), v) &= \texttt{store}(\kappa, M, b, n, v) && \text{if } \mathcal{A}(\tau) = \texttt{By_value}(\kappa) \\
\texttt{storeval}(\tau, M, (b, n), v) &= \texttt{None} && \text{otherwise}
\end{aligned}$$

### 2.4 Static Semantics (Typing Rules)

We have also formalized in Coq typing rules and a type checking algorithm for Clight. The algorithm is presented as a function from abstract syntax trees without type annotations to the abstract syntax trees with type annotations over expressions given in figure 1. We omit the typing rules by lack of space. Note that the dynamic semantics are defined for arbitrarily annotated expressions, not just well-typed expressions; however, the semantics can get stuck or produce results that disagree with ISO C when given an incorrectly-annotated expression. The translation scheme presented in section 3 demands well-typed programs and may fail to preserve semantics otherwise.

## 3 Translation from Clight to Cminor

### 3.1 Overview of Cminor

The Cminor language is the target language of our front-end compiler for C and the input language for our certified back-end. We now give a short overview of Cminor; see [8] for a more detailed description, and [7] for a complete formal specification.

Cminor is a low-level imperative language, structured like our subset of C into expressions, statements, and functions. We summarize the main differences with Clight. First, arithmetic operators are not overloaded and their behavior is independent of the static types of their operands. Distinct operators are provided for integer arithmetic and floating-point arithmetic. Conversions between integers and floats are explicit. Arithmetic is always performed over 32-bit integers and 64-bit floats; explicit truncation and sign-extension operators are provided to implement smaller integral types. Finally, the combined arithmetic-with-assignment operators of C (`+=`, `++`, etc) are not provided. For instance, the C expression `i += f` where `i` is of type `int` and `f` of type `double` is expressed as `i = intoffloat(floatofint(i) +`$_f$ `f)`.

Address computations are explicit, as well as individual load and store operations. For instance, the C expression `a[i]` where `a` is a pointer to `int` is expressed as `load(int32, a +`$_i$ `i *`$_i$ `4)`, making explicit the memory chunk being addressed (`int32`) as well as the computation of the address.

At the level of statements, Cminor has only 4 control structures: if-then-else conditionals, infinite loops, `block-exit`, and early return. The `exit` $n$ statement terminates the $(n + 1)$ enclosing `block` statements. These structures are lower-level than those of C, but suffice to implement all reducible flow graphs.

Within Cminor functions, local variables can only hold scalar values (integers, pointers, floats) and they do not reside in memory, making it impossible to take a pointer to a local variable like the C operator `&` does. Instead, each Cminor function declares the size of a stack-allocated block, allocated in memory at function entry and automatically freed at function return. The expression `addrstack`$(n)$ returns a pointer within that block at constant offset $n$. The Cminor producer can use this block to store local arrays as well as local scalar variables whose addresses need to be taken.[1]

The semantics of Cminor is defined in big-step operational style and resembles that of Clight. The following evaluation judgements are defined in [7]:

$$\begin{aligned} G, sp, L \vdash a, E, M &\to v, E', M' && \text{(expressions)}\\ G, sp, L \vdash a^*, E, M &\to v^*, E', M' && \text{(expression lists)}\\ G, sp \vdash s, E, M &\to out, E', M' && \text{(statements)}\\ G \vdash fn(v^*), M &\to v, M' && \text{(function calls)}\\ \vdash prog &\to v && \text{(whole programs)} \end{aligned}$$

The main difference with the semantics of Clight is that the local evaluation environment $E$ maps local variables to their values, instead of their memory addresses; consequently, $E$ is modified during evaluation of expressions and statements. Additional parameters are $sp$, the reference to the stack block for the current function, and $L$, the environment giving values to variables `let`-bound within expressions.

[1] While suboptimal in terms of performance of generated code, this systematic stack allocation of local variables whose addresses are taken is common practice for moderately optimizing C compilers such as `gcc` versions 2 and 3.

### 3.2 Overview of the Translation

The translation from our subset of Clight to Cminor performs three basic tasks:

- Resolution of operator overloading and explication of all type-dependent behaviors. Based on the types that annotate Clight expressions, the appropriate flavors (integer or float) of arithmetic operators are chosen; conversions between ints and floats, truncations and sign-extensions are introduced to reflect casts, both explicit in the source and implicit in the semantics of Clight; address computations are generated based on the types of array elements and pointer targets; and appropriate memory chunks are selected for every memory access.
- Translation of `while`, `do...while` and `for` loops into infinite loops with blocks and early exits. The statements `break` and `continue` are translated as appropriate `exit` constructs, as shown in figure 4.
- Placement of Clight variables, either as Cminor local variables (for local scalar variables whose address is never taken), sub-areas of the Cminor stack block for the current function (for local non-scalar variables or local scalar variables whose address is taken), or globally allocated memory areas (for global variables).[2]

The translation is specified as Coq functions from Clight abstract syntax to Cminor abstract syntax, defined by structural recursion. From these Coq functions, executable Caml code can be mechanically generated using the Coq extraction facility, making the specification directly executable. Several translation functions are defined: $\mathcal{L}$ and $\mathcal{R}$ for expressions in l-value and r-value position, respectively; $\mathcal{S}$ for statements; and $\mathcal{F}$ for functions. Some representative cases of the definitions of these functions are shown in figure 4, giving the general flavor of the translation.

The translation can fail when given invalid Clight source code, e.g. containing an assignment between arrays. To enable error reporting, the translation functions return option types: either `None` denoting an error, or $\lfloor x \rfloor$ denoting successful translation with result $x$. Systematic propagation of errors is achieved using a monadic programming style (the `bind` combinator of the error monad), as customary in purely functional programming. This monadic "plumbing" is omitted in figure 4 for simplicity.

Most translation functions are parameterized by a translation environment $\gamma$ reflecting the placement of Clight variables. It maps every variable $x$ to either `Local` (denoting the Cminor local variable named $x$), `Stack`($\delta$) (denoting a subarea of the Cminor stack block at offset $\delta$), or `Global` (denoting the address of the Cminor global symbol named $x$). This environment is constructed at the beginning of the translation of a Clight function. The function body is

[2] It would be semantically correct to stack-allocate all local variables, like the C0 verified compiler does [6,12]. However, keeping scalar local variables in Cminor local variables as much as possible enables the back-end to generate much more efficient machine code.

Casts ($\mathcal{C}^{\sigma}_{\tau}(e)$ casts $e$ from type $\tau$ to type $\sigma$):

$$\mathcal{C}^{\sigma}_{\tau}(e) = \mathcal{C}_2(\mathcal{C}_1(e,\tau,\sigma),\sigma)$$

$$\mathcal{C}_1(e,\tau,\sigma) = \begin{cases} \texttt{floatofint}(e), & \text{if } \tau = \texttt{Tint}(\_,\texttt{Signed}) \text{ and } \sigma = \texttt{Tfloat}(\_); \\ \texttt{floatofintu}(e), & \text{if } \tau = \texttt{Tint}(\_,\texttt{Unsigned}) \text{ and } \sigma = \texttt{Tfloat}(\_); \\ \texttt{intoffloat}(e), & \text{if } \tau = \texttt{Tfloat}(\_) \text{ and } \sigma = \texttt{Tint}(\_,\_); \\ e & \text{otherwise} \end{cases}$$

$$\mathcal{C}_2(e,\sigma) = \begin{cases} \texttt{cast8signed}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I8},\texttt{Signed}); \\ \texttt{cast8unsigned}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I8},\texttt{Unsigned}); \\ \texttt{cast16signed}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I16},\texttt{Signed}); \\ \texttt{cast16unsigned}(e), & \text{if } \sigma = \texttt{Tint}(\texttt{I16},\texttt{Unsigned}); \\ \texttt{singleoffloat}(e), & \text{if } \sigma = \texttt{Tfloat}(\texttt{F32}); \\ e & \text{otherwise} \end{cases}$$

Expressions in l-value position:

$$\mathcal{L}_\gamma(x) = \texttt{addrstack}(\delta) \quad \text{if } \gamma(x) = \texttt{Stack}(\delta)$$
$$\mathcal{L}_\gamma(x) = \texttt{addrglobal}(x) \quad \text{if } \gamma(x) = \texttt{Global}$$
$$\mathcal{L}_\gamma(\texttt{*}e) = \mathcal{R}_\gamma(e)$$
$$\mathcal{L}_\gamma(e_1[e_2]) = \mathcal{R}_\gamma(e_1 \texttt{ + } e_2)$$

Expressions in r-value position:

$$\mathcal{R}_\gamma(x) = x \quad \text{if } \gamma(x) = \texttt{Local}$$
$$\mathcal{R}_\gamma(e^\tau) = \texttt{load}(\kappa, \mathcal{L}_\gamma(e^\tau)) \quad \text{if } \mathcal{L}_\gamma(e) \text{ is defined and } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa)$$
$$\mathcal{R}_\gamma(e^\tau) = \mathcal{L}_\gamma(e^\tau) \quad \text{if } \mathcal{L}_\gamma(e) \text{ is defined and } \mathcal{A}(\tau) = \texttt{By\_reference}$$
$$\mathcal{R}_\gamma(x^\tau \texttt{ = } e^\sigma) = x \texttt{ = } \mathcal{C}^\tau_\sigma(\mathcal{R}(e^\sigma)) \quad \text{if } \gamma(x) = \texttt{Local}$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ = } e_2^\sigma) = \texttt{store}(\kappa, \mathcal{L}_\gamma(e_1^\tau), \mathcal{C}^\tau_\sigma(\mathcal{R}(e_2^\sigma))) \quad \text{if } \mathcal{A}(\tau) = \texttt{By\_value}(\kappa)$$
$$\mathcal{R}_\gamma(\texttt{\&}e) = \mathcal{L}_\gamma(e)$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ + } e_2^\sigma) = \mathcal{R}_\gamma(e_1^\tau) \texttt{ +}_i \mathcal{R}_\gamma(e_2^\sigma) \quad \text{if } \tau \text{ and } \sigma \text{ are integer types}$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ + } e_2^\sigma) = \mathcal{C}^{\texttt{double}}_\tau(\mathcal{R}_\gamma(e_1^\tau)) \texttt{ +}_f \mathcal{C}^{\texttt{double}}_\sigma(\mathcal{R}_\gamma(e_2^\sigma)) \quad \text{if } \tau \text{ or } \sigma \text{ are float types}$$
$$\mathcal{R}_\gamma(e_1^\tau \texttt{ + } e_2^\sigma) = \mathcal{R}_\gamma(e_1^\tau) \texttt{ +}_i \mathcal{R}_\gamma(e_2^\sigma) \texttt{ *}_i \texttt{ sizeof}(\rho) \quad \text{if } \tau \text{ is a pointer or array of } \rho$$

Statements:

$$\mathcal{S}_\gamma(\texttt{while}(e)\ s) = \texttt{block\{ loop\{ if } (\texttt{!}\mathcal{R}_\gamma(e)) \texttt{ exit 0; block\{ } \mathcal{S}_\gamma(s) \texttt{ \}\}\}}$$
$$\mathcal{S}_\gamma(\texttt{do } s \texttt{ while}(e)) = \texttt{block\{ loop\{ block\{ } \mathcal{S}_\gamma(s) \texttt{ \}; if } (\texttt{!}\mathcal{R}_\gamma(e)) \texttt{ exit 0 \}\}}$$
$$\mathcal{S}_\gamma(\texttt{for}(e_1;e_2;e_3)\ s) = \mathcal{R}_\gamma(e_1); \texttt{ block\{ loop\{ if } (\texttt{!}\mathcal{R}_\gamma(e_2)) \texttt{ exit 0; block\{ } \mathcal{S}_\gamma(s) \texttt{ \}; } \mathcal{R}_\gamma(e_3) \texttt{ \}\}}$$
$$\mathcal{S}_\gamma(\texttt{break}) = \texttt{exit 1}$$
$$\mathcal{S}_\gamma(\texttt{continue}) = \texttt{exit 0}$$

**Fig. 4.** Selected translation rules

scanned for occurrences of `&x` (taking the address of a variable). Local variables that are not scalar or whose address is taken are assigned $\texttt{Stack}(\delta)$ locations, with $\delta$ chosen so that distinct variables map to non-overlapping areas of the stack block. Other local variables are set to `Local`, and global variables to `Global`.

# 4 Proof of Correctness of the Translation

## 4.1 Relating Memory States

To prove the correctness of the translation, the major difficulty is to relate the memory states occurring during the execution of the Clight source code and that of the generated Cminor code. The semantics of Clight allocates a distinct block for every local variable at function entry. Some of those blocks (those for scalar variables whose address is not taken) have no correspondence in the Cminor memory state; others become sub-block of the Cminor stack block for the function.

To account for these differences in allocation patterns between the source and target code, we introduce the notion of *memory injections*. A memory injection $\alpha$ is a function from Clight block references $b$ to either `None`, meaning that this block has no counterpart in the Cminor memory state, or $\lfloor b', \delta \rfloor$, meaning that the block $b$ of the Clight memory state corresponds to a sub-block of block $b'$ at offset $\delta$ in the Cminor memory state.

A memory injection $\alpha$ defines a relation between Clight values $v$ and Cminor values $v'$, written $\alpha \vdash v \approx v'$ and defined as follows:

$$\alpha \vdash \texttt{Vint}(n) \approx \texttt{Vint}(n) \qquad \alpha \vdash \texttt{Vfloat}(n) \approx \texttt{Vfloat}(n) \qquad \alpha \vdash \texttt{Vundef} \approx v$$

$$\frac{\alpha(b) = \lfloor b', \delta \rfloor \qquad i' = i + \delta \pmod{2^{32}}}{\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')}$$

This relation captures the relocation of pointer values implied by $\alpha$. It also enables `Vundef` Clight values to become more defined Cminor values during the translation, in keeping with the general idea that compilation can particularize some undefined behaviors.

The memory injection $\alpha$ also defines a relation between Clight and Cminor memory states, written $\alpha \vdash M \approx M'$, consisting of the conjunction of the following conditions:

- Matching of block contents: if $\alpha(b) = \lfloor b', \delta \rfloor$ and $L(M, b) \leq i < H(M, b)$, then $L(M', b') \leq i + \delta < H(M', b')$ and $\alpha \vdash v \approx v'$ where $v$ is the contents of block $b$ at offset $i$ in $M$ and $v'$ the contents of $b'$ at offset $i'$ in $M'$.
- No overlap: if $\alpha(b_1) = \lfloor b'_1, \delta_1 \rfloor$ and $\alpha(b_2) = \lfloor b'_2, \delta_2 \rfloor$ and $b_1 \neq b_2$, then either $b'_1 \neq b'_2$, or the intervals $[L(M, b_1) + \delta_1, H(M, b_1) + \delta_1)$ and $[L(M, b_2) + \delta_2, H(M, b_2) + \delta_2)$ are disjoint.
- Fresh blocks: $\alpha(b) = \texttt{None}$ for all blocks $b$ not yet allocated in $M$.

The memory injection relations have nice commutation properties with respect to the basic operations of the memory model. For instance:

- Commutation of loads: if $\alpha \vdash M \approx M'$ and $\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')$ and $\texttt{load}(\kappa, M, b, i) = \lfloor v \rfloor$, there exists $v'$ such that $\texttt{load}(\kappa, M', b', i') = \lfloor v' \rfloor$ and $\alpha \vdash v \approx v'$.

- Commutation of stores to mapped blocks: if $\alpha \vdash M \approx M'$ and $\alpha \vdash \texttt{Vptr}(b, i) \approx \texttt{Vptr}(b', i')$ and $\alpha \vdash v \approx v'$ and $\texttt{store}(\kappa, M, b, i, v) = \lfloor M_1 \rfloor$, there exists $M_1'$ such that $\texttt{store}(\kappa, M', b', i', v') = \lfloor M_1' \rfloor$ and $\alpha \vdash M_1 \approx M_1'$.
- Invariance by stores to unmapped blocks: if $\alpha \vdash M \approx M'$ and $\alpha(b) = \texttt{None}$ and $\texttt{store}(\kappa, M, b, i, v) = \lfloor M_1 \rfloor$, then $\alpha \vdash M_1 \approx M'$.

To enable the memory injection $\alpha$ to grow incrementally as new blocks are allocated during execution, we define the relation $\alpha' \geq \alpha$ (read: $\alpha'$ extends $\alpha$) by $\forall b,\ \alpha'(b) = \alpha(b) \vee \alpha(b) = \texttt{None}$. The injection relations are preserved by extension of $\alpha$. For instance, if $\alpha \vdash M \approx M'$, then $\alpha' \vdash M \approx M'$ for all $\alpha'$ such that $\alpha' \geq \alpha$.

### 4.2 Relating Execution Environments

Execution environments differ in structure between Clight and Cminor: the Clight environment $E$ maps local variables to references of blocks containing the values of the variables, while in Cminor the environment $E'$ for local variables map them directly to values. We define a matching relation $\mathit{EnvMatch}(\gamma, \alpha, E, M, E', sp)$ between a Clight environment $E$ and memory state $M$ and a Cminor environment $E'$ and reference to a stack block $sp$ as follows:

- For all variables $x$ of type $\tau$, if $\gamma(x) = \texttt{Local}$, then $\alpha(E(x)) = \texttt{None}$ and there exists $v$ such that $\texttt{load}(\kappa(\tau), M, E(x), 0) = \lfloor v \rfloor$ and $\alpha \vdash v \approx E'(x)$.
- For all variables $x$ of type $\tau$, if $\gamma(x) = \texttt{Stack}(\delta)$, then $\alpha \vdash \texttt{Vptr}(E(x), 0) \approx \texttt{Vptr}(sp, \delta)$.
- For all $x \neq y$, we have $E(x) \neq E(y)$.
- If $\alpha(b) = \lfloor sp, \delta \rfloor$ for some block $b$ and offset $\delta$, then $b$ is in the range of $E$.

The first two conditions express the preservation of the values of local variables during compilation. The last two rule out unwanted sharing between environment blocks and their images through $\alpha$.

At any point during execution, several function calls may be active and we need to ensure matching between the environments of each call. For this, we introduce abstract call stacks, which are lists of 4-tuples $(\gamma, E, E', sp)$ and record the environments of all active functions. A call stack $cs$ is globally consistent with respect to C memory state $M$ and memory injection $\alpha$, written $\mathit{CallInv}(\alpha, M, cs)$, if $\mathit{EnvMatch}(\gamma, \alpha, E, M, E', sp)$ holds for all elements $(\gamma, E, E', sp)$ of $cs$. Additional conditions, omitted for brevity, enforce separation between Clight environments $E$ and between Cminor stack blocks $sp$ belonging to different function activations in $cs$.

### 4.3 Proof by Simulation

The proof of semantic preservation for the translation proceeds by induction over the Clight evaluation derivation and case analysis on the last evaluation rule used. The proof shows that, assuming suitable consistency conditions over the

abstract call stack, the generated Cminor expressions and statements evaluate in ways that simulate the evaluation of the corresponding Clight expressions and statements.

We give a slightly simplified version of the simulation properties shown by induction over the Clight evaluation derivation. Let $G'$ be the global Cminor environment obtained by translating all function definitions in the global Clight environment $G$. Assume $\mathit{CallInv}(\alpha, M, (\gamma, E, E', \mathit{sp}).\mathit{cs})$ and $\alpha \vdash M \approx M'$. Then there exists a Cminor environment $E'_1$, a Cminor memory state $M'_1$ and a memory injection $\alpha_1 \geq \alpha$ such that

- (R-values) If $G, E \vdash a, M \Rightarrow v, M_1$, there exists $v'$ such that $G', \mathit{sp}, L \vdash \mathcal{R}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash v \approx v'$.
- (L-values) If $G, E \vdash a, M \stackrel{l}{\Rightarrow} \mathit{loc}, M_1$, there exists $v'$ such that $G', \mathit{sp}, L \vdash \mathcal{L}_\gamma(a), E', M' \rightarrow v', E'_1, M'_1$ and $\alpha_1 \vdash \mathtt{Vptr}(\mathit{loc}) \approx v'$.
- (Statements) If $G, E \vdash s, M \Rightarrow \mathit{out}, M_1$ and $\tau_r$ is the return type of the function, there exists $\mathit{out}'$ such that $G', \mathit{sp} \vdash \mathcal{S}_\gamma(s), E', M' \rightarrow \mathit{out}', E'_1, M'_1$ and $\alpha_1, \tau_r \vdash \mathit{out} \approx \mathit{out}'$.

Moreover, the final Clight and Cminor states satisfy $\alpha_1 \vdash M_1 \approx M'_1$ and $\mathit{CallInv}(\alpha_1, M_1, (\gamma, E, E'_1, \mathit{sp}).\mathit{cs})$.

In the case of statements, the relation between Clight and Cminor outcomes is defined as follows:

$$\alpha, \tau_r \vdash \mathtt{Out\_normal} \approx \mathtt{Out\_normal} \qquad \alpha, \tau_r \vdash \mathtt{Out\_continue} \approx \mathtt{Out\_exit}(0)$$
$$\alpha, \tau_r \vdash \mathtt{Out\_break} \approx \mathtt{Out\_exit}(1) \qquad \alpha, \tau_r \vdash \mathtt{Out\_return} \approx \mathtt{Out\_return}$$

$$\frac{\alpha \vdash \mathtt{cast}(v, \tau, \tau_r) \approx v'}{\alpha, \tau_r \vdash \mathtt{Out\_return}(v, \tau) \approx \mathtt{Out\_return}(v')}$$

In addition to the outer induction over the Clight evaluation derivation, the proofs proceed by copious case analysis, over the placement $\gamma(x)$ for accesses to variables $x$, and over the types of the operands for applications of overloaded operators. As a corollary of the simulation properties, we obtain the correctness theorem for the translation:

**Theorem 1.** *Assume the Clight program $p$ is well-typed and translates without errors to a Cminor program $p'$. If $\vdash p \Rightarrow v$, and if $v$ is an integer or float value, then $\vdash p' \rightarrow v$.*

This semantic preservation theorem applies only to terminating programs. Our choice of big-step operational semantics prevents us from reasoning over non-terminating executions.

The whole proof represents approximately 6000 lines of Coq statements and proof scripts, including 1000 lines (40 lemmas) for the properties of memory injections, 1400 lines (54 lemmas) for environment matching and the call stack invariant, 1400 lines (50 lemmas) for the translations of type-dependent operators and memory accesses, and 2000 lines (51 lemmas, one per Clight evaluation rule) for the final inductive proof of simulation. By comparison, the source code

of the Clight to Cminor translator is 800 lines of Coq function definitions. The proof is therefore 7.5 times bigger than the code it proves. The whole development (design and semantics of Clight; development of the translator; proof of its correctness) took approximately 8 person.months.

## 5 Related Work

Several formal semantics of C-like languages have been defined. Norrish [10] gives a small-step operational semantics, expressed using the HOL theorem prover, for a subset of C comparable to our Clight. His semantics captures exactly the non-determinism (partially unspecified evaluation order) allowed by the ISO C specification, making it significantly more complex than our deterministic semantics. Papaspyrou [11] addresses non-determinism as well, but using denotational semantics with monads. Abstract state machines have been used to give on-paper semantics for C [4,9] and more recently for C# [3].

Many correctness proofs of program transformations have been published, both on paper and machine-checked using proof assistants; see [2] for a survey. A representative example is [5], where a non-optimizing byte-code compiler from a subset of Java to a subset of the Java Virtual Machine is verified using Isabelle/HOL. Most of these correctness proofs apply to source languages that are either smaller or semantically cleaner than C.

The work that is closest to ours is part of the Verisoft project [6,12]. Using Isabelle/HOL, they formalize the semantics of C0 (a subset of the C language) and a compiler from C0 down to DLX assembly code. C0 is a type-safe subset of C, close to Pascal, and significantly smaller than our Clight: there is no pointer arithmetic, nor side effects, nor premature execution of statements and there exists only a single integer type, thus avoiding operator overloading. They provide both a big step semantics and a small step semantics for C0, the latter enabling reasoning about non-terminating and concurrent executions, unlike our big-step semantics. Their C0 compiler is a single pass compiler that generates unoptimized machine code. It is more complex than our translation from Clight to Cminor, but considerably simpler than our whole certified compiler.

## 6 Concluding Remarks

The C language is not pretty; this shows up in the relative complexity of our formal semantics and translation scheme. However, this complexity remains manageable with the tools (the Coq proof assistant) and the methodology (big-step semantics; simulation arguments; extraction of an executable compiler from its functional Coq specification) that we used.

Future work includes 1- handling a larger subset of C, especially `struct` types; and 2- evaluating the usability of the semantics for program proof and static analysis purposes. In particular, it would be interesting to develop axiomatic semantics (probably based on separation logic) for Clight and validate them against our operational semantics.

## References


1. S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *Proc. of Int. Conf. on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 280–299, Manchester, UK, Nov. 2005. Springer-Verlag.
2. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
3. E. Börger, N. Fruja, V.Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2-3):235–284, 2005.
4. Y. Gurevich and J. Huggins. The semantics of the C programming language. In *Proc. of CSL'92 (Computer Science Logic)*, volume 702 of *LNCS*, pages 274–308. Springer Verlag, 1993.
5. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Mar. 2004. To appear in ACM TOPLAS.
6. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler. In *Proc. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 2–11, Koblenz, Germany, Sept. 2005. IEEE Computer Society Press.
7. X. Leroy. The Compcert certified compiler back-end – commented Coq development. Available on-line at `http://cristal.inria.fr/~xleroy`, 2006.
8. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. Symp. Principles Of Programming Languages (POPL)*, pages 42–54, Charleston, USA, Jan. 2006. ACM Press.
9. V. Nepomniaschy, I. Anureev, and A. Promsky. Verification-oriented language C-light and its structural operational semantics. In *Ershov Memorial Conference*, pages 103–111, 2003.
10. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, Dec. 1998.
11. N. Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, Feb. 1998.
12. M. Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, Apr. 2005.

J Autom Reasoning (2008) 41:1–31
DOI 10.1007/s10817-008-9099-0

# Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations

**Xavier Leroy · Sandrine Blazy**

Received: 26 September 2007 / Accepted: 13 February 2008 / Published online: 14 March 2008


**Abstract** This article presents the formal verification, using the Coq proof assistant, of a memory model for low-level imperative languages such as C and compiler intermediate languages. Beyond giving semantics to pointer-based programs, this model supports reasoning over transformations of such programs. We show how the properties of the memory model are used to prove semantic preservation for three passes of the Compcert verified compiler.

**Keywords** Memory model · C · Program verification · Compilation · Compiler correctness · The Coq proof assistant

## 1 Introduction

A prerequisite to the formal verification of computer programs—by model checking, program proof, static analysis, or any other means—is to formalize the semantics of the programming language in which the program is written, in a way that is exploitable by the verification tools used. In the case of program proofs, these formal semantics are often presented in operational or axiomatic styles, e.g. Hoare logic. The need for formal semantics is even higher when the program being verified itself operates over programs: compilers, program analyzers, etc. In the case of a compiler, for instance, no less than three formal semantics are required: one for the implementation language of the compiler, one for the source language, and one for

X. Leroy (✉)
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
e-mail: Xavier.Leroy@inria.fr

S. Blazy
ENSIIE, 1 square de la Résistance, 91025 Evry cedex, France
e-mail: Sandrine.Blazy@ensiie.fr

the target language. More generally speaking, formal semantics "on machine" (that is, presented in a form that can be exploited by verification tools) are an important aspect of formal methods.

Formal semantics are relatively straightforward in the case of declarative programming languages. However, many programs that require formal verification are written in imperative languages featuring pointers (or references) and in-place modification of data structures. Giving semantics to these imperative constructs requires the development of an adequate *memory model*, that is, a formal description of the memory store and operations over it. The memory model is often a delicate part of a formal semantics for an imperative programming language. A very concrete memory model (e.g. representing the memory as a single array of bytes) can fail to validate algebraic laws over loads and stores that are actually valid in the programming language, making program proofs more difficult. An excessively abstract memory model can fail to account for e.g. aliasing or partial overlap between memory areas, thus causing the semantics to be incorrect.

This article reports on the formalization and verification, using the Coq proof assistant, of a memory model for C-like imperative languages. C and related languages are challenging from the standpoint of the memory model, because they feature both pointers and pointer arithmetic, on the one hand, and isolation and freshness guarantees on the other. For instance, pointer arithmetic can result in aliasing or partial overlap between the memory areas referenced by two pointers; yet, it is guaranteed that the memory areas corresponding to two distinct variables or two successive calls to `malloc` are disjoint. This stands in contrast with both higher-level imperative languages such as Java, where two distinct references always refer to disjoint areas, and lower-level languages such as machine code, where unrestricted address arithmetic invalidates all isolation guarantees.

The memory model presented here is used in the formal verification of the Compcert compiler [3, 15], a moderately-optimizing compiler that translates the Clight subset of the C programming language down to PowerPC assembly code. The memory model is used by the formal semantics of all languages manipulated by the compiler: the source language, the target language, and 7 intermediate languages that bridge the semantic gap between source and target. Certain passes of the compiler perform non-trivial transformations on memory allocations and accesses: for instance, local variables of a Clight function, initially mapped to individually-allocated memory blocks, are at some point mapped to sub-blocks of a single stack-allocated activation record, which at a later point is extended to make room for storing spilled temporaries. Proving the correctness (semantic preservation) of these transformations requires extensive reasoning over memory states, using the properties of the memory model given further in the paper.

The remainder of this article is organized as follows. Section 2 axiomatizes the values that are stored in memory states and the associated memory data types. Section 3 specifies an abstract memory model and illustrates its use for reasoning over programs. Section 4 defines the concrete implementation of the memory model used in Compcert and shows that it satisfies both the abstract specification and additional useful properties. Section 5 describes the transformations over memory states performed by three passes of the Compcert compiler. It then defines the memory invariants and proves the simulation results between memory operations that play a crucial role in proving semantics preservation for these three passes.

Section 6 briefly comments on the Coq mechanization of these results. Related work is discussed in Section 7, followed by conclusions and perspectives in Section 8.

All results presented in this article have been mechanically verified using the Coq proof assistant [2, 8]. The complete Coq development is available online at http://gallium.inria.fr/~xleroy/memory-model/. Consequently, the paper only sketches the proofs of some of its results; the reader is referred to the Coq development for the full proofs.

## 2 Values and Data Types

We assume given a set `val` of values, ranged over by $v$, used in the dynamic semantics of the languages to represent the results of calculations. In the Compcert development, `val` is defined as the discriminated union of 32-bit integers `int`($n$), 64-bit double-precision floating-point numbers `float`($f$), memory locations `ptr`($b, i$) where $b$ is a memory block reference $b$ and $i$ a byte offset within this block, and the constant `undef` representing an undefined value such as the value of an uninitialized variable.

We also assume given a set `memtype` of memory data types, ranged over by $\tau$. Every memory access (`load` or `store` operation) takes as argument a memory data type, serving two purposes: (1) to indicate the size and natural alignment of the data being accessed, and (2) to enforce compatibility guarantees between the type with which a data was stored and the type with which it is read back. For a semantics for C, we can use C type expressions from the source language as memory data types. For the Compcert intermediate languages, we use the following set of memory data types, corresponding to the data that the target processor can access in one `load` or `store` instruction:

| | |
|---|---|
| $\tau ::=$ `int8signed` \| `int8unsigned` | 8-bit integers |
| \| `int16signed` \| `int16unsigned` | 16-bit integers |
| \| `int32` | 32-bit integers or pointers |
| \| `float32` | 32-bit, single-precision floats |
| \| `float64` | 64-bit, double-precision floats |

The first role of a memory data type $\tau$ is to determine the size $|\tau|$ in bytes that a data of type $\tau$ occupies in memory, as well as the natural alignment $\langle\tau\rangle$ for data of this type. The alignment $\langle\tau\rangle$ models the address alignment constraints that many processors impose, e.g., the address of a 32-bit integer must be a multiple of 4. Both size and alignment are positive integers.[1,2]

(A1) $|\tau| > 0$ and $\langle\tau\rangle > 0$

[1] In this article, we write A for axioms, that is, assertions that we will not prove; S for specifications, that is, expected properties of the abstract memory model which the concrete model, as well as any other implementation, satisfies; D for derived properties, provable from the specifications; and P for properties of the concrete memory model.

[2] Throughout this article, variables occurring free in mathematical statements are implicitly universally quantified at the beginning of the statement.

To reason about some memory transformations, it is useful to assume that there exists a maximal alignment `max_alignment` that is a multiple of all possible alignment values:

(A2) $\langle\tau\rangle$ divides `max_alignment`

For the semantics of C, $|\tau|$ is the size of the type $\tau$ as returned by the `sizeof` operator of C. A possible choice for $\langle\tau\rangle$ is the size of the largest scalar type occurring in $\tau$. For the Compcert intermediate languages, we take:

$$|\texttt{int8signed}| = |\texttt{int8unsigned}| = 1$$
$$|\texttt{int16signed}| = |\texttt{int16unsigned}| = 2$$
$$|\texttt{int32}| = |\texttt{float32}| = 4$$
$$|\texttt{float64}| = 8$$

Concerning alignments, Compcert takes $\langle\tau\rangle = 1$ and $\texttt{max\_alignment} = 1$, since the target architecture (PowerPC) has no alignment constraints. To model a target architecture with alignment constraints such as the Sparc, we would take $\langle\tau\rangle = |\tau|$ and $\texttt{max\_alignment} = 8$.

We now turn to the second role of memory data types, namely a form of dynamic type-checking. For a strongly-typed language, a memory state is simply a partial mapping from memory locations to values: either the language is statically typed, guaranteeing at compile-time that a value written with type $\tau$ is always read back with type $\tau$; or the language is dynamically typed, in which case the generated machine code contains enough run-time type tests to enforce this property. However, the C language and most compiler intermediate languages are weakly typed. Consider a C "union" variable:

```
union { int i; float f; } u;
```

It is possible to assign an integer to `u.i`, then read it back as a float via `u.f`. This will not be detected at compile-time, and the C compiler will not generate code to prevent this. Yet, the C standard [13] specifies that this code has undefined behavior. More generally, after writing a data of type $\tau$ to a memory location, this location can only be read back with the same type $\tau$ or a compatible type; the behavior is undefined otherwise [13, Section 6.5, items 6 and 7]. To capture this behavior in a formal semantics for C, the memory state associates type-value pairs $(\tau, v)$, and not just values, to locations. Every `load` with type $\tau'$ at this location will check compatibility between the actual type $\tau$ of the location and the expected type $\tau'$, and fail if they are not compatible.

We abstract this notion of compatibility as a relation $\tau \sim \tau'$ between types. We assume that a type is always compatible with itself, and that compatible types have the same size and the same alignment:

(A3) $\tau \sim \tau$
(A4) If $\tau_1 \sim \tau_2$, then $|\tau_1| = |\tau_2|$ and $\langle\tau_1\rangle = \langle\tau_2\rangle$

Several definitions of the $\sim$ relation are possible, leading to different instantiations of our memory model. In the strictest instantiation,, $\tau \sim \tau'$ holds only if $\tau = \tau'$; that is, no implicit casts are allowed during a `store-load` sequence. The C standard

actually permits some such casts [13, Section 6.5, item 7]. For example, an integer $n$ can be stored as an `unsigned char`, then reliably read back as a `signed char`, with result `(signed char)` $n$. This can be captured in our framework by stating that `unsigned char` $\sim$ `signed char`. For the Compcert intermediate languages, we go one step further and define $\tau_1 \sim \tau_2$ as $|\tau_1| = |\tau_2|$.

To interpret implicit casts in a `store-load` sequence, we need a function `convert` : `val` $\times$ `memtype` $\to$ `val` that performs these casts. More precisely, writing a value $v$ with type $\tau$, then reading it back with a compatible type $\tau'$ results in value `convert`$(v, \tau')$. For the strict instantiation of the model, we take `convert`$(v, \tau') = v$. For the interpretation closest to the C standard, we need `convert`$(v, \tau') = (\tau')\ v$, where the right-hand side denotes a C type cast. Finally, for the Compcert intermediate languages, `convert` is defined as:

$$
\begin{aligned}
\texttt{convert}(\texttt{int}(n), \texttt{int8unsigned}) &= \texttt{int}(\text{8-bit zero extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int8signed}) &= \texttt{int}(\text{8-bit sign extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int16unsigned}) &= \texttt{int}(\text{16-bit zero extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int16signed}) &= \texttt{int}(\text{16-bit sign extension of } n)\\
\texttt{convert}(\texttt{int}(n), \texttt{int32}) &= \texttt{int}(n)\\
\texttt{convert}(\texttt{ptr}(b, i), \texttt{int32}) &= \texttt{ptr}(b, i)\\
\texttt{convert}(\texttt{float}(f), \texttt{float32}) &= \texttt{float}(f \text{ normalized to single precision})\\
\texttt{convert}(\texttt{float}(f), \texttt{float64}) &= \texttt{float}(f)\\
\texttt{convert}(v, \tau) &= \texttt{undef} \quad \text{in all other cases}
\end{aligned}
$$

Note that this definition of `convert`, along with the fact that $\tau_1 \not\sim \tau_2$ if $|\tau_1| \neq |\tau_2|$, ensures that low-level implementation details such as the memory endianness or the bit-level representation of floats cannot be observed by Compcert intermediate programs. For instance, writing a float $f$ with type `float32` and reading it back with compatible type `int32` results in the undefined value `undef` and not in the integer corresponding to the bit-pattern for $f$.

## 3 Abstract Memory Model

We now give an abstract, incomplete specification of a memory model that attempts to formalize the memory-related aspects of C and related languages. We have an abstract type `block` of references to memory blocks, and an abstract type `mem` of memory states. Intuitively, we view the memory state as a collection of separated blocks, identified by block references $b$. Each block behaves like an array of bytes, and is addressed using byte offsets $i \in \mathbb{Z}$. A memory location is therefore a pair $(b, i)$ of a block reference $b$ and an offset $i$ within this block. The constant `empty` : `mem`

represents the initial memory state. Four operations over memory states are provided as total functions:

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

Option types are used to represent potential failures. A value of type $\texttt{option}\,t$ is either $\varepsilon$ (pronounced "none"), denoting failure, or $\lfloor x \rfloor$ (pronounced "some $x$"), denoting success with result $x : t$.

Allocation of a fresh memory block is written $\texttt{alloc}(m, l, h)$, where $m$ is the initial memory state, and $l \in \mathbb{Z}$ and $h \in \mathbb{Z}$ are the low and high bounds for the fresh block. The allocated block has size $h - l$ bytes and can be accessed at byte offsets $l, l + 1, \ldots, h - 2, h - 1$. In other terms, the low bound $l$ is inclusive but the high bound $h$ is exclusive. Allocation can fail and return $\varepsilon$ if not enough memory is available. Otherwise, $\lfloor b, m' \rfloor$ is returned, where $b$ is the reference to the new block and $m'$ the updated memory state.

Conversely, $\texttt{free}(m, b)$ deallocates block $b$ in memory $m$. It can fail if e.g., $b$ was already deallocated. In case of success, an updated memory state is returned.

Reading from memory is written $\texttt{load}(\tau, m, b, i)$. A data of type $\tau$ is read from block $b$ of memory state $m$ at byte offset $i$. If successful, the value thus read is returned. The memory state is unchanged.

Symmetrically, $\texttt{store}(\tau, m, b, i, v)$ writes value $v$ at offset $i$ in block $b$ of $m$. If successful, the updated memory state is returned.

We now axiomatize the expected properties of these operations. The properties are labeled S to emphasize that they are specifications that any implementation of the model must satisfy. The first hypotheses are "good variable" properties defining the behavior of a `load` following an `alloc`, `free` or `store` operation.

(S5) If $\texttt{alloc}(m,l,h) = \lfloor b, m' \rfloor$ and $b' \neq b$, then $\texttt{load}(\tau, m', b', i) = \texttt{load}(\tau, m, b', i)$

(S6) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $\texttt{load}(\tau, m', b', i) = \texttt{load}(\tau, m, b', i)$

(S7) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $\tau \sim \tau'$, then $\texttt{load}(\tau', m', b, i) = \texttt{convert}(v, \tau')$

(S8) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $b' \neq b \ \vee\ i' + |\tau'| \leq i \ \vee\ i + |\tau| \leq i'$, then $\texttt{load}(\tau', m', b', i') = \texttt{load}(\tau', m, b', i')$

Hypotheses S5 and S6 state that allocating a block $b$ or freeing a block $b$ preserves loads performed in any other block $b' \neq b$. Hypothesis S7 states that after writing value $v$ with type $\tau$ at offset $i$ in block $b$, reading from the same location with a compatible type $\tau'$ succeeds and returns the value $\texttt{convert}(v, \tau')$. Hypothesis S8 states that storing a value of type $\tau$ in block $b$ at offset $i$ commutes with loading a value of type $\tau'$ in block $b'$ at offset $i'$, provided the memory areas corresponding to the `store` and the `load` are separate: either $b' \neq b$, or the range $[i, i + |\tau|)$ of byte offsets modified by the `store` is disjoint from the range $[i', i' + |\tau'|)$ read by the `load`.

Note that the properties above do not fully specify the `load` operation: nothing can be proved about the result of loading from a freshly allocated block, or freshly

deallocated block, or just after a store with a type and location that do not fall in the S7 and S8 case. This under-specification is intentional and follows the C standard. The concrete memory model of Section 4 will fully specify these behaviors.

The "good variable" properties use hypotheses $b' \neq b$, that is, separation properties between blocks. To establish such properties, we axiomatize the relation $m \models b$, meaning that the block reference $b$ is valid in memory $m$. Intuitively, a block reference is valid if it was previously allocated but not yet deallocated; this is how the $m \models b$ relation will be defined in Section 4.

(S9) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $\neg(m \models b)$.
(S10) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m' \models b' \Leftrightarrow b' = b \vee m \models b'$
(S11) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \models b' \Leftrightarrow m \models b'$
(S12) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $m' \models b' \Leftrightarrow m \models b'$
(S13) If $m \models b$, then there exists $m'$ such that $\texttt{free}(m, b) = \lfloor m' \rfloor$.

Hypothesis S9 says that the block returned by `alloc` is fresh, i.e., distinct from any other block that was valid in the initial memory state. Hypothesis S10 says that the newly allocated block is valid in the final memory state, as well as all blocks that were valid in the initial state. Block validity is preserved by `store` operations (S11). After a $\texttt{free}(m, b)$ operation, all initially valid blocks other than $b$ remain valid, but it is unspecified whether the deallocated block $b$ is valid or not (S12). Finally, the `free` operation is guaranteed to succeed when applied to a valid block (S13).

The next group of hypotheses axiomatizes the function $\mathcal{B}(m, b)$ that associates low and high bounds $l, h$ to a block $b$ in memory state $m$. We write $\mathcal{B}(m, b) = [l, h)$ to emphasize the meaning of bounds as semi-open intervals of allowed byte offsets within block $b$.

(S14) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $\mathcal{B}(m', b) = [l, h)$.
(S15) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $b' \neq b$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.
(S16) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.
(S17) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.

A freshly allocated block has the bounds that were given as argument to the `alloc` function (S14). The bounds of a block $b'$ are preserved by an `alloc`, `store` or `free` operation, provided $b'$ is not the block being allocated or deallocated.

For convenience, we write $\mathcal{L}(m, b)$ and $\mathcal{H}(m, b)$ for the low and high bounds attached to $b$, respectively, so that $\mathcal{B}(m, b) = [\mathcal{L}(m, b), \mathcal{H}(m, b))$.

Combining block validity with bound information, we define the "valid access" relation $m \models \tau \ @ \ b, i$, meaning that in state $m$, it is valid to write with type $\tau$ in block $b$ at offset $i$.

$$m \models \tau \ @ \ b, i \stackrel{\text{def}}{=} m \models b \ \wedge \ \langle \tau \rangle \text{ divides } i \ \wedge \ \mathcal{L}(m, b) \leq i \ \wedge \ i + |\tau| \leq \mathcal{H}(m, b)$$

In other words, $b$ is a valid block, the range $[i, i + |\tau|)$ of byte offsets being accessed is included in the bounds of $b$, and the offset $i$ is an integer multiple of the alignment $\langle \tau \rangle$. If these conditions hold, we impose that the corresponding `store` operation succeeds.

(S18) If $m \models \tau \ @ \ b, i$ then there exists $m'$ such that $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$.

Here are some derived properties of the valid access relation, easily provable from the hypotheses above.

(D19) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $\langle \tau \rangle$ divides $i$ and $l \leq i$ and $i + |\tau| \leq h$, then $m' \models \tau \ @ \ b, i$.
(D20) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $m \models \tau \ @ \ b', i$, then $m' \models \tau \ @ \ b', i$.
(D21) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \models \tau \ @ \ b', i \ \Leftrightarrow \ m \models \tau \ @ \ b', i$.
(D22) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $m' \models \tau \ @ \ b', i \ \Leftrightarrow \ m \models \tau \ @ \ b', i$.

*Proof* D19 follows from S10 and S14. D20 follows from S10 and S15, noticing that $b' \neq b$ by S9. D21 follows from S11 and S16, and D22 from S12 and S17. □

To finish this section, we show by way of an example that the properties axiomatized above are sufficient to reason over the behavior of a C pointer program using axiomatic semantics. Consider the following C code fragment:

```
int * x = malloc(2 * sizeof(int));
int * y = malloc(sizeof(int));
x[0]    = 0;
x[1]    = 1;
*y      = x[0];
x[0]    = x[1];
x[1]    = *y;
```

We would like to show that in the final state, `x[0]` is 1 and `x[1]` is 0. Assuming that errors are automatically propagated using a monadic interpretation, we can represent the code fragment above as follows, using the operations of the memory model to make explicit memory operations. The variable `m` holds the current memory state. We also annotate the code with logical assertions expressed in terms of the memory model. The notation $\Gamma$ stands for the three conditions $\texttt{x} \neq \texttt{y}$, $\texttt{m} \models \texttt{x}$, $\texttt{m} \models \texttt{y}$.

```
(x, m) = alloc(m, 0, 8);
         /* m ⊨ x */
(y, m) = alloc(m, 0, 4);
         /* Γ */
m = store(int, x, 0, 0);
         /* Γ, load(m, x, 0) = ⌊0⌋ */
m = store(int, x, 4, 1);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋ */
t = load(int, x, 0);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋, t = 0 */
m = store(int, y, 0, t);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋,
                                          load(m, y, 0) = ⌊0⌋ */
t = load(int, x, 4);
         /* Γ, load(m, x, 0) = ⌊0⌋, load(m, x, 4) = ⌊1⌋,
                                  load(m, y, 0) = ⌊0⌋, t = 1 */
m = store(int, x, 0, t);
         /* Γ, load(m, x, 0) = ⌊1⌋, load(m, x, 4) = ⌊1⌋,
                                          load(m, y, 0) = ⌊0⌋ */
```

```
t = load(int, y, 0);
          /* Γ, load(m, x, 0) = ⌊1⌋, load(m, x, 4) = ⌊1⌋,
                                          load(m, y, 0) = ⌊0⌋, t = 0 */
m = store(int, x, 4, t);
          /* Γ, load(m, x, 0) = ⌊1⌋, load(m, x, 4) = ⌊0⌋,
                                          load(m, y, 0) = ⌊0⌋ */
```

Every postcondition can be proved from its precondition using the hypotheses listed in this section. The validity of blocks `x` and `y`, as well as the inequality $\texttt{x} \neq \texttt{y}$, follow from S9, S10 and S11. The assertions over the results of `load` operations and over the value of the temporary `t` follow from the good variable properties S7 and S8. Additionally, we can show that the `store` operations do not fail using S18 and the additional invariants $\texttt{m} \models \texttt{int} \mathbin{@} \texttt{x}, 0$ and $\texttt{m} \models \texttt{int} \mathbin{@} \texttt{x}, 4$ and $\texttt{m} \models \texttt{int} \mathbin{@} \texttt{y}, 0$, which follow from D19, D20 and D21.

## 4 Concrete Memory Model

We now develop a concrete implementation of a memory model that satisfies the axiomatization in Section 3. The type `block` of memory block references is implemented by the type $\mathbb{N}$ of nonnegative integers. Memory states (type `mem`) are quadruples $(N, B, F, C)$, where

- $N : \texttt{block}$ is the first block not yet allocated;
- $B : \texttt{block} \to \mathbb{Z} \times \mathbb{Z}$ associates bounds to each block reference;
- $F : \texttt{block} \to \texttt{boolean}$ says, for each block, whether it has been deallocated (`true`) or not (`false`);
- $C : \texttt{block} \to \mathbb{Z} \to \texttt{option}\ (\texttt{memtype} \times \texttt{val})$ associates a content to each block $b$ and each byte offset $i$. A content is either $\varepsilon$, meaning "invalid", or $\lfloor \tau, v \rfloor$, meaning that a value $v$ was stored at this location with type $\tau$.

We define block validity $m \models b$, where $m = (N, B, F, C)$, as $b < N \ \wedge \ F(b) = \texttt{false}$, that is, $b$ was previously allocated ($b < N$) but not previously deallocated ($F(b) = \texttt{false}$). Similarly, the bounds $\mathcal{B}(m, b)$ are defined as $B(b)$.

The definitions of the constant `empty` and the operations `alloc`, `free`, `load` and `store` follow. We write $m = (N, B, F, C)$ for the initial memory state.

$$
\begin{aligned}
&\texttt{empty} = \\
&\quad (0,\ \lambda b.\,[0, 0),\ \lambda b.\,\texttt{false},\ \lambda b.\lambda i.\,\varepsilon) \\
&\texttt{alloc}(m, l, h) = \\
&\quad \text{if } \texttt{can\_allocate}(m, h - l) \text{ then } \lfloor b, m' \rfloor \text{ else } \varepsilon \\
&\quad \text{where } b = N \\
&\quad \text{and } m' = (N + 1, B\{b \leftarrow [l, h)\}, F\{b \leftarrow \texttt{false}\}, C\{b \leftarrow \lambda i.\,\varepsilon\}) \\
&\texttt{free}(m, b) = \\
&\quad \text{if not } m \models b \text{ then } \varepsilon \\
&\quad \text{else } \lfloor N, B\{b \leftarrow [0, 0)\}, F\{b \leftarrow \texttt{true}\}, C \rfloor \\
&\texttt{store}(\tau, m, b, i, v) = \\
&\quad \text{if not } m \models \tau \mathbin{@} b, i \text{ then } \varepsilon \\
&\quad \text{else } \lfloor N, B, F, C\{b \leftarrow c'\} \rfloor \\
&\quad \text{where } c' = C(b)\{i \leftarrow \lfloor \tau, v \rfloor, i + 1 \leftarrow \varepsilon, \ldots, i + |\tau| - 1 \leftarrow \varepsilon\}
\end{aligned}
$$

```
load(τ, m, b, i) =
  if not m ⊨ τ @ b, i then ε
  else if C(b)(i) = ⌊τ′, v⌋ and τ′ ∼ τ
      and C(b)(i + j) = ε for j = 1, ..., |τ| − 1
  then ⌊convert(v, τ)⌋
  else ⌊undef⌋
```

Allocation is performed by incrementing linearly the $N$ component of the memory state. Block identifiers are never reused, which greatly facilitates reasoning over "dangling pointers" (references to blocks previously deallocated). The new block is given bounds $[l, h)$, deallocated status `false`, and invalid contents $\lambda i.\ \varepsilon$.[3] An unspecified, boolean-valued `can_allocate` function is used to model the possibility of failure if the request ($h - l$ bytes) exceeds the available memory. In the Compcert development, `can_allocate` always returns `true`, therefore modeling an infinite memory.

Freeing a block simply sets its deallocated status to `true`, rendering this block invalid, and its bounds to $[0, 0)$, reflecting the fact that this block no longer occupies any memory space.

A memory store first checks block and bounds validity using the $m \models \tau\ @\ b, i$ predicate, which is decidable. The contents $C(b)$ of block $b$ are set to $\lfloor \tau, v \rfloor$ at offset $i$, recording the store done at this offset, and to $\varepsilon$ at offsets $i + 1, \ldots, i + |\tau| - 1$, invalidating whatever data was previously stored at these addresses.

A memory load checks several conditions: first, that the block and offset being addressed are valid and within bounds; second, that block $b$ at offset $i$ contains a valid data $\lfloor v, \tau' \rfloor$; third, that the type $\tau'$ of this data is compatible with the requested type $\tau$; fourth, that the contents of offsets $i + 1$ to $i + |\tau| - 1$ in block $b$ are invalid, ensuring that the data previously stored at $i$ in $b$ was not partially overwritten by a store at an overlapping offset.

It is easy to show that this implementation satisfies the specifications given in Section 3.

**Lemma 1** *Properties S5 to S18 are satisfied.*

*Proof* Most properties follow immediately from the definitions of `alloc`, `free`, `load` and `store` given above. For the "good variable" property S7, the `store` assigned contents $\lfloor \tau, v \rfloor, \varepsilon, \ldots, \varepsilon$ to offsets $i, \ldots, i + |\tau| - 1$, respectively. Since $|\tau'| = |\tau|$ by A1, the checks performed by `load` succeed. For the other "good variable" property, S8, the assignments performed by the `store` over $C(b)$ at offsets $i, \ldots, i + |\tau| - 1$ have no effect over the values of offsets $i', \ldots, i' + |\tau'| - 1$ in $C(b')$, given the separation hypothesis $(b' \neq b \ \vee\ i' + |\tau'| \leq i \ \vee\ i + |\tau| \leq i')$. □

Moreover, the implementation also enjoys a number of properties that we now state. In the following sections, we will only use these properties along with those

[3] Since blocks are never reused, the freshly-allocated block $b$ already has deallocated status `false` and contents $\lambda i.\ \varepsilon$ in the initial memory state $(N, B, F, C)$. Therefore, in the definition of `alloc`, the updates $F\{b \leftarrow \texttt{false}\}$ and $C\{b \leftarrow \lambda i.\ \varepsilon\}$ are not strictly necessary. However, they allow for simpler reasoning over the `alloc` function, making it unnecessary to prove the invariants $F(b) = \texttt{false}$ and $C(b) = \lambda i.\ \varepsilon$ for all $b \geq N$.

of Section 3, but not the precise definitions of the memory operations. The first two properties state that a `store` or a `load` succeeds if and only if the corresponding memory reference is valid.

(P24) $m \models \tau \ @ \ b, i \ \Leftrightarrow \ \exists m', \texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$
(P25) $m \models \tau \ @ \ b, i \ \Leftrightarrow \ \exists v, \texttt{load}(\tau, m, b, i) = \lfloor v \rfloor$

Then come additional properties capturing the behavior of a `load` following an `alloc` or a `store`. In circumstances where the "good variable" properties of the abstract memory model leave unspecified the result of the `load`, these "not-so-good variable" properties guarantee that the `load` predictably returns $\lfloor \texttt{undef} \rfloor$.

(P26) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $\texttt{load}(\tau, m', b, i) = \lfloor v \rfloor$, then $v = \texttt{undef}$.
(P27) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $\tau \not\sim \tau'$ and $\texttt{load}(\tau', m', b, i) = \lfloor v' \rfloor$, then $v' = \texttt{undef}$.
(P28) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$ and $\texttt{load}(\tau', m', b, i') = \lfloor v' \rfloor$, then $v' = \texttt{undef}$.

*Proof* For P26, the contents of $m'$ at $b, i$ are $\varepsilon$ and therefore not of the form $\lfloor \tau, v \rfloor$. For P27, the test $\tau \sim \tau'$ in the definition of `load` fails. For P28, consider the contents $c$ of block $b$ in $m'$. If $i < i'$, the `store` set $c(i') = \varepsilon$. If $i > i'$, the `store` set $c(i' + j) = \lfloor \tau, v \rfloor$ for some $j \in [1, |\tau'|)$. In both cases, one of the checks in the definition of `load` fails. □

Combining properties S7, S8, P25, P27 and P28, we obtain a complete characterization of the behavior of a `load` that follows a `store`. (See Fig. 1 for a graphical illustration of the cases.)

(D29) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $m \models \tau' \ @ \ b', i'$, then one and only one of the following four cases holds:

- Compatible: $b' = b$ and $i' = i$ and $\tau \sim \tau'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{convert}(v, \tau') \rfloor$.
- Incompatible: $b' = b$ and $i' = i$ and $\tau \not\sim \tau'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{undef} \rfloor$.
- Disjoint: $b' \neq b$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$, in which case $\texttt{load}(\tau', m', b', i') = \texttt{load}(\tau', m, b', i')$.
- Overlapping: $b' = b$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{undef} \rfloor$.

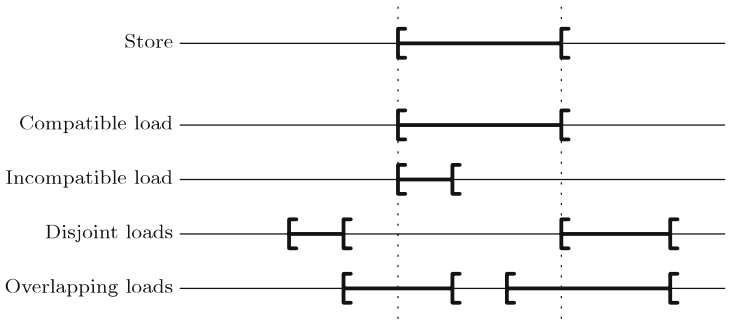



**Fig. 1** A `store` followed by a `load` in the same block: the four cases of property D29

As previously mentioned, an interesting property of the concrete memory model is that `alloc` never reuses block identifiers, even if some blocks have been deallocated before. To account for this feature, we define the relation $m \# b$, pronounced "block $b$ is fresh in memory $m$", and defined as $b \geq N$ if $m = (N, B, F, C)$. This relation enjoys the following properties:

(P30) $m \# b$ and $m \models b$ are mutually exclusive.
(P31) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m \# b$.
(P32) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m' \# b' \Leftrightarrow b' \neq b \wedge m \# b'$.
(P33) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \# b' \Leftrightarrow m \# b'$.
(P34) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $m' \# b' \Leftrightarrow m \# b'$.

Using the freshness relation, we say that two memory states $m_1$ and $m_2$ have the same domain, and write $\mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$, if $\forall b, (m_1 \# b \Leftrightarrow m_2 \# b)$. In our concrete implementation, two memory states have the same domain if and only if their $N$ components are equal. Therefore, `alloc` is deterministic with respect to the domain of the current memory state: `alloc` chooses the same free block when applied twice to memory states that have the same domain, but may differ in block contents.

(P35) If $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ and $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$ and $\mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$, then $b_1 = b_2$ and $\mathrm{Dom}(m_1') = \mathrm{Dom}(m_2')$.

The last property of the concrete implementation used in the remainder of this paper is the following: a block $b$ that has been deallocated is both invalid and empty, in the sense that its low and high bounds are equal.

(P36) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $\neg(m' \models b)$.
(P37) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $\mathcal{L}(m', b) = \mathcal{H}(m', b)$.

## 5 Memory Transformations

We now study the use of the concrete memory model to prove the correctness of program transformations as performed by compiler passes. Most passes of the Compcert compiler preserve the memory behavior of the program: some modify the flow of control, others modify the flow of data not stored in memory, but the memory states before and after program transformation match at every step of the program execution. The correctness proofs for these passes exploit none of the properties of the memory model. However, three passes of the Compcert compiler change the memory behavior of the program, and necessitate extensive reasoning over memory states to be proved correct. We now outline the transformations performed by these three passes.

The first pass that modifies the memory behavior is the translation from the source language Clight to the intermediate language Cminor. In Clight, all variables are allocated in memory: the evaluation environment maps variables to references of memory blocks that contain the current values of the variables. This is consistent with the C specification and the fact that the address of any variable can be taken and used as a memory pointer using the `&` operator. However, this feature renders register allocation and most other optimizations very difficult, because aliasing between a pointer and a variable is always possible.

Therefore, the Clight to Cminor translation detects scalar local variables whose address is never taken with the `&` operator, and "pulls them out of memory": they become Cminor local variables, whose current values are recorded in an environment separate from the memory state, and whose address cannot be taken. Other Clight local variables remain memory-allocated, but are grouped as sub-areas of a single memory block, the Cminor stack block, which is automatically allocated at function entry and deallocated at function exit. (See Fig. 2, left.)

Consequently, the memory behavior of the source Clight program and the transformed Cminor program differ greatly: when the Clight program allocates $N$ fresh blocks at function entry for its $N$ local variables, the Cminor program allocates only one; the `load` and `store` operations performed by the Clight semantics every time a local variable is accessed either disappear in Cminor or becomes `load` and `store` in sub-areas of the Cminor stack block.

The second pass that affects memory behavior is register allocation. In RTL, the source language for this translation, local variables and temporaries are initialized to the `undef` value on function entry. (This initialization agrees with the semantics of Clight, where reading an uninitialized local variable amounts to loading from a freshly allocated block.) After register allocation, some of these RTL variables and temporaries are mapped to global hardware registers, which are not initialized to the `undef` value on function entry, but instead keep whatever value they had in the caller function at point of call. This does not change the semantics of well-defined RTL programs, since the RTL semantics goes wrong whenever an `undef` value is involved in an arithmetic operation or conditional test. Therefore, values of uninitialized RTL variables do not participate in the computations performed by the program, and can be changed from `undef` to any other value without changing the semantics of the program. However, the original RTL program could have stored these values of uninitialized variables in memory locations. Therefore, the memory states before and after register allocation have the same shapes, but the contents of some memory locations can change from `undef` to any value, as pictured in Fig. 2, center.

The third and last pass where memory states differ between the original and transformed codes is the spilling pass performed after register allocation. Variables and temporaries that could not be allocated to hardware registers must be "spilled" to memory, that is, stored in locations within the stack frame of the current function. Additional stack frame space is also needed to save the values of callee-save registers

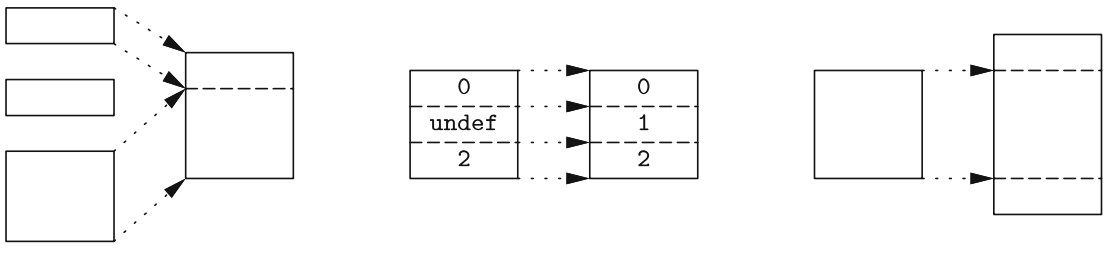



**Fig. 2** Transformations over memory states in the Compcert compiler

on function entry. Therefore, the spilling pass needs to enlarge the stack frame that was laid out at the time of Cminor generation, to make room for spilled variables and saved registers. The memory state after spilling therefore differs from the state before spilling: stack frame blocks are larger, and the transformed program performs additional `load` and `store` operations to access spilled variables. (See Fig. 2, right.)

In the three examples of program transformations outlined above, we need to formalize an invariant that relates the memory states at every point of the executions of the original and transformed programs, and prove appropriate simulation results between the memory operations performed by the two programs. Three such relations between memory states are studied in the remainder of this section: memory extensions in Section 5.2, corresponding to the spilling pass; refinement of stored values in Section 5.3, corresponding to the register allocation pass; and memory injections in Section 5.4, corresponding to the Cminor generation pass. These three relations share a common basis, the notion of memory embeddings, defined and studied first in Section 5.1.

### 5.1 Generic Memory Embeddings

An *embedding* $E$ is a function of type $\texttt{block} \to \texttt{option}(\texttt{block} \times \mathbb{Z})$ that establishes a correspondence between blocks of a memory state $m_1$ of the original program and blocks of a memory state $m_2$ of the transformed program. Let $b_1$ be a block reference in $m_1$. If $E(b_1) = \varepsilon$, this block corresponds to no block in $m_2$: it has been eliminated by the transformation. If $E(b_1) = \lfloor b_2, \delta \rfloor$, the block $b_1$ in $m_1$ corresponds to the block $b_2$ in $m_2$, or a sub-block thereof, with offsets being shifted by $\delta$. That is, the memory location $(b_1, i)$ in $m_1$ is associated to the location $(b_2, i + \delta)$ in $m_2$. We say that a block $b$ of $m_1$ is *unmapped* in $m_2$ if $E(b) = \varepsilon$, and *mapped* otherwise.

We assume we are given a relation $E \vdash v_1 \hookrightarrow v_2$ between values $v_1$ of the original program and values $v_2$ of the transformed program, possibly parametrized by $E$. (Sections 5.2, 5.3 and 5.4 will particularize this relation).

We say that $E$ embeds memory state $m_1$ in memory state $m_2$, and write $E \vdash m_1 \hookrightarrow m_2$, if every successful load from a mapped block of $m_1$ is simulated by a successful load from the corresponding sub-block in $m_2$, in the following sense:

$$\begin{aligned} E(b_1) \;=\; \lfloor b_2, \delta \rfloor \;\wedge\; \texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor \\ \Rightarrow\; \exists v_2, \texttt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor \;\wedge\; E \vdash v_1 \hookrightarrow v_2 \end{aligned}$$

We now state and prove commutation and simulation properties between the memory embedding relation and the operations of the concrete memory model. First, validity of accesses is preserved, in the following sense.

**Lemma 2** *If $E(b_1) = \lfloor b_2, \delta \rfloor$ and $E \vdash m_1 \hookrightarrow m_2$, then $m_1 \models \tau \mathbin{@} b_1, i$ implies $m_2 \models \tau \mathbin{@} b_2, i + \delta$.*

*Proof* By property P25, there exists $v_1$ such that $\texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists $v_2$ such that $\texttt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor$. The result follows from property P25. □

When is the memory embedding relation preserved by memory stores? There are three cases to consider, depicted in Fig. 3. In the leftmost case, the original

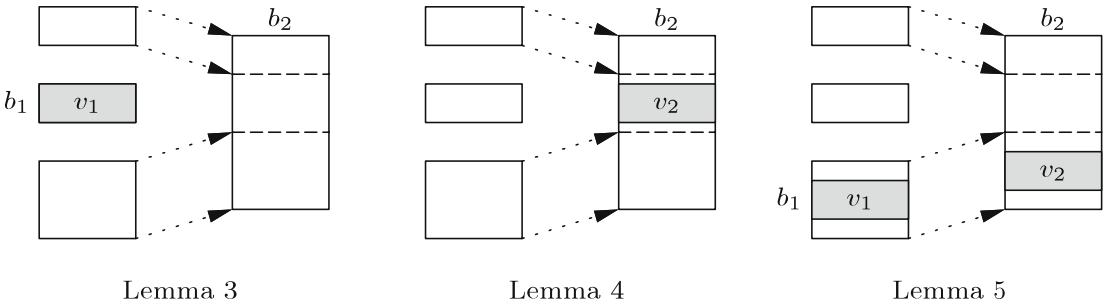



**Fig. 3** The three simulation lemmas for memory stores. The *grayed areas* represent the locations of the stores. $v_1$ is a value stored by the original program and $v_2$ a value stored by the transformed program

program performs a store in memory $m_1$ within a block that is not mapped, while the transformed program performs no store, keeping its memory $m_2$ unchanged.

**Lemma 3** *If* $E(b_1) = \varepsilon$ *and* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{store}(\tau, m_1, b_1, i, v) = \lfloor m_1' \rfloor$, *then* $E \vdash m_1' \hookrightarrow m_2$.

*Proof* Consider a load in $m_1'$ from a mapped block: $E(b_1') = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau', m_1', b_1', i') = \lfloor v_1 \rfloor$. By hypothesis $E(b_1) = \varepsilon$, we have $b_1' \neq b_1$. By S8, it follows that $\texttt{load}(\tau', m_1, b_1', i') = \texttt{load}(\tau', m_1', b_1', i') = \lfloor v_1 \rfloor$. The result follows from hypothesis $E \vdash m_1 \hookrightarrow m_2$. □

In the second case (Fig. 3, center), the original program performs no store in its memory $m_1$, but the transformed program stores some data in an area of its memory $m_2$ that is disjoint from the images of the blocks of $m_1$.

**Lemma 4** *Let* $b_2, i, \tau$ *be a memory reference in* $m_2$ *such that*

$$\forall b_1, \delta, \quad E(b_1) = \lfloor b_2, \delta \rfloor \;\Rightarrow\; \mathcal{H}(m_1, b_1) + \delta \leq i \;\vee\; i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$$

*If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{store}(\tau, m_2, b_2, i, v) = \lfloor m_2' \rfloor$, *then* $E \vdash m_1 \hookrightarrow m_2'$.

*Proof* Consider a load in $m_1$ from a mapped block: $E(b_1) = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau', m_1, b_1, i') = \lfloor v_1 \rfloor$. By P25, this load is within bounds: $\mathcal{L}(m_1, b_1) \leq i'$ and $i' + |\tau'| \leq \mathcal{H}(m_1, b_1)$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists $v_2$ such that $\texttt{load}(\tau, m_2, b_2', i' + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$. We check that the separation condition of S8 holds. This is obvious if $b_2' \neq b_2$. Otherwise, by hypothesis on $b_2$, either $\mathcal{H}(m_1, b_1) + \delta \leq i$ or $i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$. In the first case, $i' + \delta + |\tau'| \leq \mathcal{H}(m_1, b_1) + \delta \leq i$, and in the second case, $i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta \leq i' + \delta$. Therefore, $\texttt{load}(\tau', m_2', b_2', i' + \delta) = \texttt{load}(\tau', m_2, b_2', i' + \delta) = \lfloor v_2 \rfloor$, and the desired result follows. □

In the third case (Fig. 3, right), the original program stores a value $v_1$ in a mapped block of $m_1$, while in parallel the transformed program stores a matching value $v_2$ at the corresponding location in $m_2$. For this operation to preserve the memory embedding relation, it is necessary that the embedding $E$ is *nonaliasing*. We say that

an embedding $E$ is nonaliasing in a memory state $m$ if distinct blocks are mapped to disjoint sub-blocks:

$$\begin{aligned} b_1 \neq b_2 \wedge E(b_1) = \lfloor b'_1, \delta_1 \rfloor \wedge E(b_2) = \lfloor b'_2, \delta_2 \rfloor \\ \Rightarrow b'_1 \neq b'_2 \\ \vee [\mathcal{L}(m, b_1) + \delta_1, \mathcal{H}(m, b_1) + \delta_1) \cap [\mathcal{L}(m, b_2) + \delta_2, \mathcal{H}(m, b_2) + \delta_2) = \emptyset \end{aligned}$$

The disjointness condition between the two intervals can be decomposed as follows: either $\mathcal{L}(m, b_1) \geq \mathcal{H}(m, b_1)$ (block $b_1$ is empty), or $\mathcal{L}(m, b_2) \geq \mathcal{H}(m, b_2)$ (block $b_2$ is empty), or $\mathcal{H}(m, b_1) + \delta_1 \leq \mathcal{L}(m, b_2) + \delta_2$, or $\mathcal{H}(m, b_2) + \delta_2 \leq \mathcal{L}(m, b_1) + \delta_1$.

**Lemma 5** *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. *Let* $v_1$, $v_2$ *be two values and* $\tau$ *a type such that* $v_1$ *embeds in* $v_2$ *after conversion to any type* $\tau'$ *compatible with* $\tau$*:*

$$\forall \tau', \ \tau \sim \tau' \Rightarrow E \vdash \texttt{convert}(v_1, \tau') \hookrightarrow \texttt{convert}(v_2, \tau')$$

*If* $E \vdash m_1 \hookrightarrow m_2$ *and* $E$ *is nonaliasing in the memory state* $m_1$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $\texttt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m'_1 \rfloor$, *then there exists a memory state* $m'_2$ *such that* $\texttt{store}(\tau, m_2, b_2, i + \delta, v_2) = \lfloor m'_2 \rfloor$ *and moreover* $E \vdash m'_1 \hookrightarrow m'_2$.

*Proof* The existence of $m'_2$ follows from Lemma 2 and property P24. Consider a load in $m'_1$ from a mapped block: $E(b'_1) = \lfloor b'_2, \delta' \rfloor$ and $\texttt{load}(\tau', m'_1, b'_1, i') = \lfloor v'_1 \rfloor$. By property D29, there are four cases to consider.

- Compatible: $b'_1 = b_1$ and $i' = i$ and $\tau \sim \tau'$. In this case, $v'_1 = \texttt{convert}(v_1, \tau')$. By S7, we have $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \texttt{load}(\tau', m'_2, b_2, i + \delta) = \lfloor \texttt{convert}(v_2, \tau') \rfloor$. The result $E \vdash v'_1 \hookrightarrow \texttt{convert}(v_2, \tau')$ follows from the hypothesis over $v_1$ and $v_2$.
- Incompatible: $b'_1 = b_1$ and $i' = i$ and $\tau \not\sim \tau'$. In this case, $v'_1 = \texttt{undef}$. By P27 and P25, we have $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \texttt{load}(\tau', m'_2, b_2, i + \delta) = \lfloor \texttt{undef} \rfloor$. The result follows from the hypothesis $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$.
- Disjoint: $b'_1 \neq b_1$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$. In this case, $\texttt{load}(\tau', m_1, b'_1, i') = \lfloor v'_1 \rfloor$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists $v'_2$ such that $\texttt{load}(\tau', m_2, b'_2, i' + \delta') = \lfloor v'_2 \rfloor$ and $E \vdash v'_1 \hookrightarrow v'_2$. Exploiting the nonaliasing hypothesis over $E$ and $m_1$, we show that the separation hypotheses of property S8 hold, which entails $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \lfloor v'_2 \rfloor$ and the expected result.
- Overlapping: $b'_1 = b_1$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$. In this case $v'_1 = \texttt{undef}$. We show $\texttt{load}(\tau', m'_2, b'_2, i' + \delta') = \lfloor \texttt{undef} \rfloor$ using P28, and conclude using the hypothesis $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. □

We now turn to relating allocations with memory embeddings, starting with the case where two allocations are performed in parallel, one in the original program, the other in the transformed program.

**Lemma 6** *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l_1, h_1) = \lfloor b_1, m'_1 \rfloor$ *and* $\texttt{alloc}(m_2, l_2, h_2) = \lfloor b_2, m'_2 \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $l_2 \leq l_1 + \delta$ *and* $h_1 + \delta \leq h_2$ *and* `max_alignment` *divides* $\delta$, *then* $E \vdash m'_1 \hookrightarrow m'_2$.

*Proof* Consider a load in $m'_1$ from a mapped block: $E(b'_1) = \lfloor b'_2, \delta \rfloor$ and $\texttt{load}(\tau, m'_1, b'_1, i) = \lfloor v_1 \rfloor$. If $b'_1 \neq b_1$, we have $\texttt{load}(\tau, m_1, b'_1, i) = \lfloor v_1 \rfloor$ by S5, and there exists $v_2$ such that $\texttt{load}(\tau, m_2, b'_2, i + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$. It must be the case that $b'_2 \neq b_2$, otherwise the latter `load` would have failed (by S9 and P25). The expected result $\texttt{load}(\tau, m'_2, b'_2, i + \delta) = \lfloor v_2 \rfloor$ follows from S5.

If $b'_1 = b_1$, we have $\texttt{load}(\tau, m'_1, b_1, i) = \lfloor \texttt{undef} \rfloor$ by P26, and $l_1 \leq i, i + |\tau| \leq h_1$ and $|\tau|$ divides $i$ by P25 and P26. It follows that $m'_2 \models \tau \ @ \ b_2, i + \delta$, and therefore $\texttt{load}(\tau, m'_2, b_2, i + \delta) = \lfloor \texttt{undef} \rfloor$ by P25 and P26. This is the expected result since $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. □

To complement Lemma 6, we also consider the cases where allocations are performed either in the original program or in the transformed program, but not necessarily in both. (See Fig. 4.) We omit the proof sketches, as they are similar to that of Lemma 6.

**Lemma 7** *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m'_2 \rfloor$, *then* $E \vdash m_1 \hookrightarrow m'_2$.

**Lemma 8** *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m'_1 \rfloor$ *and* $E(b_1) = \varepsilon$, *then* $E \vdash m'_1 \hookrightarrow m_2$.

**Lemma 9** *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{v}$ *for all values* $v$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m'_1 \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $m_2 \models b_2$ *and* $\mathcal{L}(m_2, b_2) \leq l + \delta$ *and* $h + \delta \leq \mathcal{H}(m_2, b_2)$ *and* `max_alignment` *divides* $\delta$, *then* $E \vdash m'_1 \hookrightarrow m_2$.

Finally, we consider the interaction between `free` operations and memory embeddings. Deallocating a block in the original program always preserves embedding.

**Lemma 10** *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_1, b_1) = \lfloor m'_1 \rfloor$, *then* $E \vdash m'_1 \hookrightarrow m_2$.

*Proof* If $\texttt{load}(\tau, m'_1, b'_1, i) = \lfloor v_1 \rfloor$, it must be that $b'_1 \neq b_1$ by P25 and P36. We then have $\texttt{load}(\tau, m_1, b'_1, i) = \lfloor v_1 \rfloor$ by S6 and conclude by hypothesis $E \vdash m_1 \hookrightarrow m_2$. □

Deallocating a block in the transformed program preserves embedding if no valid block of the original program is mapped to the deallocated block.

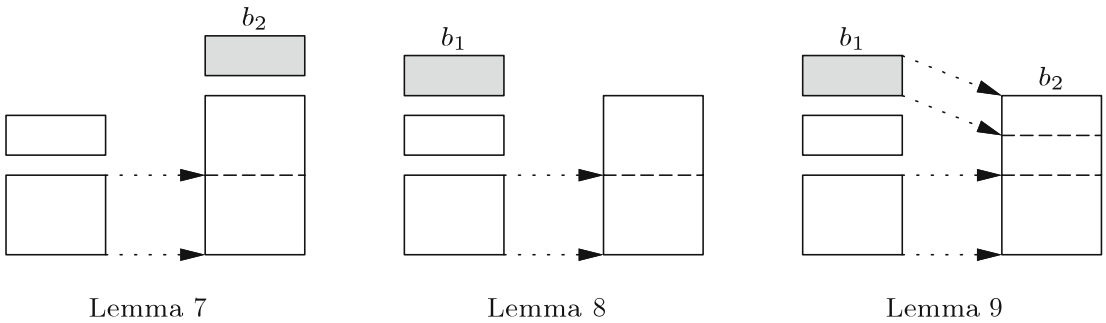



**Fig. 4** The three simulation lemmas for memory allocations. The *grayed areas* represent the freshly allocated blocks

**Lemma 11** *Assume* $\forall b_1, \delta, \quad E(b_1) = \lfloor b_2, \delta \rfloor \Rightarrow \neg(m_1 \models b_1)$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$, *then* $E \vdash m_1 \hookrightarrow m_2'$.

*Proof* Assume $E(b_1) = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$. It must be the case that $b_2' \neq b_2$, otherwise $m_1 \models b_1$ would not hold, contradicting P25. The result follows from the hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property S6. □

Combining Lemmas 10 and 11, we see that embedding is preserved by freeing a block $b_1$ in the original program and in parallel freeing a block $b_2$ in the transformed program, provided that no block other than $b_1$ maps to $b_2$.

**Lemma 12** *Assume* $\forall b, \delta, E(b) = \lfloor b_2, \delta \rfloor \Rightarrow b = b_1$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_1, b_1) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$, *then* $E \vdash m_1' \hookrightarrow m_2'$.

Finally, it is useful to notice that the nonaliasing property of embeddings is preserved by `free` operations.

**Lemma 13** *If* $E$ *is nonaliasing in* $m_1$, *and* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$, *then* $E$ *is nonaliasing in* $m_1'$.

*Proof* The block $b$ becomes empty in $m_1'$: by P37, $\mathcal{L}(m_1', b) = \mathcal{H}(m_1', b)$. The result follows from the definition of nonaliasing embeddings. □

## 5.2 Memory Extensions

We now instantiate the generic framework of Section 5.1 to account for the memory transformations performed by the spilling pass: the transformed program allocates larger blocks than the original program, and uses the extra space to store data of its own (right part of Fig. 2).

Figure 5 illustrates the effect of this transformation on the memory operations performed by the original and transformed programs. Each `alloc` in the original program becomes an `alloc` operation with possibly larger bounds. Each `store`, `load` and `free` in the original program corresponds to an identical operation in the transformed program, with the same arguments and results. The transformed program contains additional `store` and `load` operations, corresponding to spills

```
sp := alloc(0,8)
store(int, sp, 0, 42)
x := load(int, sp, 0)
...
...
...
y := x + x
free(sp)
```

```
sp := alloc(-4,8)
store(int, sp, 0, 42)
x := load(int, sp, 0)
store(int, sp, -4, x)   // spill
...
x := load(int, sp, -4)  // reload
y := x + x
free(sp)
```

**Fig. 5** Example of insertion of spill code. *Left*: original program, *right*: transformed program. The variable `x` was spilled to the stack location at offset −4. The variable `y` was not spilled

and reloads of variables, operating on memory areas that were not accessible in the original program (here, the word at offset $-4$ in the block `sp`).

To prove that this transformation preserves semantics, we need a relation between the memory states of the original and transformed programs that (1) guarantees that matching pairs of `load` operations return the same value, and (2) is preserved by `alloc`, `store` and `free` operations.

In this section, we consider a fixed embedding $E_{id}$ that is the identity function: $E_{id}(b) = \lfloor b, 0 \rfloor$ for all blocks $b$. Likewise, we define embedding between values as equality between these values: $E_{id} \vdash v_1 \hookrightarrow v_2$ if and only if $v_1 = v_2$.

We say that a transformed memory state $m_2$ extends an original memory state $m_1$, and write $m_1 \subseteq m_2$, if $E_{id}$ embeds $m_1$ in $m_2$, and both memory states have the same domain:

$$m_1 \subseteq m_2 \quad \stackrel{\text{def}}{=} \quad E_{id} \vdash m_1 \hookrightarrow m_2 \ \wedge \ \mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$$

The $\subseteq$ relation over memory states is reflexive and transitive. It implies the desired equality between the results of a `load` performed by the initial program and the corresponding `load` after transformation.

**Lemma 14** *If* $m_1 \subseteq m_2$ *and* $\texttt{load}(\tau, m_1, b, i) = \lfloor v \rfloor$, *then* $\texttt{load}(\tau, m_2, b, i) = \lfloor v \rfloor$.

*Proof* Since $E_{id} \vdash m_1 \hookrightarrow m_2$ holds, and $E_{id}(b) = \lfloor b, 0 \rfloor$, there exists a value $v'$ such that $\texttt{load}(\tau, m_2, b, i) = \lfloor v' \rfloor$ and $E_{id} \vdash v \hookrightarrow v'$. The latter entails $v' = v$ and the expected result. □

We now show that any `alloc`, `store` or `free` operation over $m_1$ is simulated by a similar memory operation over $m_2$, preserving the memory extension relation.

**Lemma 15** *Assume* $\texttt{alloc}(m_1, l_1, h_1) = \lfloor b_1, m_1' \rfloor$ *and* $\texttt{alloc}(m_2, l_2, h_2) = \lfloor b_2, m_2' \rfloor$. *If* $m_1 \subseteq m_2$ *and* $l_2 \leq l_1$ *and* $h_1 \leq h_2$, *then* $b_1 = b_2$ *and* $m_1' \subseteq m_2'$.

*Proof* The equality $b_1 = b_2$ follows from P35. The embedding $E_{id} \vdash m_1' \hookrightarrow m_2'$ follows from Lemma 6. The domain equality $\mathrm{Dom}(m_1') = \mathrm{Dom}(m_2')$ follows from P32. □

**Lemma 16** *Assume* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b) = \lfloor m_2' \rfloor$. *If* $m_1 \subseteq m_2$, *then* $m_1' \subseteq m_2'$.

*Proof* Follows from Lemma 12 and property P34. □

**Lemma 17** *If* $m_1 \subseteq m_2$ *and* $\texttt{store}(\tau, m_1, b, i, v) = \lfloor m_1' \rfloor$, *then there exists* $m_2'$ *such that* $\texttt{store}(\tau, m_2, b, i, v) = \lfloor m_2' \rfloor$ *and* $m_1' \subseteq m_2'$.

*Proof* Follows from Lemma 5 and property P33. By construction, the embedding $E_{id}$ is nonaliasing for any memory state. □

Finally, the transformed program can also perform additional stores, provided they fall outside the memory bounds of the original program. (These stores take place when a variable is spilled to memory.) Such stores preserve the extension relation.

**Lemma 18** *Assume* $m_1 \subseteq m_2$ *and* $\texttt{store}(\tau, m_2, b, i, v) = \lfloor m_2' \rfloor$. *If* $i + |\tau| \leq \mathcal{L}(m_1, b)$ *or* $\mathcal{H}(m_1, b) \leq i$, *then* $m_1 \subseteq m_2'$.

*Proof* Follows from Lemma 4 and property P33. □

### 5.3 Refinement of Stored Values

In this section, we consider the case where the original and transformed programs allocate identically-sized blocks in lockstep, but some of the `undef` values produced and stored by the original program can be replaced by more defined values in the transformed program. This situation, depicted in the center of Fig. 2, occurs when verifying the register allocation pass of Compcert. Figure 6 outlines an example of this transformation.

We say that a value $v_1$ is refined by a value $v_2$, and we write $v_1 \leq v_2$, if either $v_1 = \texttt{undef}$ or $v_1 = v_2$. We assume that the `convert` function is compatible with refinements: $v_1 \leq v_2 \Rightarrow \texttt{convert}(v_1, \tau) \leq \texttt{convert}(v_2, \tau)$. (This is clearly the case for the examples of `convert` functions given at the end of Section 2.)

We instantiate again the generic framework of Section 5.1, using the identity embedding $E_{id} = \lambda b\,.\,\lfloor b, 0 \rfloor$ and the value embedding

$$E_{id} \vdash v_1 \hookrightarrow v_2 \quad \stackrel{\text{def}}{=} \quad v_1 \leq v_2.$$

We say that a transformed memory state $m_2$ refines an original memory state $m_1$, and write $m_1 \leq m_2$, if $E_{id}$ embeds $m_1$ in $m_2$, and both memory states have the same domain:

$$m_1 \leq m_2 \quad \stackrel{\text{def}}{=} \quad E_{id} \vdash m_1 \hookrightarrow m_2 \ \wedge\ \mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$$

The $\leq$ relation over memory states is reflexive and transitive.

The following simulation properties are immediate consequences of the results from Section 5.1.

**Lemma 19** *Assume* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$. *If* $m_1 \leq m_2$, *then* $b_1 = b_2$ *and* $m_1' \leq m_2'$.

**Lemma 20** *Assume* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b) = \lfloor m_2' \rfloor$. *If* $m_1 \leq m_2$, *then* $m_1' \leq m_2'$.

*Proof* Follows from Lemma 12 and property P34. □

```
// x implicitly initialized to undef
sp := alloc(0,8)
store(int, sp, 0, x)
...
y := load(int, sp, 0)
...
free(sp)
```

```
// R1 not initialized
sp := alloc(0,8)
store(int, sp, 0, R1)
...
R2 := load(int, sp, 0)
...
free(sp)
```

**Fig. 6** Example of register allocation. *Left*: original code, *right*: transformed code. Variables `x` and `y` have been allocated to registers `R1` and `R2`, respectively

**Lemma 21** *If* $m_1 \leq m_2$ *and* $\texttt{load}(\tau, m_1, b, i) = \lfloor v_1 \rfloor$, *then there exists a value* $v_2$ *such that* $\texttt{load}(\tau, m_2, b, i) = \lfloor v_2 \rfloor$ *and* $v_1 \leq v_2$.

**Lemma 22** *If* $m_1 \leq m_2$ *and* $\texttt{store}(\tau, m_1, b, i, v_1) = \lfloor m_1' \rfloor$ *and* $v_1 \leq v_2$, *then there exists* $m_2'$ *such that* $\texttt{store}(\tau, m_2, b, i, v_2) = \lfloor m_2' \rfloor$ *and* $m_1' \leq m_2'$.

### 5.4 Memory Injections

We now consider the most difficult memory transformation encountered in the Compcert development, during the translation from Clight to Cminor: the removal of some memory allocations performed by the Clight semantics and the coalescing of other memory allocations into sub-areas of a single block (see Fig. 2, left).

The pseudocode in Fig. 7 illustrates the effect of this transformation on the memory behavior of the program. Here, the transformation elected to "pull x out of memory", using a local variable x in the transformed program to hold the contents of the block pointed by x in the original program. It also merged the blocks pointed by y and z into a single block pointed by sp, with y corresponding to the sub-block at offsets [0, 8) and z to the sub-block at offsets [8, 10).

To relate the memory states in the original and transformed programs at any given point, we will again reuse the results on generic memory embeddings established in Section 5.1. However, unlike in Sections 5.2 and 5.3, we cannot work with a fixed embedding $E$, but need to build it incrementally during the proof of semantic preservation.

In the Compcert development, we use the following relation between values $v_1$ of the original Clight program and $v_2$ of the generated Cminor program, parametrized by an embedding $E$:

$$E \vdash \texttt{undef} \hookrightarrow v_2 \qquad E \vdash \texttt{int}(n) \hookrightarrow \texttt{int}(n) \qquad E \vdash \texttt{float}(n) \hookrightarrow \texttt{float}(n)$$

$$\frac{E(b_1) = \lfloor b_2, \delta \rfloor \qquad i_2 = i_1 + \delta}{E \vdash \texttt{ptr}(b_1, i_1) \hookrightarrow \texttt{ptr}(b_2, i_2)}$$

```
x := alloc(0, 4)                    sp := alloc(0, 10)
y := alloc(0, 8)
z := alloc(0, 2)
store(int, x, 0, 42)                x := 42
... load(int, x, 0) ...             ... x ...
store(double, y, 0, 3.14)           store(double, sp, 0, 3.14)
... load(short, z, 2) ...           ... load(short, sp, 8) ...
free(x)
free(y)
free(z)                             free(sp)
```

**Fig. 7** Example of the Clight to Cminor translation. *Left*: original program, *right*: transformed program. Block x in the original program is pulled out of memory; its contents are stored in the local variable x in the transformed program. Blocks y and z become sub-blocks of sp, at offsets 0 and 8 respectively

In other words, `undef` Clight values can be refined by any Cminor value; integers and floating-point numbers must not change; and pointers are relocated as prescribed by the embedding $E$. Notice in particular that if $E(b) = \varepsilon$, there is no Cminor value $v$ such that $E \vdash \mathtt{ptr}(b, i) \hookrightarrow v$. This means that the source Clight program is not allowed to manipulate a pointer value pointing to a memory block that we have decided to remove during the translation.

We assume that the $E \vdash v_1 \hookrightarrow v_2$ relation is compatible with the `convert` function: $E \vdash v_1 \hookrightarrow v_2$ implies $E \vdash \mathtt{convert}(v_1, \tau) \hookrightarrow \mathtt{convert}(v_2, \tau)$. (This clearly holds for the examples of `convert` functions given at the end of Section 2.)

We say that an embedding $E$ injects a Clight memory state $m_1$ in a Cminor memory state $m_2$, and write $E \vdash m_1 \mapsto m_2$, if the following four conditions hold:

$$
\begin{aligned}
E \vdash m_1 \mapsto m_2 \quad \stackrel{\text{def}}{=} \quad & E \vdash m_1 \hookrightarrow m_2 && (1)\\
\wedge \ & \forall b_1, \ \ m_1 \# b_1 \ \Rightarrow \ E(b_1) = \varepsilon && (2)\\
\wedge \ & \forall b_1, b_2, \delta, \ \ E(b_1) = \lfloor b_2, \delta \rfloor \ \Rightarrow \ \neg(m_2 \# b_2) && (3)\\
\wedge \ & E \text{ is nonaliasing for } m_1 && (4)
\end{aligned}
$$

Condition (1) is the embedding of $m_1$ into $m_2$ in the sense of Section 5.1. Conditions (2) and (3) ensure that fresh blocks are not mapped, and that images of mapped blocks are not fresh. Condition (4) ensures that the embedding does not cause sub-blocks to overlap.

Using this definition, it is easy to show simulation results for the `load` and `store` operations performed by the original program.

**Lemma 23** *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$ and $E(b_1) = \lfloor b_2, \delta \rfloor$, then there exists a value $v_2$ such that $\mathtt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$.*

*Proof* By (1) and definition of $E \vdash m_1 \hookrightarrow m_2$. □

**Lemma 24** *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$ and $E(b_1) = \lfloor b_2, \delta \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$, then there exists $m_2'$ such that $\mathtt{store}(\tau, m_2, b_2, i + \delta, v_2) = \lfloor m_2' \rfloor$ and $E \vdash m_1' \mapsto m_2'$.*

*Proof* Follows from Lemma 5. Conditions (2), (3) and (4) are preserved because of properties S16 and P33. □

**Lemma 25** *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$ and $E(b_1) = \varepsilon$, then $E \vdash m_1' \mapsto m_2'$.*

*Proof* Follows from Lemma 3 and properties S16 and P33. □

In the Compcert development, given the algebra of values used (see Section 2), we can define the following variants `loadptr` and `storeptr` of `load` and `store` where the memory location being accessed is given as a pointer value:

```
loadptr(τ, m, a) =
  match a with ptr(b, i) ⇒ load(τ, m, b, i) | _ ⇒ ε
```

$$\begin{array}{l}\texttt{storeptr}(\tau, m, a, v) = \\ \quad \texttt{match}\ a\ \texttt{with}\ \texttt{ptr}(b, i) \Rightarrow \texttt{store}(\tau, m, b, i, v) \mid \_ \Rightarrow \varepsilon\end{array}$$

Lemmas 23 and 24 can then be restated in a more "punchy" way, taking advantage of the way $E \vdash v_1 \hookrightarrow v_2$ is defined over pointer values:

**Lemma 26** *If $E \vdash m_1 \mapsto m_2$ and $\texttt{loadptr}(\tau, m_1, a_1) = \lfloor v_1 \rfloor$ and $E \vdash a_1 \hookrightarrow a_2$, then there exists a value $v_2$ such that $\texttt{loadptr}(\tau, m_2, a_2) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$.*

**Lemma 27** *If $E \vdash m_1 \mapsto m_2$ and $\texttt{storeptr}(\tau, m_1, b_1, a_1, v_1) = \lfloor m_1' \rfloor$ and $E \vdash a_1 \hookrightarrow a_2$ and $E \vdash v_1 \hookrightarrow v_2$, then there exists $m_2'$ such that $\texttt{storeptr}(\tau, m_2, b_2, a_2, v_2) = \lfloor m_2' \rfloor$ and $E \vdash m_1' \mapsto m_2'$.*

We now relate a sequence of deallocations performed in the original program with a single deallocation performed in the transformed program. (In the example of Fig. 7, this corresponds to the deallocations of `x`, `y` and `z` on one side and the deallocation of `sp` on the other side.) If $l$ is a list of block references, we define the effect of deallocating these blocks as follows:

$$\begin{array}{l}\texttt{freelist}(m, l) = \\ \quad \texttt{match}\ l\ \texttt{with} \\ \quad\ \texttt{nil} \Rightarrow \lfloor m \rfloor \\ \quad \mid b :: l' \Rightarrow \texttt{match}\ \texttt{free}(m, b)\ \texttt{with}\ \varepsilon \Rightarrow \varepsilon \mid \lfloor m' \rfloor \Rightarrow \texttt{freelist}(m', l')\end{array}$$

**Lemma 28** *Assume $\texttt{freelist}(m_1, l) = \lfloor m_1' \rfloor$ and $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$. Further assume that $E(b_1) = \lfloor b_2, \delta \rfloor \Rightarrow b_1 \in l$ for all $b_1, \delta$; in other words, all blocks mapped to $b_2$ are in $l$ and therefore are being deallocated from $m_1$. If $E \vdash m_1 \mapsto m_2$, then $E \vdash m_1' \mapsto m_2'$.*

*Proof* First, notice that for all $b_1 \in l$, $\neg(m_1' \models b_1)$, by S13 and P36. Part (1) of the expected result then follows from Lemmas 10 and 11. Parts (2) and (3) follow from P34. Part (4) follows by repeated application of Lemma 13. □

Symmetrically, we now consider a sequence of allocations performed by the original program and relate them with a single allocation performed by the transformed program. (In the example of Fig. 7, this corresponds to the allocations of `x`, `y` and `z` on one side and the allocation of `sp` on the other side.) A difficulty is that the current embedding $E$ needs to be changed to map the blocks allocated by the original program; however, changing $E$ should not invalidate the mappings for pre-existing blocks.

We say that an embedding $E'$ is compatible with an earlier embedding $E$, and write $E \leq E'$, if, for all blocks $b$, either $E(b) = \varepsilon$ or $E'(b) = E(b)$. In other words, all blocks that are mapped by $E$ remain mapped to the same target sub-block in $E'$. This relation is clearly reflexive and transitive. Moreover, it preserves injections between values:

**Lemma 29** *If $E \vdash v_1 \hookrightarrow v_2$ and $E \leq E'$, then $E' \vdash v_1 \hookrightarrow v_2$.*

We first state and prove simulation lemmas for one allocation, performed either by the original program or by the transformed program. The latter case is straightforward:

**Lemma 30** *If* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$, *then* $E \vdash m_1 \mapsto m_2'$.

*Proof* Follows from Lemma 7 and property P32. □

For an allocation performed by the original program, we distinguish two cases: either the new block is unmapped (Lemma 31), or it is mapped to a sub-block of the transformed program (Lemma 32).

**Lemma 31** *Assume* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$. *Write* $E' = E\{b_1 \leftarrow \varepsilon\}$. *Then,* $E' \vdash m_1' \mapsto m_2$ *and* $E \leq E'$.

*Proof* By part (2) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property P31, it must be the case that $E(b_1) = \varepsilon$. It follows that $E' = E$, and therefore we have $E \leq E'$ and $E' \vdash m_1 \hookrightarrow m_2$. Applying Lemma 8, we obtain part (1) of the expected result $E' \vdash m_1' \mapsto m_2$. Part (2) follows from P32. Parts (3) and (4) are straightforward. □

**Lemma 32** *Assume* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $m_2 \models b_2$ *and* $\mathcal{L}(m_2, b_2) \leq l + \delta$ *and* $h + \delta \leq \mathcal{H}(m_2, b_2)$ *and* `max_alignment` *divides* $\delta$. *Further assume that for all blocks* $b'$ *and offsets* $\delta'$,

$$E(b') = \lfloor b_2, \delta' \rfloor \;\Rightarrow\; \mathcal{H}(m_1, b') + \delta' \leq l + \delta \;\vee\; h + \delta \leq \mathcal{L}(m_1, b') + \delta' \qquad (*)$$

*Write* $E' = E\{b_1 \leftarrow \lfloor b_2, \delta \rfloor\}$. *If* $E \vdash m_1 \mapsto m_2$, *then* $E' \vdash m_1' \mapsto m_2$ *and* $E \leq E'$.

*Proof* By part (2) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property P31, it must be the case that $E(b_1) = \varepsilon$. It follows that $E \leq E'$.

We first show that $E' \vdash m_1 \hookrightarrow m_2$. Assume $E'(b) = \lfloor b', \delta' \rfloor$ and $\texttt{load}(\tau, m_1, b, i) = \lfloor v \rfloor$. It must be the case that $b \neq b_1$, since $b_1$ is not valid in $m_1$. Therefore, $E(b) = \lfloor b', \delta' \rfloor$ and the result follows from part (1) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and from Lemma 29.

Using Lemma 9, we obtain part (1) of the expected result $E' \vdash m_1' \mapsto m_2$. Part (2) follows from P32. Part (3) follows from the fact that $b_2$ is not fresh (property P30). Finally, part (4) follows from hypothesis (*) and property S15. □

We now define the `alloclist` function, which, given a list $L$ of (low, high) bounds, allocates the corresponding blocks and returns both the list $B$ of their references and the final memory state. In the example of Fig. 7, the allocation of `x`, `y` and `z` corresponds to an invocation of `alloclist` with the list $L = (0, 4); (0, 8); (0, 2)$.

```
alloclist(m, L) =
  match L with
   nil ⇒ ⌊nil, m⌋
  | (l, h) :: L' ⇒
     match alloc(m, l, h) with
      ε ⇒ ε
```

$$
\begin{array}{l}
\mid \lfloor b, m' \rfloor \Rightarrow \\
\quad \texttt{match alloclist}(m', L') \texttt{ with} \\
\quad\ \varepsilon \Rightarrow \varepsilon \\
\quad \mid \lfloor B, m'' \rfloor \Rightarrow \lfloor b :: B, m'' \rfloor
\end{array}
$$

Along with the list $L = (l_1, h_1) \ldots (l_n, h_n)$ of allocation requests to be performed in the original program, we assume given the bounds $(l, h)$ of a block to be allocated in the transformed program, and a list $P = p_1, \ldots, p_n$ of elements of type `option` $\mathbb{Z}$, indicating how these allocated blocks should be mapped in the transformed program. If $p_i = \varepsilon$, the $i$-th block is unmapped, but if $p_i = \lfloor \delta_i \rfloor$, it should be mapped at offset $\delta_i$. In the example of Fig. 7, we have $l = 0$, $h = 10$, $p_1 = \varepsilon$, $p_2 = \lfloor 0 \rfloor$, and $p_3 = \lfloor 8 \rfloor$.

We say that the quadruple $(L, P, l, h)$ is well-formed if the following conditions hold:

1. $L$ and $P$ have the same length.
2. If $p_i = \lfloor \delta_i \rfloor$, then $l \leq l_i + \delta_i$ and $h_i + \delta_i \leq h$ and `max_alignment` divides $\delta_i$ (the image of the $i$-th block is within bounds and aligned).
3. If $p_i = \lfloor \delta_i \rfloor$ and $p_j = \lfloor \delta_j \rfloor$ and $i \neq j$, then $h_i + \delta_i \leq l_j + \delta_j$ or $h_j + \delta_j \leq l_i + \delta_i$ (blocks are mapped to disjoint sub-blocks).

**Lemma 33** *Assume that $(L, P, l, h)$ is well-formed. Assume* `alloclist`$(m_1, L) = \lfloor B, m_1' \rfloor$ *and* `alloc`$(m_2, l, h) = \lfloor b, m_2' \rfloor$. *If $E \vdash m_1 \mapsto m_2$, there exists an embedding $E'$ such that $E' \vdash m_1' \mapsto m_2'$ and $E \leq E'$. Moreover, writing $b_i$ for the $i$-th element of $B$ and $p_i$ for the $i$-th element of $P$, we have $E'(b_i) = \varepsilon$ if $p_i = \varepsilon$, and $E'(b_i) = \lfloor b, \delta_i \rfloor$ if $p_i = \lfloor \delta_i \rfloor$.*

*Proof* By Lemma 30, we have $E \vdash m_1 \mapsto m_2'$. We then show the expected result by induction over the length of the lists $L$ and $P$, using an additional induction hypothesis: for all $b', \delta, i$, if $E(b') = \lfloor b, \delta \rfloor$ and $p_i = \lfloor \delta' \rfloor$, then $h_i + \delta_i \leq \mathcal{L}(m_1, b') + \delta$ or $\mathcal{H}(m_1, b') + \delta \leq l_i + \delta_i$. In other words, the images of mapped blocks that remain to be allocated are disjoint from the images of the mapped blocks that have already been allocated. The proof uses Lemmas 31 and 32. We finish by proving the additional induction hypothesis for the initial state, which is easy since the initial embedding $E$ does not map any block to a sub-block of the fresh block $b$. □

## 6 Mechanical Verification

We now briefly comment on the Coq mechanization of the results presented in this article, which can be consulted on-line at http://gallium.inria.fr/~xleroy/memory-model/. The Coq development is very close to what is presented here. Indeed, almost all specifications and statements of theorems given in this article were transcribed directly from the Coq development. The only exception is the definition of well-formed multiple allocation requests at the end of Section 5.4, which is presented as inductive predicates in the Coq development, such predicates being easier to reason upon inductively than definitions involving $i$-th elements of lists. Also, the Coq development proves additional lemmas not shown in this paper: 8 derived properties in the style of properties D19-D22, used to shorten the proofs of Section 5, and 16 properties of auxiliary functions occurring in the concrete

implementation of the model, used in the proofs of Section 4, especially that of Lemma 1.

The mechanized development uses the Coq module system [6] to clearly separate specifications from implementations. The specification of the abstract memory model from Section 3, as well as the properties of the concrete memory model from Section 4, are given as module signatures. The concrete implementation of the model is a structure that satisfies these two signatures. The derived properties of Sections 3 and 4, as well as the memory transformations of Section 5, are presented as functors, i.e., modules parametrized by any implementation of the abstract signature or concrete signature, respectively. This use of the Coq module system ensures that the results we proved, especially those of Section 5, do not depend on accidental features of our concrete implementation, but only on the properties stated earlier.

The Coq module system was effective at enforcing this kind of abstraction. However, we hit one of its limitations: no constructs are provided to extend a posteriori a module signature (interface) with additional declarations and logical properties. The Standard ML and Objective Caml module systems support such extensions through the `open` and `include` constructs, respectively. By lack of such constructs in Coq, the signature of the abstract memory model must be manually duplicated in the signature of the concrete memory model, and later changes to the abstract signature must be manually propagated to the concrete signature. For our development, this limitation was a minor annoyance, but it is likely to cause serious problems for developments that involve a large number of refinement steps.

The Coq development represents approximately 1070 non-blank lines of specifications and statements of theorems, and 970 non-blank lines of proof scripts. Most of the proofs are conducted manually, since Coq does not provide much support for proof automation. However, our proofs intensively use the `omega` tactic, a decision procedure for Presburger arithmetic that automates reasoning about linear equalities and inequalities. The `eauto` (Prolog-style resolution) and `congruence` (equational reasoning via the congruence closure algorithm) tactics were also occasionally useful, but the `tauto` and `firstorder` tactics (propositional and first-order automatic reasoning, respectively) were either too weak (`tauto`) or too inefficient (`firstorder`) to be useful.

As pointed out by one of the reviewers, our formalization is conducted mostly in first-order logic: functions are used as data in Sections 4 and 5, but only to implement finite maps, which admit a simple, first-order axiomatization. A legitimate question to ask, therefore, is whether our proofs could be entirely automated using a modern theorem prover for first-order logic. We experimented with this approach using three automatic theorem provers: Ergo [7], Simplify [10] and Z3 [9]. The Why platform for program proof [12] was used to administer the tests and to translate automatically between the input syntaxes of the provers. Most of the Coq development was translated to Why's input syntax. (The only part we did not translate is the concrete implementation of the memory model, because it uses recursive functions that are difficult to express in this syntax.) Each derived property and lemma was given to the three provers as a goal, with a time limit of 5 min of CPU time.[4]

[4]The test was run on a 2.4 GHz Intel Core 2 Duo processor, with 2 Gb of RAM, running the MacOS 10.4 operating system. The versions of the provers used are: Ergo 0.7.2, compiled with OCaml version 3.10.1; Simplify 1.5.5; Z3 1.1, running under CrossOver Mac.

**Table 1** Experiments with three automated theorem provers

| | Ergo | Simplify | Z3 | At least one |
|---|---|---|---|---|
| Derived properties from Sections 3 and 4 | 15/15 | 15/15 | 15/15 | 15/15 |
| Generic memory embeddings (Section 5.1) | 7/12 | 1/12 | 6/12 | 9/12 |
| Memory extensions (Section 5.2) | 0/7 | 1/7 | 5/7 | 5/7 |
| Refinement of stored values (Section 5.3) | 2/8 | 3/8 | 6/8 | 6/8 |
| Memory injections (Section 5.4) | 4/8 | 4/8 | 7/8 | 7/8 |
| Total | 28/50 | 24/50 | 39/50 | 42/50 |

Table 1 summarizes the results of this experiment. A total of 50 goals were given to the three provers: the derived properties D19-D22 and D29 from Sections 3 and 4, all lemmas from Section 5, and some auxiliary lemmas present in the Coq development. Of these 50 goals, 42 were proved by at least one of the three provers. The 8 goals that all three provers fail to establish within the 5-min time limit correspond to Lemmas 5, 6, 12, 15, 17, 19, 22, and 31 from Section 5. These preliminary results are encouraging: while interactive proof remains necessary for some of the most difficult theorems, integration of first-order theorem proving within a proof assistant has great potential to significantly shorten our proofs.

## 7 Related Work

Giving semantics to imperative languages and reasoning over pointer programs has been the subject of much work since the late 1960's. Reynolds [23] and Tennent and Ghica [26] review some of the early work in this area. In the following, we mostly focus on semantics and verifications that have been mechanized.

For the purpose of this discussion, memory models can be roughly classified as either "high level", where the model itself provides some guarantees of separation, enforcement of memory bounds, etc., or "low-level", where the memory is modeled essentially as an array of bytes and such guarantees must be enforced through additional logical assertions.

A paradigmatic example of high-level modeling is the Burstall-Bornat encoding of records (`struct`), where each field is viewed as a distinct memory store mapping addresses to contents [4, 5]. Such a representation captures the fact that distinct fields of a `struct` value are separated: it becomes obvious that assigning to one field though a pointer (`p->f = x` in C) leaves unchanged the values of any other field. In turn, this separation guarantee greatly facilitates reasoning over programs that manipulate linked data structures, as demonstrated by Mehta and Nipkow [19] and the Caduceus program prover of Filliâtre and Marché [11]. However, this representation makes it difficult to account for other features of the C language, such as `union` types and some casts between pointer types.

Examples of low-level modeling of memory include Norrish's HOL semantics for C [21] and the work of Tuch et al. [27]. There, a memory state is essentially a mapping from addresses to bytes, and allocation, loads and stores are axiomatized in these terms. The axioms can either leave unspecified all behaviors undefined in the C standard, or specify additional behaviors arising from popular violations

of the C standard such as casts between incompatible pointer types. Reasoning about programs and program transformations is more difficult than with a high-level memory model; Tuch et al. [27] use separation logic to alleviate these difficulties.

The memory model presented in this article falls half-way between high-level models and low-level models. It guarantees several useful properties: separation between blocks obtained by distinct calls to `alloc`, enforcement of bounds during memory accesses, and the fact that loads overlapping a prior store operation predictably return the `undef` value. These properties play a crucial role in verifying the program transformations presented in Section 5. In particular, a lower-level memory model where a load overlapping a previous store could return an unspecified value would invalidate the simulation lemmas for memory injections (Section 5.4). On the other hand, the model offers no separation guarantees over accesses performed within the same memory block. The natural encoding of a `struct` value as a single memory block does not, by itself, validate the Burstall-Bornat separation properties; additional reasoning over field offsets is required.

Separation logic, introduced by O'Hearn et al. [22, 24], and the related spatial logic of Jia and Walker [14], provide an elegant way to reason over memory separation properties of pointer programs. Central to these approaches is the separating conjunction $P * Q$, which guarantees that the logical assertions $P$ and $Q$ talk about disjoint areas of the memory state. Examples of use of separation logic include correctness proofs for memory allocators and garbage collectors [17, 18]. It is possible, but not very useful, to define a separation logic on top of our memory model, where in a separating conjunction $P * Q$, every memory block is wholly owned by either $P$ or $Q$ but not both. Appel and Blazy [1] develop a finer-grained separation logic for the Cminor intermediate language of Compcert where disjoint parts of a given block can be separated.

While intended for sequential programs, the memory model described in this paper can also be used to describe concurrent executions in a strongly-consistent shared memory context, where the memory effect of a concurrent program is equivalent to an interleaving of the load and store operations performed by each of its threads. Modern multiprocessor systems implement weakly-consistent forms of shared memory, where the execution of a concurrent program cannot be described as such interleavings of atomic load and store operations. The computer architecture community has developed sophisticated hardware memory models to reason about weakly-consistent memory. For instance, Shen et al. [25] use a term rewriting system to define a memory model that decomposes load and store operations into finer-grained operations. This model formalizes the notions of data replication and instruction reordering. It aims at defining the legal behaviors of a distributed shared-memory system that relies on execution trace of memory accesses. Another example of architecture-centric memory model is that of Yang et al. [28].

Going back to memory models for programming languages, features of architectural models for weakly-consistent shared memory also appear in specifications of programming languages that support shared-memory concurrency. A famous example is Java, whose specification of its memory model has gone through several iterations. Manson et al. [16] describe and formalize the latest version of the Java memory model. Reasoning over concurrent, lock-free programs in Java or any other shared-memory, weakly-consistent concurrency model remains challenging. Reasoning over transformations of such programs is an open problem.

Software written in C, especially systems code, often makes assumptions about the layout of data in memory and the semantics of memory accesses that are left unspecified by the C standard. Recent work by Nita et al. [20] develops a formal framework to characterize these violations of the C standard and to automatically detect the portability issues they raise. Central to their approach is the notion of a *platform*, which is an abstract description of the assumptions that non-portable code makes about concrete data representations. Some aspects of their notion of platform are captured by our memory model, via the size and alignment functions and the type compatibility relation from Section 2. However, our model does not account for many other aspects of platforms, such as the layout and padding algorithm for `struct` types.

## 8 Conclusions

We have presented and formalized a software memory model at a level of abstraction that is intermediate between the high-level view of memory that underlies the C standard and the low-level view of memory that is implemented in hardware. This memory model is adequate for giving semantics and reasoning over intermediate languages typically found in compilers. In particular, the main features of our model (separation between blocks, bounds checking, and the `undef` value resulting from ill-defined loads) played a crucial role in proving semantics preservation for the Compcert verified compiler.

This model can also be used to give a concrete semantics for the C language that specifies the behavior of a few popular violations of the C standard, such as arbitrary casts between pointers, as well as pointer arithmetic within `struct` types. However, many other violations commonly used in systems code or run-time systems for programming languages (for instance, copying arrays of characters 4 or 8 elements as a time using integer or floating-point loads and stores) cannot be accounted for in our model. We have considered several variants of our model that could give meaning to these idioms, but have not yet found one that would still validate all simulation properties of Section 5. This remains an important direction for future work, since such a model would be useful not only to reason over systems C code, but also to prove that the semantics of such code is preserved during compilation by the Compcert compiler.

Another direction for future work is to construct and prove correct refinements from a high-level model such as the Burstall-Bornat model used in Caduceus [11] to our model, and from our model to a low-level, hardware-oriented memory model. Such refinements would strengthen the proof of semantic preservation of the Compcert compiler, which currently uses a single memory model for the source and target languages.

**Acknowledgements** Our early explorations of memory models for the Compcert project were conducted in collaboration with Benjamin Grégoire and François Armand, and benefited from discussions with Catherine Dubois and Pierre Letouzey. Sylvain Conchon, Jean-Christophe Filliâtre and Benjamin Monate helped us experiment with automatic theorem provers. We thank the anonymous reviewers as well as Nikolay Kosmatov for their careful reading of this article and their helpful suggestions for improvements.

## References


1. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007, Lecture Notes in Computer Science, vol. 4732, pp. 5–21. Springer, New York (2007)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer, New York (2004)
3. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods, Lecture Notes in Computer Science, vol. 4085, pp. 460–475. Springer, New York (2006)
4. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC '00: Proc. Int. Conf. on Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 1837, pp. 102–126. Springer, New York (2000)
5. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Mach. Intell. **7**, 23–50 (1972)
6. Chrząszcz, J.: Modules in type theory with generative definitions. Ph.D. thesis, Warsaw University and University of Paris-Sud (2004)
7. Conchon, S., Contejean, E., Kanig, J.: The Ergo automatic theorem prover. Software and documentation available at http://ergo.lri.fr/ (2005–2008)
8. Coq development team: The Coq proof assistant. Software and documentation available at http://coq.inria.fr/ (1989–2008)
9. De Moura, L., Bjørner, N., et al.: Z3: an efficient SMT solver. Software and documentation available at http://research.microsoft.com/projects/z3 (2006–2008)
10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
11. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: 6th Int. Conf. on Formal Engineering Methods, ICFEM 2004, Lecture Notes in Computer Science, vol. 3308, pp. 15–29. Springer, New York (2004)
12. Filliâtre, J.C., Marché, C., Moy, Y., Hubert, T.: The Why software verification platform. Software and documentation available at Software and documentation available at http://why.lri.fr/ (2004–2008)
13. ISO: International Standard ISO/IEC 9899:1999, Programming languages – C. ISO, Geneva (1999)
14. Jia, L., Walker, D.: ILC: a foundation for automated reasoning about pointer programs. In: Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Lecture Notes in Computer Science, vol. 3924, pp. 131–145. Springer, New York (2006)
15. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd Symposium Principles of Programming Languages, pp. 42–54. ACM, New York (2006)
16. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: 32nd Symposium Principles of Programming Languages, pp. 378–391. ACM, New York (2005)
17. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an operating system using separation logic. In: Formal Methods and Software Engineering, 8th Int. Conf. ICFEM 2006, Lecture Notes in Computer Science, vol. 4260, pp. 400–419. Springer, New York (2006)
18. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Programming Language Design and Implementation 2007, pp. 468–479. ACM, New York (2007)
19. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2), 200–227 (2005)
20. Nita, M., Grossman, D., Chambers, C.: A theory of platform-dependent low-level software. In: 35th Symposium Principles of Programming Languages. ACM, New York (2008)
21. Norrish, M.: C formalized in HOL. Technical report UCAM-CL-TR-453. Ph.D. thesis, University of Cambridge (1998)
22. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic, 15th Int. Workshop, CSL 2001, Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer, New York (2001)

23. Reynolds, J.C.: Intuitionistic reasoning about shared data structures. In: Davies, J., Roscoe, B., Woodcock, J. (eds.) Millenial Perspectives in Computer Science, pp. 303–321. Palgrave, Hampshire (2000)
24. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
25. Shen, X., Arvind, R.L.: Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In: ISCA '99: Proc. Int. Symp. on Computer Architecture, pp. 150–161. IEEE Computer Society, Los Alamitos (1999)
26. Tennent, R.D., Ghica, D.R.: Abstract models of storage. Higher-Order and Symbolic Computation **13**(1–2), 119–129 (2000)
27. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: 34th Symposium Principles of Programming Languages, pp. 97–108. ACM, New York (2007)
28. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurr. Comput.: Practice and Experience **17**(5–6), 465–487 (2005)

# Separation Logic for Small-Step Cminor

Andrew W. Appel[1,⋆] and Sandrine Blazy[2,⋆]

[1] Princeton University
[2] ENSIIE

**Abstract.** Cminor is a mid-level imperative programming language; there are proved-correct optimizing compilers from C to Cminor and from Cminor to machine language. We have redesigned Cminor so that it is suitable for Hoare Logic reasoning and we have designed a Separation Logic for Cminor. In this paper, we give a small-step semantics (instead of the big-step of the proved-correct compiler) that is motivated by the need to support future concurrent extensions. We detail a machine-checked proof of soundness of our Separation Logic. This is the first large-scale machine-checked proof of a Separation Logic w.r.t. a small-step semantics. The work presented in this paper has been carried out in the Coq proof assistant. It is a first step towards an environment in which concurrent Cminor programs can be verified using Separation Logic and also compiled by a proved-correct compiler with formal end-to-end correctness guarantees.

## 1 Introduction

The future of program verification is to connect machine-verified source programs to machine-verified compilers, and run the object code on machine-verified hardware. To connect the verifications end to end, the source language should be specified as a structural operational semantics (SOS) represented in a logical framework; the target architecture can also be specified that way. Proofs of source code can be done in the logical framework, or by other tools whose soundness is proved w.r.t. the SOS specification; these may be in safety proofs via type-checking, correctness proofs via Hoare Logic, or (in source languages designed for the purpose) correctness proofs by a more expressive proof theory. The compiler—if it is an optimizing compiler—will be a stack of phases, each with a well specified SOS of its own. There will be proofs of (partial) correctness of each compiler phase, or witness-driven recognizers for correct compilations, w.r.t. the SOS's that are inputs and outputs to the phases.

Machine-verified hardware/compiler/application stacks have been built before. Moore described a verified compiler for a "high-level assembly language" [13]. Leinenbach *et al.* [11] have built and proved a compiler for *C0*, a small C-like language, as part of a project to build machine-checked correctness proofs

⋆ Appel supported in part by NSF Grants CCF-0540914 and CNS-0627650. This work was done, in part, while both authors were on sabbatical at INRIA.

K. Schneider and J. Brandt (Eds.): TPHOLs 2007, LNCS 4732, pp. 5–21, 2007.

of source programs, Hoare Logic, compiler, micro-kernel, and RISC processor. These are both simple one- or two-pass nonoptimizing compilers.

Leroy [12] has built and proved correct in Coq [1] a compiler called *CompCert* from a high-level intermediate language *Cminor* to assembly language for the Power PC architecture. This compiler has 4 intermediate languages, allowing optimizations at several natural levels of abstraction. Blazy *et al.* have built and proved correct a translator from a subset of C to Cminor [5]. Another compiler phase on top (not yet implemented) will then yield a proved-correct compiler from C to machine language. We should therefore reevaluate the conventional wisdom that an entire practical optimizing compiler cannot be proved correct.

A software system can have components written in different languages, and we would like end-to-end correctness proofs of the whole system. For this, we propose a new variant of Cminor as a machine-independent intermediate language to serve as a common denominator between high-level languages. Our new Cminor has a usable Hoare Logic, so that correctness proofs for some components can be done directly at the level of Cminor.

Cminor has a "calculus-like" view of local variables and procedures (*i.e.* local variables are bound in an environment), while Leinenbach's C0 has a "storage-allocation" view (*i.e.* local variables are stored in the stack frame). The calculus-like view will lead to easier reasoning about program transformations and easier use of Cminor as a target language, and fits naturally with a multi-pass optimizing compiler such as CompCert; the storage-allocation view suits the one-pass nonoptimizing C0 compiler and can accommodate in-line assembly code.

Cminor is a promising candidate as a common intermediate language for end-to-end correctness proofs. But we have many demands on our new variant of Cminor, only the first three of which are satisfied by Leroy's Cminor.

- Cminor has an operational semantics represented in a logical framework.
- There is a proved-correct compiler from Cminor to machine language.
- Cminor is usable as the high-level target language of a C compiler.
  - Our semantics is a *small-step* semantics, to support reasoning about input/output, concurrency, and nontermination.
  - Cminor is machine-independent over machines in the "standard model" (*i.e.* 32- or 64-bit single-address-space byte-addressable multiprocessors).
  - Cminor can be used as a mid-level target language of an ML compiler [8], or of an OO-language compiler, so that we can integrate correctness proofs of ML or OO programs with the proofs of their run-time systems and libraries.
  - As we show in this paper, Cminor supports an axiomatic Hoare Logic (in fact, Separation Logic), proved sound with respect to the small-step semantics, for reasoning about low-level (C-like) programs.
  - In future work, we plan to extend Cminor to be concurrent in the "standard model" of thread-based preemptive lock-synchronized weakly consistent shared-memory programming. The sequential soundness proofs we present here should be reusable in a concurrent setting, as we will explain.

Leroy's original Cminor had several Power-PC dependencies, is slightly clumsy to use as the target of an ML compiler, and is a bit clumsy to use in Hoare-style

reasoning. But most important, Leroy's semantics is a big-step semantics that can be used only to reason about terminating sequential programs. We have redesigned Cminor's syntax and semantics to achieve all of these goals. That part of the redesign to achieve target-machine portability was done by Leroy himself. Our redesign to ease its use as an ML back end and for Hoare Logic reasoning was fairly simple. Henceforth in this paper, Cminor will refer to the new version of the Cminor language.

The main contributions of this paper are a small-step semantics suitable for compilation and for Hoare Logic; and the first machine-checked proof of soundness of a sequential Hoare Logic (Separation Logic) w.r.t. a small-step semantics. Schirmer [17] has a machine-checked *big-step* Hoare-Logic soundness proof for a control flow much like ours, extended by Klein *et al.* [10] to a C-like memory model. Ni and Shao [14] have a machine-checked proof of soundness of a Hoare-like logic w.r.t. a small-step semantics, but for an assembly language and for much simpler assertions than ours.

## 2 Big-Step Expression Semantics

The C standard [2] describes a memory model that is byte- and word-addressable (yet portable to big-endian and little-endian machines) with a nontrivial semantics for uninitialized variables. Blazy and Leroy formalized this model [6] for the semantics of Cminor. In C, pointer arithmetic within any malloc'ed block is defined, but pointer arithmetic between different blocks is undefined; Cminor therefore has non-null pointer values comprising an abstract block-number and an int offset. A `NULL` pointer is represented by the integer value 0. Pointer arithmetic between blocks, and reading uninitialized variables, are undefined but not illegal: expressions in Cminor can evaluate to *undefined* (Vundef) without getting stuck.

Each memory load or store is to a non-null pointer value with a "chunk" descriptor *ch* specifying number of bytes, signed or unsigned, int or float. Storing as 32-bit-int then loading as 8-bit-signed-byte leads to an undefined value. Load and store operations on memory, $m \vdash v_1 \overset{ch}{\mapsto} v_2$ and $m' = m[v_1 \overset{ch}{:=} v_2]$, are partial functions that yield results only if reading (resp., writing) a chunk of type *ch* at address $v_1$ is legal. We write $m \vdash v_1 \overset{ch}{\mapsto} v$ to mean that the result of loading from memory $m$ at address $v_1$ a chunk-type *ch* is the value $v$.

The *values* of Cminor are *undefined* (Vundef), integers, pointers, and floats. The int type is an abstract data-type of 32-bit modular arithmetic. The expressions of Cminor are literals, variables, primitive operators applied to arguments, and memory loads.

There are 33 primitive operation symbols *op*; two of these are for accessing global names and local stack-blocks, and the rest is for integer and floating-point arithmetic and comparisons. Among these operation symbols are casts. Cminor casts correspond to all portable C casts. Cminor has an infinite supply ident of variable and function identifiers *id*. As in C, there are two namespaces—each *id*

can be interpreted in a local scope (using $\mathsf{Evar}\,(id)$) or in a global scope (using $\mathsf{Eop}$ with the operation symbol for accessing global names).

$$\begin{aligned} i &: \mathsf{int} ::= [0, 2^{32}) \\ v &: \mathsf{val} ::= \mathsf{Vundef} \mid \mathsf{Vint}\,(i) \mid \mathsf{Vptr}\,(b,i) \mid \mathsf{Vfloat}\,(f) \\ e &: \mathsf{expr} ::= \mathsf{Eval}\,(v) \mid \mathsf{Evar}\,(id) \mid \mathsf{Eop}\,(op, el) \mid \mathsf{Eload}\,(ch, e) \\ el &: \mathsf{exprlist} ::= \mathsf{Enil} \mid \mathsf{Econs}\,(e, el) \end{aligned}$$

*Expression Evaluation.* In original Cminor, expression evaluation is expressed by an inductive big-step relation. Big-step statement execution is problematic for concurrency, but big-step *expression* evaluation is fine even for concurrent programs, since we will use the separation logic to prove noninterference.

Evaluation is deterministic. Leroy chose to represent evaluation as a relation because Coq had better support for proof induction over relations than over function definitions. We have chosen to represent evaluation as a partial function; this makes some proofs easier in some ways: $f(x) = f(x)$ is simpler than $f\,x\,y \Rightarrow f\,x\,z \Rightarrow y = z$. Before Coq's new functional induction tactic was available, we developed special-purpose tactics to enable these proofs. Although we specify expression evaluation as a function in Coq, we present evaluation as a judgment relation in Fig. 1. Our evaluation function is (proved) equivalent to the inductively defined judgment $\Psi; (sp; \rho; \phi; m) \vdash e \Downarrow v$ where:

- $\Psi$ is the "program," consisting of a global environment ($\mathsf{ident} \to \mathsf{option\,block}$) mapping identifiers to function-pointers and other global constants, and a global mapping ($\mathsf{block} \to \mathsf{option\,function}$) that maps certain ("text-segment") addresses to function definitions.
- $sp$ : $\mathsf{block}$. The "stack pointer" giving the address and size of the memory block for stack-allocated local data in the current activation record.
- $\rho$ : $\mathsf{env}$. The local environment, a finite mapping from identifiers to values.
- $\phi$ : $\mathsf{footprint}$. It represents the memory used by the evaluation of an expression (or a statement). It is a mapping from memory addresses to permissions. Leroy's Cminor has no footprints.
- $m$ : $\mathsf{mem}$. The memory, a finite mapping from blocks to block contents [6]. Each block represents the result of a C `malloc`, or a stack frame, a global static variable, or a function code-pointer. A block content consists of the dimensions of the block (low and high bounds) plus a mapping from byte offsets to byte-sized memory cells.
- $e$ : $\mathsf{expr}$. The expression being evaluated.
- $v$ : $\mathsf{val}$. The value of the expression.

Loads outside the footprint will cause expression evaluation to get stuck. Since the footprint may have different permissions for loads than for stores to some addresses, we write $\phi \vdash \mathsf{load}_{ch}\,v$ (or $\phi \vdash \mathsf{store}_{ch}\,v$) to mean that all the addresses from $v$ to $v + |ch| - 1$ are readable (or writable).

To model the possibility of exclusive read/write access or shared read-only access, we write $\phi_0 \oplus \phi_1 = \phi$ for the "disjoint" sum of two footprints, where $\oplus$

$$\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Eval}\,(v) \Downarrow v \qquad \frac{x \in \mathrm{dom}\,\rho}{\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Evar}\,(x) \Downarrow \rho(x)}$$

$$\frac{\Psi; (sp; \rho; \phi; m) \vdash el \Downarrow vl \qquad \Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v}{\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Eop}\,(op, el) \Downarrow v}$$

$$\frac{\Psi; (sp; \rho; \phi; m) \vdash e_1 \Downarrow v_1 \qquad \phi \vdash \mathsf{load}_{ch}\, v_1 \qquad m \vdash v_1 \overset{ch}{\mapsto} v}{\Psi; (sp; \rho; \phi; m) \vdash \mathsf{Eload}\,(ch, e_1) \Downarrow v}$$

**Fig. 1.** Expression evaluation rules

is an associative and commutative operator with several properties such as $\phi_0 \vdash \mathsf{store}_{ch}\, v \;\Rightarrow\; \phi_1 \not\vdash \mathsf{load}_{ch}\, v$, $\phi_0 \vdash \mathsf{load}_{ch}\, v \;\Rightarrow \phi \vdash \mathsf{load}_{ch}\, v$ and $\phi_0 \vdash \mathsf{store}_{ch}\, v \;\Rightarrow \phi \vdash \mathsf{store}_{ch}\, v$. One can think of $\phi$ as a set of fractional permissions [7], with 0 meaning no permission, $0 < x < 1$ permitting read, and 1 giving read/write permission. A store permission can be split into two or more load permissions, which can be reconstituted to obtain a store permission. Instead of fractions, we use a more general and powerful model of sharable permissions similar to one described by Parkinson [16, Ch. 5].

Most previous models of Separation Logic (*e.g.*, Ishtiaq and O'Hearn [9]) represent heaps as partial functions that can be combined with an operator like $\oplus$. Of course, a partial function can be represented as a pair of a domain set and a total function. Similarly, we represent heaps as a footprint plus a Cminor memory; this does not add any particular difficulty to the soundness proofs for our Separation Logic.

To perform arithmetic and other operations, in the third rule of Fig. 1, the judgment $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ takes an operator $op$ applied to a list of values $vl$ and (if $vl$ contains appropriate values) produces some value $v$. Operators that access global names and local stack-blocks make use of $\Psi$ and $sp$ respectively to return the address of a global name or a local stack-block address.

*States.* We shall bundle together $(sp; \rho; \phi; m)$ and call it the *state*, written as $\sigma$. We write $\Psi; \sigma \vdash e \Downarrow v$ to mean $\Psi; (sp_\sigma; \rho_\sigma; \phi_\sigma; m_\sigma) \vdash e \Downarrow v$.

*Notation.* We write $\sigma[:= \rho']$ to mean the state $\sigma$ with its environment component $\rho$ replaced by $\rho'$, and so on (*e.g.* see rules 2 and 3 of Fig. 2 in Section 4).

*Fact.* $\Psi; sp \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v$ and $m \vdash v_1 \overset{ch}{\mapsto} v$ are both deterministic relations, *i.e.* functions.

**Lemma 1.** *$\Psi; \sigma \vdash e \Downarrow v$ is a deterministic relation. (Trivial by inspection.)*

**Lemma 2.** *For any value $v$, there is an expression $e$ such that $\forall \sigma.\ (\Psi; \sigma \vdash e \Downarrow v)$.*

*Proof. Obvious; $e$ is simply* $\mathsf{Eval}\, v$*. But it is important nonetheless: reasoning about programs by rewriting and by Hoare Logic often requires this property, and it was absent from Leroy's Cminor for* Vundef *and* Vptr *values.* ■

An expression may fetch from several different memory locations, or from the same location several times. Because $\Downarrow$ is deterministic, we cannot model a situation where the memory is updated by another thread after the first fetch and before the second. But we want a semantics that describes real executions on real machines. The solution is to evaluate expressions in a setting where we can guarantee *noninterference*. We will do this (in our extension to Concurrent Cminor) by guaranteeing that the footprints $\phi$ of different threads are disjoint.

*Erased Expression Evaluation.* The Cminor compiler (CompCert) is proved correct w.r.t. an operational semantics that does not use footprints. Any program that successfully evaluates with footprints will also evaluate ignoring footprints. Thus, for sequential programs where we do not need noninterference, it is sound to prove properties in a footprint semantics and compile in an erased semantics. We formalize and prove this in the full technical report [4].

## 3 Small-Step Statement Semantics

The statements of sequential Cminor are:

$$\begin{aligned} s : \mathsf{stmt} \quad ::= {} & x := e \mid [e_1]_{ch} := e_2 \mid \mathsf{loop}\, s \mid \mathsf{block}\, s \mid \mathsf{exit}\, n \\ & \mid \mathsf{call}\, xl\, e\, el \mid \mathsf{return}\, el \mid s_1; s_2 \mid \mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \mid \mathsf{skip}. \end{aligned}$$

The assignment $x := e$ puts the value of $e$ into the local variable $x$. The store $[e_1]_{ch} := e_2$ puts (the value of) $e_2$ into the memory-chunk $ch$ at address given by (the value of) $e_1$. (Local variables are not addressable; global variables and heap locations are memory addresses.) To model exits from nested loops, $\mathsf{block}\, s$ runs $s$, which should not terminate normally but which should $\mathsf{exit}\, n$ from the $(n+1)^{th}$ enclosing block, and $\mathsf{loop}\, s$ repeats $s$ infinitely or until it returns or exits. $\mathsf{call}\, xl\, e\, el$ calls function $e$ with parameters (by value) $el$ and results returned back into the variables $xl$. $\mathsf{return}\, el$ evaluates and returns a sequence of results, $(s_1; s_2)$ executes $s_1$ followed by $s_2$ (unless $s_1$ returns or exits), and the statements $\mathsf{if}$ and $\mathsf{skip}$ are as the reader might expect.

Combined with infinite loops and $\mathsf{if}$ statements, blocks and exits suffice to express efficiently all reducible control-flow graphs, notably those arising from C loops. The C statements $\mathsf{break}$ and $\mathsf{continue}$ are translated as appropriate $\mathsf{exit}$ statements. Blazy *et al.* [5] detail the translation of these C statements into Cminor.

*Function Definitions.* A program $\Psi$ comprises two mappings: a mapping from function names to memory blocks (*i.e.*, abstract addresses), and a mapping from memory blocks to function definitions. Each function definition may be written as $f = (xl, yl, n, s)$, where $\mathsf{params}(f) = xl$ is a list of formal parameters, $\mathsf{locals}(f) = yl$ is a list of local variables, $\mathsf{stackspace}(f) = n$ is the size of the local stack-block to which $sp$ points, and the statement $\mathsf{body}(f) = s$ is the function body.

*Operational Semantics.* Our small-step semantics for statements is based on continuations, mainly to allow a uniform representation of statement execution that facilitates the design of lemmas. Such a semantics also avoids all search

$$\Psi \vdash (\sigma,\ (s_1; s_2)\cdot\kappa) \longmapsto (\sigma, s_1 \cdot s_2 \cdot \kappa) \qquad \frac{\Psi;\sigma \vdash e \Downarrow v \quad \rho' = \rho_\sigma[x := v]}{\Psi \vdash (\sigma,\ (x := e)\cdot\kappa) \longmapsto (\sigma[:=\rho'], \kappa)}$$

$$\frac{\Psi;\sigma \vdash e_1 \Downarrow v_1 \qquad \Psi;\sigma \vdash e_2 \Downarrow v_2 \qquad \phi_\sigma \vdash \mathsf{store}_{ch}\, v_1 \qquad m' = m_\sigma[v_1 \overset{ch}{:=} v_2]}{\Psi \vdash (\sigma,\ ([e_1]_{ch} := e_2)\cdot\kappa) \longmapsto (\sigma[:=m'], \kappa)}$$

$$\frac{\Psi;\sigma \vdash e \Downarrow v \qquad \mathsf{is\_true}\, v}{\Psi \vdash (\sigma,\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\cdot\kappa) \longmapsto (\sigma, s_1\cdot\kappa)}$$

$$\frac{\Psi;\sigma \vdash e \Downarrow v \qquad \mathsf{is\_false}\, v}{\Psi \vdash (\sigma,\ (\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2)\cdot\kappa) \longmapsto (\sigma, s_2\cdot\kappa)} \qquad \Psi \vdash (\sigma, \mathsf{skip}\cdot\kappa) \longmapsto (\sigma,\kappa)$$

$$\Psi \vdash (\sigma,\ (\mathsf{loop}\, s)\cdot\kappa) \longmapsto (\sigma, s\cdot\mathsf{loop}\, s\cdot\kappa) \qquad \Psi \vdash (\sigma, (\mathsf{block}\, s)\cdot\kappa) \longmapsto (\sigma, s\cdot\mathsf{Kblock}\,\kappa)$$

$$\frac{j \geq 1}{\Psi \vdash (\sigma, \mathsf{exit}\, 0 \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma,\kappa)}$$

$$\frac{j \geq 1}{\Psi \vdash (\sigma, \mathsf{exit}\,(n+1)\cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa) \longmapsto (\sigma, \mathsf{exit}\, n\cdot\kappa)}$$

**Fig. 2.** Sequential small-step relation. We omit here call and return, which are in the full technical report [4].

rules (congruence rules), which avoids induction over search rules in both the Hoare-Logic soundness proof and the compiler correctness proof.[1]

**Definition 1.** *A* continuation $k$ *has a state* $\sigma$ *and a* control *stack* $\kappa$*. There are sequential control operators to handle local control flow (*Kseq*, written as* $\cdot$*), intraprocedural control flow* (Kblock)*, and function-return* (Kcall)*; this last carries not only a control aspect but an activation record of its own. The control operator* Kstop *represents the safe termination of the computation.*

$$\begin{aligned} \kappa : \mathsf{control} &::= \mathsf{Kstop} \mid s\cdot\kappa \mid \mathsf{Kblock}\,\kappa \mid \mathsf{Kcall}\, xl\, f\, sp\, \rho\, \kappa \\ k : \mathsf{continuation} &::= (\sigma,\kappa) \end{aligned}$$

The sequential small-step function takes the form $\Psi \vdash k \longmapsto k'$ (see Fig. 2), and we define as usual its reflexive transitive closure $\longmapsto^*$. As in C, there is no boolean type in Cminor. In Fig. 2, the predicate $\mathsf{is\_true}\, v$ holds if $v$ is a pointer or a nonzero integer; $\mathsf{is\_false}$ holds only on 0. A store statement $[e_1]_{ch} := e_2$ requires the corresponding store permission $\phi_\sigma \vdash \mathsf{store}_{ch}\, v_1$.

Given a control stack $\mathsf{block}\, s\cdot\kappa$, the small-step execution of the block statement $\mathsf{block}\, s$ enters that block: $s$ becomes the next statement to execute and the control stack becomes $s \cdot \mathsf{Kblock}\,\kappa$.

[1] We have proved in Coq the equivalence of this small-step semantics with the big-step semantics of CompCert (for programs that terminate).

Exit statements are only allowed from blocks that have been previously entered. For that reason, in the two rules for exit statements, the control stack ends with $(\mathsf{Kblock}\,\kappa)$ control. A statement $(\mathsf{exit}\,n)$ terminates the $(n+1)^{th}$ enclosing block statements. In such a block, the stack of control sequences $s_1 \cdots s_j$ following the exit statement is not executed. Let us note that this stack may be empty if the exit statement is the last statement of the most enclosing block. The small-step execution of a statement $(\mathsf{exit}\,n)$ exits from only one block (the most enclosing one). Thus, the execution of an $(\mathsf{exit}\,0)$ statement updates the control stack $(\mathsf{exit}\,0 \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa)$ into $\kappa$. The execution of an $(\mathsf{exit}\ n+1)$ statement updates the control stack $(\mathsf{exit}\,(n+1) \cdot s_1 \cdot \cdots \cdot s_j \cdot \mathsf{Kblock}\,\kappa)$ into $\mathsf{exit}\,n \cdot \kappa$.

**Lemma 3.** *If* $\Psi; \sigma \vdash e \Downarrow v$ *then* $\Psi \vdash (\sigma, (x := e) \cdot \kappa) \longmapsto k'$ *iff* $\Psi \vdash (\sigma, (x := \mathsf{Eval}\,v) \cdot \kappa)) \longmapsto k'$ *(and similarly for other statements containing expressions).*
*Proof. Trivial: expressions have no side effects. A convenient property nonetheless, and not true of Leroy's original Cminor.* ■

**Definition 2.** *A continuation* $k = (\sigma, \kappa)$ *is* stuck *if* $\kappa \neq \mathsf{Kstop}$ *and there does not exist* $k'$ *such that* $\Psi \vdash k \longmapsto k'$.

**Definition 3.** *A continuation* $k$ *is* safe *(written as* $\Psi \vdash \mathsf{safe}(k)$*) if it cannot reach a stuck continuation in the sequential small-step relation* $\longmapsto^*$.

## 4 Separation Logic

Hoare Logic uses triples $\{P\}\, s\, \{Q\}$ where $P$ is a precondition, $s$ is a statement of the programming language, and $Q$ is a postcondition. The assertions $P$ and $Q$ are predicates on the program state. The reasoning on memory is inherently global. Separation Logic is an extension of Hoare Logic for programs that manipulate pointers. In Separation Logic, reasoning is local [15]; assertions such as $P$ and $Q$ describe properties of part of the memory, and $\{P\}\, s\, \{Q\}$ describes changes to part of the memory. We prove the soundness of the Separation Logic via a shallow embedding, that is, we give each assertion a semantic meaning in Coq. We have $P, Q : \mathsf{assert}$ where $\mathsf{assert} = \mathsf{prog} \to \mathsf{state} \to \mathsf{Prop}$. So $P\Psi\sigma$ is a proposition of logic and we say that $\sigma$ satisfies $P$.

*Assertion Operators.* In Fig. 3, we define the usual operators of Separation Logic: the empty assertion **emp**, separating conjunction $*$, disjunction $\vee$, conjunction $\wedge$, implication $\Rightarrow$, negation $\neg$, and quantifier $\exists$. A state $\sigma$ satisfies $P * Q$ if its footprint $\phi_\sigma$ can be split into $\phi_1$ and $\phi_2$ such that $\sigma[:= \phi_1]$ satisfies $P$ and $\sigma[:= \phi_2]$ satisfies $Q$. We also define some novel operators such as expression evaluation $e \Downarrow v$ and base-logic propositions $\lceil A \rceil$.

O'Hearn and Reynolds specify Separation Logic for a little language in which expressions evaluate independently of the heap [15]. That is, their expressions access only the program variables and do not even have *read* side effects on the memory. Memory reads are done by a command of the language, not within expressions. In Cminor we relax this restriction; expressions can read the heap.

$$
\begin{aligned}
\mathbf{emp} &=_{\text{def}} \lambda\Psi\sigma.\ \phi_\sigma = \emptyset \\
P * Q &=_{\text{def}} \lambda\Psi\sigma.\ \exists\phi_1.\exists\phi_2.\ \ \phi_\sigma = \phi_1 \oplus \phi_2\ \wedge\ P(\sigma[:=\phi_1])\ \wedge\ Q(\sigma[:=\phi_2]) \\
P \vee Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \vee Q\sigma \\
P \wedge Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \wedge Q\sigma \\
P \Rightarrow Q &=_{\text{def}} \lambda\Psi\sigma.\ P\sigma \Rightarrow Q\sigma \\
\neg P &=_{\text{def}} \lambda\Psi\sigma.\ \neg(P\sigma) \\
\exists z.P &=_{\text{def}} \lambda\Psi\sigma.\ \exists z.\, P\sigma \\
\lceil A \rceil &=_{\text{def}} \lambda\Psi\sigma.\ A \qquad \text{where } \sigma \text{ does not appear free in } A \\
\mathbf{true} &=_{\text{def}} \lambda\Psi\sigma.\mathbf{True} \qquad\qquad \mathbf{false} =_{\text{def}} \lceil\mathbf{False}\rceil \\
e \Downarrow v\ &=_{\text{def}} \mathbf{emp} \wedge \lceil \mathsf{pure}(e) \rceil \wedge \lambda\Psi\sigma.\, (\Psi; \sigma \vdash e \Downarrow v) \\
\lceil e \rceil_{\mathsf{expr}} &=_{\text{def}} \exists v.\, e \Downarrow v \wedge \lceil \mathsf{is\_true}\, v \rceil \\
\mathsf{defined}(e) &=_{\text{def}} \lceil e \overset{\text{int}}{==} e \rceil_{\mathsf{expr}} \vee \lceil e \overset{\text{float}}{==} e \rceil_{\mathsf{expr}} \\
e_1 \overset{ch}{\mapsto} e_2 &=_{\text{def}} \exists v_1.\exists v_2.(e_1 \Downarrow v_1) \wedge (e_2 \Downarrow v_2) \wedge (\lambda\sigma, m_\sigma \vdash v_1 \overset{ch}{\mapsto} v_2 \wedge \phi_\sigma \vdash \mathsf{store}_{ch}\, v_1) \wedge \mathsf{defined}(v_2)
\end{aligned}
$$

**Fig. 3.** Main operators of Separation Logic

But we say that an expression is *pure* if it contains no Eload operators—so that it cannot read the heap.

In Hoare Logic one can use expressions of the programming language as assertions—there is an implicit coercion. We write the assertion $e \Downarrow v$ to mean that expression $e$ is pure and evaluates to value $v$ in the operational semantics. This is an expression of Separation Logic, in contrast to $\Psi; \sigma \vdash e \Downarrow v$ which is a judgment in the underlying logic. In a previous experiment, our Separation Logic permitted impure expressions in $e \Downarrow v$. But, this complicated the proofs unnecessarily. Having $\mathbf{emp} \wedge \lceil \mathsf{pure}(e) \rceil$ in the definition of $e \Downarrow v$ leads to an easier-to-use Separation Logic.

Hoare Logic traditionally allows expressions $e$ of the programming language to be used as expressions of the program logic. We will define explicitly $\lceil e \rceil_{\mathsf{expr}}$ to mean that $e$ evaluates to a true value (*i.e.* a nonzero integer or non-null pointer). Following Hoare's example, we will usually omit the $\lceil\ \rceil_{\mathsf{expr}}$ braces in our Separation Logic notation.

Cminor's integer equality operator, which we will write as $e_1 \overset{\text{int}}{==} e_2$, applies to integers or pointers, but in several cases it is "stuck" (expression evaluation gives no result): when comparing a nonzero integer to a pointer; when comparing Vundef or $\mathsf{Vfloat}(x)$ to anything. Thus we can write the assertion $\lceil e \overset{\text{int}}{==} e \rceil_{\mathsf{expr}}$ (or just write $e \overset{\text{int}}{==} e$) to test that $e$ is a defined integer or pointer in the current state, and there is a similar operator $e_1 \overset{\text{float}}{==} e_2$.

Finally, we have the usual Separation Logic singleton "maps-to", but annotated with a chunk-type $ch$. That is, $e_1 \overset{ch}{\mapsto} e_2$ means that $e_1$ evaluates to $v_1$, $e_2$ evaluates to $v_2$, and at address $v_1$ in memory there is a defined value $v_2$ of the given chunk-type. Let us note that in this definition, $\mathsf{defined}(v_1)$ is implied by the third conjunct. $\mathsf{defined}(v_2)$ is a design decision. We could leave it out and have a slightly different Separation Logic.

$$\frac{P \Rightarrow P' \qquad \Gamma; R; B \vdash \{P'\}s\{Q'\} \qquad Q' \Rightarrow Q}{\Gamma; R; B \vdash \{P\}s\{Q\}} \qquad \Gamma; R; B \vdash \{P\}\mathsf{skip}\{P\}$$

$$\frac{\Gamma; R; B \vdash \{P\}s_1\{P'\} \qquad \Gamma; R; B \vdash \{P'\}s_2\{Q\}}{\Gamma; R; B \vdash \{P\}s_1; s_2\{Q\}}$$

$$\frac{\rho' = \rho_\sigma[x := v] \qquad P = (\exists v.\, e \Downarrow v \ \wedge\ \lambda\sigma.\ Q\,\sigma[:= \rho'])}{\Gamma; R; B \vdash \{P\}x := e\{Q\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \mathsf{pure}\,(e_2) \qquad P = (e \overset{ch}{\mapsto} e_2 \ \wedge\ \mathsf{defined}(e_1))}{\Gamma; R; B \vdash \{P\}[e]_{ch}{:=}e_1\{e \overset{ch}{\mapsto} e_1\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \Gamma; R; B \vdash \{P \wedge e\}s_1\{Q\} \qquad \Gamma; R; B \vdash \{P \wedge \neg e\}s_2\{Q\}}{\Gamma; R; B \vdash \{P\}\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\{Q\}}$$

$$\frac{\Gamma; R; B \vdash \{I\}s\{I\}}{\Gamma; R; B \vdash \{I\}\mathsf{loop}\ s\{\mathbf{false}\}} \qquad \frac{\Gamma; R; Q \cdot B \vdash \{P\}s\{\mathbf{false}\}}{\Gamma; R; B \vdash \{P\}\mathsf{block}\ s\{Q\}}$$

$$\Gamma; R; B \vdash \{B(n)\}\mathsf{exit}\ n\{\mathbf{false}\}$$

$$\frac{\Gamma; R; B \vdash \{P\}s\{Q\} \qquad \text{modified vars}(s) \ \cap\ \text{free vars}(A) \ =\ \emptyset}{\Gamma; (\lambda vl.A * R(vl)); (\lambda n.A * B(n)) \vdash \{A * P\}s\{A * Q\}}$$

**Fig. 4.** Axiomatic Semantics of Separation Logic (without call and return)

*The Hoare Sextuple.* Cminor has commands to call functions, to exit (from a block), and to return (from a function). Thus, we extend the Hoare triple $\{P\}\,s\,\{Q\}$ with three extra contexts to become $\Gamma; R; B \vdash \{P\}s\{Q\}$ where:

$\Gamma$ : assert describes context-insensitive properties of the global environment;
$R$ : list val $\rightarrow$ assert is the *return environment*, giving the current function's post-condition as a predicate on the list of returned values; and
$B$ : nat $\rightarrow$ assert is the *block environment* giving the exit conditions of each block statement in which the statement $s$ is nested.

Most of the rules of sequential Separation Logic are given in Fig. 4. In this paper, we omit the rules for return and call, which are detailed in the full technical report. Let us note that the $\Gamma$ context is used to update global function names, none of which is illustrated in this paper.

The rule for $[e]_{ch}{:=}e_1$ requires the same store permission than the small-step rule, but in Fig. 4, the permission is hidden in the definition of $e \overset{ch}{\mapsto} e_2$. The rules for $[e]_{ch}{:=}e_1$ and $\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$ require that $e$ be a pure expression. To reason about an such statements where $e$ is impure, one reasons by program transformation using the following rules. It is not necessary to rewrite the actual source program, it is only the local reasoning that is by program transformation.

$$\frac{x, y \text{ not free in } e, e_1, Q \qquad \Gamma; R; B \vdash \{P\}\, x := e;\ y := e_1;\ [x]_{ch} := y\, \{Q\}}{\Gamma; R; B \vdash \{P\} [e]_{ch} := e_1 \{Q\}}$$

$$\frac{x \text{ not free in } s_1, s_2, Q \qquad \Gamma; R; B \vdash \{P\}\, x := e; \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\, \{Q\}}{\Gamma; R; B \vdash \{P\}\, \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\, \{Q\}}$$

The statement $\mathsf{exit}\, i$ exits from the $(i+1)^{th}$ enclosing $\mathsf{block}$. A block environment $B$ is a sequence of assertions $B_0, B_1, \ldots, B_{k-1}$ such that $(\mathsf{exit}\, i)$ is safe as long as the precondition $B_i$ is satisfied. We write $\mathsf{nil_B}$ for the empty block environment and $B' = Q \cdot B$ for the environment such that $B'_0 = Q$ and $B'_{i+1} = B_i$.

Given a block environment $B$, a precondition $P$ and a postcondition $Q$, the axiomatic semantics of a $(\mathsf{block}\, s)$ statement consists in executing some statements of $s$ given the same precondition $P$ and the block environment $Q \cdot B$ (*i.e.* each existing block nesting is incremented). The last statement of $s$ to be executed is an exit statement that yields the **false** postcondition. An $(\mathsf{exit}\, n)$ statement is only allowed from a corresponding enclosing block, *i.e.* the precondition $B(n)$ must exist in the block environment $B$ and it is the precondition of the $(\mathsf{exit}\, n)$ statement.

*Frame Rules.* The most important feature of Separation Logic is the frame rule, usually written

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

The appropriate generalization of this rule to our language with control flow is the last rule of Fig. 4. We can derive from it a *special frame rule* for simple statements $s$ that do not exit or return:

$$\frac{\forall R, B. (\Gamma; R; B \vdash \{P\}\, s\, \{Q\}) \qquad \text{modified vars}(s) \ \cap\ \text{free vars}(A) \ =\ \emptyset}{\Gamma; R; B \vdash \{A * P\}\, s\, \{A * Q\}}$$

*Free Variables.* We use a semantic notion of free variables: $x$ is not free in assertion $A$ if, in any two states where only the binding of $x$ differs, $A$ gives the same result. However, we found it necessary to use a syntactic (inductive) definition of the variables modified by a command. One would think that command $c$ "modifies" $x$ if there is some state such that by the time $c$ terminates or exits, $x$ has a different value. However, this definition means that the modified variables of `if false then` $B$ `else` $C$ are *not* a superset of the modified variables of $C$; this lack of an inversion principle led to difficulty in proofs.

*Auxiliary Variables.* It is typical in Hoare Logic to use auxiliary variables to relate the pre- and postconditions, e.g., the variable $a$ in $\{x = a\}\, x := x+1\, \{x = a+1\}$. In our shallow embedding of Hoare Logic in Coq, the variable $a$ is a Coq variable, not a Cminor variable; formally, the user would prove in Coq the proposition, $\forall a, (\Gamma; R; B \vdash \{P\} s \{Q\})$ where $a$ may appear free in any of $\Gamma, R, B, P, s, Q$. The existential assertion $\exists z. Q$ is useful in conjunction with this technique.

Assertions about functions require special handling of these quantified auxiliary variables. The assertion that some value $f$ is a function with precondition $P$ and postcondition $Q$ is written $f : \underline{\forall} x_1 \underline{\forall} x_2 \ldots \underline{\forall} x_n,\ \{P\}\{Q\}$ where $P$ and $Q$

are functions from value-list to assertion, each $\underline{\forall}$ is an operator of our separation logic that binds a Coq variable $x_i$ using higher-order abstract syntax.

*Application.* In the full technical report [4], we show how the Separation Logic (*i.e.* the rules of Fig. 4) can be used to prove partial correctness properties of programs, with the classical in-place list-reversal example. Such proofs rely on a set of tactics, that we have written in the tactic definition language of Coq, to serve as a proof assistant for Cminor Separation Logic proofs [3].

## 5 Soundness of Separation Logic

Soundness means not only that there is a model for the logic, but that the model is *the* operational semantics for which the compiler guarantees correctness! In principle we could prove soundness by syntactic induction over the Hoare Logic rules, but instead we will give a semantic definition of the Hoare sextuple $\Gamma; R; B \vdash \{P\}\, s\, \{Q\}$, and then prove each of the Hoare rules as a derived lemma from this definition.

A simple example of semantic specification is that the Hoare Logic $P \Rightarrow Q$ is defined, using the underlying logical implication, as $\forall \Psi\sigma.\ P\,\Psi\,\sigma \ \Rightarrow\ Q\,\Psi\,\sigma$. From this, one could prove soundness of the Hoare Logic rule on the left (where the $\Rightarrow$ is a symbol of Hoare Logic) by expanding the definitions into the lemma on the right (where the $\Rightarrow$ is in the underlying logic), which is clearly provable in higher-order logic:

$$\frac{P \Rightarrow Q \qquad Q \Rightarrow R}{P \Rightarrow R} \qquad\qquad \frac{\forall \Psi\sigma.(P\Psi\sigma \Rightarrow Q\Psi\sigma) \quad \forall \Psi\sigma.(Q\Psi\sigma \Rightarrow R\Psi\sigma)}{\forall \Psi\sigma.(P\Psi\sigma \Rightarrow R\Psi\sigma)}$$

**Definition 4.** *(a) Two states $\sigma$ and $\sigma'$ are* equivalent *(written as $\sigma \cong \sigma'$) if they have the same stack pointer, extensionally equivalent environments, identical footprints, and if the footprint-visible portions of their memories are the same. (b) An* assertion *is a predicate on states that is extensional over equivalent environments (in Coq it is a dependent product of a predicate and a proof of extensionality).*

**Definition 5.** *For any control $\kappa$, we define the assertion* **safe** $\kappa$ *to mean that the combination of $\kappa$ with the current state is safe:*

$$\textbf{safe}\ \kappa\ =_{\text{def}}\ \lambda\Psi\sigma.\forall\sigma'.\ (\sigma \cong \sigma'\ \Rightarrow\ \Psi \vdash \mathsf{safe}\,(\sigma', \kappa))$$

**Definition 6.** *Let $A$ be a* frame*, that is, a closed assertion (i.e. one with no free Cminor variables). An assertion $P$* guards *a control $\kappa$ in the frame $A$ (written as $P \,\square_A\, \kappa$) means that whenever $A * P$ holds, it is safe to execute $\kappa$. That is,*

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

*We extend this notion to say that a return-assertion $R$ (a function from value-list to assertion) guards a return, and a block-exit assertion $B$ (a function from block-nesting level to assertions) guards an exit:*

$$R \,\boxbslash_A\, \kappa\ =_{\text{def}}\ \forall vl.R(vl) \,\square_A\, \mathsf{return}\ vl \cdot \kappa \qquad\qquad B \,\boxslash_A\, \kappa\ =_{\text{def}}\ \forall n.B(n) \,\square_A\, \mathsf{exit}\ n \cdot \kappa$$

**Lemma 4.** *If* $P \,\Box_A\, s_1 \cdot s_2 \cdot \kappa$ *then* $P \,\Box_A\, (s_1; s_2) \cdot \kappa$.

**Lemma 5.** *If* $R \,\boxed{\mathbb{R}}_A\, \kappa$ *then* $\forall s, R \,\boxed{\mathbb{R}}_A\, s \cdot \kappa$. *If* $B \,\boxed{\mathbb{B}}_A\, \kappa$ *then* $\forall s, B \,\boxed{\mathbb{B}}_A\, s \cdot \kappa$.

**Definition 7 (Frame).** *A* frame *is constructed from the global environment* $\Gamma$*, an arbitrary frame assertion* $A$*, and a statement* $s$*, by the conjunction of* $\Gamma$ *with the assertion* $A$ *closed over any variable modified by* $s$*:*

$$\mathsf{frame}(\Gamma, A, s) \;=_{\text{def}}\; \Gamma * \mathsf{closemod}(s, A)$$

**Definition 8 (Hoare sextuples).** *The Hoare sextuples are defined in "continuation style," in terms of implications between continuations, as follows:*

$$\Gamma; R; B \vdash \{P\}\, s\, \{Q\} \;=_{\text{def}}\; \forall A, \kappa.\\ R \,\boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,s)}\, \kappa \wedge B \,\boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,s)}\, \kappa \wedge Q \,\Box_{\mathsf{frame}(\Gamma,A,s)}\, \kappa \;\Rightarrow\; P \,\Box_{\mathsf{frame}(\Gamma,A,s)}\, s \cdot \kappa$$

From this definition we prove the rules of Fig. 4 as derived lemmas.

It should be clear from the definition—after one gets over the backward nature of the continuation transform—that the Hoare judgment specifies partial correctness, not total correctness. For example, if the statement $s$ infinitely loops, then the continuation $(\sigma, s \cdot \kappa)$ is automatically safe, and therefore $P \,\Box_A\, s \cdot \kappa$ always holds. Therefore the Hoare tuple $\Gamma; R; B \vdash \{P\} s \{Q\}$ will hold for that $s$, regardless of $\Gamma, R, B, P, Q$.

*Sequence.* The soundness of the sequence statement is the proof that if the hypotheses $\mathsf{H}_1 : \Gamma; R; B \vdash \{P\}\, s_1\, \{P'\}$ and $\mathsf{H}_2 : \Gamma; R; B \vdash \{P'\}\, s_2\, \{Q\}$ hold, then we have to prove $\mathsf{Goal} : \Gamma; R; B \vdash \{P\}\, s_1; s_2\, \{Q\}$ (see Fig. 4). If we unfold the definition of the Hoare sextuples, $\mathsf{H}_1$, $\mathsf{H}_2$ and $\mathsf{Goal}$ become:

$$(\forall A, \kappa)\frac{R \,\boxed{\mathbb{R}}_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, \kappa \qquad B \,\boxed{\mathbb{B}}_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, \kappa \qquad Q \,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, \kappa}{P \,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, (s_1; s_2) \cdot \kappa}\ \mathsf{Goal}$$

We prove $P \,\Box_{\mathsf{frame}(\Gamma,A,(s_1;s_2))}\, (s_1; s_2) \cdot k$ using Lemma 4:[2]

$$\dfrac{\dfrac{\dfrac{R \,\boxed{\mathbb{R}}\, k}{R \,\boxed{\mathbb{R}}\, s_2 \cdot k}\,\mathsf{Lm.\ 5} \qquad \dfrac{B \,\boxed{\mathbb{B}}\, k}{B \,\boxed{\mathbb{B}}\, s_2 \cdot k}\,\mathsf{Lm.\ 5} \qquad \dfrac{R \,\boxed{\mathbb{R}}\, k \quad B \,\boxed{\mathbb{B}}\, k \quad Q \,\Box\, k}{P' \,\Box\, s_2 \cdot k}\,\mathsf{H}_2}{P \,\Box\, s_1 \cdot s_2 \cdot k}\,\mathsf{H}_1}{P \,\Box\, (s_1; s_2) \cdot k}\ \mathsf{Lm.\ 4}$$

*Loop Rule.* The loop rule turns out to be one of the most difficult ones to prove. A loop continues executing until the loop-body performs an exit or return. If loop $s$ executes $n$ steps, then there will be 0 or more complete iterations of $n_1, n_2, \ldots$ steps, followed by $j$ steps into the last iteration. Then either there is an exit (or return) from the loop, or the loop will keep going. But if the exit is from an inner-nested block, then it does not terminate the loop (or even this iteration). Thus we need a formal notion of when a statement exits.

[2] We will elide the frames from proof sketches by writing $\Box$ without a subscript; this particular proof relies on a lemma that $\mathsf{closemod}(s_1, \mathsf{closemod}((s_1; s_2), A)) = \mathsf{closemod}((s_1; s_2), A)$.

Consider the statement $s = \mathsf{if}\, b\, \mathsf{then}\, \mathsf{exit}\, 2\, \mathsf{else}\, (\mathsf{skip}; x := y)$, executing in state $\sigma$. Let us execute $n$ steps into $s$, that is, $\Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^n (\sigma', \kappa')$. If $n$ is small, then the behavior should not depend on $\kappa$; only when we "emerge" from $s$ is $\kappa$ important. In this example, if $\rho_\sigma b$ is a true value, then as long as $n \leq 1$ the statement $s$ can *absorb* $n$ steps independent of $\kappa$; if $\rho_\sigma b$ is a false value, then $s$ can absorb up to 3 steps. To reason about absorption, we define the concatenation $\kappa_1 \circ \kappa_2$ of a control prefix $\kappa_1$ and a control $\kappa_2$ as follows:

$$\begin{array}{ll} \mathsf{Kstop} \circ \kappa =_{\text{def}} \kappa & (\mathsf{Kblock}\, \kappa') \circ \kappa =_{\text{def}} \mathsf{Kblock}\, (\kappa' \circ \kappa) \\ (s \cdot \kappa') \circ \kappa =_{\text{def}} s \cdot (\kappa' \circ \kappa) & (\mathsf{Kcall}\, xl\, f\, sp\, \rho\, \kappa') \circ \kappa =_{\text{def}} \mathsf{Kcall}\, xl\, f\, sp\, \rho\, (\kappa' \circ \kappa) \end{array}$$

$\mathsf{Kstop}$ is the empty prefix; $\mathsf{Kstop} \circ \kappa$ does not mean "stop," it means $\kappa$.

**Definition 9 (absorption).** *A statement $s$ in state $\sigma$* $\mathsf{absorbs}$ *$n$ steps (written as* $\mathsf{absorb}(n, s, \sigma)$*) iff* $\forall j \leq n. \exists \kappa_{\text{prefix}}. \exists \sigma'.\ \forall \kappa.\ \Psi \vdash (\sigma, s \cdot \kappa) \longmapsto^j (\sigma', \kappa_{\text{prefix}} \circ \kappa)$.

*Example 1.* An $\mathsf{exit}$ statement by itself absorbs no steps (it immediately uses its control-tail), but $\mathsf{block}\, (\mathsf{exit}\, 0)$ can absorb the 2 following steps:
$\Psi \ \vdash\ (\sigma, \mathsf{block}\, (\mathsf{exit}\, 0) \cdot \kappa) \longmapsto (\sigma, \mathsf{exit}\, 0 \cdot \mathsf{Kblock}\, \kappa) \longmapsto (\sigma, \kappa)$

**Lemma 6.** *1.* $\mathsf{absorb}(0, s, \sigma)$.
*2.* $\mathsf{absorb}(n+1, s, \sigma) \ \Rightarrow\ \mathsf{absorb}(n, s, \sigma)$.
*3. If* $\neg\mathsf{absorb}(n, s, \sigma)$*, then* $\exists i < n. \mathsf{absorb}(i, s, \sigma) \wedge \neg\mathsf{absorb}(i+1, s, \sigma)$*. We say that $s$ absorbs at most $i$ steps in state $\sigma$.*

**Definition 10.** *We write $(s;)^n s'$ to mean* $\underbrace{s; s; \ldots; s;}_{n}\ s'$.

**Lemma 7.** $$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} (s;)^n \mathsf{loop}\, \mathsf{skip} \{\mathbf{false}\}}$$

*Proof. For $n = 0$, the infinite-loop* $(\mathsf{loop}\, \mathsf{skip})$ *satisfies any precondition for partial correctness. For $n+1$, assume $\kappa$, $R \,\boxed{\mathsf{R}}\, \kappa$, $B \,\boxed{\mathsf{B}}\, \kappa$; by the induction hypothesis (with $R \,\boxed{\mathsf{R}}\, \kappa$ and $B \,\boxed{\mathsf{B}}\, \kappa$) we know $I \,\square\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$. We have $R \,\boxed{\mathsf{R}}\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ and $B \,\boxed{\mathsf{B}}\, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ by Lemma 5. We use the hypothesis $\Gamma; R; B \vdash \{I\} s \{I\}$ to augment the result to $I \,\square\, (s; (s;)^n \mathsf{loop}\, \mathsf{skip}) \cdot \kappa$.* ■

**Theorem 1.** $$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} \mathsf{loop}\, s \{\mathbf{false}\}}$$

*Proof. Assume $\kappa$, $R \,\boxed{\mathsf{R}}\, \kappa$, $B \,\boxed{\mathsf{B}}\, \kappa$. To prove $I \,\square\, \mathsf{loop}\, s \cdot \kappa$, assume $\sigma$ and $I\sigma$ and prove* $\mathsf{safe}\, (\sigma, \mathsf{loop}\, s \cdot \kappa)$*. We must prove that for any $n$, after $n$ steps we are not stuck. We unfold the loop $n$ times, that is, we use Lemma 7 to show* $\mathsf{safe}\, (\sigma, (s;)^n \mathsf{loop}\, \mathsf{skip} \cdot \kappa)$*. We can show that if this is safe for $n$ steps, so is $\mathsf{loop}\, s \cdot \kappa$ by the principle of absorption. Either $s$ absorbs $n$ steps, in which case we are done; or $s$ absorbs at most $j < n$ steps, leading to a state $\sigma'$ and a control (respectively) $\kappa_{prefix} \circ (s;)^{n-1} \mathsf{loop}\, \mathsf{skip} \cdot \kappa$ or $\kappa_{prefix} \circ \mathsf{loop}\, s \cdot \kappa$. Now, because $s$ cannot absorb $j+1$ steps, we know that either $\kappa_{prefix}$ is empty (because $s$ has terminated normally) or $\kappa_{prefix}$ starts with a* $\mathsf{return}$ *or* $\mathsf{exit}$*, in which case we*

*escape (resp. past the* loop skip *or the* loop $s$*) into* $\kappa$*. If* $\kappa_{prefix}$ *is empty then we apply strong induction on the case* $n - j$ *steps; if we escape, then* $(\sigma', \kappa)$ *is safe iff* $(\sigma, \mathsf{loop}\, s \cdot \kappa)$ *is safe. (For example, if* $j = 0$*, then it must be that* $s =$ return *or* $s =$ exit *, so in one step we reach* $\kappa_{prefix} \circ (\mathsf{loop}\, s \cdot \kappa)$ *with* $\kappa_{prefix} =$ return *or* $\kappa_{prefix} =$ exit *.)* ■

## 6 Sequential Reasoning About Sequential Features

Concurrent Cminor, like most concurrent programming languages used in practice, is a sequential programming language with a few concurrent features (locks and threads) added on. We would like to be able to reason about the sequential features using purely sequential reasoning. If we have to reason about all the many sequential features without being able to assume such things as determinacy and sequential control, then the proofs become much more difficult.

One would expect this approach to run into trouble because critical assumptions underlying the sequential operational semantics would not hold in the concurrent setting. For example, on a shared-memory multiprocessor we cannot assume that (x:=x+1; x:=x+1) has the same effect as (x:=x+2); and on any real multiprocessor we cannot even assume *sequential consistency*—that the semantics of $n$ threads is some interleaving of the steps of the individual threads.

We will solve this problem in several stages. Stage 1 of this plan is the current paper. Stages 2, 3, and 4 are work in progress; the remainder is future work.

1. We have made the language, the Separation Logic, and our proof extensible: the set of control-flow statements is fixed (inductive) but the set of straight-line statements is extensible by means of a parameterized module in Coq. We have added to each state $\sigma$ an *oracle* which predicts the meaning of the extended instruction (but which does nothing on the core language). All the proofs we have described in this paper are on this extensible language.
2. We define spawn, lock, and unlock as extended straight-line statements. We define a concurrent small-step semantics that assumes noninterference (and gets "stuck" on interference).
3. From this semantics, we calculate a single-thread small-step semantics equipped with the oracle that predicts the effects of synchronizations.
4. We define a Concurrent Separation Logic for Cminor as an extension of the Sequential Separation Logic. Its soundness proof uses the sequential soundness proof as a lemma.
5. We will use Concurrent Separation Logic to guarantee noninterference of source programs. Then (x:=x+1; x:=x+1) *will* have the same effect as (x:=x+2).
6. We will prove that the Cminor compiler (CompCert) compiles each footprint-safe source thread into an equivalent footprint-safe machine-language thread. Thus, noninterfering source programs will produce noninterfering machine-language programs.

7. We will demonstrate, with respect to a formal model of weak-memory-consistency microprocessor, that noninterfering machine-language programs give the same results as they would on a sequentially consistent machine.

## 7 The Machine-Checked Proof

We have proved in Coq the soundness of Separation Logic for Cminor. Each rule is proved as a lemma; in addition there is a main theorem that if you prove all your function bodies satisfy their pre/postconditions, then the program "call main()" is safe. We have informally tested the adequacy of our result by doing tactical proofs of small programs [3].

| Lines | Component |
|---|---|
| 41 | Axioms: dependent unique choice, relational choice, extensionality |
| 8792 | Memory model, floats, 32-bit integers, values, operators, maps (exactly as in CompCert [12]) |
| 4408 | Sharable permissions, Cminor language, operational semantics |
| 462 | Separation Logic operators and rules |
| 9874 | Soundness proof of Separation Logic |

These line counts include some repetition of specifications (between Modules and Module Types) in Coq's module system.

## 8 Conclusion

In this paper, we have defined a formal semantics for the language Cminor. It consists of a big-step semantics for expressions and a small-step semantics for statements. The small-step semantics is based on continuations mainly to allow a uniform representation of statement execution. The small-step semantics deals with nonlocal control constructs (return, exit) and is designed to extend to the concurrent setting.

Then, we have defined a Separation Logic for Cminor. It consists of an assertion language and an axiomatic semantics. We have extended classical Hoare triples to sextuples in order to take into account nonlocal control constructs. From this definition of sextuples, we have proved the rules of axiomatic semantics, thus proving the soundness of our Separation Logic.

We have also proved the semantic equivalence between our small-step semantics and the big-step semantics of the CompCert certified compiler, so the Cminor programs that we prove in Separation Logic can be compiled by the CompCert certified compiler. We plan to connect a Cminor certified compiler directly to the small-step semantics, instead of going through the big-step semantics.

Small-step reasoning is useful for sequential programming languages that will be extended with concurrent features; but small-step reasoning about nonlocal control constructs mixed with structured programming (loop) is not trivial. We have relied on the determinacy of the small-step relation so that we can define concepts such as $\mathsf{absorb}(n, s, \sigma)$.

## References


1. The Coq proof assistant, `http://coq.inria.fr`
2. American National Standard for Information Systems – Programming Language – C: American National Standards Institute (1990)
3. Appel, A.W.: Tactics for separation logic (January 2006), `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`
4. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor (extended version). Technical Report RR 6138, INRIA (March 2007), `https://hal.inria.fr/inria-00134699`
5. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)
6. Blazy, S., Leroy, X.: Formal verification of a memory model for C-like imperative languages. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 280–299. Springer, Heidelberg (2005)
7. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL'05, pp. 259–270. ACM Press, New York (2005)
8. Dargaye, Z.: Décurryfication certifiée. JFLA (Journées Françaises des Langages Applicatifs), 119–133 (2007)
9. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: POPL'01, January 2001, pp. 14–26. ACM Press, New York (2001)
10. Klein, G., Tuch, H., Norrish, M.: Types, bytes, and separation logic. In: POPL'07, January 2007, pp. 97–108. ACM Press, New York (2007)
11. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM'05. IEEE Conference on Software Engineering and Formal Methods, IEEE Computer Society Press, Los Alamitos (2005)
12. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: POPL'06, pp. 42–54. ACM Press, New York (2006)
13. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning 5(4), 461–492 (1989)
14. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: POPL'06, January 2006, pp. 320–333. ACM Press, New York (2006)
15. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
16. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
17. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)